ALGORITHMS AND SOFTWARE SYSTEMS FOR
LEARNING AND RESEARCH

A DISSERTATION
SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
for the degree
DOCTOR OF EDUCATION

by
ADRIAN HEINZ

Committee Approval:

Dr. Jay Bagga Committee Chairman Date

Dr. Samuel Hsieh Committee Member Date

Dr. Michael McGrew Committee Member Date

Dr. Thalia Mulvihill Committee Member Date

Department Head Approval:

Dr. Paul Buis Chairman of Department Date

Graduate Office Check:

Dr. Robert Morris Dean of Graduate School Date

BALL STATE UNIVERSITY
MUNCIE, INDIANA
July 2009
Software systems have experienced an impressive growth in the last few decades and have impacted a wide variety of areas. In this respect, two fields benefit greatly. Learning and research. In this work, we present several software systems that we have created to assist in the process of learning and to help researchers by performing complex computations and generating data. We demonstrate three web-based educational video games that we developed to teach science to middle school students. We also describe several software systems that we created for research in graph theory and model checking. Finally, we discuss our results, contributions and future directions.
ALGORITHMS AND SOFTWARE SYSTEMS
FOR LEARNING AND RESEARCH

A DISSERTATION
SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
for the degree
DOCTOR OF EDUCATION

by
ADRIAN HEINZ

ADVISOR – DR. JAY BAGGA

BALL STATE UNIVERSITY
MUNCIE, INDIANA
JULY 2009
Acknowledgements

Throughout my academic career I have been privileged to work with exceptional educators and outstanding researchers. They all deserve credit on this work.

Profound words of gratitude go to my dissertation advisor Professor Jay Bagga for his continuous support throughout my years at Ball State University. His unselfish guidance as a mentor and role model has enlightened my life.

Many thanks to all members of Science Literacy Project (SLP) including Dr. Bizhan Nasseh, Dr. Laurie Mullen, Dr. Lisa Huffman and Dr. James Jones for their professionalism and hard-work during the development process. Special thanks to Dr. James Jones for providing me with the research results. Also, major words of appreciation to Todd Meister for his extraordinary assistance with the ASP website and to Zack Martin of Outside Source Design for his immense help on the creation of the SQL Server database.

Our research colleague Mahbub Majumder also deserves credits and thanks for his contributions on our graceful labeling research.

Thanks to members of the Dissertation Committee including Dr. Mike McGrew, Dr. Sam Hsieh and Dr. Thalia Mulvihill for their review of this work and their encouragement throughout my academic career.

I am also thankful to Computer Science Department Chair Dr. Paul Buis and
all faculty and staff of the department for making my graduate studies an enriching experience.

Special words of affection to my parents, brothers and sister for all their love and to the hundreds of friends around the world that have made a lasting impression in my life. Their friendship is a priceless treasure.

Finally, I am indebted to my alma mater Ball State University for providing me with the tools to succeed in life and to realize my dreams.
Contents

Acknowledgements ii

1 Introduction 1
   1.1 Survey of literature ................................. 4

2 Educational Perspectives 6
   2.1 Science Teaching and Learning ...................... 6
   2.2 Software systems in education ..................... 8

3 Graph Algorithms and Their Applications 10
   3.1 Background and definitions ......................... 12
      3.1.1 Terminology, topics and applications ............. 13
   3.2 Graceful labeling .................................... 18
      3.2.1 Research findings ............................... 19
   3.3 Manohar .............................................. 24
      3.3.1 Graphical User Interface ......................... 24
      3.3.2 Implementation details ........................... 28
      3.3.3 Enumeration of graceful labelings of paths ....... 29
      3.3.4 Enumeration of graceful labelings of suns ........ 31
   3.4 Colossus ............................................. 37
      3.4.1 Graphical User Interface ......................... 37
      3.4.2 Implementation ................................. 41
   3.5 Graph Algorithm Constructor ......................... 41
      3.5.1 Graphical user interface ......................... 41
4 E-learning

4.1 SLP ......................................................... 49
4.2 Implementation Details ................................. 50
  4.2.1 Software Development Process ..................... 50
  4.2.2 Author’s Contributions ............................ 52
  4.2.3 Introduction to Adobe Flash and ActionScript ........ 53
  4.2.4 Sample Flash Movie - Pirate Store .................. 57
  4.2.5 Sample ActionScript Class - WordProblem.as ....... 60
4.3 Testing and Results ...................................... 62
4.4 Pirate Math ............................................... 63
  4.4.1 Introductory Screen .................................. 64
  4.4.2 Main Menu .......................................... 64
  4.4.3 Coin Toss .......................................... 65
  4.4.4 Pirate Gold ......................................... 68
  4.4.5 Treasure Map ........................................ 69
  4.4.6 Pirate Store ......................................... 71
4.5 Chemistry Circus ........................................... 74
  4.5.1 Introductory Screen .................................. 74
  4.5.2 Demo ................................................. 75
  4.5.3 Game Play .......................................... 75
4.6 Cellular Divide and Conquer ............................. 77
  4.6.1 Main menu .......................................... 78
  4.6.2 Help Screen ........................................ 78
  4.6.3 Game Play .......................................... 80
5 Model Checking
5.1 Critical systems ........................................ 87
5.2 Formal methods ........................................ 87
   5.2.1 SPIN .................................................. 88
5.3 Assembly Line Simulator .................................. 90
   5.3.1 Machine description .................................. 91
   5.3.2 SPIN model ........................................... 92

6 Conclusion .................................................. 102
6.1 Research .................................................. 103
6.2 Future directions .......................................... 103

A Pirate Store ActionScript Code ................................ 106

B Word Problem Class ......................................... 111

C Assembly line simulator flow diagrams ...................... 120

D Assembly line simulator source code ........................ 125

E Source code and executable files ............................ 131

List of Tables .................................................. 133

List of Figures .................................................. 136

Bibliography ..................................................... 1
Chapter 1

Introduction

Computers and their software applications are becoming powerful tools in education and are also a valuable resource to researchers. A clear example is E-learning, a recent educational paradigm involving computer applications, which is reshaping the traditional teaching process of lectures, homework and tests. Schunk [58] mentions in his book that a technological classroom was constrained to movies, slide projectors, radio and others. Nowadays, the wide variety of software available for educational purposes creates endless opportunities. Students can visualize models of natural phenomena, interact with peers from far away distances, consult large databases, join discussion groups that study a particular subject and receive expert instruction online. In the area of research, the advantages are numerous. For instance, software systems can perform complex computations, analyze large amounts of data, perform simulations or even guide [44] an uncrewed NASA rover on Mars. For all these reasons, software systems play a significant role in these areas and this role will be predominant in the future.

In this work, we describe several software systems that we have developed and show how they contribute to facilitate the learning process by using innovative teaching techniques. Furthermore, these systems have also helped us make important progress in our research. Our software systems cover three main areas: graph theory, e-learning and model checking.
In the area of **graph theory**, our software system *Manohar* allowed us to discover important properties of graceful labelings of graphs leading to the publication of two scientific papers [14, 15]. *Manohar* and our other graph software systems *Colossus*, *Graph Algorithm Constructor* and *JEdit* have been presented in national and international conferences including India [12, 13] and Vietnam [11]. Also, in 2001 *JEdit* was accepted for a presentation in the prestigious Graph Drawing Conference held in Vienna, Austria [9].

Regarding **E-learning**, we developed three web-based educational video games to teach science to middle-school students. This work was part of a project called *Science Learning Project (SLP)* [62], which was funded by a grant from the U.S. Department of Energy. The three games implemented are *Pirate Math*, *Chemistry Circus* and *Cellular Divide and Conquer* in the fields of Mathematics, Chemistry and Biology respectively. The games were used by over 200 students in four schools.

Concerning **Model Checking**, we implemented an *assembly line simulator* to process machinery parts and verified certain safety properties using *SPIN*, an award-winner model checking tool. This model and its results were presented at the International Colloquium on Theoretical Aspects of Computing held in 2005 in Hanoi, Vietnam [10].

This work is organized in several chapters that cover all of the above mentioned areas.

In chapter 2, *Educational Perspectives*, we discuss the process of science teaching and learning by describing well-known learning theories. We also show how our software systems incorporate those learning theories and illustrate how we used our systems in the classroom.

In chapter 3, *Graph algorithms and their applications*, we begin by presenting some
background information about graph theory, its terminology and applications. Next we provide an introduction to graceful labelings, describe known results, and current research in the area and discuss open problems. We then describe our software system Manohar, which computes and enumerates graceful labelings of certain classes of graphs. We continue with Colossus, a system for computing visibility graphs of polygons. We show several features such as real-time computation and manipulation of orthogonal polygons. We then describe Graph Algorithm Constructor, which is the foundation of our ultimate goal, a system that allows creation and execution of graph algorithms without writing computer source code. Even though this system is not yet functional, we present its current development state and future work. We conclude this chapter with Jedit, a Java-based system for drawing graphs and running graph algorithms. We describe some of the most important algorithms implemented and features added to the new version 5.

In chapter 4, E-Learning we present the Science Literacy Project (SLP). We begin by listing project members, teams role and project goals. We then describe the development process detailing each of its phases. We continue with an introduction to Adobe Flash and its scripting language ActionScript. We briefly describe the development of one module and an ActionScript class. We continue by presenting the three web-based games developed in the areas of mathematics, chemistry and biology. These games are Pirate Math, Chemistry Circus and Cellular Divide and Conquer respectively. For each game, we identify the learning objectives, misconceptions targeted, story-line, graphical interface and game play. We conclude by showing results and conclusions.

In chapter 5, Model Checking, we present an assembly line simulator to process machinery parts. We begin by analyzing critical systems and why model checking is used for simulation and verification purposes. We then give a brief introduction to SPIN and its specification language PROMELA. We continue with an overview of assembly lines and present our simulator by describing the real machine and our simulation model. We show sample runs and verification of certain safety properties.
Finally, in Chapter 6, we summarize our work, discuss our results and provide the foundation for future research.

