SOME GRAPH ALGORITHMS FOR MOLECULAR SWITCHING DEVICES

MOHAMMAD MONOWAR HOSSAIN

Some Graph Algorithms for Molecular Switching Devices

by

Mohammad Monowar Hossain

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Master of Science

Department of Computer Science
Memorial University of Newfoundland

November 2006

St. John's Newfoundland

Your file Votre référence
ISBN: 978-0-494-31256-8
Our file Notre référence
ISBN: 978-0-494-31256-8

Abstract

Molecular computing is a promising area for researchers from several disciplines. Currently many theoretical and application-oriented scientists are turning towards molecular computing with the goal to develop a molecular computer. Digital circuits for molecular devices are designed at the molecular level. A digital circuit will be thousands of times smaller if we can develop switching elements from appropriate molecules by using a direct chemical procedure. To develop such circuits we need to understand the nature of molecular switching in principle. The concept soliton automaton has been introduced to model this phenomena using graph matchings. The goal of my thesis was to develop and implement graph algorithms for soliton automata.

Acknowledgements

First would like to thank my supervisor Dr. Miklos Bartha for his continuous guidance and suggestions. Without his guidance and support this thesis would not have taken this final form. I would also like to thank all the faculty and staff of Computer Science Department for their assistance and support throughout my program. Finally, I would like to thank my family and friends and others who have contributed to my research and thesis.

Table of Contents

List of Figures

Chapter 1

Introduction

Molecular computing is a promising area for researchers from several disciplines. Currently many theoretical and application-oriented scientists are turning towards molecular computing with the goal to develop a molecular computer. Electronics is the major obstacle in reducing the size and increasing the speed of conventional switching devices. New principles like bioelectronics or molecular electronics have been introduced to overcome these problems. The idea of molecular switching dates back to the early 1930's, when science fiction introduced some molecular devices [17]. Feynman was one of the pioneer researchers who brought up the idea of building real molecular devices. He introduced his ideas in [31]. Aviram and Ranter also dealt with molecular electronic devices in their paper [32], which encouraged F. L. Cater to continue their study. Some further interesting ideas (e.g. biological systems) have been proposed by Adleman [33] and Conrad[34].

Digital circuits for molecular devices are designed at the molecular level. A digital circuit will be thousands of times smaller if we can develop switching elements from appropriate molecules by a direct chemical procedure. For this type of molecular circuits, chemical molecules will be used as electronic switches and will be interconnected by some sort of ultra-fine conducting wires. Carter [20] introduced an idea to construct such type of conductors. In his proposed technique, Carter used single strands of electrically conductive plastic polyacetylene and electrons. Electrons are thought to travel along polyacetylene in little packets called soliton. Molecular scale electronic devices are constructed from molecular switches and polyacetylene chains, which are called soliton circuits.

Polyacetylene consists of a chain of carbon atoms held together by alternating double and single bonds. Each carbon atom is also bonded to a hydrogen atom. Polyacetylene has two stable states, which differ in the position of the alternating double and single bonds with respect to the carbon atoms. A soliton is a moving wave which causes conversion between the two states of polyacetylene. When a soliton wave passes through a polyacetylene, it effectively selects one arrangement of bonds and ignores others. Fig. 1.1 shows how a soliton wave affects the bonds in a polyacetylene.

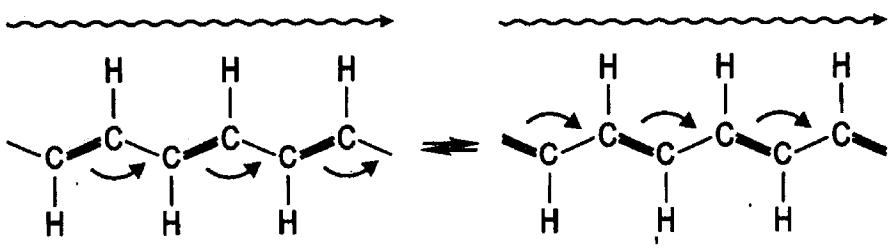

Figure. 1.1: Effect of soliton in a polyacetylene

Several soliton-based computational models have been proposed in the last few years. A good survey of such models can be found in [35]. In this thesis we have used a mathematical model of soliton circuits known as soliton automata. The concept of soliton automata was first introduced by J. Dassow and H. Jürgensen in 1990 [22]. The intention behind this model was to investigate the logical aspects of soliton switching.

Graph theory plays a significant role in the study of soliton automata because the underlying object of soliton automaton is a finite undirected graph, called a soliton graph. In soliton graphs vertices correspond to the atoms or certain group of atoms, whereas the edges represent chemical bonds or chains of bonds. The multiplicity of bonds (single or double) is fixed by a weight assignment to the edges. We assume that molecules consist of carbon and hydrogen atoms only. In a soliton graph, some vertices are distinguished as external, and their role is to accept or donate electrons for the remaining part of the molecular network. In the later chapters of this thesis, we will provide a detailed discussion on soliton automata and their underlying soliton graphs. In this chapter we are limiting our focus to the intuitive understanding of soliton graphs and automata.

The analysis of soliton automata is a complex task, and only few special cases have been analyzed so far. These are: soliton automata with a single external vertex [23], automata with a single cycle [24], automata obtained by the general product of strongly deterministic soliton automata [25], and the transition monoids of tree-based soliton automata have been studied in [26]. M. Bartha and E. Gombas first recognized the connection between soliton

automata and matching theory [1], which opens up a new perspective for the study of soliton automata. They used the concept of perfect internal matching (matching covering all internal vertices of a graph) for characterizing soliton graphs. In a soliton graph, the edges belonging to a perfect internal matching correspond to the double bonds in an appropriate state of a molecule chain. After the paper [1], soliton graphs and automata have been systematically studied in a sequence of papers on the ground of matching theory [2][4][5][10].

When using matching theory to describe soliton automata, we are analyzing the internal structure of a chemical compound (e.g. polyacetylene). Using matching theory for this purpose is not a new idea. For many years, chemists have used graph matchings to analyze different chemical compounds which have an alternating pattern of single and double bond. Graphs corresponding to such compounds are called **Hückel graphs** in the literature. For more information on Hückel graphs, see [28, Section 8.7]. Hückel and soliton graphs are quite similar, although there is an important difference between them. Hückel graphs generally have a perfect matching, whereas soliton graphs only possess a matching that covers all of the internal vertices (vertices with degree greater than one).

The study of soliton switching is not limited to theory. There is also some practical research going on. As an example, a research project funded by Circadian Technologies shows promising results. A series of papers showed that an appropriate chemical structure, which is able to communicate with solitons, could be used as an electronic switching

device [36][37][38][39]. In this thesis, however, we concentrate on the mathematical model of soliton circuits. A thorough analysis of the mathematical model can tell us a lot about the behavior of a soliton circuit, so we can verify certain properties of such circuits without actually building them. Therefore, the elaboration of a proper mathematical model is a significant step towards developing molecular switching devices.

There is an intriguing connection between soliton automata and the recent efforts by Abramsky [41] and others to revive Girard's geometry of interaction program. As it turns out, soliton automata can be given the structure of a self-dual compact closed category, which qualifies such automata as simple but very efficient reversible computation models. Research in this direction is under development.

In this thesis we have worked out three algorithms, which are all related to soliton graphs. These algorithms will either test some important property of soliton graphs, or perform a simplifying transformation on such graphs. To develop these algorithms, we need a thorough understanding of matching theory and the structure of soliton graphs. We devote separate chapters to matching theory and soliton automata in order to provide the necessary background. Most of the terms used here to explain the algorithms will be dealt with in later chapters.

The first algorithm reduces a graph, using its incidence matrix. The second algorithms can decide if a given graph is an elementary deterministic soliton graph or not. The third

algorithm can test whether a given graph is a deterministic viable soliton graph containing an alternating cycle. Different intermediate steps of the second and third algorithms will also determine some other important properties of soliton graphs (e.g. being a generalized tree or a baby chestnut).

After the design of the above algorithms, I have implemented and tested all three of them. The algorithms are discussed in separate chapters, which also indicate the major underlying theoretical results.

The thesis consists of eight chapters. Chapter 1 is a short introduction containing a preliminary discussion on molecular switching devices and their history, soliton graphs and automata. In Chapter 2, we introduce the idea of molecular switching and F. L. Carter's mechanism for soliton switching. Chapter 3 is a revision of basis concepts in matching theory, which are related to this thesis. In Chapter 4, the reader will find the exact definition of soliton automata and soliton graphs. This chapter contains most of the theoretical background used in the algorithms. This chapter also contains a detailed discussion on the structure of soliton graphs. Chapter 5, 6, and 7 contain the description of the first, second, and third algorithm, respectively. Finally, Chapter 8 concludes this thesis.

Chapter 2

Molecular Switching and Carter's Mechanism

Before discussing soliton automata and soliton graphs in detail, we present some heuristics about the concepts of molecular switching. This chapter will provide some answers as to how Carter's experimental technique to explore molecular level switching leads to the development of mathematical models like Soliton Automata. Most of the observations in this chapter originate form Carter's paper 'Conformational switching at the molecular level' [18]. The reader is referred to that work for more details.

2.1 Basic Concepts

A soliton is like a particle that can move in one or two dimensions on a microscopic scale. The single-double bond rearrangement of a conjugated system is the key idea of

propagating a soliton through such a system. When a soliton passes through a conjugated system, it causes the exchange of single and double bonds along its passage. In most of the examples of his paper, Carter used polyacetylene as a conjugated system. As mentioned before, Polyacetylene consists of a chain of carbon atoms held together by alternating double and single bonds and each carbon atom is also bonded to a hydrogen atom. See Fig. 1.1.

2.2 Push-Pull Olefin

'Push-pull' is an important concept introduced by Carter. He showed special interest in the push-pull distributed olefin (1,1-N, N-dimethyl-2-nitroethenamine) (See Fig. 2.1) because it can be embedded in a transpolyacetylene and also can be switched off by the propagation of a soliton. See Fig. 2.2.

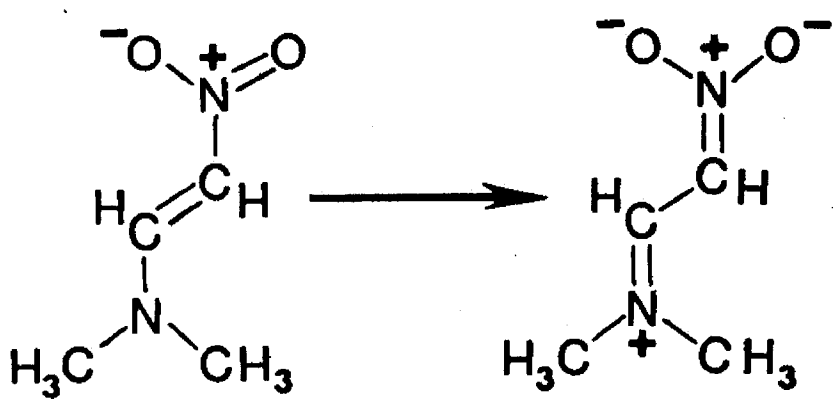

Figure. 2.1: Push-Pull OLEFIN

After the horizontal soliton passage, resulting in the arrangement of bonds shown at the bottom of Fig. 2.2, the vertical switch of the embedded push-pull olefin molecule is no longer possible.

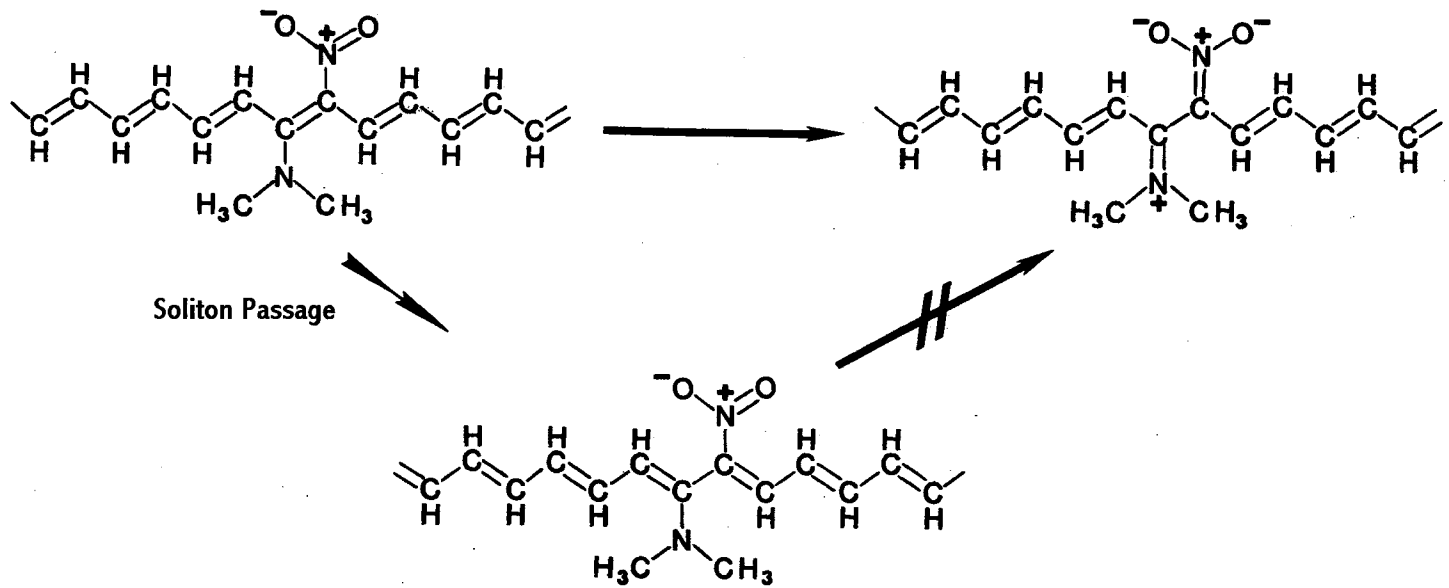

Figure. 2.2: Soliton Switching

2.3 Soliton Gang Switching

Carter extended the concept of soliton switching by introducing gang switching. The concept of gang switching implies that there may be more than one chain of polyacetylene and push-pull structure or extended chromophores (chromophore is a part of a molecule which is responsible for color). Carter gave an example where he had two chains and two different push-pull structures. Fig. 2.3 shows that example. In this example the passage of a soliton down chain 1 will turn the first chromophore on and the second off; a soliton moving down chain 2 will turn both of them off. Turning a chromophore off means that it is not possible to propagate a soliton through the vertical chain of alternating single and double bonds in that chromophore.

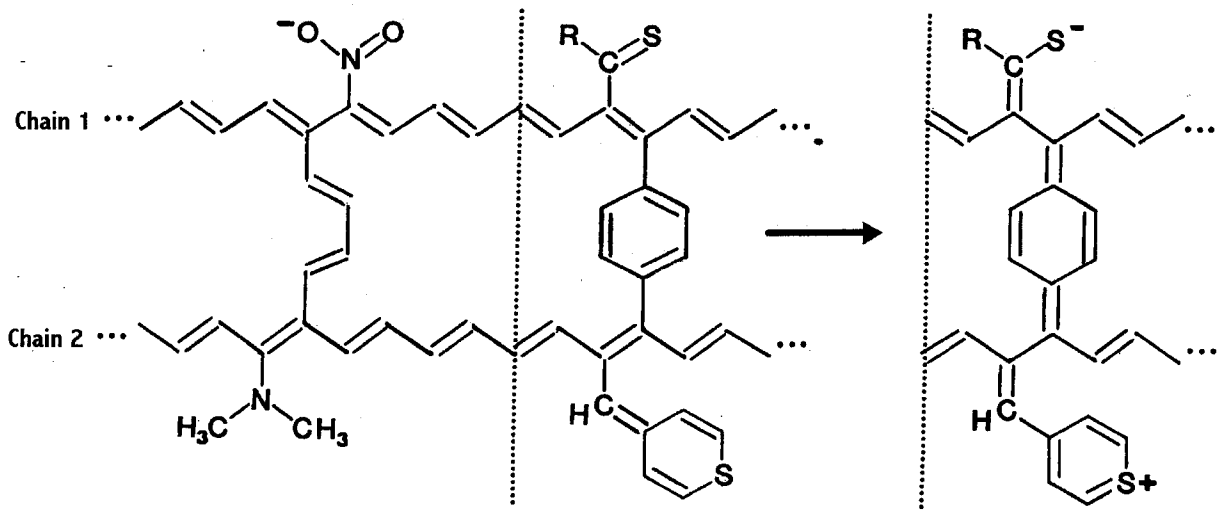

Fig. 2.3: Soliton switching involving two transpolyacetylene chains and two chromophores.

Carter even generalized the concept of soliton gang switching. In Fig. 2.4, where A, C, and D, are generalized electron acceptors, conjugated connectors, and electron donors, respectively. In this figure, chromophores are separated from each other by dotted lines, which imply eight different chromophore-chain relationships relative to the conformations of chains 1, 2 and 3. See [18] for details.

In another example, Carter extends the generalization concept to a very advanced level, which goes beyond our meager knowledge of soliton propagation. He has replaced the electron acceptor and donor groups with molecular 'wires' or filaments of -(SN) -. Fig. 2.5 illustrates this example.

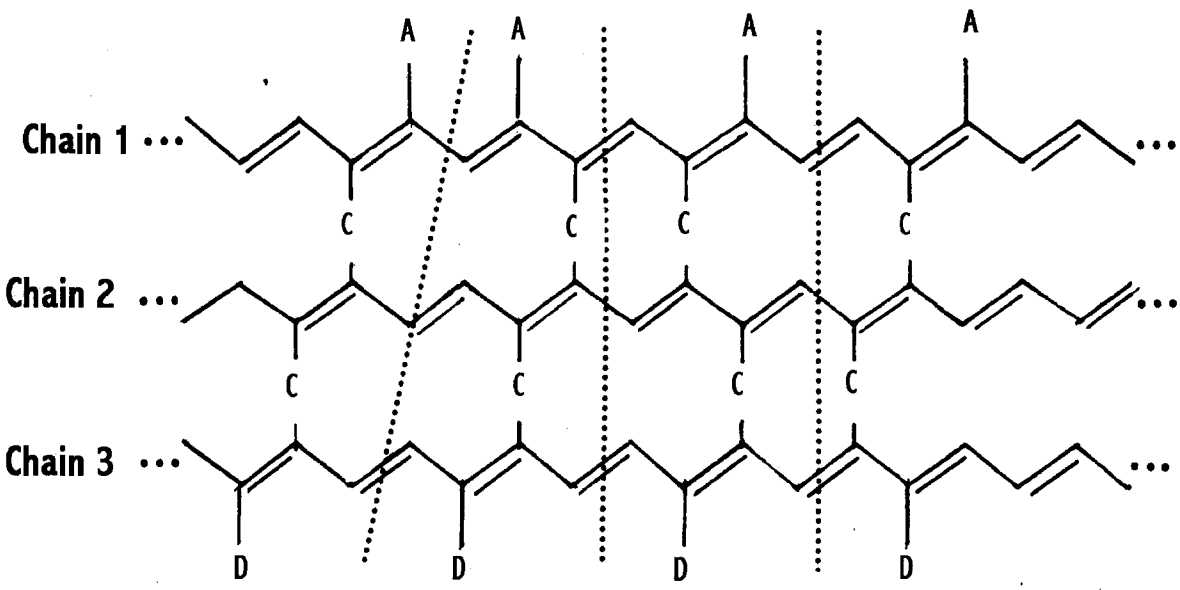

Figure. 2.4: Soliton Gang Switching

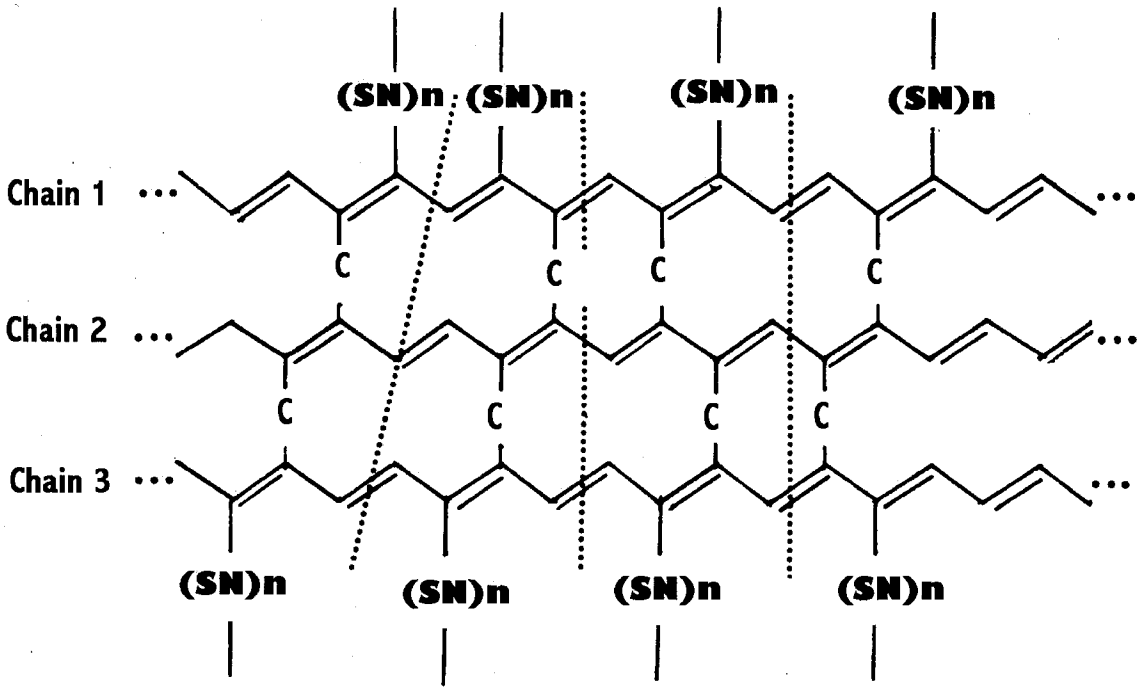

Figure. 2.5: Soliton Gang Switching with –(SN)- acceptor

2.4 Soliton Valving

Carter also described the valving behavior of soliton propagation in a conjugated system. Fig. 2.6 is illustrates an example. In this example the passage of a soliton from X to Y (or from Y to X) moves the double bond at the branch carbon from chain X to the chain Y. Moreover, in the upper right portion of the figure, a soliton moving from Y to Z shifts the double bond to the chain Z.

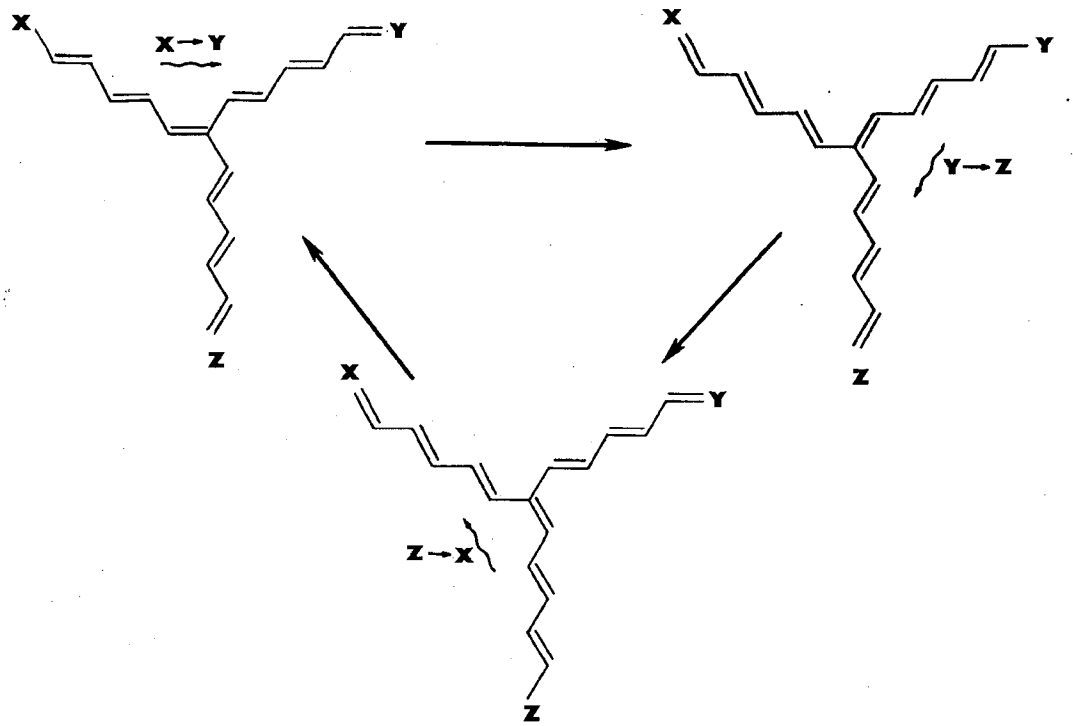

Figure. 2.6: Soliton Valving

2.5 Soliton Memory Element

Finally Carter proposed a structure called a soliton memory element, which shows us a real world use of soliton propagation. In his model, the access time of an information bit and the number of bits clearly depend on the soliton velocity and the length of the conjugated polymer linking the soliton generator and the electron tunnel switch. The structure of this memory element is illustrated in Fig. 2.7.

This memory element can store four bits (in duplicate) at a time (two bits on the upper chain and two bits on the lower chain). In this memory element solitons are temporarily stored on transpolyacetelylene chains. These chains connect the soliton generator first with the soliton reverser which reflects the solitons and with the control groups (CGs), which regulate the depth of the potential wells and hence the pseudostationary state energies of soliton. The control groups are gates to the multi-barrier electron tunnel detector (switch). See [18] for more details.

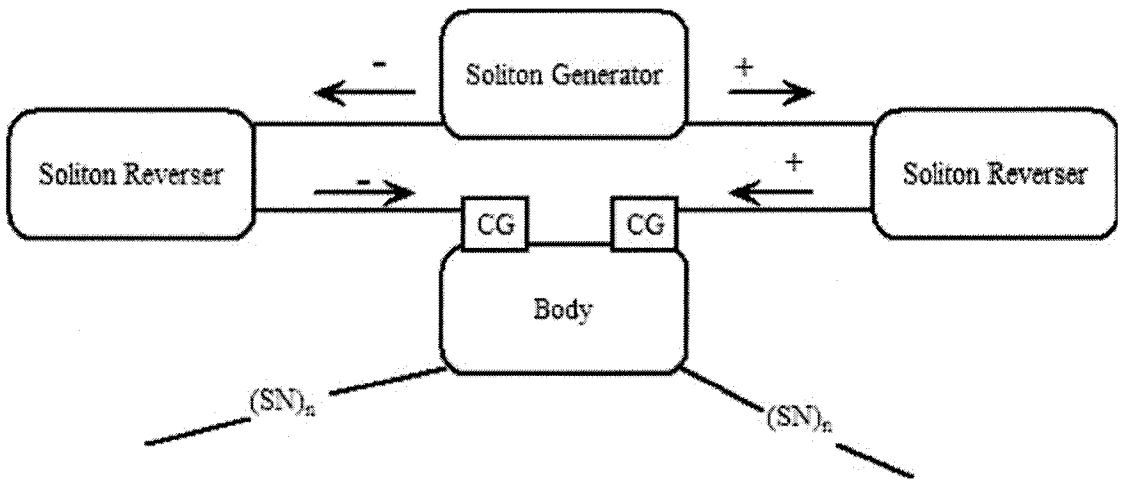

Figure. 2.7: Soliton Memory Element

Chapter 3

Graph and Matching Theory

The underlying object of a soliton automaton is a soliton graph, and the definition of soliton graphs is based on graph matchings. Thus, matching theory becomes an integral part of this thesis. One must have a strong background in graph theory in order to understand the structural analysis of soliton graphs. The goal of this chapter is to provide an overview of the most important concepts on graphs and graph matchings. The reader is referred to [28] for a comprehensive study of matching theory. The notation and terminology used in this thesis is compatible with that work.

• Graph

By a ***graph*** we mean a finite undirected graph with multiple edges and loops allowed. For a graph G, $V(G)$ and $E(G)$ will denote the set of vertices and the set of edges of G, respectively. An edge $e \in E(G)$ connects two vertices $v_1, v_2 \in V(G)$, which are said to be

adjacent in G. The vertices v_1 and v_2 are called the ***endpoints*** of e. If $v_1 = v_2$, then e is called a ***loop*** around v_1.

- **Degree of a vertex**

In graph G, the degree of a vertex v is the number of occurrences of v as an endpoint of some edge in $E(G)$. According to this definition, every loop around v contributes two occurrences to the count.

- **External, Internal and Isolated**

If the degree of a vertex v is one, then that vertex is called ***external***. If the degree of v is greater than one, then v is ***internal***, and v is ***isolated*** if its degree is 0. An edge $e \in E(G)$ is said to be an external edge if one of its endpoints is an external vertex. Internal edges are those that are not external. The sets of external and internal vertices of G will be denoted by $Ext(G)$ and $Int(G)$, respectively.

Graph G is said to be ***open*** if it has at least one external vertex, and G is ***closed*** if all the vertices of G are internal.

- **Graph Matching:**

A ***matching*** M of graph G is a subset of $E(G)$ such that no vertex of G occurs more than once as an endpoint of some edge in M. This definition implies that loops are not allowed to participate in M. The endpoints of the edges contained in M are said to be covered by M.

- **Perfect and Perfect internal matching**

A matching is called ***perfect*** if it covers all vertices of G. A ***perfect internal matching*** is one that covers all of $Int(G)$. Clearly, the notions *perfect matching* and *perfect internal matching* coincide for closed graphs.

- **Subgraph**

A ***subgraph*** G' of G is a collection of vertices and edges of G. However, in our treatment of open graphs we do not want to allow that new external vertices (i.e., ones that are not present in G) emerge in G'. Therefore, when vertex $v \in Int(G)$ becomes external in G', we will augment G' with a loop edge around v. This augmentation will be understood automatically in all subgraphs of G. The subgraph of G induced by a set of vertices $X \subseteq V(G)$ will be denoted by $G[X]$, or just by $[X]$ if G is understood. By the standard definition, the edges of $G[X]$ are those of G having both endpoints in X.

- **Bipartite graph**

A graph is called ***bipartite*** if its set of vertices $V(G)$ can be partitioned into two sets A and B such that every edge in $E(G)$ has one endpoint in A and the other in B. Often the sets A and B are called ***color classes*** of G and (A,B) a ***bipartition*** of G.

- **Allowed, Forbidden, Mandatory and Constant edges**

An edge $e \in E(G)$ is called ***allowed*** if e is part of some perfect internal matching of G, and e is ***forbidden*** if this is not the case. Edge e is ***mandatory*** if it is present in all perfect internal matchings of G, and e is ***constant*** if it is either forbidden or mandatory.

- **Elementary Graph**

Graph G is ***elementary*** if its allowed edges form a connected subgraph covering all of the external vertices, and G is ***1-extendable*** if all of its edges, except the loops if any, are allowed.

- **Canonical Partition**

The canonical partition of an elementary graph G is determined by the following equivalence relation $\sim$ on $V(G)$. For any two internal vertices $u,v \in V(G)$, $u \sim v$ if the edge $e=(u,v)$ becomes forbidden in $G+e$. (The graph $G+e$ is obtained from G by adding the edge e).

• Nice and G-Permissible Graph

Let graph G have a perfect internal matching. A subgraph G' of G is ***nice*** if it has a perfect internal matching, and every perfect internal matching of G' can be extended to a perfect internal matching of G. A perfect internal matching of G is ***G'-permissible*** if it is the extension of an appropriate perfect internal matching of G'. Not all perfect internal matchings of G are necessarily G'-permissible.

• Elementary Components of Graph

The subgraph of G determined by its allowed edges usually has several connected components, which are known as the ***elementary components*** of G. An elementary component C is ***external*** if it contains external vertices of G, otherwise C is ***internal***. An elementary component can be as small as a single external vertex of G. Such a component is the only exception from the general rule that each elementary component is an elementary graph.

A ***mandatory elementary component*** is a single mandatory edge $e \in E(G)$ with a loop around one or both of its endpoints, depending on whether e is external or internal. An edge connecting two external vertices is not mandatory in G, therefore it is not a mandatory elementary component either.

• Walk, Trail and Cycle

A ***walk*** in graph G is an alternating sequence of vertices and edges, starting and ending with a vertex, such that each edge in the sequence is incident with the vertex immediately preceding and following it. A ***trail*** is a walk in which no edge occurs more than once, and a path is a trail with no repetition in the sequence of vertices. A ***cycle*** is a trail that returns to its starting point after covering a path, and then stops. A trail is called external if one of its endpoints is such, otherwise the trail is internal.

• M-Alternating Trail, Walk and Fork

Let M be a perfect internal matching of G. A trail $\alpha = v_0, e_1 \ldots e_n, v_n$ is alternating with respect to M (or M-alternating, for short) if for every $1 \leq i \leq n-1$, $e_i \in M$ if and only if $e_{i+1} \notin M$. An alternating trail can return to itself only at its endpoints. Therefore we shall specify alternating trails just by giving the set of their edges, indicating the starting point and other particulars of the trails only in words when necessary.

If $\alpha = v_0, e_1 \ldots e_n, v_n$ is an ***M-alternating path*** and $e_1 \in M$ ($e_1 \notin M$), then we can say that α is ***positive*** (respectively, ***negative***) at its v_0 endpoint. An external alternating path leading to an internal vertex is positive (negative) if it is such at its internal endpoint. An internal alternating path is positive (negative) if it is such at both ends. A positive ***M-alternating fork*** is a pair of disjoint positive external M-alternating paths leading to two different

internal vertices. Even if this sounds odd, a positive alternating fork is said to connect its two internal endpoints.

• Crossing, M-alternating Loop, Cycle and Network

A perfect internal matching of G is often called a ***state***. For any state M, an M-alternating path connecting two external vertices of G is called a ***crossing***. An ***M-alternating loop*** around vertex v is an odd *M-alternating cycle* starting from v. Clearly, the first and the last edge of any *M-alternating* loop must not be in M. Since we now have a distinct name for odd alternating cycles, we shall reserve the term "alternating cycle" for even length ones. An M-alternating unit is either a crossing or an (even length) alternating cycle with respect to M. An ***external alternating path*** is one that has an external endpoint. Making an *M-alternating* unit α (or switching on α) means changing the status of each edge appearing in α regarding its being present or not present in M.