1.1 Survey of literature

Graph theory and graph algorithms have been studied extensively. This study has led to the publication of a large number of research papers and books in graph theory, and a wide variety of its subareas. The main subareas covered in this work are graph labeling and visibility graphs. Many graph labelings are based on the 1967 paper by Alex Rosa [55], where he identified three types of labelings called $\alpha$ – valuation, $\beta$ – valuation and $\rho$ – valuation. He defined a function $f$ as a $\beta$ – valuation of a graph $G$ with $q$ edges if $f$ is an injection from the vertices of $G$ to the set $0, 1, ..., q$ such that, when each edge $xy$ is assigned the label $|f(x) - f(y)|$, the resulting edge labels are distinct. Such valuation was later called graceful by Golomb [31] and it is the term used nowadays. The well-known Ringel-Kotzig conjecture states that all trees are graceful [39]. Even though this conjecture is still open, many classes of trees are known to be graceful. Such classes include: paths and caterpillars [55], trees with at most four end-vertices [68], trees with diameter five [38] and trees with at most 27 vertices [3]. In this work, we use the book by Gross et al. [32] for terminology and applications of graph theory. The graceful labeling section is based on several sources including Rosa [55] and Gallian [30]. In the area of visibility graphs, O’Rourke’s book [51] provides an excellent introduction to the topic and describes its applications in art gallery problems.

Electronic learning or E-learning has also been studied extensively. The book by Clark et al. [21] gives an introduction by defining of E-learning and providing description of its development process. Rosenberg’s book [56] lists benefits and describes the current E-learning revolution. Holmes et al. [34] exhibits E-learning resources and analyzes challenges and opportunities. Many other areas are emerging from electronic learning. For instance, McCracken [47] discusses virtual learning communities.
where communications and interactions are dependent upon media-based tools. In this work, we present a discussion of E-learning based on the books by Clark et al [21] and Allen [5]. The Adobe Flash tutorial is based on the book by Yeung [67] and the ActionScript tutorial is derived from the book by Lott [46] and Moock [48].

Model checking has been used for the formal verification of a wide variety of systems. Kars [40] presents a case study where model checking techniques were used for the verification of algorithms of the flood control barrier built in the nineties close to Rotterdam in the Netherlands. Schneider et al. [57] describe the use of the SPIN model checker to verify the correct working of the handoff algorithms for the dual control CPUs in the space mission Cassini. Joshi et al. [37] verified the correct working of the resource arbiter that manages the use of all motors of the Mars Exploration Rovers. Our work in Model Checking is based on the book by Holzmann [35] and the description of critical systems is based on Knight [42].
Chapter 2

Educational Perspectives

The study of science has to be promoted from an early age so that middle and high school students have a strong background to pursue advance degrees in scientific disciplines. This can be achieved by presenting the material as relevant and enjoyable. Students should see sciences as fun and useful disciplines having real everyday applications. In that respect, the use of technology in education is a valuable tool to attain this objective since it makes students active participants in the learning process. This technology can range from a simple computer animation for elementary school students to a complex flight simulator for military pilots. In this work, we demonstrate how technology can be a valuable tool to achieve this objective. In particular, we concentrate in E-learning and the use of software systems to do research and to teach science. We show how we employ our software systems with middle school as well as college students.

2.1 Science Teaching and Learning

According to Pearsall [52], the classic classroom environment where students learn by lectures, assigned readings, quizzes and labs is not as effective as it has been typically assumed. Educators have been frequently frustrated by the failure of students to understand basic concepts of science. Much of the time students are unable to discuss and reflect on difficult material. She states that there is increasing evidence
to suggest that these methods are not helping students acquire an understanding of the science concepts being taught.

Alberts [23] explains that people relate to their experiences by creating mental models to understand the universe. When faced with a new event or phenomenon, they use these models to interpret the information, to make generalizations or to make predictions. An article published in Academictips.org website [1] describes a technique used to remember people’s names or foreign words by association with a familiar face or idea. An example from personal experience occurred while assisting Japanese friends to memorize the Spanish word “azúcar” (sugar). As they were struggling to remember it, an association with the familiar Japanese female name “Azuka” helped them learn the new foreign word.

The book by Schunk [58] describes several theories that have been created to facilitate the process of learning. Two popular theories are behaviorism and constructivism.

In behaviorism, learning occurs when a new behavior is acquired through conditioning. There are two types of conditioning. In classical conditioning the new behavior arises from a response to stimulus. This was observed by the Russian psychologist Ivan Petrovich Pavlov who noticed that sometimes dogs drooled even though there was no food in sight. This was because every time the dogs received food, the server was wearing a lab coat. Therefore, the dogs reacted to the lab coat anticipating that food was going to be served. Pavlov was successful in obtaining the same reaction by utilizing a bell instead of lab coats. In his experiment, the server did not wear a lab coat but instead, played a bell before serving food. The dogs quickly related the sound of the bell with food and after a while, by only hearing the bell, they reacted by drooling. In operant conditioning desired behavior is reinforced by a reward, while undesired behavior is discouraged by punishment. Deeley [24] presents an example of this in a system used to train dogs where the trainer feeds the dog (reward) if the behavior was appropriate and punishes the animal otherwise.
CHAPTER 2. EDUCATIONAL PERSPECTIVES

In constructivism, learning is seen as a process where the learner creates new ideas or concepts from current and previous knowledge or experience. A perfect example is the MOON Project [61]. This project started in 2001 at Ball State University and currently links several universities in the United States and even other countries such as Australia and Japan. This project is aimed at elementary and middle school students who through a network or “research buddies” are in charge of studying the moon every night, discussing their findings and conceptualizing the phases of the moon. In this way, students are not directly taught the material but instead, they are active participants in the process of discovering new knowledge.

As shown in the next section, our educational software combines innovative teaching methods with appealing graphics, animations and sounds with concepts based on behaviorism and constructivism.

2.2 Software systems in education

We have developed several software systems for teaching and research. We have used our software with graduate, undergraduates and middle school students to help them visualize and experiment with new concepts. We now provide details on the way we utilize these systems.

Our software system JEdit [33] is a Java based application for drawing graphs, performing graph operations and executing algorithms. This system has been used in a graph theory course at Ball State University for over a decade. This course provides an introduction to graph theory by describing terminology, algorithms and applications. Students are initially exposed to the new material by traditional methods including classroom presentations, reading assignments and homework problems. Once students acquire a basic understanding of graph theory, they are presented with our software system JEdit. In this system, they put the new knowledge into practice by drawing graphs and experimenting with custom scenarios to discover how certain
operations and algorithms work. A feature of JEdit allows the step-by-step execution of an algorithm. This valuable feature allows students to conceptualize the general idea about the way algorithms perform operations. In addition, JEdit is extensible so that new algorithms can easily be added by writing the appropriate Java code without the need to modify existing source code. This characteristic has been very useful. During the class, each student is assigned a project consisting on the implementation of a new algorithm. In this way, students get hands-on experience not only in graph theory but in software development as well. This a fine example of a constructivism-based approach since students “discover” the material and “create” new knowledge by direct experimentation. JEdit is described in detailed in section 3.6.

We have also developed three web-based educational video games to teach science to middle school students. These games are detailed in section 4.1. The science areas covered are mathematics, chemistry and biology and the games developed are Pirate Math, Chemistry Circus and Cellular Divide and Conquer respectively. The games use an attractive combination of multimedia graphics, sounds and animations to engage students in the learning process. In addition, some of our games use humorous vocabulary to entertain and attract student’s attention such as in Pirate Math. This game uses pirate terminology when describing game objectives. The games teach science by showing similarities between the new material and concepts already familiar to students. For instance, Chemistry Circus compares the process of balancing chemical equations with the process of balancing the bar of an acrobat walking on a tight-robe. A similar concept is used in the Coin Toss module of Pirate Math where pirate chests are used to represent variables so that students clearly visualize that variables can change their value just as chests can contain a variable number of coins. Similarly, Cellular Divide and Conquer uses a clock to illustrate the phases of mitosis and cytokinesis. Finally, every game rewards correct answers and punishes incorrect ones in conforming to operant conditioning in the behaviorism theory of learning.
Chapter 3

Graph Algorithms and Their Applications

A graph is a combinatorial structure that is used to model a collection of objects and relationships between pairs of such objects. In other words, a graph is a network for which each element may be connected to others. Such collections of objects appear quite frequently in the real world. For instance, the cities to which a particular airline provides service can be thought as a network; each city would be an object and the flights available between cities would be the links that connect them. Similarly, the webpages that constitute a website can also be considered a network since every page would be an object and hyperlinks would connect them.

Precisely speaking, a graph consists of two sets called vertices (or nodes) and edges that connect vertices. Sometimes, it is useful to assign properties to these vertices and edges. These properties are usually color and weight, but any other useful information can also be used [32].

Due to the large number of applications of graph theory and the ease with which graph structures can be modeled by computers, research in this field has experienced an impressive growth in the last three decades. In this respect, software systems are an invaluable tool to experiment with graphs since they assist researchers in the study
of large amounts of data and execution of complex algorithms. For instance, a famous conjecture that all trees are graceful \cite{54} was stated in 1964. At that time, computer technology was not developed enough, and it was difficult to experiment with large trees. Nowadays, such technology makes this research possible. For instance, in 1998 Aldred and McKay used computer software to prove that all trees up to 27 vertices are graceful \cite{3}. Another example is enumeration of graceful labelings. We used our software Manohar to compute all graceful labelings of a cycle with 20 vertices. These graceful labelings number over a million \cite{14}. It is clear that software applications are naturally suited for this kind of research.

Moreover, software systems are also valuable tools for teaching graph theory since students can visualize and experiment with step-by-step execution of algorithms. This interactive feature helps not only in the understanding of graph theory concepts, but also in the development of critical thinking skills since students can also experiment with several graph configurations, modify algorithms and try different case scenarios.

There is also another important reason to develop software systems for graph algorithms. Researchers in graph theory come from a wide variety of fields. Even though these researchers need to implement graph algorithms in a computer system, they may not have enough programming background to develop them. Therefore, it is necessary to create software systems that people with little computer programming experience could use. This goal has been accomplished in our software applications. These applications have helped us in our own research in graph theory. By using these systems we were able to generate large amounts of data, carefully analyze it and discover important properties of graphs. The results have been published \cite{15, 14} and presented in national and international conferences including those in India \cite{13, 12}, Vietnam \cite{11} and Austria \cite{9}. Please refer to appendix E for the entire source code and executable files of our software systems.