An M-alternating network Γ is a set of pairwise disjoint M-alternating units. Again, by making Γ in state M we mean creating a new state $S(M, \Gamma)$ by making the units in Γ one by one in an arbitrary order. It was proved in [3] that for every two states M and M' there exists an M-alternating network Γ such that $M' = S(M, \Gamma)$ and $M = S(M', \Gamma)$. This network Γ is called the mediator-alternating network between states M and M'.

- **Accessible Vertex, Impervious and Viable Edges**

An internal vertex v of G is called ***accessible*** in state M if there exists a positive external *M-alternating* path leading to v. An edge e is ***impervious*** in state M if neither of its endpoints are accessible in M. Edge e is ***viable*** if it is not impervious. Fig. 3.1 is showing a graph containing an impervious edge e. In this figure, double lines connecting two vertices indicate edges in the given matching M.

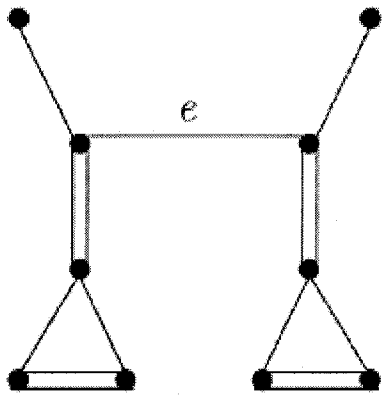

Figure. 3.1: An impervious edge e

- **Some important claims and Corollaries**

Some of the important claims and corollaries regarding the above definitions are listed below:

Claim 3.1. *[10] An internal vertex v is accessible in state M if and only if v is accessible in all states of G.*

Proof: Let us augment G by a new external edge at v, that is, by an edge $e = (v, v')$, where $v' \notin V(G)$. If $G + e$ denotes the augmented graph, then $G + e$ still has a perfect internal matching, moreover, G is a nice subgraph of $G + e$. Obviously, there is only one way to extend any perfect internal matching of G to $G + e$, i.e., by not including the edge e in that matching. We shall therefore identify each state of G by its unique extension to $G + e$. By assumption, there exists an M-alternating crossing α in $G + e$ passing through the edge e. Consider the state $S(M,\alpha)$, and switch to any G-permissible state M' of $G + e$ by making the mediator alternating network Γ between $S(M, \alpha)$ and M'. It is clear that Γ contains a unique crossing β going through e. Stripping β from the edge e results in the desired positive external M'-alternating path in G leading to vertex v.

By virtue of *Claim 3.1* we can say that an internal vertex v is accessible in G without specifying the state M relative to which this concept was originally defined.

Corollary 3.2. *[10] An edge e is impervious in some state of G if and only if e is impervious in all states of G.*

Claim 3.3. *Every internal vertex of an open elementary graph G is accessible.*

Proof: It was proved in [3] that, for every two allowed edges e_1, e_2 of an elementary graph, there exists a state M such that both e_1 and e_2 are contained in an appropriate M-alternating unit. Let v be an arbitrary internal vertex of G. Clearly, there exists an edge $e \in M$ incident

with v. If e is external, then we are done. Otherwise, since e is allowed, for any external edge e' of G there exists a state M' and a crossing with respect to M' such that goes through e and e'. Thus, v is indeed accessible (e.g. in state M').

Claim 3.4. *Let C_1 and C_2 be two different external elementary components of G. There exists no alternating path β with respect to any state M connecting C_1 and C_2 in such a way that the two endpoints of β, but no other vertices, lie in C_1 and C_2.*

Proof: Assume, by contradiction, that there exists an M-alternating path β connecting vertex v_1 in C_1 with vertex v_2 in C_2 as described in the claim. Clearly, β must be negative at both ends. Moreover, v_i $(i = 1,2)$ can be external only if $C_i = \{ v_i \}$. Take a positive external M-alternating path α_i leading to v_i inside C_i if v_i is internal, otherwise let α_i be the empty path. The path α_i exists by Claim 3.3 above. Combining α_1, β, and α_2 then results in a crossing through both components C_1 and C_2, which contradicts that $C_1 \neq C_2$.

Claim 3.5. *If v_1 and v_2 are two internal vertices of an elementary graph G, then $v_1 \sim v_2$ if and only if one of the following conditions are met in any state M of G:*

(a) there exists a positive M-alternating path connecting v_1 and v_2,

(b) there exists a positive M-alternating fork connecting v_1 and v_2.

Proof. Consider the extra edge $e = (v_1, v_2)$ in the graph $G + e$. Since G is a nice subgraph of $G + e$, the edge e cannot be mandatory. Therefore e is not forbidden if and only if there exists an M_e-alternating unit passing through e in any state M_e of $G + e$.

Identifying the G-permissible states of $G + e$ with those of G, this is equivalent to saying that e is not forbidden in $G + e$ if and only if there exists an M-alternating unit passing through e in any state M of G. This unit opens up to either a positive M-alternating path or a positive M-alternating fork connecting v_1 and v_2 when the edge e is deleted from $G+e$.

Chapter 4

Soliton Automata and Their Structural Decomposition

Soliton automata are the main focus of this research. The algorithms that we have worked out in this thesis are related to soliton automata (actually soliton graphs which are the underlying objects of soliton automata). Different steps of these algorithms can determine many important properties of soliton automata. This chapter will provide a brief overview of soliton automata. There are many complex results on soliton automata, which are very hard to cover in one chapter. Therefore, in this chapter we will mainly concentrate on those results that will help us to elaborate the algorithms of this thesis. Most of the results listed in this chapter originate from papers [2][4][10][11].

4.1 Finite State Automata

An ***alphabet*** is a finite, nonempty set of symbols. A ***non-deterministic finite state automaton*** is a triple $A=(S, X, \delta)$, where S is a non-empty finite set, the *set of states*, X is an alphabet, the *input alphabet*, and $\delta : S \times X \rightarrow 2^S$ is the *transition function*. Generally we use the term "automaton" to mean "non-deterministic finite automaton".

An automaton $A=(S, X, \delta)$ is ***deterministic*** if for each $s \in S$ and $x \in X$, $|\delta(s,x)| \leq 1$.

4.2 Soliton Graphs

The underlying object of a soliton automaton is a so called soliton graph. Such a graph is the topological model of a hydrocarbon molecule (or chain of molecules) along which soliton waves travel. In this model, soliton graphs come with a perfect internal matching, i.e., a matching that covers all the vertices with degree at least two. These vertices, called internal, model carbon atoms, whereas vertices with degree one, called external, represent a suitable chemical interface with the outside world. The states of the corresponding automaton are perfect internal matchings of the underlying graph, and transitions are carried out by switching on alternating walks.

A ***soliton graph*** is an open graph G having a perfect internal matching. External vertices in G provide an interface by which the corresponding soliton automaton $A(G)$ can be controlled from the outside world. The states of $A(G)$ are the perfect internal matchings of G, and the inputs are pairs of external vertices in G. A state change of $A(G)$ in state (perfect internal matching) M on input (v_1, v_2) is carried out by selecting an alternating walk α

connecting v_1 to v_2 with respect to the current state M, and exchanging the status of each edge along α regarding its being present or not present in M. The resulting new state will be denoted by $S(M,\alpha)$. There might be several alternating walks connecting the same pair (v_1, v_2) of external vertices. The formal definition of alternating (soliton) walks and automata is given in the next two sections.

4.3 Soliton Walks

Let M be a state of G. The set of *external alternating walks,* together with the concept of switching on such walks is defined recursively as follows –

1. The walk $\alpha = v_0 e v_1$, where $e = (v_0, v_1)$ with v_0 being external, is an external alternating walk, and the set $S(M,\alpha) \subseteq E(G)$ is defined by $S(M,\alpha) = M \oplus \{e\}$. (The operation $\oplus$ is symmetric difference of sets).
2. If $\alpha = v_0 e_1 ... e_n v_n$ is an external alternating walk ending at an internal vertex v_n, and $e_{n+1} = (v_n, v_{n+1})$ is such that $e_{n+1} \in S(M,\alpha)$ if and only if $e_n \in S(M,\alpha)$, then $\alpha' = \alpha e_{n+1} v_{n+1}$ is an external alternating walk and $S(M,\alpha') = S(M,\alpha) \oplus \{e_{n+1}\}$. It is required, however, that $e_{n+1} \neq e_n$ unless $e_n \in S(M,\alpha)$ is a loop.

It is clear by the above definition that $S(M,\alpha)$ is a state if and only if the endpoint v_n of α is external, in which case the walk is called complete. A ***soliton walk*** is a complete external alternating walk, which therefore connects two external vertices of G. Intuitively,

when making a soliton walk α in state M, one changes the sign of every edge immediately after having traversed that edge. *No backtrack* is allowed on the edge e that was traversed last, unless this is a must, i.e., e is a loop currently being positive. To exclude the possibility of making a backtrack altogether, we assign a *direction* to the loops occurring in soliton walks (e.g. clockwise/anticlockwise). Then we insist that once a loop has been traversed in one direction, it must be traversed in the same direction immediately afterwards. In Fig. 4.1 let $M = \{e, h_1, h_2\}$. A possible soliton walk from u to v with respect to M is $\alpha = uewgz_1h_1z_2l_2z_3h_2z_4l_1z_1gwfv$. Switching on α then results in $S(M, \alpha) = \{f, l_1, l_2\}$.

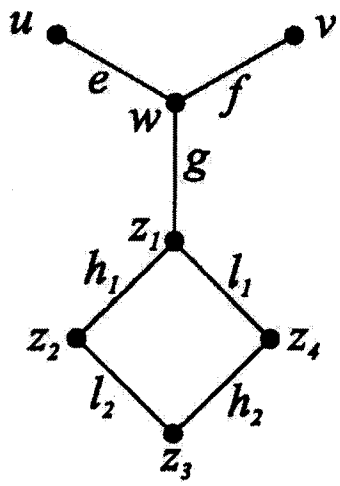

Figure. 4.1: An example soliton walk

4.4 Soliton Automata

The underlying object of a ***soliton automaton*** is a graph G having a perfect internal matching. Graph G gives rise to an automaton $A(G)$, the states of which are the perfect internal matchings of G. The input alphabet for $A(G)$ is the set of all (ordered) pairs of external vertices in G, and the state transition function δ is defined by -

$\delta(M, (v_1, v_2)) = \{S(M,\alpha) | \alpha$ is a soliton walk from v_1 to $v_2\}$

A soliton automaton $A(G)$ is *deterministic* if for every state M and input (v_1, v_2),

$|\delta(M, (v_1, v_2))| \leq 1$,

where

$\delta(M, (v_1, v_2)) = \{S(M, \alpha) | \alpha$ is a soliton walk from v_1 to $v_2\}$.

Soliton graph G is deterministic if the automaton $A(G)$ is such.

4.5 Viable Soliton Graphs

An edge $e \in E(G)$ is viable in state M if there exists an M-alternating path $e_1, \ldots, e_n$ from some external vertex of G to one of the endpoints of e such that

(i) $e \neq e_i$ for any $i \in [n]$;

(ii) e_n and e are M-alternating in the sense that $e_n \in M$ if and only if $e \notin M$.

The edge e is impervious if it is not viable (in state M).

Thus, an edge e is viable in state M if there exists an M-alternating path that starts from an external vertex, reaches one endpoint of e without passing through e itself, and it can be continued on e in an alternating fashion. It is easy to see that the present definition of viable and impervious edges is equivalent to the one given in Chapter 3.

If a graph is free form impervious allowed edges, then it is called a viable soliton graph. The concept of viable soliton graphs is important for one of our algorithms (Chapter 7). By that algorithm we can decide if a given graph G is a deterministic viable soliton graph containing an alternating cycle.

4.6 Principal Canonical Class and Principal Vertex

Elementary components are classified as external or internal, depending on whether or not they contain an external vertex. An elementary component of G is viable if it does not contain impervious allowed edges. A viable internal elementary component C is one-way if all external alternating paths (with respect to any state M) enter C in vertices belonging to the same canonical class of C. This unique canonical class is called ***principal*** and vertices belonging to this class are principal vertices. A viable elementary component is two-way if it is not one-way.

4.7 Chestnuts

A connected graph G is called a ***chestnut*** if it has a representation in the form $G = \gamma + \alpha_1 + ... + \alpha_k$ with $k \geq 1$, where β is a cycle of even length and each α_i ($i \in [k]$) is tree subject to the following conditions:

i. $V(\alpha_i) \cap V(\alpha_j) = \varnothing$ for $1 \leq i \neq j \leq k$;

ii. $V(\alpha_i) \cap V(\gamma)$ consists of a unique vertex – denoted by v_i – for each $i \in [k]$

iii. v_i and v_j are at an even distance on γ for any distinct $i, j \in [k]$

iv. Every vertex $w_i \in V(\alpha_i)$ with $d(w_i) > 2$ is at even distance from v_i in α_i for each $i \in [k]$

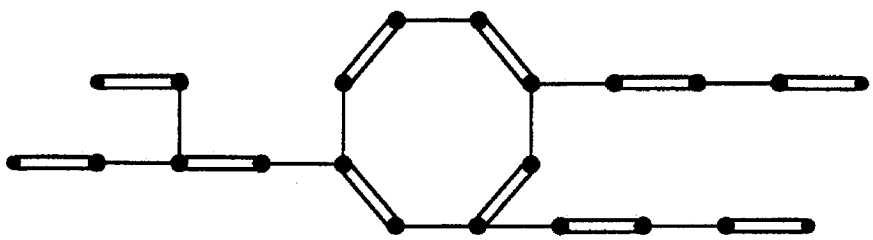

Figure. 4.2: A Chestnut

The following theorem provides a characterization of chestnuts. For a proof, the reader is referred to [7].

Theorem 4.7.1. *Let G be a connected deterministic soliton graph having no impervious edges. The graph G has a non-mandatory internal elementary component, if and only if G is a chestnut.*

Chestnuts are bipartite graphs. A vertex of a chestnut G is called outer if its distance from any of the vertices v_i appearing in ii of the definition above is even, and inner if this distance is odd. Then the inner and outer vertices indeed define a bipartition of G. Moreover, the degree of each inner vertex is at most 2. Coming up with a perfect internal matching for G is simple: just mark the cycle γ in an alternating way, and then the continuation is uniquely determined by the structure of the trees α_i. Thus, G has exactly two states. It is also easy to see that the inner internal vertices are accessible, while the outer ones are inaccessible. Thus, the cycle γ forms an internal elementary component with its outer vertices constituting the principal canonical class of this component.

In terms of families of elementary components introduces in [10], the cycle γ forms a stand-alone internal family in G. The rest of G's families are all single mandatory edges along the trees α_i, or they are degenerate ones consisting of a single inner external vertex. Their rank in the partial order $\preceq_G$ [10] is consistent with their position in the respective trees α_i, following a decreasing order from the leafs to the root. The family $\{\gamma\}$ is the minimum element of $\preceq_G$, and G has no impervious edges. See [10] for the precise definition of the precise partial order $\preceq_G$ of families of soliton graph G.

As it was proved in [22], every chestnut G is a deterministic soliton graph. Moreover, G is strongly deterministic in the sense that, for each pair (v_1,v_2) of external vertices, there exists at most one soliton walk from v_1 to v_2 in each state of G. For every connected soliton graph G having no impervious edges, but possessing a non-mandatory internal elementary component, G is deterministic if and only if G is a chestnut.

4.8 Redex and Secondary Loop

A redex r in graph G consists of two adjacent edges $e = (u, z)$ and $f = (z, v)$ such that $u \neq v$ are both internal and the degree of z is 2. The vertex z is called the centre of r, while u and v (e and f) are two focal vertices (respectively, focal edges) of r.

Let r be a redex in G. *Contracting* r in G means creating a new graph G_r from G by deleting the centre of r and merging the two focal vertices of r into one vertex s. The vertex s is called the *sink* of r in G_r.

Suppose that G is a soliton graph. For a state M of G, let M_r denote the restriction of M to edges in G_r. Clearly, M_r is a state of G_r. Sometimes, however, we shall identify M with M_r if r is understood from the context. This identification is safe, as the state M can be reconstructed from M_r in a unique way. In other words, the connection $M \rightarrow M_r$ is a one-to-one correspondence between the states of G and those of G_r. Graph G and state M will often be referred to as the *unfolding* of G_r and M_r, respectively, with respect to redex r.

For any walk α in G, let *trace(α)* denote the restriction of α to edges in G_r . Clearly, *trace(α)* is a walk in G_r. It is also easy to see that if α is a soliton walk in G with respect to *M,* then *trace(α)* is a soliton walk in G_r. Moreover, the walk α can again be uniquely recovered from its trace by unfolding. (Remember the orientation imposed on loops in soliton walks.) Consequently, the connection $\alpha \rightarrow$ *trace(α)* is also a one-to-one correspondence between soliton walks in G and soliton walks in G_r.

The following two statements have been proved in [11].

Proposition 4.8.1. *The soliton automata* A_G *and* A_{G_r} *are isomorphic.*

Proposition 4.8.2. *For any state M,* α is *an M -alternating cycle in* G *if and only if trace* (α) *is an* M_r*- alternating cycle in* G_r.

It follows from Propositions 4.8.1 and 4.8.2 that any edge *e* in G_r is allowed in G_r if and only if *e* is allowed in *G*. As to the two focal edges of *r*, they can either be allowed or not in *G*, even when G_r is elementary. This issue is addressed by Lemma 4.8.3 -

Lemma 4.8.3. *Let r be a redex in soliton graph G, and assume that* G_r *is elementary. Then G is elementary if and only if both focal edges of r are allowed in G, or, equivalently, each focal vertex of r has at least one allowed edge of* G_r *incident with it.*

Proof: It is sufficient to note that either focal edge of *r* is forbidden in *G* if and only if the other focal edge is mandatory. Moreover, an arbitrary internal edge *e* of *G* is mandatory if and only if all edges adjacent to *e* are forbidden.

Another natural simplifying operation on graphs is the removal of a loop from around a vertex v if, after the removal v still remains internal. Such loops are called ***secondary***.

Let G_v denote the graph obtained from G by removing a secondary loop e at vertex v. Clearly, if G is a soliton graph, then so is G_v, and the states of G_v are exactly the same as those of G. The automata A_G and A_{G_v}, however, need not be isomorphic. This follows from the fact that any external alternating walk reaching v on a positive edge can turn back in G after having made the loop e twice, while this may not be possible for the same walk without the presence of e. Nevertheless, it is still true that for every elementary soliton graph G, G is deterministic if and only if G_v is such.

There are loops, however, the removal of which preserves isomorphism of soliton automata. These loops are exactly the ones around the inaccessible vertices of G. Each such loop is impervious, so that its removal does not affect the automaton behavior of G.

4.9 Reduced Graphs

The results listed in the forthcoming three sections are cited from [11]. Graph G is said to be reduced if it is free from redexes and secondary loops. Every graph G can be transformed into a reduced one $r(G)$ by a suitable reduction procedure.

For an arbitrary graph G, contract all redexes and remove all secondary loops in an iterative manner to obtain a reduced graph $r(G)$. Observe that this reduction procedure has the so

called ***Church-Rosser property***, that is, if G admits two different one-step reductions to graphs G_1 and G_2, then either G_1 is isomorphic to G_2, or G_1 and G_2 can further be reduced to a common graph $G_{1,2}$ In this context, one reduction step means contracting a redex or removing a single secondary loop. As an immediate consequence of the Church-Rosser property, the graph $r(G)$ above is unique up to graph isomorphism.

In a similar fashion, the process of contracting all redexes and removing all impervious secondary loops is called ***i-reduction*** and the graph obtained from G after *i-reduction* is denoted by $r_i(G)$.

4.10 Generalized Trees

A generalized tree is a connected graph not containing even-length cycles. By this definition, if there are odd-length cycles present in a generalized tree, then those cycles must be pairwise edge-disjoint. Some important results on generalized trees are listed below:

Theorem 4.10.1. *Let G be a reduced elementary soliton graph. If G contains an even-length cycle, then it also has an alternating cycle with respect to some state of G.*

Proof: For a proof, the reader is referred to [11].

Theorem 4.10.2. *For any graph* G, *if r(G) is a generalized tree, then* G *is a deterministic soliton graph. Conversely, if* G *is a deterministic elementary soliton graph, then r(G) is a generalized tree.*

Proof: See [11].

Corollary 4.10.3. *An elementary soliton graph* G *is deterministic if and only if* G *reduces to a generalized tree.*

4.11 Baby Chestnuts

A baby chestnut is a chestnut $\gamma + \alpha_1$ such that γ is a pair of parallel edges and each branch of α_1 consists of a single edge or two adjacent edges. Fig. 4.3 shows a typical baby chestnut-

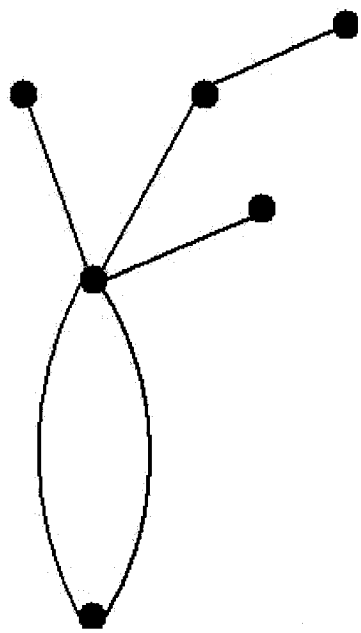

Figure. 4.3: Baby chestnut

Some important results on deterministic soliton graphs and chestnuts are listed below –

Theorem 4.11.1. *Let G be a viable connected soliton graph possessing a non-mandatory internal elementary component. Then G is deterministic if and only if* $r_i(G)$ *is a baby chestnut.*

Proof *(Only if:)* By Theorem 4.7.1, G is a chestnut augmented by some impervious edges connecting the outer internal vertices with each other. Since each internal inner vertex, different from the base ones, is the center of a redex, we can eliminate all of these vertices, except of course the last inner vertex on γ, which will no longer identify a redex. After removing the secondary impervious loops generated during redex elimination, $r_i(G)$ becomes a baby chestnut.

(If:) Blowing up γ by inverse redex elimination, or stretching the trees α_i in this manner preserves the property of being a chestnut, and any impervious loops added during this procedure may only stretch into impervious edges. Thus, the graph resulting from a baby chestnut after any number of blow-ups and stretches is still a chestnut with some additional impervious edges, provided that impervious mandatory edges have not been introduced during the unfolding procedure.

Theorem 4.11.2. *Let G be a connected viable soliton graph. Then G is deterministic if and only if it satisfies one of the following two conditions.*

1. *G i-reduces to a baby chestnut.*
2. *Each external elementary component of G reduces to a generalized tree, and the subgraph of G determined by its internal elementary components has a unique perfect matching.*

Proof: Immediate by Theorems 4.7.1, 4.11.1 and Corollary 4.10.3.

Chapter 5

An Algorithm for Graph Reduction by Using the Incidence Matrix of Graphs

This, and the next two chapters describe the algorithms that we have worked out in this thesis. In the present chapter we discuss the first algorithm, which aims at graph reduction. This algorithm is very important because the output of this algorithm is used as an input to the other two algorithms of this thesis. Graph reduction was introduced in Chapter 4, so the reader is referred back to that chapter for the terms used in graph reduction.

5.1 Representing a graph by its Adjacency Matrix

There are two common ways to represents graphs in computers – adjacency list and adjacency matrix. In our algorithm we have used the adjacency matrix approach. In this section we will briefly introduce the mechanism of graph representation using its adjacency matrix.

To construct the adjacency matrix of a graph, first we number the vertices of the graph in an arbitrary manner such as 1, 2, 3, … , |V|. For a graph *G=(V,E),* the adjacency matrix will be a $|V| \times |V|$ sized integer matrix. For the adjacency matrix $A=(a_{ij})$,

$a_{ij} = n$ *if there are n parallel edges connecting vertices i and j.*

We can use the adjacency matrix representation for both directed and undirected graphs. The adjacency matrix will take $\Theta(V^2)$ memory space and is independent of the number of edges in the graph. Adjacency matrices can also be used for representing weighted graphs. The adjacency matrix of an example graph is given below:

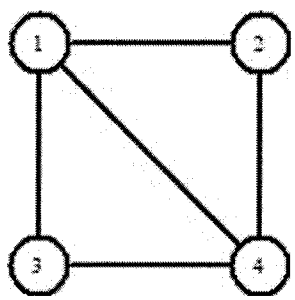

Figure. 5.1: A Graph

The adjacency matrix representation of the graph in Fig. 5.1 is:

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

5.2 Steps of the Algorithm

The actual reduction algorithm is discussed in this section. We know that a graph is reduced if it is free from redexes and secondary loops, so the main goal of this algorithm is to remove all redexes and secondary loops. The process is divided into two steps given below:

Step 1:

Consider the adjacency matrix A of G, and scan A in order to eliminate all secondary loops, and to construct the list R of all redexes in the simplified graph. Let each redex be represented in R by its center vertex.

Step 2:

Do while R is not empty: take the first redex k from R, and eliminate it by updating A in such a way that row/column j is added to row/column i, where i and j are the focal vertices

of the redex k. Abandon rows/columns k and j in A, and delete the vertices k, i, j (if present) from R. Reset $A(i, i)$ to 0 or 1 by removing all secondary loops around i, and add i to R if the updated matrix A indicates that i has become (the center of) a redex.

For a graph G with n vertices, the algorithm above constructs $r(G)$ in $O(n^2)$ time. Indeed, the elimination of one redex, together with the deletion of the newly introduced secondary loops, takes $O(n)$ time, and the number of redexes is smaller than n.

5.3 Example of the Reduction Algorithm

In this section we provide a complete example of the graph reduction process. Fig. 5.2 shows the graph to be reduced. There are eight vertices in this graph and many redexes.

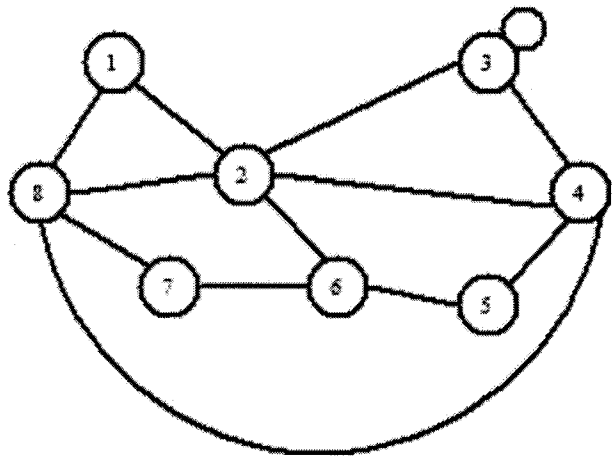

Figure. 5.2: Graph to be reduced

Fig. 5.3 shows the graph after the first iteration step of the algorithm. In this step, the secondary loop from vertex 3 is removed.

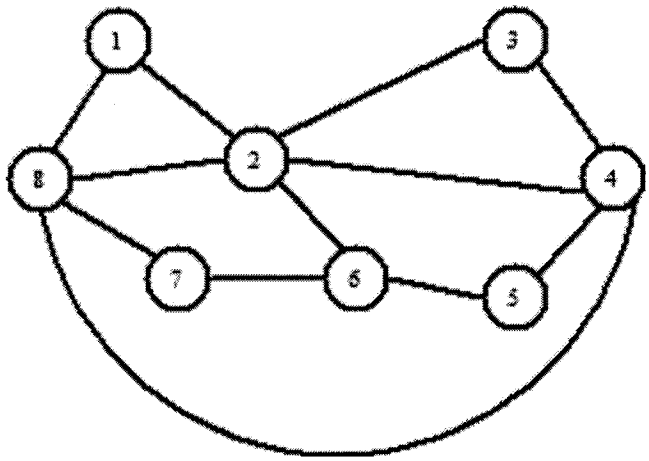

Figure. 5.3: Graph after the first iteration

Fig. 5.4 shows the graph after the second iteration step. Here we removed vertex 3 which was the centre of a redex. After removing vertex 3, the focal vertices (vertex 2 and 4) are merged into vertex 2. So here vertex 2 is the sink vertex of this reduction. There is a new secondary loop, emerging at vertex 2.

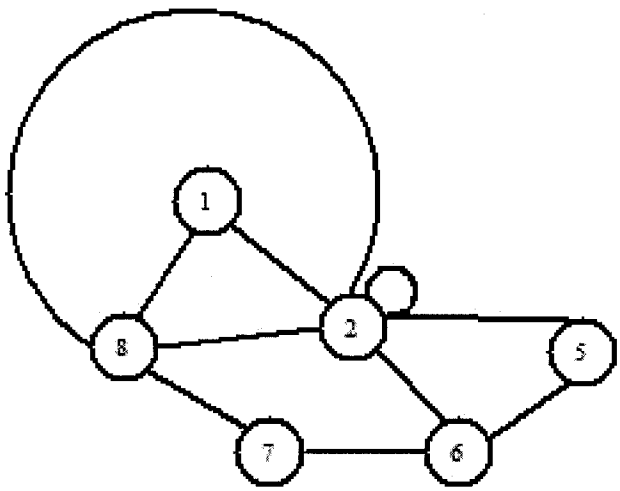

Figure. 5.4: Graph after the second iteration

Fig. 5.5 shows the resulting graph after the third iteration step of the algorithm. In this iteration step, the secondary loop at vertex 2 is removed.

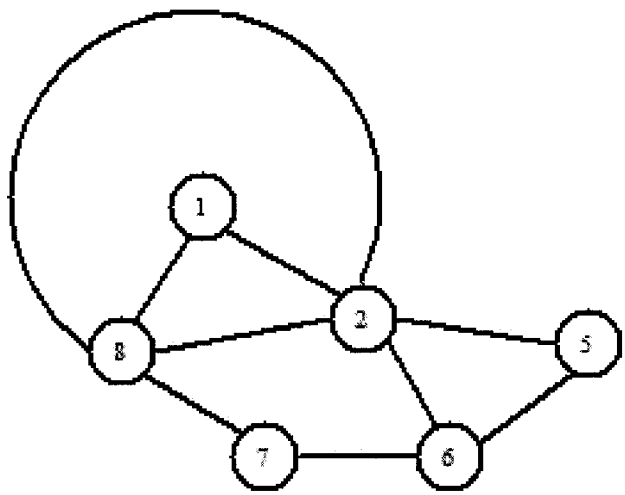

Figure. 5.5: Graph after the third iteration

Fig. 5.6 shows the graph after the fourth iteration. Here we removed vertex five which was the center of a redex. After removing vertex 5, the focal vertices (vertex 2 and 6) are merged into vertex 2. So here vertex 2 is the sink vertex of this reduction. Again, after this reduction there is a new secondary loop emerging at vertex 2.

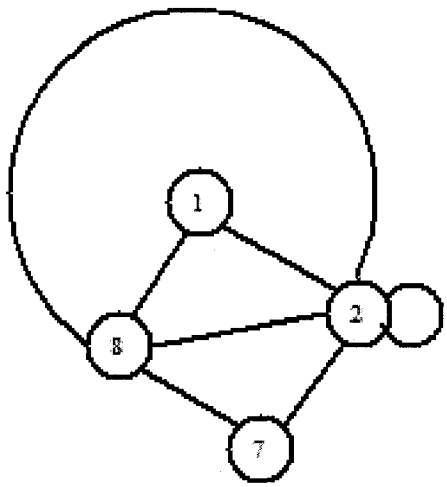

Figure. 5.6: Graph after the fourth iteration

Fig. 5.7 shows the graph after the fifth iteration step of the algorithm. In this iteration step, the secondary loop at vertex 2 is removed.

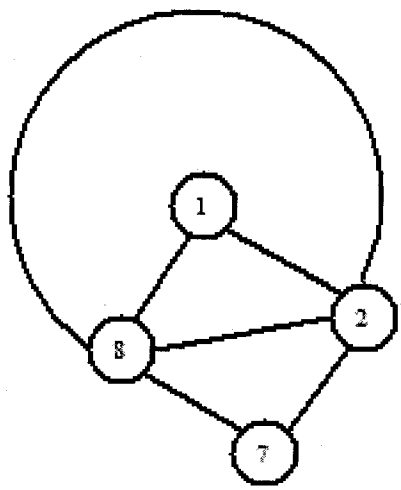

Figure. 5.7: Graph after the fifth iteration

Fig. 5.8 shows the graph after the sixth iteration step. Here we removed vertex 7, which was the center of a redex. After removing vertex 7, the focal vertices (vertex 2 and 8) are merged into vertex 2. Vertex 2 is the sink vertex of this reduction process. After this reduction step, there are two new secondary loops, emerging at vertex 2.

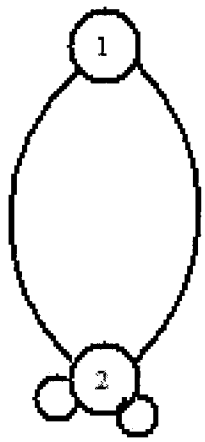

Figure. 5.8: Graph after the sixth iteration

Fig. 5.9 shows the resulting graph after the seventh iteration step of the algorithm. In this iteration step, one of the secondary loops at vertex 2 is removed.

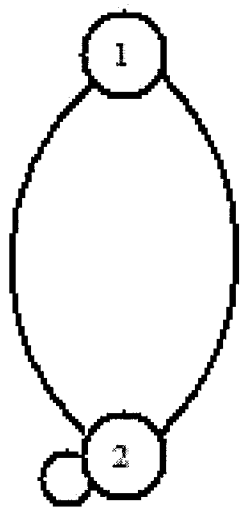

Figure. 5.9: Graph after the seventh iteration

Fig. 5.10 shows the resulting graph after the eighth iteration step. In this iteration step, another secondary loop at vertex 2 is removed. The resulting graph is our final result.