Moreover, these software systems have been used to teach graph theory to students with various backgrounds by illustrating the execution of graph algorithms and
well-known properties of graphs. Also, students are given assignments requiring the addition of new algorithms to our systems.

This chapter is subdivided into several sections. We begin with some background of graph theory, terminology that will be used throughout this work and present some applications. Next, we describe several software systems that we have developed. The first one is Manohar, a system for computing graceful labelings of graphs. We describe Manohar’s main features, algorithms and its graphical user interface. Next, we present Colossus, a system for computing visibility graphs of polygons. We also discuss Graph Algorithm Constructor. This system is the foundation of our ultimate goal, to implement graph algorithms without the need to write computer code. Although this system is still a work in progress, we show its current status and describe the future directions. Finally, we review JEdit, a system for drawing graphs and creating graph algorithms. We present some of its algorithms and new features introduced in version 5.0.

3.1 Background and definitions

The first paper on Graph theory is considered to be the one published in 1736 by Leonhard Euler [17]. In this paper, Euler describes the problem of the Seven Bridges of Königsberg [59]. This is a classic problem about seven bridges in the city of Königsberg, which was the capital of the German province of East Prussia. This place is now known as Kaliningrad, and it is in Russia. The city was divided into several land areas by the river Pregel. There were seven bridges connecting these land areas as shown in figure 1. The thick dark lines represent the bridges, and the land areas are labeled A, B, C and D.

People in the city wondered if starting at any land area, there was a walk that would visit A, B, C and D by crossing each bridge exactly once and returning to the starting point. The proof proposed by the Swiss mathematician Leonhard Euler uses an abstraction model similar to the one shown in figure 2. Notice that this model is
Figure 1: Seven Bridges of Königsberg

A graph. The land areas are represented by nodes and the bridges are the links that join them. Using this model, Euler proved that no such walk exists. The publication of this result is considered to be the “birth” of Graph Theory.

Figure 2: Graph representation of the problem

Since the publication of Euler’s result, there have been numerous publications and books written in the area. We have used [32] for fundamental definitions and terminology. Several concepts of visibility graphs, properties and definitions are taken from [51]. Contributions in this area are part of the ongoing work in visibility graphs at Ball State University [7, 8, 26]. Also, we present terminology and results in graceful labeling from [16, 29, 30, 49, 55].

3.1.1 Terminology, topics and applications

Gross and Yellen [32] provide the following terminology. A graph $G = (V, E)$ is a combinatorial structure containing two finite sets $V$ and $E$. The set $V$ consists of elements $v_1, ..., v_n$ called vertices (or nodes). The set $E$ contains elements $e_1, ..., e_m$, etc.
usually referred to as **edges**. Each edge \( e_k \) is a pair \((v_i, v_j)\) of vertices where \( v_i \) and \( v_j \) are called the endpoints of \( e_k \). We say that \( v_i \) and \( v_j \) are **adjacent vertices** since they are joined by \( e_k \). If two edges have an endpoint in common, they are referred to as **adjacent edges**. Notice that an edge may not necessarily connect two vertices. A **self-loop** connects a vertex to itself. If the edge is not a self-loop, it is called **proper edge**. A set of more than one edge with identical endpoints is a **multi-edge**. An edge in which one of its endpoints is designated as the **tail** and the other as the **head** is a **directed edge** or **arc**. If the edges of a graph are directed, the graph is called **directed graph** or **digraph**. A graph that contains no self-loop and no multi-edge is a **simple graph**. In this work, we deal only with simple graphs. Hereafter, the term graph will mean a simple graph, unless otherwise stated. For a vertex \( v \) of graph \( G \), the **degree** of \( v \), denoted \( \text{deg}(v) \) is the number of edges incident on \( v \). A vertex with degree 0 is called **isolated vertex**. A list of all the vertex degrees arranged in non-decreasing order is called the **degree sequence** of the graph. An alternating sequence \( W = < v_0, e_1, v_1, e_2, ..., v_{n-1}, e_n, v_n > \) of vertices and edges, such that \( \text{endpts}(e_i) = \{v_{i-1}, v_i\} \) for \( i = 1, ..., n \) is called a **walk** from \( v_0 \) to \( v_n \). Using this definition, a vertex \( v \) is reachable from vertex \( u \) if there is a walk from \( u \) to \( v \). In addition, if for every vertex \( v_i \) of the graph \( G \), there is a walk to any vertex \( v_j \) of \( G \), then it is said that \( G \) is **connected**.

A **path** \( P \) is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence and no vertices are repeated. A path with \( n \) vertices is denoted \( P_n \). A **cycle** is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence and only the start/end vertex repeats. A cycle with \( n \) vertices is denoted \( C_n \). A **tree** is an acyclic connected graph where acyclic means that the graph has no cycles. A **caterpillar** is a tree for which if all leaf vertices and their incident edges are removed, the resulting graph is a path. Following this criteria, for a given caterpillar, all its vertices are within distance 1 of a central path. A **lobster** graph is a tree in which all the vertices are within distance 2 of a central path [30] [32].
The corona operation was introduced in 1970 by Frucht and Harary [29]. It is represented by the symbol \( \odot \). Using graphs \( G \) and \( H \), the corona operation \( G \odot H \), takes one copy of \( G \), where \( G \) has \( p \) vertices, and \( p \) copies of \( H \), and then joins them by an edge the \( k^{th} \) vertex of \( G \) to every vertex in the \( k^{th} \) copy of \( H \) [16]. Given a cycle \( C_n \) and a graph \( H \) with \( k \) isolated vertices, the graph \( \text{SUN}(n, k) \) is defined as \( C \odot H \). Figure 3 illustrates a sample \( \text{SUN}(5, 2) \).

Numerous applications exist in a wide variety of areas including computer science, mathematics, topology, geography, astronomy, sociology and chemistry. We mention some examples. In the field of computer science, it is often necessary to sort a list of elements. Heapsort is a well-known sorting algorithm that relies on a data structure called heap which is a specialized graph structure [18]. The field of computational chemistry uses graph theory for manipulation of chemical information. An example is illustrated in figure 4, which depicts two hydrogen-suppressed molecular graphs [45, 66]. A graph structure is used to model Social-Acquaintance Networks, where vertices represent people, such as students and edges connecting vertices represent a relationship such as familiarity with each other when the course began [32].
Planarity

An important concept in graph theory, is the notion of a **planar graph**. We say a graph $G$ is planar if it can be drawn on a plane such that no two edges cross. In other words, $G$ has a planar embedding on the plane [32]. For instance, consider the graph depicted in figure 5. Even though the drawing $A$ has crossings, the same graph can be drawn with no crossing as shown in $B$. Hence, the graph is planar. A classic problem involving planar graphs is given in [20]. This problem is known as the three houses and three utilities problem. Given three houses and three utilities, say gas, electric and water, find a way to connect each of the houses to each utility avoiding utility lines to cross. This problem can be modeled by a graph structure where each house and each utility is represented by a vertex as illustrated in figure 6 where $H_1, H_2, H_3$ are houses and $E, G, W$ are electricity, gas and water respectively. The goal is to find a planar graph connecting houses with utilities. As shown in the figure, sequentially connecting electric, gas and water to houses 1, 2, 3 cannot be done since it leads to a drawing where vertex $W$ cannot be connected to $H_3$ since it would create crossings. In fact, this is a well-known graph called $K_{3,3}$. Kuratowski [43] proved in 1930 that this graph is non-planar.
Visibility

Graph theory is also used extensively in computational geometry. In the plane, we define a line segment as a subset of a line which is contained between two points. These points are called the endpoints of the segment. A polygon is an area of the plane enclosed by a finite set of line segments. These line segments are the boundary of the polygon. A polygon whose boundary does not cross itself is a simple polygon. Figure 7 depicts polygons $P_1$ and $P_2$. Notice that the boundary of $P_1$ does not cross itself and therefore it is a simple polygon while $P_2$ is not. In this work, we only deal with simple polygons. Hence, the word polygon means simple polygon unless otherwise stated. The inside (bounded region) of a polygon is the interior region and the outside is the exterior region.

Using these definitions, we say that 2 points $x$ and $y$ can see each other if and only if the entire closed segment $xy$ is never exterior to the polygon. Also, a visibility graph is a graph of intervisible locations where each vertex is a point location and each edge is a visible connection between them. In this way, an edge is drawn between two locations if and only if they can see each other. This visibility relation leads to the concept of ears and mouths. A point $x$ is an ear if and only if its adjacent endpoints
can see each other. A point \( x \) is a mouth if and only if the closed segment between its adjacent endpoints is entirely outside the polygon [51]. A well-known application of visibility graph is the art gallery problem. Given an art gallery room, the problem asks what is the minimum number of stationary guards needed to observe the whole gallery? In this case, a polygon is created based on the floor plan of the room and its visibility graph is used to model the problem.

### 3.2 Graceful labeling

Graceful labelings were introduced in 1967 by Rosa [55]. Rosa defined \( \beta \)-valuation and several other valuations to help with the decomposition of the complete graph into isomorphic subgraphs. In particular, \( \beta \)-valuations were developed as means to solve the conjecture of Ringel [54] that \( K_{2n+1} \) can be decomposed into \( 2n + 1 \) subgraphs that are all isomorphic to a given tree with \( n \) edges. Golomb [31] later called such labelings graceful, and it is the term that it is used nowadays. For a graph \( G \) with \( q \) edges, a **graceful labeling** is a function \( f \) which is an injection from the vertices of \( G \) to the set \( 0, 1, \ldots, q \) in such a way that when each edge \( xy \) is given the label \( |f(x) - f(y)| \), the resulting edge labels are distinct. If \( G \) has at least one graceful labeling, then \( G \) is a **graceful graph**. An \( \alpha \)-valuation is a graceful labeling with the requirement that it exists a value \( x \), where \( x \in 0, \ldots, n \), with the property that for any edge \((v_i, v_j)\) where \( v_i < v_j \), it always holds that \( v_i \leq x < v_j \) [55].

The well-known Ringel-Kotzig conjecture states all trees are graceful. Due to the large number of scientific papers published about this conjecture, Kotzig [39] has called the effort to prove it a “disease”. Although this conjecture is still open, some classes of graphs are known to be graceful. In his original paper, Rosa [55] proved that all paths and caterpillars are graceful and also presented the following result.

**Theorem 3.2.1** The cycle \( C_n \) is graceful if and only if \( n \equiv 0 \) (mod 4) or \( n \equiv 3 \) (mod 4).
Morgan [49] later proved that all lobsters with a perfect matching are graceful. For an account of the research in this area, we refer the reader to Gallian’s ongoing survey [30].

Graph labelings have applications in radio broadcasting. Suppose there is a geographical region where radio transmitters are given broadcasting frequencies. Transmitters that are close to each other should not share the same frequency since it would cause interference. The problem of minimizing the number of different frequencies assigned is usually modeled by a graph where each radio transmitter is represented by a vertex and an edge is drawn between transmitters that should not have the same frequency [32].

While most of the research in the area has been centered toward finding new classes of graceful graphs, our research has evolved into two areas. The first one is finding a graceful labeling and the second one is enumerating graceful labelings. In the first case, we are given a graph and the objective is to find a graceful labeling or to prove that no such labeling exists. In the second case, we are given a graph that is known to be graceful and we look for all its graceful labelings. To help our research, we have developed original algorithms and implemented them in our software system Manohar. We give a detailed description of Manohar in section 3.3.

3.2.1 Research findings

As part of a collaborative research work with Jay Bagga and Mahbub Majumder, we discovered important properties of graph labelings for certain classes of graphs. We started by carefully analyzing the data generated with our software system Manohar, and we noticed several recurrent patterns about the structure of graceful labelings. This analysis led us to make conjectures about their properties. Further work allowed us to prove several of these properties verifying our original conjectures. These results were presented at the Thirty-Eighth Southeastern International Conference on Combinatorics, Graph Theory, and Computing in Boca Raton, FL. and published in
[14, 15]. We will now give an overview of these properties.

In order to describe our results, it is necessary to introduce some notation. We describe our findings in terms of a cycle $C_n$ where $n$ is 0 or 3 ($mod$ 4). Notice that in a graceful labeling of $C_n$, there are $n$ labels from the set $\{0, 1, 2, ..., n\}$ of $n + 1$ elements and therefore, one label is missing. We denote this missing label by $m$. Given the cycle $C_n$ with vertices $v_1, v_2, ..., v_n$ in counterclockwise order and given a graceful labeling $f$ of $C_n$ with a missing label $m$, we define sets $H$, $L$ and $I$ as the sets of high, low and intermediary labels respectively. A label $f(v_i) \in H$ if $f(v_i) > f(v_{i-1})$ and $f(v_i) > f(v_{i+1})$ (all indices are mod $n$). A label $f(v_i) \in L$ if $f(v_i) < f(v_{i-1})$ and $f(v_i) < f(v_{i+1})$. Also, if $f(v_i) \notin H$ and $f(v_i) \notin L$ and $f(v_i) \neq m$ then $f(v_i) \in I$. Figure 8 shows a graceful labeling of a cycle $C_8$. For this labeling $m = 5$, $H = \{6, 7, 8\}$, $L = \{0, 1, 2\}$ and $I = \{3, 4\}$.

We further define $s_H$, $s_L$, and $s_I$ as the sums of elements in $H$, $L$ and $I$ respectively. We also use $h$ to denote $|H|$. If $f$ is a graceful labeling of $C_n$, then the labeling obtained by

$$n - f(v_1), n - f(v_2), ..., n - f(v_n)$$

is also a graceful labeling of $C_n$. We call it the complementary labeling $\overline{f}$.
We now present an overview of our results. These results have been published in [15]. Please refer to that publication for complete details and proofs.

**Theorem 3.2.2** For a given graceful labeling \( f \) of \( C_n \)

1. \( |H| + |L| + |I| = n \),
2. \( s_H + s_L + s_I + m = \frac{n(n+1)}{2} \)
3. \( |H| = |L| = \frac{n-|I|}{2} \)
4. \( s_H - s_L = \frac{n(n+1)}{4} \)
5. \( m = s_H - 3s_L - s_I \)

**Proof.** The proofs of (1) and (2) are straightforward.

Without loss of generality, suppose \( f(v_1) \) is a high label. Let \( i \) be the smallest index such that \( f(v_1) > f(v_2) > ... > f(v_i) < f(v_{i+1}) \). Such an \( i \) must exist since \( f(v_1) > f(v_2) \) and \( f(v_1) > f(v_n) \). Then \( f(v_i) \) is a low label. It follows that if we ignore the intermediate labels, then the high and low labels alternate on the cycle, so that \( |H| = |L| \). By using property (1), we see that \( |H| = |L| = \frac{n-|I|}{2} \). This proves (3). We next prove (4). Again assume without loss of generality that \( f(v_1) \) is a high label. Then the edge labels of the two edges containing \( v_1 \) are \( f(v_1) - f(v_2) \) and \( f(v_1) - f(v_n) \). Thus, in the expression for the sum of all the edge labels of the cycle, each high label appears twice with a + sign, each low label appears twice with a − sign, and the intermediate labels appear once with a + sign and once with a − sign. Hence this sum is \( 2s_H - 2s_L \). On the other hand, since the labeling is graceful, the sum of all the edge labels must be \( \frac{n(n+1)}{2} \). This proves (4). Finally, (5) follows from (2) and (4). ■

**Lemma 3.2.3** If \( f \) is a graceful labeling of \( C_n \), then

1. \( \frac{h(h-1)}{2} \leq s_L \leq \frac{h(2n-h+1)}{2} - \frac{n(n+1)}{4} \)
2. \[ \frac{h(h-1)}{2} + \frac{n(n+1)}{4} \leq s_H \leq \frac{h(2n-h+1)}{2} \]

**Theorem 3.2.4** For every graceful labeling of \( C_n \),

1. \( |I| \leq \sqrt{n+1} - 1 \)
2. \( |H| = |L| \geq \frac{n+1-\sqrt{n+1}}{2} \)

**Theorem 3.2.5** For every graceful labeling of \( C_n \),

1. \( \{0, 1, 2, ..., \left\lfloor \frac{n}{4} \right\rfloor - 1\} \subseteq L \)
2. \( \{\left\lfloor \frac{3n}{4}\right\rfloor + 1, \left\lfloor \frac{3n}{4}\right\rfloor + 2, ..., n\} \subseteq H \)

**Theorem 3.2.6** For every graceful labeling of \( C_n \),

\( \left\lceil \frac{n}{4} \right\rceil \leq m \leq \left\lfloor \frac{3n}{4} \right\rfloor \)

**Theorem 3.2.7** \( C_n \) has an \( \alpha \) valuation if and only if

1. \( n = 4t \)
2. \( m = t \) or \( m = 3t \)

In each case, it also follows that \( I = \Phi \).

**Theorem 3.2.8** Suppose \( n = 4t - 1 \) and \( C_n \) has a graceful labeling \( f \).

1. If \( m = t \), then \( I = \{2t\} \), \( L = \{0, 1, ..., 2t - 1\} \) and \( H = \{2t + 1, ..., 4t - 1\} \)
2. If \( m = 3t - 1 \), then \( I = \{2t - 1\} \), \( L = \{0, 1, ..., 2t - 2\} \), and \( H = \{2t, ..., 4t - 1\} - \{3t - 1\} \).

By using our system Manohar, we enumerated graceful labelings for cycle \( C_n \), with \( 0 \leq n \leq 20 \). Table 1 shows some results for each pair \( (m, n) \). Notice that the missing label \( m \) is always between \( \left\lceil \frac{n}{4} \right\rceil \) and \( \left\lfloor \frac{3n}{4} \right\rfloor \) as shown in Theorem 3.2.6.
Other approaches have also been used to determine if a graph is graceful. For instance, Eshghi and Azimi [27] used mathematical programming techniques to check for gracefulness of some classes of graphs. Their algorithm was used on cycles $C_n$ for certain values of $n$. Table 2 compares the performance of Eshghi-Azimi algorithm with ours. Eshghi and Azimi used a Pentium IV machine with 256 MB RAM and we used a Pentium IV 2GHZ PC with 512 MB RAM.

<table>
<thead>
<tr>
<th>$n$</th>
<th>Performance (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E &amp; A [27]</td>
</tr>
<tr>
<td>8</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>15</td>
<td>&lt; .05</td>
</tr>
<tr>
<td>20</td>
<td>&lt; 105.32</td>
</tr>
<tr>
<td>55</td>
<td>N/A</td>
</tr>
<tr>
<td>72</td>
<td>N/A</td>
</tr>
<tr>
<td>112</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 2: Performance comparison
3.3 Manohar

Manohar is a software system for computing and enumerating graceful labelings of certain graphs. We use the term computing to refer to the process of finding a graceful labeling for a given graph \( G \), while the term enumerating is used to describe the process of finding all graceful labelings of \( G \). Manohar performs graceful computation of graceful labelings of any tree \( T \) and enumeration of graceful labelings of path \( P_n \), cycle \( C_n \) and graph \( \text{SUN}(n,k) \).

In this section, we begin by describing Manohar’s graphical user interface, and then proceed to describe each of the graceful operations listed above. Implementation details and algorithms are provided at the end of this section.

3.3.1 Graphical User Interface

Manohar provides a Graphical User Interface (GUI) to create graphs and perform labeling computations. A graph is created by adding vertices and connecting them with edges. The main sections of the GUI are illustrated in figure 9. Notice that the graph displayed in the figure is a lobster with 18 vertices.

![Figure 9: Manohar’s GUI Sections](image)
The main screen area is used to draw the input graph. This graph can be created in several ways. The first one is by using the add vertex and add edge buttons together with the selection button. The second way is by clicking on the create path button. It is also possible to use a combination of both techniques. Suppose that one wants to create a caterpillar with a central path of 100 vertices and a pendant vertex attached to the middle vertex of the path. The fastest way to accomplish this would be to use the create path button with an input of 100, then create an extra vertex and connect it to the middle vertex of the previously generated path.

The buttons and the menu area are used to perform operations on the current graph as well as computing and enumerating graceful labelings. Figure 10 lists the buttons and their purpose.