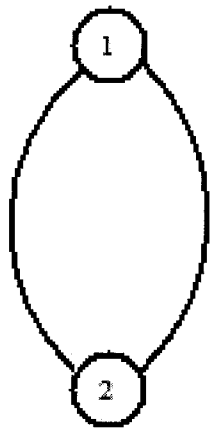

Figure. 5.10: Graph after the eight iteration

5.4 Discussion of Implementation

In this section we will provide a brief explanation of how the first algorithm is implemented. The implementation uses the Java programming language. There are two major classes – *GraphReduction* and *ReductionOperation*. Most of the operations are done inside the *GraphReduction* class, while *ReductionOperation* class is mainly responsible for reconstructing matrices after each reduction step (abandon rows/columns of the given matrix and generate a new matrix for the next iteration).

Class *GraphReduction* accepts inputs from a file and also prints different steps of the reduction process into the file. For maintaining a redex list I have used an array R. There is a function called *findFocal* for finding out the focal vertices of a redex. The function *removeLoop* is responsible for removing loops from the matrices at different stages of the reduction process. Updating the redex list is an important task for this algorithm so there are two functions for performing this task. Function *buildLisrR* is responsible for reconstructing a new redex list. On the other hand, function *checklist* checks for the necessity to reconstruct the redex list.

Function *ClassReduction* also contains *printFile* and *printR* function for printing results in the output file. Function *printFile* is responsible for printing the matrices of the different stages of the reduction process into a file where function *printR* prints the updated redex lists of different stages into the output file.

Class *ReductionOperation* has only one function that is *columnOperation*. Matrix, redex and its focal vertices are passed into *columnOperation* as parameters. This function adds rows and columns, abandons rows and columns k and j. We have used many graphs as input for testing the accuracy of the implementation and the program successfully produced accurate results for all of them. Some of the graphs that have been used for testing the program are given below:

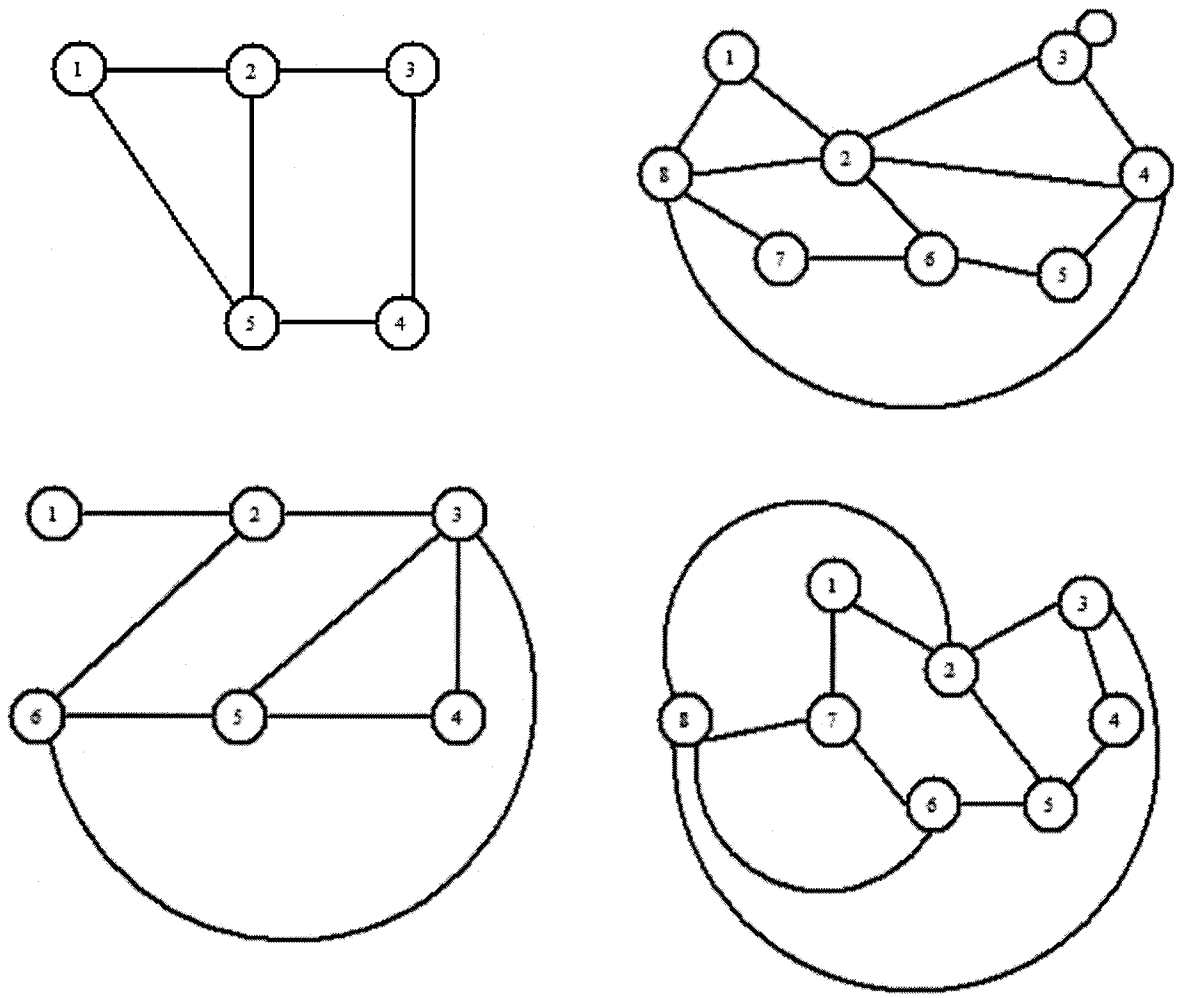

Figure. 5.11: Graphs used for testing the first algorithm

Chapter 6

An Algorithm to decide if an Arbitrary Graph is an Elementary Deterministic Soliton Graph

In this chapter we describe the second algorithm of this thesis. By the help of this algorithm we can decide if a given graph is an elementary deterministic soliton graph or not. For this algorithm we also have to use the reduction algorithm as a preamble. Different steps of this algorithm can determine some important properties of the given graphs. For example the second part of the algorithm can decide if the reduced graph is a generalized tree or not and third part of the algorithm can decide if the original graph is an elementary graph, based on the knowledge that the reduced graph is a generalized tree.

Depth-first search is another important part of our algorithm. In the second step of this algorithm we have to use depth-first search for detecting cycles in the reduced graph.

Therefore in this chapter we devote a section to presenting the depth-first search algorithm and its modified version that we have used for detecting cycles in graphs.

6.1 Steps of the algorithm

In this section we discuss the three steps of this algorithms. A brief explanation of each step is given below:

Step 1:

In this step we use the reduction algorithm (first algorithm) to reduce the given graph *G*. The reduced graph *r(G)* is used as input for the next step.

Step 2:

In this step we check if the reduced graph is a generalized tree or not. This entails the following:

- By the help of the depth-first search algorithm, see if the reduced graph *r(G)* contains an even-length cycle. As part of this algorithm, each odd-length cycle of r*(G)* is marked.

- When an even-length cycle is found, the algorithm is terminated because *r(G)* is not a generalized tree, therefore *G* is not a deterministic soliton graph. [Theorem 4.10.2]

- If *r(G)* is a generalized tree then we move to the next step.

Step 3:

In this step we check if the original graph G is an elementary graph or not. To this end, we have to reverse the reduction procedure. The details are given below –

- Start unfolding $r(G)$ back into G by reversing the steps of the reduction algorithm.

- In the process of reversing the reduction, when we add a loop to the graph, that loop is added as a forbidden edge. We use *Lemma 4.8.3* to decide if the unfolding of a redex keeps the graph elementary or not.

- Graph G is elementary (also deterministic) if and only if a positive answer is obtained every time a redex is unfolded.

The time complexity of the depth-first search algorithm is known to be $O(n^2)$. In fact, a smarter implementation of this algorithm, using the adjacency list to represent a graph, has a liner time complexity in terms of the number of edges. Thus, Step 1 and 2 take $O(n^2)$ time to execute. As to the complexity of Step 3, reconstructing graph G by inverse reduction takes as much time as its demolition did, that is $O(n^2)$. Consequently, the overall time complexity of algorithm 2 is $O(n^2)$.

6.2 The Depth-first Search Algorithm and Its Modification

We can find a cycle in a graph by using the depth-first search algorithm (DFS). This algorithm therefore becomes an integral part of our decision algorithm. The reader is referred to [27,29,30] for a detailed analysis of the DFS algorithm. The algorithm that we are presenting here is also known as colored DFS. For detecting a cycle we use a modified version of colored DFS.

DFS explores new edges from a recently discovered vertex v, which still has some unexplored edges. It explores all the edges incident with v, then it backtracks to explore edges leaving the vertex from which v was discovered. DFS continues this backtracking process until it discovers all the vertices that are reachable form the original source vertex. If there are any undiscovered vertices remaining, then it selects one such vertex as a new source vertex and repeats the search starting form that vertex. The process lasts until all vertices have been discovered.

The above paragraph explains the general working process of the depth-first search algorithm. The pseudo code of the colored DFS is given below –

```
DFS(G)
1 for each vertex u ∈ V[G]
2       do color[u] ← WHITE
```

```
3       π[u] ← NILL
4       time ← 0
5       for each vertex u∈V[G]
6           do if color[u] = WHITE
7               then DFS-Visit (u)
```

```
DFS-Visit (u)
1  color[u] ← GRAY   // White vertex u has just been discovered
2  d[u] ← time ← time+1
3  for each v ∈ Adj[u]        //Explore edge (u,v)
4       do if color[v] = WHITE
5           then π[v] ← u
6               DFS-Visit (v)
7  color[u] ← BLACK        // Blacken u, it is finished
8  f[u] ← time ← time +1
```

In the above pseudo code *π[u]* holds the predecessor information of a vertex. There are also two types of timestamps – *d[u]*, which records when *u* is discovered and grayed, and *f[u]*, which records when the search finishes after examining *u*'s adjacency list and blackening that vertex. Initially all vertices are *white* then they become *grayed* between time *d[u]* and time *f[u]* and finally they become *blacken*. In the above pseudo code *time* is

a global variable, which is used for timestamping and array *color* stores the color status of all the vertices of the given graph. Array *Adj* holds the vertices adjacent to a vertex.

As mentioned before, we use a modified version of DFS algorithm to detect cycles in graphs. The key behind this modified DFS algorithm is that, if a node is seen the second time before all of its descendants have been visited then there must be a cycle. For example, if there is a cycle containing node X then node X must be reachable from one of its descendants. Therefore, when the DFS is visiting that descendant, it will see X again, before it has finished visiting all of X's descendants which implies that a cycle exists.

The modified colored DFS algorithm is as follows. As we know, all nodes are initially colored white, and when a node is encountered, it is marked grey. Finally when its descendants are completely visited then that vertex is marked black. Now, if a grey node is encountered before all of its descendants have been visited, then there is a cycle. The pseudo code of the modified colored DFS algorithm is given below:

DFS(G)

1 for each vertex u $\in$ *V[G]*

2 do color[u] $\leftarrow$ WHITE

3 π[u] $\leftarrow$ NILL

4 time $\leftarrow$ 0

5 for each vertex *u* $\in$ *V[G]*

```
6           do if color[u] = WHITE
7               then DFS-Visit (u)
```

DFS-Visit (u)

```
1   color[u] ← GRAY   // White vertex u has just been discovered
2   d[u] ← time ← time+1
3   for each v ∈ Adj[u]    //Explore edge (u,v)
4       if color[v] = GRAY and π[u] ≠ v then
5           return "Cycle exists"
6       else if color[v] = WHITE
7           then π[v] ← u
8               DFS-Visit (v)
9   color[u] ← BLACK   // Blacken u, it is finished
10  f[u] ← time ← time +1
```

6.3 Special graphs considered for generalized tree

According to the definition, a connected graph not containing any even-length cycles is a generalized tree. If there are odd-length cycles present in a generalized tree, then those cycles must be pairwise disjoint. In this section we present some examples which will clarify the properties of generalized trees.

Fig. 6.1 shows two examples of generalized trees. Both of the graphs have only odd length cycles and these cycles are pairwise disjoint.

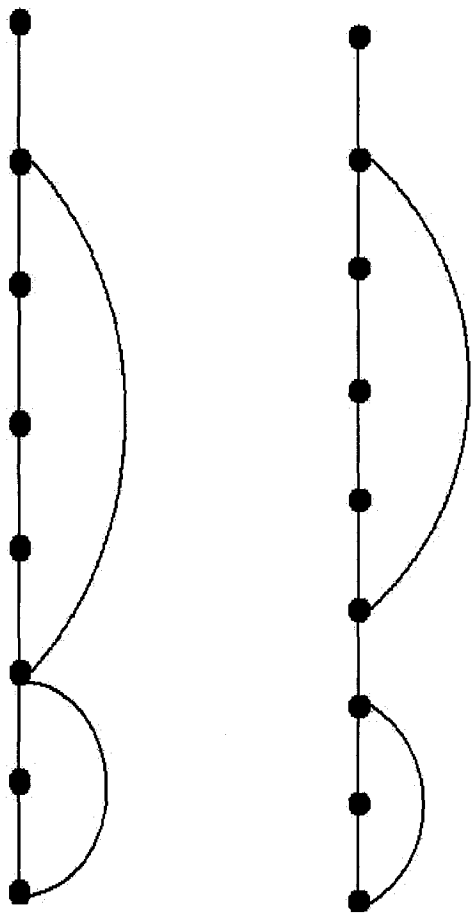

Figure. 6.1: Generalized trees

Fig. 6.2 shows two examples of graphs containing two overlapping odd length cycles. This implies that the graphs contain an even-length cycle, too, so that they cannot be generalized trees.

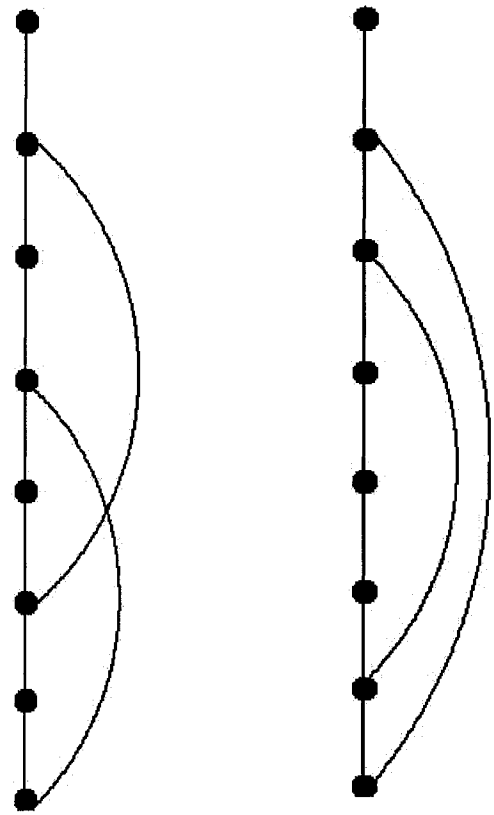

Figure. 6.2: Graphs containing overlapping odd length cycles

6.4 Discussion of Implementation

This section provides a brief discussion of the implementation of the algorithm of this chapter. I will highlight most of the classes and some of the functions that have been designed to perform important tasks.

In the second step we have to decide if the input graph is a generalized tree or not. To perform this check I have created three Java classes. The first class is *CheckGeneralizedTree*. Class *CheckGeneralizedTree* will mainly take input from the user and the input file. Class *CheckGeneralizedTree* also uses class *Vertex* for storing some important information on a vertex of a given graph.

Class *Vertex* is used to store different important information on a vertex such as its adjacency list, its parent and its color (color information will be used for the modified color DFS algorithm). The parent and color information of a vertex is very important for detecting a cycle in a graph. Class *CheckGeneralizedTree* also uses an object of class *DFSVisit* for detecting a cycle in the graph and also to figure out the length of the cycles. *CheckGeneralizedTree* passes all the information of a vertex as an object of *Vertex* class to the function *dfs_visit* of the class *DFSVisit*.

Class *DFSVisit* is the most important class for the second step, where I have implemented the modified color DFS algorithm to detect cycles in the given graph. Inside *DFSVisit, dfs_visit* is the core function which perform most the tasks. As mentioned before, function *dfs_vist* rather than taking the whole graph as an input, takes important information on a vertex in the object *Vertex*. If a cycle is detected, then this function also counts the length of that cycle. If more than one odd-length cycle is detected that it also checks if these cycles are pairwise disjoint.

To implement the last step (step 3), I have used three Java classes – *PrepInputForElementary*, *ReverseSteps* and *CheckElementary*. The exact reduction steps from the graph reduction program are stored in class *PrepInputForElementary*. This class sends this information to the class *ReverseSteps*.

Class *ReverseSteps* is mainly responsible for reversing the reduction steps and after every step checks if the graph is still elementary or not. In each reverse reduction step, *CheckElementary* is used to perform the checking. All inputs and outputs are maintained in plain text files.