<table>
<thead>
<tr>
<th>Button</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Open file" /></td>
<td>Open file</td>
<td>Retrieves a previously drawn graph by opening a file saved with Manohar</td>
</tr>
<tr>
<td><img src="image" alt="Save graph" /></td>
<td>Save graph</td>
<td>Saves the current graph to a file</td>
</tr>
<tr>
<td><img src="image" alt="Selection mode" /></td>
<td>Selection mode</td>
<td>Enables selection mode. This mode allows selection of vertices and modification of their location</td>
</tr>
<tr>
<td><img src="image" alt="Add vertex" /></td>
<td>Add vertex</td>
<td>Adds a vertex to the current graph</td>
</tr>
<tr>
<td><img src="image" alt="Add edge" /></td>
<td>Add edge</td>
<td>Adds an edge between vertices</td>
</tr>
<tr>
<td><img src="image" alt="Vertices in a circle" /></td>
<td>Vertices in a circle</td>
<td>Arranges the vertices in a circle</td>
</tr>
<tr>
<td><img src="image" alt="Zooms out" /></td>
<td>Zooms out</td>
<td>Decreases current zoom</td>
</tr>
<tr>
<td><img src="image" alt="Zooms in" /></td>
<td>Zooms in</td>
<td>Increases current zoom</td>
</tr>
<tr>
<td><img src="image" alt="Create Path" /></td>
<td>Create Path</td>
<td>Creates a path with the number of vertices specified by the user</td>
</tr>
<tr>
<td><img src="image" alt="Graceful Computation" /></td>
<td>Graceful Computation</td>
<td>Computes the graceful labelings of the current tree</td>
</tr>
<tr>
<td><img src="image" alt="Delete" /></td>
<td>Delete</td>
<td>Deletes the current graph</td>
</tr>
</tbody>
</table>

Figure 10: Manohar buttons

The graceful computation button computes the graceful labeling of the current input graph. It is important to note that Manohar requires the input graph to be a tree, otherwise a message informs the user that the current graph is not a tree. After clicking on the button, the computation of graceful labelings starts. If a graceful labeling is found, it is displayed in a window. This window contains an okay button to continue computing graceful labelings. Once all labelings found are displayed, the
computation stops. Notice that even though several labelings may be found, there may be other graceful labelings for the input tree that were not found. This is due to the fact that the operation performed is computation and not enumeration.

In order to perform enumeration, it is necessary to use the **algorithm** menu instead of using the graceful computation button. This menu allows the execution of enumeration algorithms for paths, cycles and suns as shown in Figure 11. When this menu is used, the graph drawn on the screen is not utilized.

![Algorithm menu](image)

**Figure 11: Algorithm menu**

To execute the algorithm that enumerates graceful labelings of path, one selects the menu item **enumerate path graceful labelings**. If the check box **only graceful** is unchecked, then not only the graceful labelings will be displayed but also the entire computation on how the labelings were obtained (this process is explained in later sections). Conversely, if this checkbox is checked, then only graceful labelings will be displayed. After selection of the **enumerate path graceful labelings** menu item, a window prompts the user for the number of vertices on the path. Manohar then proceeds to execute the algorithm and the results are saved on text files in the **output folder**. This output folder is a special folder where Manohar outputs all the results. For the case of the path enumeration algorithm, a new folder is created under the output folder. For a path where the highest label is 11 (and therefore 12 vertices $v_0, v_1, ..., v_{11}$), the folder would be called **PATH-n=11**. If the output is relatively small, only one file is created. However, for large values of $n$, many files may be generated. In this later case, each file is numbered in the order in which it is created. The only exception is the last file which is not numbered. Figure 12 shows a partial sample output for a path of 12 vertices (labels go from 0 to 11).
Figure 12: Path enumeration sample output

Notice that the number in square brackets to the right is the sequence number. In other words, it indicates the order in which each graceful labeling was generated. The total number of these labelings is the sequence number of the last labeling on the last file since that sequence number equals the total.

In a similar way, we can also execute algorithms for enumerating graceful labelings of cycles and suns. Recall that theorem 3.2.1 states that a cycle $C_n$ is graceful if and only if $n \equiv 0 \pmod{4}$ or $n \equiv 3 \pmod{4}$.

Figure 13: Cycle enumeration sample output

Notice that for a cycle $C$ with $n$ vertices, there are exactly $n$ edges. As there are $n + 1$ vertex labels, namely $0, 1, ..., n$, exactly one label will be missing. This fact is observed in the sample partial output for a cycle with 15 as the highest label as
shown in figure 13. In the graceful labeling listed at the top (labeling 4753), the label 5 is missing. The missing labels are displayed to the very right of the graceful labelings.

The last menu item is *enumerate sun graceful labelings*. By selecting this menu item, Manohar enumerates graceful labelings for a $SUN(n, k)$ where $n$ is the number of vertices in the cycle and $k$ is the number of rays or pendant vertices to each vertex in the cycle. After the user enters values for $n$ and $k$, the algorithm starts the enumeration process. For a value of $k = 0$, the output resembles that of a cycle since there are no rays. For values of $k > 0$, the ray labels are listed to the right of the cycle labeling separated by the $|$ character. This later scenario is observed in figure 14 where a sample partial output for a $SUN(5, 2)$ is displayed.

![Figure 14: Sun enumeration sample output](image)

To better understand this output, consider the graceful labeling 211 shown at the top. For this labeling, label 3 is missing. The labels in the cycle are 9, 14, 0, 15. Each vertex cycle has two pendant vertices since $k = 2$. The cycle vertex with label 9 has two ray vertices with labels 4 and 8 attached to it. In similar fashion, the cycle vertex labeled 1 has ray vertices with labels 12 and 13. Other cycle vertices also have their corresponding labels and ray vertices labels arranged in the same way.

### 3.3.2 Implementation details

Manohar has been developed using Visual Basic .Net and OpenGL as the graphics package. It uses several classes and data structures to handle manipulation of graphs and their operations. The main algorithms implemented are the ones that enumerate graceful labelings of paths, cycles and suns. These three algorithms share the same fundamentals to obtain graceful labelings. That is, start computing the highest edge label and continue the computation of edge labels in decreasing order following a
tree-like structure. A detailed description of this process is given in [14], where the algorithm for enumeration of graceful labelings of a cycle is presented. In this work, we will provide an overview of the algorithm for paths and a detailed description and proof of the algorithm for suns.

### 3.3.3 Enumeration of graceful labelings of paths

The enumeration of graceful labelings of paths algorithm finds graceful labelings of a path $P_n$ by computing sublabelings in a tree-like fashion. We define $f$ as a labeling of $P_n$ where $f = < a_1, a_2, ..., a_n >$ is an ordered sequence of vertex labels. A sublabeling is an ordered union of separate subsequences of $f$. We denote $S_k$ as the sublabeling obtained at step $k$, where $(1 \leq k \leq n)$. This sublabeling produces edge labels $k, k+1, ..., n$. Notice that for any graceful labeling $f$, $S_n = < 0, n >$ and $S_1 = f$. A level $L$ is the edge label needed to produce a sublabeling at a given step. Therefore, it is necessary to find two vertex labels $v_i$ and $v_j$ with $0 \leq v_i, v_j \leq n$ such that $|v_i - v_j| = L$.

The algorithm starts at level $L = n$. Notice that the only vertex labels $v_i$ and $v_j$ for which $|v_i - v_j| = 0$ are 0 and $n$. Therefore, $S_n = < 0, n >$. In the next step, $L = n - 1$, for this case, the vertex labels which can generate $n - 1$ are $n - 1, 0$ and $n, 1$. Since there are two possibilities, the algorithm splits into two execution branches. One branch (let’s call it A) will have sublabeling $< n - 1, 0, n >$ and the other branch (let’s call it B) will have sublabeling $< 0, n, 1 >$. Next, the algorithm continues for level $L = n - 2$ by generating subbranches of A and B. Figure 15 illustrates this process.

At a general step $k + 1$, the sublabeling $S_{k+1}$ generates edge labels $k + 1, k + 2, ..., n$. For step $k$, level $L = k$, is necessary to find two vertex labels $v_i$ and $v_j$ between 0 and $n$ such that $|v_i - v_j| = L$ to generate $S_k$. In order to use a vertex label, it has to be available. A vertex label is available if it is not in $S_{k+1}$ or if it is in an extreme vertex of $S_{k+1}$. To this respect, there are three cases to consider.
Case 1. \( v_i \) is an extreme vertex of \( S_{k-1} \), \( v_j \notin S_{k-1} \). Vertex label \( v_j \) is added adjacent to \( v_i \) to create \( S_k \).

Case 2. \( v_i \notin S_{k-1} \), \( v_j \notin S_{k-1} \). Vertex labels \( v_i \) and \( v_j \) are joined into subsequences \( <v_i, v_j> \) and \( <v_j, v_i> \). These two subsequences will be inserted next to each side of every subsequence of \( S_{k-1} \) creating a new execution branch each time.

Case 3. \( v_i \) and \( v_j \) are both extreme vertices of \( S_{k-1} \). In this case, if the labels \( v_i \) and \( v_j \) are adjacent ends of some \( <s_{r-1}> \) and \( s_r \), then \( <s_{r-1}> \) and \( s_r \) are merged into a single subsequence.

If vertex labels \( v_i \) and \( v_j \) cannot be found, then none of the above cases are possible and then execution branch dies unsuccessfully. However, the algorithm continues computing other execution branches.

The following example illustrates this. Consider a path \( P_{13} \) with vertices \( v_0, v_1, ..., v_{12} \). At a given step \( k+1 = 5 \) of the execution tree of the algorithm, the sublabeling \( S_5 = <4, 9, 2, 10, 1, 11, 0, 12, 6> \) generates the edge labels \( 12, 11, ..., 5 \). To proceed
with the next step \( k = 4 \) with the current level \( L = 4 \), it is necessary to find vertex labels \( v_i \) and \( v_j \) such that \( |v_i - v_j| = 4 \). Notice that the vertex label 8 is not in \( S_5 \) and that vertex label 4 is in an extreme vertex of \( S_5 \), therefore both vertex labels are available. This is in fact an example of case 1 and hence, a new vertex with label 8 is added adjacent to the one with label 4 to create \( S_4 \) and continue the execution on a new branch with \( S_4 =< 8, 4, 9, 2, 10, 1, 11, 0, 12, 6 > \).