Class *CheckElementary* is responsible for checking if the unfolded graph is still elementary or not. The basis of this checking is Lemma 4.8.3. This class is also responsible for producing the final result. We know that a graph is elementary (also deterministic) if and only if a positive answer is obtained every time a redex is unfolded. Class *CheckElementary* also uses the class *Vertex*. All information of a vertex is stored as an object of *Vertex* class. This class uses a function *GetDegree* to determine the degree of a vertex. For this algorithm we used many graphs to test the accuracy of the implementation. Some of these graphs are given below:

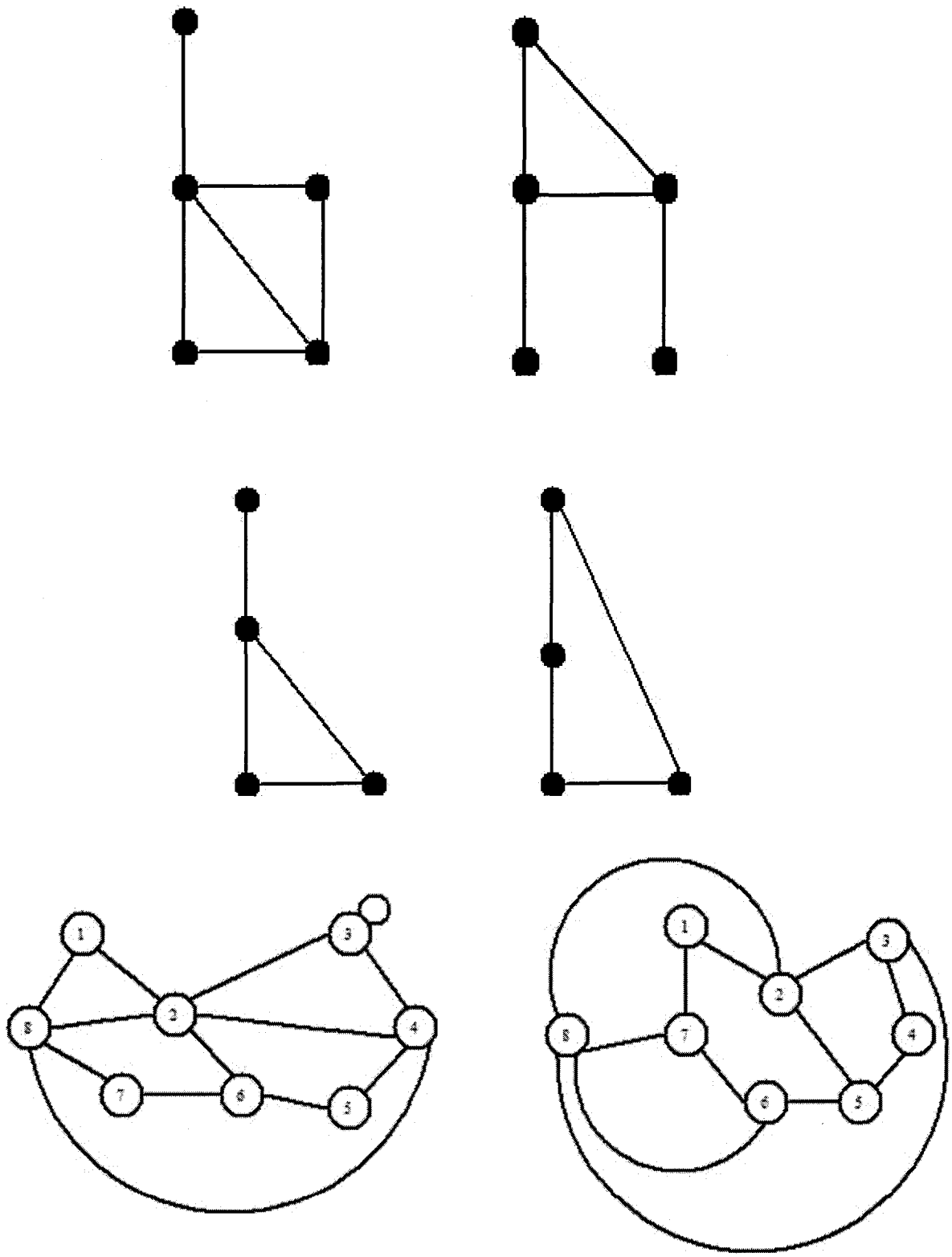

Figure. 6.3: Graphs used for testing the second algorithm

Chapter 7

An Algorithm to decide if an Arbitrary Graph is a Deterministic Viable Soliton Graph Containing an Alternating Cycle

In this chapter we present an algorithm to decide if an arbitrary graph is a viable soliton graph containing an alternating cycle. This is the third algorithm of the thesis. Baby chestnuts and impervious loops are the key concepts of this algorithm. First, the input graph is reduced by the help of the reduction algorithm. Then it is checked if the resulting graph is a baby chestnut or not. Finally, the reduction is reversed to see if all the loops eliminated during the reduction process are impervious, and impervious mandatory edges are not introduced.

7.2 Steps of the algorithm

The three steps of this algorithm are presented below:

Step 1:

The first step of the algorithm is reduction, as discussed in Chapter 5.

Step 2:

In the second step, we check if the reduced graph *r(G)* is a baby chestnut or not. If it is not, then by Theorem 4.11.2, *G* is not a deterministic soliton graph containing an alternating cycle. If the reduced graph *r(G)* is a baby chestnut then we move to the next step.

Step 3:

In the third step of the algorithm we reverse the reduction procedure to find out if all loops that have been eliminated are impervious or not. Also, in a way analogous to the second algorithm, we need to check if an inverse reduction step preserves the viable property. This amounts to the following:

- Every time a loop is added, we have to check if the loop is impervious or not. A loop around a vertex v of a viable soliton graph G is impervious if and only if v is a principal vertex in a family of elementary components in G. Technically, we must keep track of the principal vertices of the graph during the unfolding procedure to see if a particular loop vertex is principal or not.

- If we find a loop that is eliminated during reduction but is not around a principal vertex, then our answer is negative (i.e. graph G is not a deterministic viable soliton graph containing an alternating cycle). If all loops can be restored in G without finding a viable one, in other words, if all loops are added to a principal vertex, then the answer is positive.

- Whenever a new vertex is introduced, use Lemma 4.8.3 to check if the unfolded graph remains viable.

7.2 Further discussion on baby chestnuts, impervious loops, principal vertices and viable soliton graphs

Most of the concepts that we are going to deal with in this section have already been defined in Chapter 4. In order to see if a graph is a baby chestnut, the following features must be checked:

- A cycle γ consisting of a pair of parallel edges
- A tree attached to one of the two vertices of γ, the branches of which consist of at most two edges.

Impervious loops are important in this algorithm. Such loops arise during the reduction procedure from impervious edges of the original graph G. Fig. 7.1 shows a graph with an

impervious edge (connecting two principal vertices), which becomes an impervious loop during reduction. See Fig. 7.2.

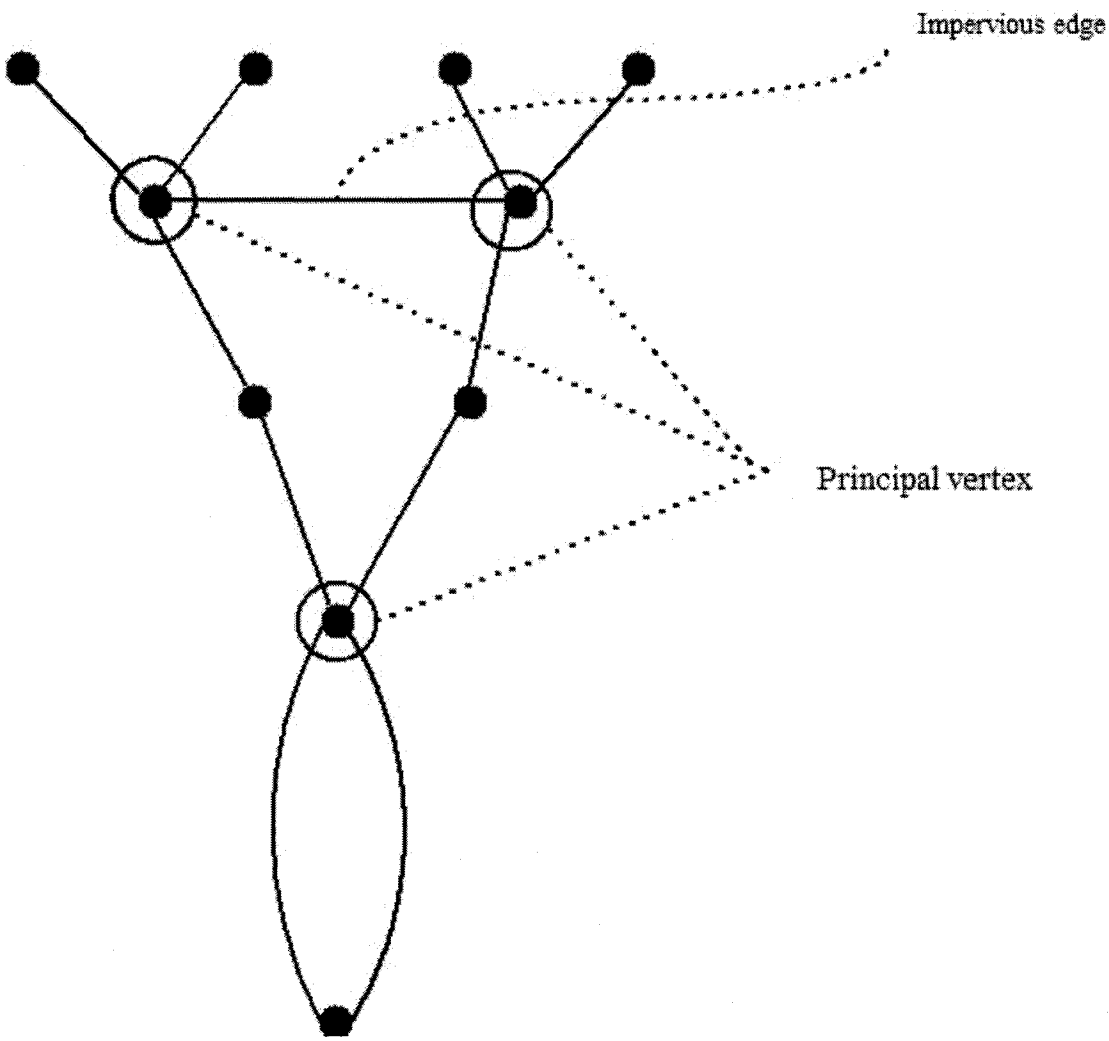

Figure. 7.1: Graph with an impervious edge

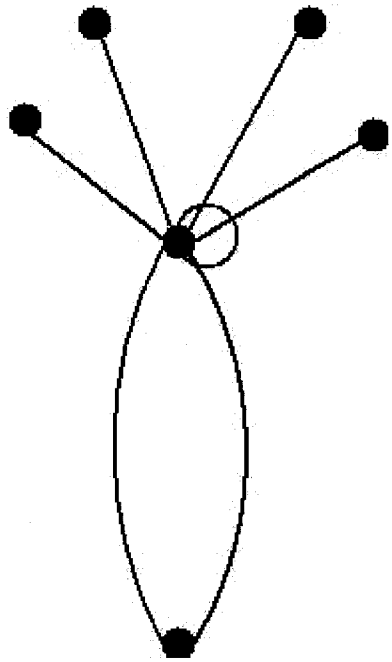

Figure. 7.2: Baby chestnut with an impervious loop

There is an easy way to locate the principal vertices in graphs arising from a baby chestnut by inverse reduction. Clearly, the only principal vertex in a baby chestnut is the root of its tree component. In the course of inverse reduction, the original baby chestnut unfolds into a chestnut graph with some extra impervious edges, which result from impervious loops being added to the graph. Nevertheless, the principal vertices of such a graph are exactly the ones of the chestnut, that is, the vertices that are at an even distance from the root of the tree they belong to. See Fig. 7.3.

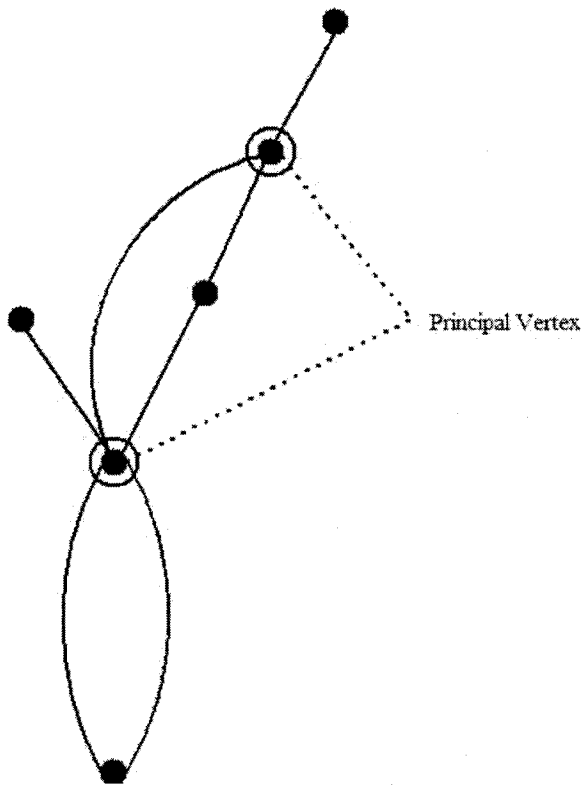

Figure. 7.3: Chestnut with principal vertices

When unfolding an impervious loop *e* during inverse reduction, the following two cases may occur:

Case 1:

The loop *e* becomes an impervious edge of the graph that is still viable. See Fig. 7.4.

Case 2:

The loop *e* becomes a forbidden edge that is adjacent to an impervious mandatory edge. See Fig. 7.5.

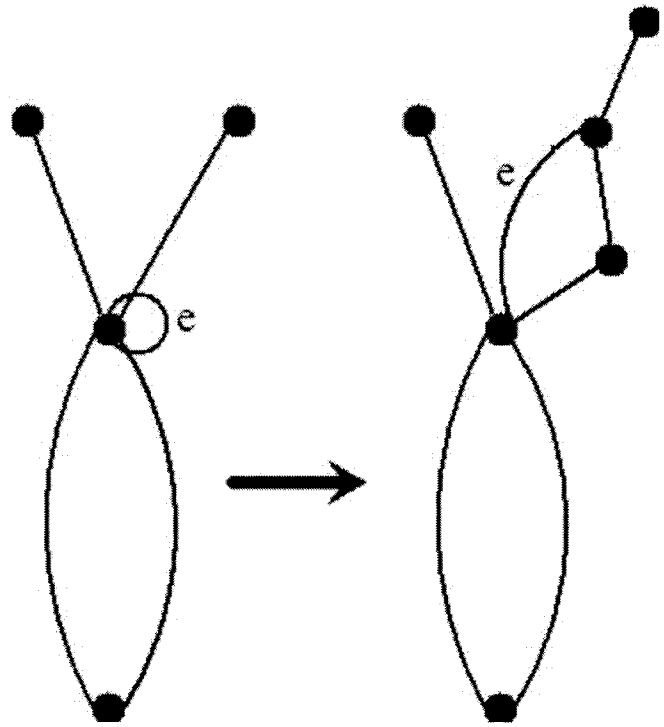

Figure. 7.4: Case 1 when unfolding an impervious loop

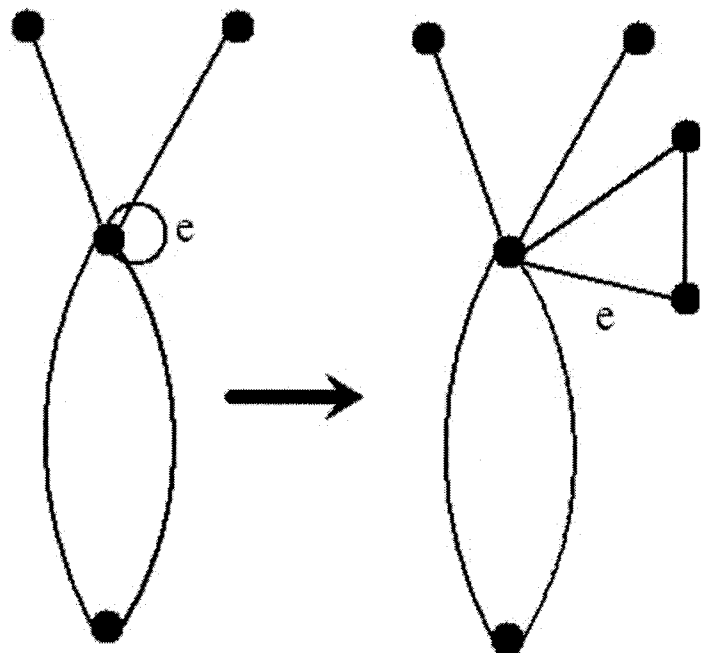

Figure. 7.5: Case 2 when unfolding an impervious loop

Obviously, our algorithm must detect the "wrong" Case 2, and if it happens, terminate immediately. In general, we terminate the algorithm whenever one of the two focal vertices of the newly introduced redex is not incident with at least one viable edge. See Lemma 4.8.3.

Regarding the time complexity of algorithm 3, we need to investigate Step 2 and 3. Clearly, Step 2 takes linear time in terms of the number of vertices. The time complexity of Step 3 is the same as that of Step 1, that is $O(n^2)$. As in the case of algorithm 2, the reason is that the reconstruction of graph G by inverse reduction takes as much time as its demolition did. Thus, the overall time complexity of algorithm 3 is $O(n^2)$.

7.3 Discussion of Implementation

As in the last two chapters, this section is intended to provide a brief discussion on the implementation of this algorithm. The algorithm has three major steps to decide if a given graph is a deterministic viable soliton graph containing an alternating soliton graph or not. The first step is again reduction. In the second step of the algorithm we have to check the reduced graph for being a baby chestnut. In the implementation I have used three Java classes. One class is named *Vertex* and other two classes are named *CheckBCNut* and *BabyChestNut*.

The role of class *Vertex* in this algorithm is analogous to its role in the second algorithm. We need to store some important information on vertices of the given reduced graph. Like before, class *Vertex* can store a vertex's adjacency list and parent information. Class *BabyCheastNut* performs the major operations of this algorithm. This class contains a function *FindBabyCheastNut* which takes a list of *Vertex* type objects and some other information as parameters and then performs necessary checking to decide if the given graph is a baby chestnut or not. Finally, it sends the result back to the class *CheckBCNut.*

Class *CheckBCNut* works like a bridge between the classes *Vertex* and *BabyChestNut* . It reads the input graph from a file, then creates a *Vertex* object for each vertex and sends them to *BabyChestNut* for checking. After receiving the requested information form BabyChestNut , it returns the final result.

In the third step we have to reverse the reduction steps and check the loops that were eliminated. When putting the eliminated loops back to the graph, we need to check if they are impervious or not. For reversing the steps of the reduction process, I used the same procedure that was introduced in the second algorithm. Class *ReverseRead* starts the program by calling the functions of class *Scanner.*

As in the second algorithm class *Scanner* is responsible for producing the reverse steps output file. Class *ChcekImperviousLoop* is responsible for performing the major operations in step three. This class will check if an eliminated loop is impervious or not. It has two

functions for performing this operation: *CheckForImperviousLoop* and *CheckSecondVertex*. *CheckForImperviousLoop* performs almost all the checking. I have used function *CheckSecondVertex* to locate the principal vertices of a chestnut.

For testing the correctness of the programs, I have used couple of test graphs. Some of these graphs are given below:

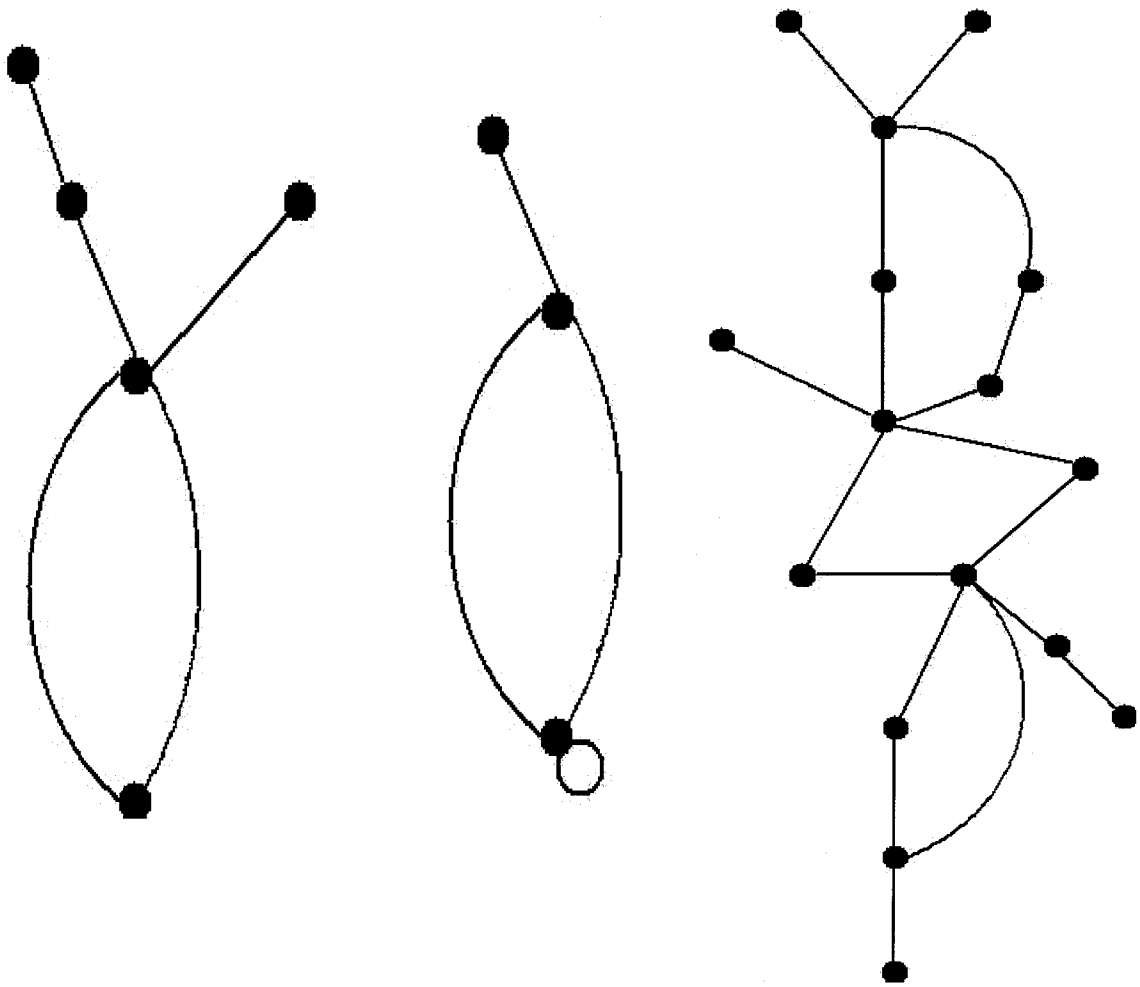

Figure. 7.6: Graphs used for testing the third algorithm

Chapter 8

Conclusion and Future Work

Molecular switching might turn out to be a significant step toward nanotechnology. We can construct unbelievably smaller devices if we are successful in designing the switches at the molecular level. In order to explore the computational power of such devices, we need to study them on abstract models. Soliton automata seem to be a perfect mathematical model for this purpose. The algorithms discussed in this thesis largely contribute to the research on soliton graphs and automata.

We have discussed three algorithms in this thesis. The first algorithm reduces an arbitrary graph, making it free from redexes and secondary loops. This algorithm plays an important role in the other two algorithms as well, because those algorithms rely on reduction as a preliminary step. The second algorithm can decide if a given graph is an elementary soliton graph. The method of this algorithm is to first see if the input graph reduces to a generalized tree, and if so, reverse the reduction to see if the elementary property is

preserved. The third algorithm can decide if a given graph is a deterministic viable soliton graph containing an alternating cycle. To this end, the algorithm we first check if the input graph reduces to a baby chestnut. Then, again, the reduction is reversed to see the viable property and the deterministic property are both preserved.

There are some issues that have not been dealt with in greater detail. We did not thoroughly analyze the time and space complexity of our algorithms. Therefore, one of the important future endeavours will be the analysis of the time and space complexity of these algorithms.

Another aspect of future work is to test the implementation with more complex and large graphs. We have tested our implementation with several standard graphs, and the programs returned correct results for all of them. Testing with more complex graphs, however, could lead to the fine-tuning of our implementation.

Reverse reduction is an important step in the last two algorithms. For implementing reverse reduction I have used a file to store information. However, there is an alternative approach (store information in the memory), which I want to explore in the future, and compare its efficiency with the file approach.

Bibliography

[1] M. Bartha, E. Gombas, A structure theorem for maximum internal matchings in graphs, *Information Processing Letters* **40** (1991), 289-294.

[2] M. Bartha, E. Gombas, On graphs with perfect internal matchings, *Acta Cybernetica* **12** (1995), 111-124.

[3] M. Bartha, H. Jürgensen, Characterizing finite undirected multigraphs as indexed algebras, Research Report # 252, Department of Computer Science, The University of Western Ontario, London, Canada, 1989.

[4] M. Bartha, M. Krész, Elementary decomposition of soliton automata, *Acta Cybernetica* **14** (2000), 631-652.

[5] M. Bartha, M. Krész, On the immediate predecessor relationship in soliton graphs, *Congressus Numerantium* **150** (2001), 15-31.

[6] M. Bartha, M. Krész, Characterizing soliton graphs, in *Abstracts of the Fourth Joint Conference on Mathematics and Computer Science* (2001).

[7] M. Bartha, M. Krész, Deterministic soliton graphs, *manuscript* (2004).

[8] M. Bartha, M. Krész, Isolating the families of soliton graphs, *Pure Mathematics and Applications* **13** (2002), 49-62.

[9] M. Bartha, M. Krész, Soliton graphs and graph-expressions, in *Abstracts of the 3rd CSCS Conference* (2002).

[10] M. Bartha, M. Krész, Structuring the elementary components of graphs having a perfect internal matching, *Theoretical Computer Science* **299** (2003), 179-210.

[11] M. Bartha, M. Krész, Deterministic soliton automata defined by elementary graphs, in *Proceedings of the Kalmar Workshop on Logic and Computer Science* (2003) pp. 69-79.

[12] M. Bartha, M. Krész, Thtte type theorems in graphs having a perfect internal matching, *Information Processing Letters* **91** (2004), 277-284.

[13] M. Bartha, M. Krész, Redex elimination in graphs, in *Abstracts of the Fifth Joint Conference on Mathematics and Computer Science* (2004).

[14] M. Bartha, M. Krész, The Berge formula, and a calculus of barriers for maximum internal matchings in graphs, in *Abstracts of the Graph Theory 2004 Conference* (2004).

[15] M. Bartha, M. Krész, Algorithms for soliton automata, *manuscript* (2004).

[16] M. Krész, Alternating cycles in soliton graphs, *WSEAS Transactions on Mathematics* 1 (2002),165-170.

[17] M. Krész, Soliton automata: a computational model on the principle of graph matchings, Ph. D Thesis, Department of Computer Science, University of Szeged, Hungary, 2004.

[18] F. L. Carter, Conformational switching at the molecular level, in *Molecular Electronic Devices* (F. L. Carter Ed.), Marcel Dekker, Inc., New York, 1982, pp. 51-72.

[19] F. L. Carter (Ed.), *Molecular Electronic Devices,* Marcel Dekker, Inc., New York, 1982.

[20] F. L. Carter, The molecular device computer: Point of departure for large scale cellular automata, *Physica D* **10** (1984), 175-194.

[21] F. L. Carter (Ed.), *Molecular Electronic Devices II,* Marcel Dekker, Inc., New York, 1987.

[22] J. Dassow, H. Jürgensen, Soliton automata, *J. Comput. System Sci.* **40** (1990), 154-181.

[23] J. Dassow, H. Jürgensen, Soliton automata with a single exterior node, *Theoretical Computer Science* **84** (1991), 281-292.

[24] J. Dassow, H. Jürgensen, Soliton automata with at most one cycle, *Journal of Computer and System Sciences* **46** (1993), 155-197.

[25] F. Gécseg, H. Jürgensen, Automata represented by products of soliton automata, *Theoretical Computer Science* **74** (1990), 163-181

[26] J. Dassow, H. Jürgensen, The transition monoids of soliton trees, In G. Păun (editor): *Mathematical Linguistics and Related Topics*. Papers in Honour of Solomon Marcus on His 70th Birthday. 76-87. Editura Academiei Române, Bucuresti, 1995.

[27] R. E. Tarjan, Depth-first search and linear graph algorithms, *SIAM J. Comput.* **1** (1972), 146-160.

[28] L. Lovász, M. D. Plummer, *Matching Theory,* North Holland, Amsterdam, 1986.

[29] T. H. Coremen, C. E. Leiserson, R. L. Rivest, *Introduction To Algorithms*, MIT Press, 1999.

[30] E. G. Goodaire, M. M. Parmenter, *Discrete Mathematics with Graph Theory (Second Edition)*, Prentice Hall, New Jersey, 2000

[31] R. P. Feynman, There's plenty of room at the bottom, in *Miniaturization* (H. D. Gilbert Ed.), Reinhold, New York, 1961, pp. 282.

[32] A. Aviram, M. A. Ratner, Molecular rectifiers, *Chem. Phys. Letters* 29 (1974), 277-283.

[33] L. M. Adleman, Molecular computation of solutions to combinatorial problems, *Science* 222 (1994),1021-1024.

[34] M. Conrad, Molecular computing: The lock and key pardigm, *IEEE Computer* **25** (1994), 11-20.

[35] A. Adamatzky (Ed.), *Collision-Based Computing,* Springer-Verlag, 2002.

[36] M. P. Groves, C. F. Carvalho, C. D. Marlin, and R. H. Prager, Using Soliton Circuits to Build Molecular Memories, *Australian Computer Science Communications* **15** (1993) pp. 37-45.

[37] M. P. Groves, C. D. Marlin, Using Soliton Circuits to Build Molecular Computers, *Australian Computer Science Communications* **17** (1995) pp. 188-193.

[38] M. P. Groves, C. F. Carvalho, and R. H. Prager, Switching the polyacetylene soliton, *Materials Science and Engineering* **C3** (1995) pp. 181-185.

[39] M. P. Groves, Soliton Circuit Design Using Molecular Gate Arrays, in *Proceedings of the 20th Australasian Computer Science Conference* (1997) pp. 245-252.

[40] M. P. Groves, Towards verification of soliton circuits, in *Molecular Electronic Devices* (F. L. Carter, R. E. Siatkowski, H. Wohltjen Ed.), North-Holland, Amsterdam, 1988, pp. 287-302.

[41] S. Abramsky, A Structural Approach to Reversible Computation, in LCCS 2001: Proceedings of the International Workshop on Logic and Complexity in Computer Science, edited by D. Beauquier and Y. Matiyasevich, LACL 2001, 1-16

Appendix A

Code Examples of Algorithm 1

```
import java.io.*;
import java.util.StringTokenizer;


// Class GraphReduction accepts inputs from a file and also prints different steps of the
reduction process into the file

class GraphReduction
{
        public static void main(String args[]) throws IOException
        {

                ReductionOperation ro = new ReductionOperation();

                GraphReduction gR = new GraphReduction();

                int [][] matrix = new int[matrixSize][matrixSize];

                int [][] loopFreeMatrix = new int [matrixSize][];

        loopFreeMatrix = gR.removeLoop(matrix); // removing loops for first time

                // code for printing the first result matrix to output file

                  println("\nRemoving Inner loops ....\n");

                for (int i = 0; i < matrixSize; i++)
                {
                        for (int j = 0; j < matrixSize; j++)
                        {
```

```
                    print(loopFreeMatrix[i][j] + "\t");

                } // end of inner for loop

            } // end of outer for loop

            // For maintaining a redex list used an array R

            int []R = new int[10];

            R = gR.buildListR(matrix);

            int redexListCount = 0;

            redexListCount = R.length;

            gR.printR(pw,R,redexListCount); // printing R into the output file

            int [][] rec = new int[matrixSize][matrixSize];

            int rK = 0, rI = 0, rJ =0;

            int [][] focalVertex = new int[1][2];

            int check = -1; //check variable will determine that if we have to to build a
new redex list or not

            int newMatrixSize = matrixSize;

            rec = loopFreeMatrix;

            int [][] old_rec = new int[matrixSize][matrixSize];

            // Other parts of the reduction operation

            for(int i = 0; i <= redexListCount;i++)
            {

                if(check == 1) // we have to build a new redex list
                {
```

```
                    i = 0;
                    R = gR.buildListR(rec);
                    if(R.length == 0) break;
                    focalVertex = gR.findFocal(rec,R[i]);
                    rK = R[i];
                    rI = focalVertex[0][0];
                    rJ = focalVertex[0][1];
                    redexListCount = R.length;
                    old_rec = rec;
rec = ro.columnOperation(rec,newMatrixSize,newMatrixSize,rK,rI,rJ);
                    newMatrixSize = rec.length;
                    println("Before removing loop...\n");
                    matrix_marker++;
                    // printing information for reverse writing
                    println("<"+matrix_marker+">");
                    gR.printFile(pw,rec);
                    gR.printFile(pw2,rec);
                    println("end_matrix");
                    if(newMatrixSize > 2)
                    {
                            rec = gR.removeLoop(rec);
                    }
                    else if(newMatrixSize <= 1) break;
                    gR.printR(pw,R,redexListCount);
```

```
                    println("Reduction continue ....\n");

                    gR.printFile(pw,rec);

                    i++;

                } // end of 'if' part

                check = gR.checkList(rec,old_rec,R); // checking the status of R

        } // end of for loop

} //end of main function

// removeLoop() function remove the loops from the given graph

public int [][] removeLoop(int matrix[][])
{
        int mSize = matrix.length;

        for (int i = 0; i < mSize; i++)
        {
                for (int j = 0; j < mSize; j++)
                {
                        if( (i == j) && (matrix [i][j] >=1 ))
                        {
                                matrix[i][j] = 0;
                        }
                } // end of inner loop
        } // end of outer for loop

        return matrix;

} // end of removeLoop function

// findFocal() function will find the focal vertices of a redex

public int[][] findFocal (int matrix[][], int j)
{
        int matrixSize = matrix.length;
        int focalCount = 0;

int [][] focal = new int[1][2]; // focal is an array with one row and two columns
```

```
        for(int i=0; i < matrixSize; i++)
        {
                if(matrix[j][i] == 1)
                {
                        focal[0][focalCount] = i;
                        focalCount++;
                }
        }
        return focal;
} // end of findFocal function

// buildList() function will build the new list R for given matrix

public int[] buildListR(int matrix[][])
{
        int [] R = new int[10];

        for(int i=0; i < 10; i++)
        {
                R[i] = -1;
        }

        int redexCount = 0;
        int redexListCount = 0;
        int nonRedexCount = 0;

        int matrixSize = matrix.length;

        for(int i = 0; i < matrixSize; i++)
        {
                for(int j = 0; j < matrixSize; j++)
                {
                        if(matrix[i][j] == 1)
                        {
                                redexCount++;
                        }
                        if(matrix[i][j] > 1)
                        {
                                nonRedexCount++;
                        }
                }

        if(redexCount == 2 && nonRedexCount == 0)
```

```
                {
                        R[redexListCount] = i;
                        redexListCount++;
                }

                redexCount = 0;

                nonRedexCount = 0;
                }

                int [] new_R = new int[redexListCount];

                for(int i=0; i < redexListCount; i++)
                {
                        new_R[i] = R[i];
                }

                return new_R;

        }// end of buildListR function

        // checkList() function check the necessity to build new list R

        public int checkList(int [][]matrix, int [][] old_matrix, int[] R)
        {
                int mLength = matrix.length;
                int mOldLength = old_matrix.length;

                int rLength = R.length;
                int check = 0;
                if (mLength < mOldLength) check = 1;
                if(rLength == 1) check = 1;
                if(check == 1) return 1;
                else return 0;

        } // end of checkList() function

} // end of graphReduction class
```