On the other hand, notice that vertex labels 3 and 7 are not in \( S_5 \) which means that they are both available and also \( |3 - 7| = 4 \). This is case 2 and consequently these vertex labels are joined into subsequences \( < 3, 7 > \) and \( < 7, 3 > \) and inserted next to each side of every subsequence of \( S_5 \) to create a sublabeling \( S_4 \) for every new execution branch created. In this example, there will be several new branches with \( S_4 \) taking values \( S_4 =< 7, 3 > < 4, 9, 2, 10, 1, 11, 0, 12, 6 > \), \( S_4 =< 3, 7 > < 4, 9, 2, 10, 1, 11, 0, 12, 6 > \), \( S_4 =< 4, 9, 2, 10, 1, 11, 0, 12, 6 > < 3, 7 > \) and \( S_4 =< 4, 9, 2, 10, 1, 11, 0, 12, 6 > < 7, 3 > \). The algorithm continues computing execution branches for \( L < 4 \) until either \( L = 1 \) and therefore a graceful labeling is found or it is not possible to find vertex labels \( v_i \) and \( v_j \) and the branch dies unsuccessfully while the algorithm continues computing the remaining execution branches.

**Complexity**

The complexity of this algorithm is bounded by the number of graceful labelings of paths. The growth of this number has been studied by enumerating graceful permutations. Klove [41] showed that this growth is exponential. Aldred et al. [4] found that the number of graceful labelings for \( P_n \) is bounded below by \( (\frac{5}{3})^n \). This bound was later improved by Adamaszek [2] to \( (2.37)^n \).

**3.3.4 Enumeration of graceful labelings of suns**

In this section, we present an algorithm for enumerating graceful labelings of a Sun\((n,k)\) with \( n \) vertices in the cycle and \( k \) vertices pendant to each cycle vertex. This is a generalization of the algorithm presented in 3.3.3.
Our algorithm finds graceful labelings of $Sun(n,k)$ by generating edge labels as it traverses a tree. At the root of this tree, there are three branches. The top of each branch is level $t$, which is the highest edge label and therefore $t = n(k+1)$. Successive edge labels in decreasing order are generated at different levels of the tree as shown in figure 16. Note that the edge label $t$ can only be generated from the vertex labels 0 and $t$ and therefore, these vertex labels are adjacent. For levels $k < t$, there are several possibilities for edge label $k$.

**Description**

Given a $Sun(n,k)$ with $n$ vertices in the cycle, $k$ vertices pendant to each cycle vertex and $t = n(k+1)$ the highest label; the algorithm starts the computation at level $L = t$, where level indicates that it is necessary to find a sublabeling containing two labels $v_i$ and $v_j$ such that $|v_i - v_j| = L$. At level $L = t$ there exists only two labels, namely 0 and $t$ and hence the starting sublabeling consists of two adjacent vertices with these labels. The next step is to find sublabelings for $L = t - 1$. The two alternatives available consist of adding the label $t - 1$ adjacent to 0 or the label 1 adjacent to $t$. For each case the algorithm splits into a new computation branch. The algorithm continues computing for $L = t - 2$ by splitting into several computation branches each time and recursively calling each branch for other values of $L$. The computation for a particular branch continues until a graceful labeling is found or no graceful labeling is possible for the current branch. Figure 16 illustrates the execution tree.

To describe the general step of our algorithm, we define $c_1, c_2, ..., c_n$ as vertex labels in the cycle of $Sun(n,k)$ and $r_{ij}$ where $1 \leq j \leq k$ as the $j-th$ pendant vertex label adjacent to $c_i$. Similarly, $\overline{r}_i = \{r_{i1}, r_{i2}, ..., r_{ik}\}$ is the set of of pendant vertex labels adjacent to $c_i$.

A given labeling $f = (c_1, \overline{r}_i), ..., (c_n, \overline{r}_n)$ of $Sun(n,k)$ can be considered an ordered sequence of labels. A sublabeling is an ordered union of disjoint subsequences
of $f$. Let $t = n(k + 1)$ be the highest label, we denote by $S_q$ with $1 \leq q \leq t$ a sublabeling containing edge labels $q, q + 1, ..., t$.

For instance, for the graceful labeling:

$$f = < (3, 10), (5, 9), (13, 4), (2, 8), (14, 0), (1, 11), (6, 7) >$$

of $Sun(7, 1)$ displayed in Figure 17 at the top, the sublabeling $S_7$ of $f$ displayed at the bottom, consists of $S_7 = < (3, (10)) > < (5, (−)), (13, (4)), (2, (−)), (14, (0)), (1, (11)) >$ and therefore $S_7$ is the ordered union of two subsequences. Note that $f = S_1$. 

$$f = < (3, 10), (5, 9), (13, 4), (2, 8), (14, 0), (1, 11), (6, 7) >$$

$$S_7 = < (3, (10)) > < (5, (−)), (13, (4)), (2, (−)), (14, (0)), (1, (11)) >$$
General Step

As our algorithm computes graceful labelings by traversing a tree and splitting into computation branches, it is essential to determine at each step how to find the next edge label. We proceed to explain how this process is achieved.

For a given level $l$, denote the previous sublabeling $S_{l+1}, S_{l+2}, ..., S_{n(1+k)}$. At every step, it is necessary to find two vertex labels $v_i$ and $v_j$ such that $|v_i - v_j| = l$. There are three cases to consider.

(i) $v_i \in S_{l+1}$ and $v_j \in S_{l+1}$. Subsequences are merged into a single subsequence.

Note that in figure 18, $v_i$ and $V_j$ are at the extremity of their corresponding subsequences before merging.

(ii) $v_i \notin S_{l+1}$ and $v_j \notin S_{l+1}$. In this instance, $v_i$ and $v_j$ are added adjacent to each other. Figure 19 illustrates all possible scenarios. Observe that in the sublabelings displayed on the left, both $v_i$ and $v_j$ are added in the cycle while in the sublabelings shown on the right, $v_i$ is added in the cycle and $v_j$ as ray (top) or $v_j$ in the cycle and $v_i$ as ray (bottom).

(iii) $v_i \in S_{l+1}$ and $v_j \notin S_{l+1}$. Then $v_j$ is added adjacent to $v_i$. Notice that in Figure 20, $v_j$ can be added as a ray of $v_i$ (right) or as neighbor in the cycle (left).

If none of these cases occur at a given branch, then that computation branch dies.
CHAPTER 3. GRAPH ALGORITHMS AND THEIR APPLICATIONS

Figure 19: Case ii

Proof of Correctness

Suppose $f = \langle (c_1, (r_1)), \ldots, (c_n, (r_n)) \rangle$ is a graceful labeling of $Sun(n,k)$. The algorithm achieves $f$ exactly once.

Proof: Let $t = n(k + 1)$ be the highest vertex label. The proof is by induction on the edge labels $q = t, t-1, \ldots, 2, 1$.

Base Case: $S_t = \langle (0, (t)) \rangle$ or $S_t = \langle (t, (0)) \rangle$ or $S_t = \langle (0, (-)), (t, (-)) \rangle$ since for any $f$, the edge label $t$ can only be generated by taking 0 and $t$ together. The algorithm does this at the very first step by creating independent execution branches for each instance. For $t-1$, six branches are created as illustrated in Figure 16.

Notice that on each branch, either vertex label $t-1$ or 1 was added adjacent to 0 or $t$ respectively to obtain edge label $t-1$. The algorithm considers all these cases and hence it achieves $S_{t-1}$. Also, since $f$ is given exactly one of these alternatives
will match with $S_{t-1}$ since they are distinct options. This indicates that at edge label $t - 1$ the algorithm achieves $S_{t-1}$ uniquely.

Now, for $1 \leq l \leq t$ assume the algorithm achieves $S_{l+1}$ uniquely. Suppose that in $S_l$ the edge label $l$ is obtained by vertex labels $v_i$ and $v_j$ in $f$, so that $|v_i - v_j| = l$. We consider $S_l$ an order union of subsequences. There are three cases:

(i) $l$ is an interior edge of the cycle.

(ii) $l$ is the only edge of a subsequence.

(iii) $l$ is a pendant edge of a subsequence either as a ray or in the cycle.

As $f$ is given, precisely one of these will match $S_l$. Since the algorithm achieves $S_{l+1}$ uniquely, exactly one of the three cases described in section 3.3.4 applies to achieve $S_l$. Since $S_1 = f$ the theorem follows by induction.

Complexity

The complexity of this algorithm is bounded by the number of graceful labelings of the graph $SUN(n, k)$. Notice that the graph $SUN(n, 0)$ is in fact the cycle $C_n$. We denote the total number of graceful labelings of $SUN(n, k)$ by $L_{n,k}$. Then $L_{n,0}$ is the number of graceful labelings of $C_n$. This number is a lower bound for $L_{n,i}$ since $L_{n,i}$
≤ L_{n,i+k} for all i, k ≥ 0. In section 3.3.3 we presented some results indicating that
the number of graceful labelings of $P_n$ is bounded below by $(2.37)^n$. Based on the
structure of the tree generated by our algorithm, it appears that $C_n$ may have similar
bounds.

3.4 Colossus

Colossus is a system for drawing polygons and computing their visibility graph. The
system provides a Graphical User Interface where the user can easily create and
manipulate a polygon. After the polygon is defined, Colossus allows the real-time
computation of its visibility graph by displaying visibility edges and updating the
graph every time the user changes the position of any of the points of the polygon.

Along with the visibility graph, Colossus also colors the points to indicate ears and
mouths. The total number of each type of point is displayed in the messages
section.

3.4.1 Graphical User Interface

Colossus provides a Graphical User Interface that allows the user to draw a polygon.
This polygon is drawn by defining its point locations or endpoints.

As shown in figure 21, endpoints are displayed in several colors depending on
their current status. In this example, endpoints 1, 3, 4, 6, 8, 10 and 13 are ears; while
endpoints 2, 7, 9, 11 and 14 are mouths. Remaining endpoints do not fit in any of the
previous categories.

Colossus's main window is divided into several areas. These areas are: the main
screen, messages, buttons and menu.
The **main screen** area is where the user draws the polygon. The input polygon is created by using either the menu or the buttons.

The **messages** area displays important information about the structure of the polygon and its visibility graph.

The **buttons** and the **menu** area are used to perform operations on the current polygon as well as other activities. Figure 22 lists all the buttons with their respective functions.

![Colossus screenshot](image)

**Figure 21: Colossus screenshot**

One way to create a new polygon is by clicking the **add endpoint** button. This
button enables the insertion of endpoints so that every click on the main screen creates a new endpoint. Once all the endpoints have been inserted, it is necessary to specify the sides (or edges) of the polygon. The **add edges** button allows joining endpoints by edges. This operation is performed by clicking on the first endpoint followed by a click on the second endpoint. After these clicks, an edge will be drawn between the pair of endpoints. It is important to realize that in order to have a valid polygon, the figure must be closed, and its sides should never cross.

Another way to create a polygon is by clicking on the **create polygon** button. This button automatically creates a polygon based on the number of endpoints specified. Figure 23 shows a sample output for a polygon with 20 endpoints.

![Figure 23: Resulting effect after clicking on create polygon button for a polygon of 20 endpoints](image)

Clicking the **enable/disable visibility graph** button displays the visibility graph. Moreover, it is possible to move endpoints around by clicking on the **selection mode** button and then clicking on the target endpoint. As the computation for the visibility graph is performed real-time, any changes to the position of the endpoints of the polygon will be immediately reflected on its visibility graph. This
CHAPTER 3. GRAPH ALGORITHMS AND THEIR APPLICATIONS

operation is shown in figure 24

Figure 24: Moving endpoints when visibility mode is enabled

One key feature of Colossus is its ability to work with orthogonal polygons. This is easily accomplished by enabling the orthogonal mode from the Drawing menu. In this mode, endpoints can only be added in such a way that the orthogonality of the polygon is always preserved. Once the creation of the polygon is completed, it is possible to enable the visibility mode to see the resulting visibility graph. Figure 25 illustrates this.

Figure 25: Visibility in orthogonal mode
3.4.2 Implementation

Colossus has been entirely developed under Visual Basic .Net using the graphics library OpenGL to implement the drawing of polygons and visibility graphs. Colossus relies on several classes to perform graph operations. The JGraph.vb class contains data structures for storing graphs. These data structures rely on the Vertex.vb and Edge.vb classes which handle vertices and edges respectively. Moreover, the JGraph.vb class provides methods for manipulating the graph and performing graph operations. The Visibility.vb class is the core of Colossus. This class contains data structures and methods to compute real-time visibility of a given polygon as well as ears and mouths.

3.5 Graph Algorithm Constructor

Our ultimate goal is to write a software system that allows the creation of graph algorithms without the need to write computer code. Such system would allow individuals with knowledge of graph theory to implement graph algorithms independently of their programming expertise. Our software system Graph Algorithm Constructor serves this purpose. It provides a graphical user interface where users can create their own algorithms by using flow diagrams. Moreover, these algorithms can be executed and even used in the implementation of other algorithms. As this software is still under development, we will examine its current components and describe the future directions.

3.5.1 Graphical user interface

The Graph Algorithm Constructor’s graphical user interface (GUI) is divided into two windows. These windows are the main window and the runner window.
CHAPTER 3. GRAPH ALGORITHMS AND THEIR APPLICATIONS

The main window provides an area where users can implement their own algorithms by creating a flow diagram. A toolbar contains a list of all the elements available. An example screenshot is displayed in figure 26.

Figure 26: Graph algorithm constructor’s main window

For the current figure, the toolbar is displayed to the right of the main window, but its position can be adjusted by the user. The main window displays a sample flow diagram of a hypothetical algorithm. The elements available are divided into algorithms, operations, control structures and other elements. An algorithm is a flowchart that has been previously created with the constructor and it can now be used to implement new algorithms. It is important to note that most algorithms will have an input and an output and this must be considered during implementation. An operation is a task that it is usually performed on vertices or edges of a graph. Examples of operations are “color a vertex”, “color an edge”, “find neighbors of a particular vertex”, etc. Control structures manage the flow of operations. For instance, the loop structure repeats a series of operations until the finalize condition becomes true. Other elements include graphs, trees, vertices, edges, etc.

The runner window is used to execute algorithms. It is also based on a graphical user interface divided into two areas. The first area (on the left) is where the user enters the input required by the algorithm. This input is usually a graph, a tree, a vertex or a even a combination of elements such as a graph and a selected vertex. The second area displays an XML tree with information about the algorithm and
its input. Figure 27 shows an screenshot of the runner window for a simple sample algorithm that colors some vertices of an input graph.

![Graph algorithm constructor’s runner window](image)

**Figure 27: Graph algorithm constructor’s runner window**

### 3.5.2 Implementation details

Graph algorithm constructor has been entirely implemented in Visual Basic .Net and using the graphics library OpenGL to handle drawing of graphs and flow diagrams. Algorithms are saved on XML files. The system relies on several custom classes to manipulate graphs as well as to create, modify and execute algorithms.

### 3.5.3 Future directions

At the present time, our system Graph Algorithm Constructor is under development and can only execute very simple custom algorithms. The finished product should be flexible enough to allow the creation of a wide range of algorithms, save them in XML format and be able to execute them in the runner window. For instance, most algorithms receive an input, process it and return an output. This input may consist on a graph (e.g. planarity drawing), or a graph and two selected vertices (e.g. Dijkstra shortest path), or even a sequence of numbers (e.g. Prüfer encoding). Moreover, previously created algorithms may be used in new ones. For instance, suppose algorithm \( A \) has been already implemented. This algorithm expects input \( x \) and after processing it, returns \( y \). Next, algorithm \( B \) is created. This algorithm
expects \( p \) and returns \( q \). However, a part of algorithm \( B \) uses algorithm \( A \). In this case, \( B \) should make sure that \( A \) always receives the input \( x \) since this is the only valid input. At the same time, it should know what values of output \( y \) are expected to be returned by \( A \).

### 3.6 JEdit

The software system JEdit is an application for creating graphs and running graph algorithms. The development of JEdit started in the early '90s with the creation of the G-Net team, a research group consisting of Ball State students lead by Professor Jay Bagga. The G-Net website [6] maintains information about current events in the GNet group and a downloadable version of JEdit.

The two most complex algorithms available in JEdit are **planarity testing** and **planarity drawing**. These two algorithms were implemented as part of JEdit v4.0, which was the topic of the author’s M.S. Thesis [33]. This system was accepted for presentation at the prestigious Graph Drawing 2001 conference held in Vienna, Austria [9]. The system was displayed along other state-of-the-art graph drawing software [64].

#### 3.6.1 Graph algorithms

JEdit contains a library of well-known graph algorithms such as **Dijkstra shortest path**, **Prim minimum spanning tree**, **Prüfer encoding**, **Component search**, **Block finding**, etc. Some of these algorithms were created by members of the G-Net team and others were given as assignments for students in graph theory class since algorithms can be easily created and added into JEdit. Figure 28 displays the execution of the Prim’s minimum spanning tree algorithm.

Algorithms are classified as **automatic**, **one-click**, **two-click** or **text**. The Prim’s minimum spanning tree algorithm is an example of a one-click algorithm since the user is asked to select a vertex before execution. The algorithm starts computation of
the tree from the vertex chosen showing each step by using animation. An automatic algorithm executes directly on the current graph without any other parameters. The planarity testing and planarity drawing algorithms are examples of automatic algorithms. Notice in figure 29 that the graph displayed is drawn with multiple crossings.

The planarity checking algorithm determines if the input graph can be embedded on a grid with no crossings. If this is possible, planarity drawing proceeds to draw the graph with no crossings by embedding it on a grid. This final result is shown in figure 30. The execution is animated so that one vertex at a time is placed on the grid until they are all arranged. In this way, it is easy to see the steps involved in the execution.

3.6.2 Recent features

As the development of JEdit is an ongoing process, several features have been added by other students taking JEdit to v5.0. One of these features include the use of GraphML, a standard XML-based format for graphs. By using this format, a graph
CHAPTER 3. GRAPH ALGORITHMS AND THEIR APPLICATIONS

Figure 29: Before execution of planarity drawing

Figure 30: After execution of planarity drawing
created in JEdit can be used in other software systems and vice versa. Other features include: selection of a component, shrink/enlarge a component and others. For the most up-to-date information about JEdit, please visit the G-Net website [6]. This site is maintained by Professor Jay Bagga and some of his graduate students including the author.
Chapter 4

E-learning

The book by Clark et al [21] describes E-learning as a method of teaching where the instructional material is delivered on a computer. Some characteristics include: content relevant to the learning objectives, use of examples and activities to assist in the learning process, use of media such as pictures, sounds, words to present content, creation of knowledge and skills related to learning objectives as well as assessment and tests. Synchronous E-learning is instructor-led, while asynchronous E-learning is intended for individual study.

In his book, Allen [5] lists numerous benefits of E-learning. In the traditional classroom instructional setting, the teacher only has limited time to explain/discuss the material and some students may need additional time and practice to understand. E-learning solves this problem by adapting to learner necessities. In this respect, if a topic is already known, it can be skipped, otherwise additional exercises may be done. Another benefit is that it gives in-depth learning experiences to every learner and not just selected ones or those volunteering. It also encourages students to participate by decreasing the fear of making mistakes since the learner will not be embarrassed in front of a class. In addition, it provides immediate feedback to learner and assessment statistics to teachers. Finally, it allows a flexible schedule and in the case of online E-learning, it can be accessed from anywhere in the world where there is a computer with an internet connection available.
We have made several contributions to E-learning through our work on a Department of Energy funded research SLP (Science Literacy Project). In this chapter we present our SLP work. We begin with a description and the learning objectives. We also discuss the development process, project team and E-learning game modules created. We conclude by providing testing and assessment details.

4.1 SLP

The Science Literacy Project (SLP) is a research project funded by a grant from the Department of Energy. This project consists in the development of educational video games to teach topics in science and mathematics to middle school students. These games were developed to study the impact that educational video games have in the learning of science as well as to find out if they are a valuable tool to help eliminate common misconceptions.