// Class *ReductionOperation* function adds rows and columns, abandons rows and columns *k* and *j* by using *columnOperation* function

```
class ReductionOperation
{

public int[][] columnOperation(int matrix[][], int mRow, int mCol, int k, int i, int j)
        {
                for(int m = 0; m < mCol; m++)
                {
                        matrix[i][m] = matrix[i][m]+matrix[j][m];
                }
                // adding columns

                for(int n = 0; n < mCol; n++)
                {
                        matrix[n][i] = matrix[n][i]+matrix[n][j];
                }

                // Abandoning rows/columns of k and j

                int [][] modMatrix_1 = new int [mRow - 2][mCol];

                int mInc = 0;

                // code for row deletion

                for (int rowCount = 0; rowCount < mRow; rowCount++)
                {
                        if ( k != rowCount && j != rowCount)
                        {
                                for (int colCount = 0; colCount < mCol; colCount++)
                                {
                        modMatrix_1[mInc][colCount] = matrix[rowCount][colCount];
                                }

                                mInc++; // increase mInc
                        } // end of if
                } // end of for loop

                // New matrix for column deletion
```

```
        int [][] modMatrix_2 = new int [mRow - 2][mCol - 2];
        int m;

        // code for delete column

        for (int rowCount = 0; rowCount < mRow - 2; rowCount++)
        {
                m = 0;
                for (int  n = 0; n < mCol; n++)
                {
                        if(k != n && j != n)
                        {
                modMatrix_2[rowCount][m] = modMatrix_1[rowCount][n];
                                m++;
                        } // end of if
                }
        } // end of for loop

        return modMatrix_2;
    }
}
```

Appendix B

Code Examples of Algorithm 2

// Class *CheckGeneralizedTree* will mainly take input from the user and the input file. It also uses class *Vertex* for storing some important information on a vertex of a given graph

```
class CheckGeneralizedTree
{
        public static void main(String [] args)throws IOException
        {
                Vertex [] vertex = new Vertex[v];

                DFSVisit dv = new DFSVisit();

                // reading the input adjacency matrix from the input file

                for(int i = 0; i < v; i++)
                {
                        text = fileRead.readLine();
                        StringTokenizer st = new StringTokenizer(text);
                        int temp_list [] = new int[v];

                        for(int j = 0; j < v; j++)
                        {
                                inputString = st.nextToken();
                                matrixInput = Integer.parseInt(inputString);
                                temp_list[j] = matrixInput;

                        } // end of inner loop
                        vertex[i] = new Vertex(temp_list);
```

```
                } // end of outer for loop


        for (int i =0; i<v; i++)
        {
                        if(vertex[i].color == 0)
                                dv.dfs_visit(vertex,i,v);
                }
          }// end of main method
}
```

// Class *Vertex* is used to store different important information on a vertex such as its adjacency list, its parent and its color (color information will be used for the modified color DFS algorithm)

```
class Vertex
{
        int color; // 0 = white, 1 = grey, 2 = black
        int adjList[];
        int parent;
        Vertex(int list[])
        {
                parent = -1;
                adjList = list;
                color = 0;
        }
        int GetDegree()
        {
                int list_length = adjList.length;
                int count_degree = 0;
                for(int i=0; i<list_length; i++)
                {
                        if(adjList[i] == 1)
                        {
                                count_degree++;
                        }
                }
                return count_degree;
        }
} // end of Vertex class
```

// Class *DFSVisit* for detecting a cycle in the graph and also to find out the length of the cycles

```
class DFSVisit
{
	static int time_count = 0;
	static int last_cycle_vertex = -1;

	public void dfs_visit(Vertex [] vertex, int v_id, int v)
	{
		vertex[v_id].color = 1;
		time_count++;
		boolean cycle_check;

		for(int i=0; i<v; i++)
		{
			if(vertex[v_id].adjList[i] == 1)
			{
				if(vertex[i].color == 1 && vertex[v_id].parent != i)
				{
					System.out.println("Cycle exists");
					if(i == 0)
					{
						System.out.println(time_count);
					}
					else
					{
				time_count = (time_count+1)-2; // finidng the exact cycle length
					}

					cycle_check = check_odd_number(time_count);

					if (i < last_cycle_vertex || cycle_check == false)
					{
						System.out.println("Not generalized tree");
						last_cycle_vertex = -1;
					}

					last_cycle_vertex = (i + time_count) - 1; // storing the
last vertex from the discoverd cycle
				}
				if(vertex[i].color == 0)
				{
					vertex[i].parent = v_id;
```

```
                                dfs_visit(vertex,i,v);
                            }
                        }
                } // end of for loop
                vertex[v_id].color = 2;
        }

// Class ReverseSteps is mainly responsible for reversing the reduction steps and after every
step checks if the graph is still elementary or not

class ReverseSteps
{
        ReverseSteps(String filename) throws IOException
        {
                fr = new FileReader(filename);

                br = new BufferedReader (fr);
                build_string();
        }

        void build_string() throws IOException
        {
                while((str=br.readLine())!= null)
                {
                        recognize(str);
                }
        }

        void recognize(String s)
        {
                if(s.startsWith("<"))
                {
                        maximum_number = Character.getNumericValue(s.charAt(1));
                }
        }

        // ReverseWrite function will print the matrices from where loops were removed
        // in a reverse way

        void ReverseWrite(String fname) throws IOException
        {
                FileReader fr2 = new FileReader(fname);
                br = new BufferedReader (fr2);
```

```
                for(int i = maximum_number; i>0; i--)
                {
                        while((str=br.readLine())!= null)
                        {
                                if((str.startsWith("<")) &&
(Character.getNumericValue(str.charAt(1)) == i))
                                {
                                        while(!((str = br.readLine()).equals("end_matrix")))
                                        {
                                                pw.println(str);
                                        }
                                        pw.println("end_matrix");
                                } // end of if
                        }// end of while loop
                }// end of for loop
        } // end of ReverseWrite()

        //ReadForValue will read input from file

        void ReadForValue() throws IOException
        {
                int matrix_size = StringLength / 2 ;
                System.out.println(matrix_size);

                int [][] output_matrix = new int[matrix_size][matrix_size];

                while(true)
                {
                        st = new StringTokenizer(str);
                        token = st.countTokens();

                        if(!str.equals("end_matrix") && token != 0)
                        {
                                System.out.println(str);

                                while(st.hasMoreTokens())
                                {
                                        inputString = st.nextToken();
                                        System.out.println("input String"+inputString);
                                        matrixInput = Integer.parseInt(inputString);
                                        System.out.println("i = "+i+" j = "+j);
                                        output_matrix[i][j] = matrixInput;
                                        j++;
```

```
                    }

                    i++;
                    j = 0;
                }// end of first outer if

                if(str.equals("end_matrix"))
                {
                    CheckElementary ce = new CheckElementary();
                    result = ce.ISElementary(output_matrix);

                    if(result == false)
                    {
                System.out.println("This graph is not a elementary graph.");
                        break;
                    }
                else
                    {
                        new_matrix = true;
                    }

                } // end of second outer if

                if((str = br.readLine()) == null) break;
                StringLength = str.length();

                if(new_matrix == true)
                {
                    i = 0;
                    new_matrix = false;
                    matrix_size = StringLength / 2 ;
                    output_matrix = new int[matrix_size][matrix_size];
                }
            } // end of while loop
        } // end of ReadForValue
} // end of scanner class
```

// Class *CheckElementary* is responsible for checking if the unfolded graph is still elementary or not.

```
class CheckElementary
{
        boolean ISElementary(int[][] input_matrix)
        {
                int v = input_matrix.length;
                Vertex [] vertex = new Vertex[v];
                int matrixInput;
                String inputString;

                // reading the input adjacency matrix from the input file
                for(int i = 0; i < v; i++)
                {
                        int temp_list[] = new int[v];
                        for(int j = 0; j < v; j++)
                        {
                                temp_list[j] = input_matrix[i][j];

                        } // end of inner loop

                        vertex[i] = new Vertex(temp_list);
                } // end of outer for loop

                int vertex_degree;
                int allow_edge = 0;
                boolean elementary_graph = false;
                boolean return_value = false;
                System.out.println("Vertex Length: "+vertex.length);

                if(vertex.length == 2)
                {
                        if(vertex[0].adjList[1] == 2 && vertex[1].adjList[0] == 2)
                                return_value = true;
                }
                else
                {
                        for(int i = 0; i< v; i++)
                        {
                                vertex_degree = vertex[i].GetDegree();

                                if(vertex_degree == 2)
```

```
                    {
                        for(int j = 0; j < v; j++)
                        {
                            if(vertex[i].adjList[j] == 1)
                            {
                                if(vertex[j].GetDegree() == 2 ||
vertex[j].GetDegree() == 1)
                                {
                                    allow_edge++;
                                }
                            }
                        } // end of inner for loop

                        if(allow_edge != 2)
                        {
                        System.out.println("Not an elementary graph.");
                            break;
                        }
                        else
                        {
                            allow_edge = 0;
                            elementary_graph = true;
                        }
                    } // end of outer if
                }// end of outer for loop

                if(elementary_graph == true)
                {
                    System.out.println("This is a elementary graph.");

                    return_value = true;
                }

            else { return_value = false;}
        } // end of most outer else
        return return_value;

} // end of ISElementary
}  // end of CheckElementary class
```

Appendix C

Code Examples of Algorithm 3

/* Class *CheckBCNut* works like a bridge between the classes *Vertex* and *BabyChestNut* . It reads the input graph from a file, then creates a *Vertex* object for each vertex and sends them to *BabyChestNut* for checking. After receiving the requested information form BabyChestNut , it returns the final result. */

```
class CheckBCNut
{
        static int parallel_edge;
        public static void main(String [] args)throws IOException
        {
          // reading the input adjacency matrix from the input file

                for(int i = 0; i < v; i++)
                {
                        text = fileRead.readLine();
                        StringTokenizer st = new StringTokenizer(text);
                        int temp_list [] = new int[v];

                        for(int j = 0; j < v; j++)
                        {
                                inputString = st.nextToken();
                                matrixInput = Integer.parseInt(inputString);
                                temp_list[j] = matrixInput;

                        } // end of inner loop
                        vertex[i] = new Vertex(temp_list);
                } // end of outer for loop
```

```
        // calling responsible method and producing the final result

    int check_edges;
    for(int i=0; i<v; i++)
    {
                    check_edges = bcn.FindBabyCheastNut(vertex,i,v);

                    if(check_edges > 2)
                    {
                            System.out.println("This is not a baby cheast nut.");
                            break;
                    }
            } // end of for loop

            if (parallel_edge == 2)

                    System.out.println("This is a baby chest nut.");

            else System.out.println("This is not a baby cheast nut.");

        }// end of main method
}

// BabyCheastNut class is responsible for performing necessary checking and to decide a
given graph is a baby chestnut or not

class BabyCheastNut extends CheckBCNut
{
        int FindBabyCheastNut(Vertex [] vertex, int v_id, int v)
        {
                int count_edge = 0;

                for(int i =0; i<v; i++)
                {
                        if(vertex[v_id].adjList[i] == 1)
                        {
                                if(v_id != i && count_edge < 2)
                                {
                                        count_edge++;
                                }
                        }
                        if(vertex[v_id].adjList[i] == 2)

                                parallel_edge++;
```

```
            }

            return count_edge;
        }// end of FindBabyCheastNut()
}


/* Class ChcekImperviousLoop is responsible for performing the major operations for the
step three of the algorithm. This class will check if an eliminated loop is impervious or not.
*/


class CheckImperviousLoop
{
        boolean CheckForImperviousLoop(int matrix[][], int matrix_id)
        {
                if(matrix_id == 1) // 0 = main principal vertex , 2 = another vertex
                {
                        for(int i=0;i < matrix.length; i++)
                        {
                                second_vertex = check_second_vertex(matrix, i);

                                for(int j=0; j < matrix.length; j++)
                                {
                                        if(matrix[0][0] > 2)                                    {
                                                viable = true;
                                        }

                                if(second_vertex == true && (i==j) && matrix[i][j] > 2)
                                        {
                                                viable = true;
                                        }
                                } // end of inner for loop
                        } // end of outer for loop
                } // end of if


          boolean self_loop = false; // indicate the vertex contain a loop
        int loop_count = 0; // count the number of loop so if there is no loop it will return 0

                if(matrix_id > 1)
                {
                        for(int i =0; i<matrix.length; i++)
                        {
```

```
                        if(matrix[i][i] >= 2)
                        {
                                self_loop = true;
                                loop_count++;
                        }
                } // end of outer for loop
        }// end of if which checking matrix_id

        if(loop_count == 0)
        {
                viable = true;
        }
        return viable;
} // end of CheckForImperviousLoop

// Checking for a second vertex in the given graph

        boolean check_second_vertex(int [][]matrix, int selected_vertex)
        {
                int length = matrix.length;
                boolean second_vertex = false;

                for(int i=0; i<length; i++)
                {
                        if( i != 0 && i != 2)
                        {
                                for(int j=0; j<length; j++)
                                {
                                        if(j == selected_vertex && matrix[i][j] >= 1)
                                        {
                                                second_vertex = true;
                                        }
                                }
                        }

                }
                return second_vertex;
        }

} // end of CheckImperviousLoop Class
```

```
// Class Scanner is responsible for producing the reverse steps output file

class Scanner
{
        Scanner(String filename) throws IOException
        {
                fr = new FileReader(filename);
                br = new BufferedReader (fr);
                build_string();
        }

        void build_string() throws IOException
        {
                while((str=br.readLine())!= null)
                {
                        recognize(str);
                }
}
        void recognize(String s)
        {
                if(s.startsWith("<"))
                {
                        maximum_number = Character.getNumericValue(s.charAt(1));
                }
        }

// ReverseWrite function will print the matrices from where loops are removed in a reverse
way

        void ReverseWrite(String fname) throws IOException
        {
                FileReader fr2 = new FileReader(fname);
                br = new BufferedReader (fr2);

                for(int i = maximum_number; i>0; i--)
                {
                        fr2 = new FileReader(fname);
                        br = new BufferedReader (fr2);

                        while((str=br.readLine())!= null)
                        {
                                if((str.startsWith("<")) &&
(Character.getNumericValue(str.charAt(1)) == i))
                                {
```

```
                                        while(!((str = br.readLine()).equals("end_matrix")))
                                        {
                                                pw.println(str);
                                        }
                                        println("end_matrix");

                                } // end of if
                        }// end of while loop
                }// end of for loop
        } // end of ReverseWrite()

        void ReadForValue() throws IOException
        {

        // Similar code like algorithm 2

        boolean build_org_matrix(int [] single_array_matrix,int index_size)
        {
                int matrix_length = single_array_matrix.length;

                int [] [] Org_Matrix = new int[index_size][index_size];
                int k = 0;

                for(int i = 0; i<index_size; i++)
                {
                        for(int j=0; j<index_size; j++)
                        {
                                if(k < matrix_length)
                                {
                                        Org_Matrix [i][j] = single_array_matrix[k];
                                        k++;
                                }
                        }
                } // end of for loop

                CheckImperviousLoop cil = new CheckImperviousLoop();
                boolean viable = cil.CheckForImperviousLoop(Org_Matrix, matrix_id);
                matrix_id++;

                return viable;

        } // end of build_org_matrix()
} // end of scanner class
```