The overall project was led by Dr. Bizhan Nasseh, former Assistant Vice-president for Information Technology at Ball State University. The research team consisted of Dr. Laurie Mullen, Associate Dean of Teachers College, Dr. Lisa Huffman, Chair of the Department of Educational Psychology and Dr. James Jones, Assistant Director of research and design. The teachers/subject matter experts were Linda Draper from Alexandria School, Mike Dodrill from Burris Laboratory School, Dr. Jeff Smith and Kim Foltz from the Indiana Academy for Mathematics, Science and Humanities at Ball State University, and Deb Smith from Yorktown Middle School. The game development team was divided into two groups. The first group was made of developers from a professional web design firm called Outside Source Design [25]. This team was in charge of the development of five games. The second group consisted of a faculty member and students from Ball State University. Dr. Jay Bagga, Professor of Computer Science was the faculty in charge of the group and this author served as the lead developer of five graduate and undergraduate students from arts, music, education and computer science. Our team was assigned the development of three
4.2 Implementation Details

As a result of the development process, we created three educational games. These were Pirate Math, Chemistry Circus, and Cellular Divide and Conquer in the areas of mathematics, chemistry and biology respectively. The games track metadata with performance statistics for each student to help researchers determine their effectiveness in the learning process.

The games were implemented in Adobe Flash version 8. We also implemented a System Interface in ASP.Net to communicate with an SQL Server database for metadata tracking. Dr. Jay Bagga was in charge of the overall development process, two undergraduate computer science students and this author wrote all the functionality, the two Art students created graphics and simple animations. The music student recorded sounds and composed music, and the education student provided support by managing the team meetings and gathering resources on the web. The entire source code and executable files are found in Appendix E.

4.2.1 Software Development Process

The traditional waterfall development process consists of several stages. Poppendieck [53] mentions the following: system requirements, software requirements, preliminary program design, analysis, program design, coding/development, testing and operations. In real life, this process may not be strictly followed since some stages may overlap or may not be needed. We now describe the development process during our work on SLP.

The first phase was requirement analysis. This phase started with preliminaries meetings in October 2006 to determine the scope of the overall project. These
meetings were held at Outside Source Design (OSD) facilities and were led by its vice-president Mike Peck and Project Leader Stuart Sipahigil. Attendees from Ball State University (BSU) included Dr. Bizhan Nasseh as Project Director; Dr. Laurie Mullen, Dr. Lisa Huffman and Dr. James Jones representing the research team; Dr. Jay Bagga and Adrian Heinz representing BSU’s development team; and Dr. Jeff Smith, Kim Foltz, Mike Dodrill, Linda Draper and Deb Smith as domain experts. Meetings continued in January 2007, and resulted in the conception of requirement specifications describing overall games idea, objectives, misconceptions targeted, story line, metadata tracking, development assignments and deadlines.

Once the requirement specifications were completed, the BSU’s development team started working on three web-based games in the areas of mathematics, chemistry and biology in that order. The design phase consisted on the creation of a design document, which described the main aspects of the game by providing specific details, examples and screenshots. Once this document was completed, it was sent to the Project Director, research team and domain experts for revision and approval.

The approval of the design document marked the beginning of the Implementation Phase. This phase is where coding took place. The games were implemented using AdobeFlash® version 8 and its scripting language ActionScript® version 2. Developers worked on the game functionality by integrating the graphics, animations, music and sounds created by the Art and Music students. Once a prototype was ready, it was presented to the Project Director, research team and domain experts for revisions. These revisions were later studied by the development team and incorporated into the games when appropriate.

Once revisions for all games were completed, the Testing Phase began. Testing was conducted by members of the research team, domain experts and developers with over 200 students from four schools. During this phase, we were able to assess the overall performance of our games using the hardware configuration available in
schools and their network connection. We also investigated metadata tracking by reviewing the database while students were playing. The results from the tests allowed us to identify a number of issues that we then corrected. For instance, the Flash executable file of our mathematics game *Pirate Math* was quite large at about 10MB since it incorporated several music files. As Internet connection speeds at schools were usually much lower than those at Ball State, this game would take a long time to download. We solved this problem by creating a light version with only sounds and no music.

OSD developed the other five games, implemented an SQL-Server Database and created a library of classes in Action Script to communicate from the Adobe Flash games to the database.

### 4.2.2 Author’s Contributions

As lead developer, the author coordinated the entire BSU development team with assistance from Dr. Jay Bagga. This included regular attendance to meetings with the Project Director, researchers, domain experts and OSD staff as well as meetings within the development group. Weekly meetings with Dr. Bagga were used to report progress and organize activities for the week.

Contributions in *Pirate Math* include the implementation of the introductory screen detailing the story line, the *Coin Toss* and *Pirate Store* modules, Mathematical Equation and Word Problem classes and metadata tracking. Also, the author assisted in the implementation of *Treasure Map* and fixed errors across the entire game.

In *Chemistry Circus*, the author created the introduction, help and menu screens, scoreboard, chemical equation class, metadata tracking and the boxes below the acrobat’s bar showing number of atoms.
CHAPTER 4. E-LEARNING

Another contribution was the creation of a web-based System Interface [60] which provides access to the games, metadata retrieval and users’ maintenance area. Only registered users can enter the system since a logon screen prevents unauthorized access. Each user is assigned a role. This role can be administrator, teacher, researcher or student. Users in the first three roles are allowed to play games as well as to access other areas. It is important to note that games track metadata for only users with a student role. Such users are only allowed to play games; they have no access to the metadata maintenance area.

4.2.3 Introduction to Adobe Flash and ActionScript

In this section, we provide an overview of Adobe Flash along with its scripting language ActionScript. This is intended to be a brief introduction and not a comprehensive tutorial. For the latter, we refer the interested reader to the book by Yeung [67] which is the source of this discussion.

Adobe Flash is a tool for developing visually responsive web content and interactive interfaces across platforms and devices. It provides an abundant programming environment that allows to create data-rich Internet applications, as well as to incorporate professional video editing tools to produce a superior-quality video experience.

A new document is created by using the document window. This window contains several components as illustrated in figure 31. The timeline controls the animation by specifying the static and moving elements of the project file. The stage is where content appears. It is the visible area of the project. The work area is the area around the stage. Nothing in this area is visible to the end user after the movie has been published. The edit bar shows the current location inside the project. The toolbar has tools that are used to create and edit artwork. The panels are windows containing tools and information to facilitate the work process.

A useful technique is the use of symbols. A symbol is an object which allows
the creation of any number of instances. An instance is a copy linked to the original symbol. When a new symbol is created, it is incorporated into the project’s library for future use. In this way, a symbol containing a drawing of a house can be immediately incorporated into the project by dragging and dropping from the library and thus, avoiding having to redraw the same house every time. There are three types of symbols, a movie clip, a button and a graphics symbol. Movie clips are arguably the most powerful of the three since they can incorporate animations that operate independently of the main timeline. A button symbol contains four states: Up, Over, Down and Hit. The Up state is how the button looks before the user interacts with it, the Over state represents how the button appears when the mouse is over it, the Down defines the look of the button when it is clicked and finally the hit state defines the area that is reactive to the mouse. Any number of graphics and events can be incorporated into these states. A graphics symbol permits the inclusion of graphics but it does not support animation. In this work, we use the naming convention “mc” for movie clips, “b” for button and “g” for graphics. Following this convention the symbols mcSurfBoard, bAccept and gTree would correspond to a movie clip, a button and a graphic respectively.
Even though the use of symbols is recommended, it is sometimes useful to use simple elements available in the toolbar by dragging and dropping. Examples of such elements include line, rectangle, curve, regular polygon and text tool. A text tool can contain text. This text can be fixed in the case of static text or variable for dynamic text. Dynamic text can be modified at run time.

Users create animations by using the timeline which contains animation frames. By default, frames are executed sequentially. The timeline also provides layers which are used to organize the elements of the animation. Elements of a layer appear in front or behind elements of other layers depending on the relative position of the layer. For instance, elements of the top layer always appear at front while elements of the bottom layer always appear in the back. The frame rate is the number of frames per second at which the movie plays.

Adobe Flash uses a scripting language called ActionScript. We present an overview of this language based on the books by Lott [46] and Moock [48]. For a comprehensive tutorial, the reader is referred to those sources since this discussion only serves as a brief introduction.

ActionScript is used for several purposes such as the real-time creation of instances of symbols, manipulation of objects, control of the timeline, mathematical computations, actions, events handling and many others. Its syntax is similar to the one used in high level programming languages of the family of Java and C.

ActionScript code is linked to a particular frame by selecting the desired frame and then invoking the scripting window. It is also possible to link the code to a specific event of a symbol. We do this latter by placing the code within an event handler. For instance, suppose that we declare an instance of a button symbol named bPrintMessage. We could link scripting code to the release event by using the following code:
bPrintMessage.onRelease = function():Void {
    trace("Hello world!");
};

In this code, the name of the event is `onRelease` and it is preceded by the name of the target symbol which in this case is `bPrintMessage`. The trace command prints the message “Hello world!”. We could also include any number of statements to perform a variety of actions and even modify properties of other symbols. To illustrate this, consider a movie clip of a ball that is instantiated under the name `mcBall1` and an instance of a button called `bMyButton1` as shown in figure 32.

![Figure 32: A movie clip of ball and a button](image)

Suppose that we want to shrink the ball size by half when the user clicks `bMyButton1`. This is accomplished by modifying the `onRelease` event of the button using the following code:

```actionscript
bMyButton1.onRelease = function():Void {
    mcBall1._width *= .5;
    mcBall1._height *= .5;
}
```

When the user plays the movie, it will show the ball in its original state. However, pressing the button will shrink the ball to half of its size as illustrated in figure 33.

Actionscript code can also be created outside the Flash movie. This is accomplished by creating a class file with the `.as` extension. This file can be referenced in
CHAPTER 4. E-LEARNING

Figure 33: Ball after button is clicked

the movie by creating an instance of the class.

Armed with this preliminary knowledge of Adobe Flash and JavaScript, we present a sample Flash movie from our Pirate Math game called Pirate Store. This module relies on the WordProblem.as class to generate random word problems.

4.2.4 Sample Flash Movie - Pirate Store

Pirate Store is one of the modules of our Pirate Math Game. For a game play description of this module, refer to section 4.4.6. In this section we provide an overview of its implementation. For an introduction to Adobe Flash and ActionScript, please study section 4.2.3. For the entire ActionScript code of this module, please see appendix A.

This module uses several symbols including movie clips, buttons and graphics. These symbols are stored in the library and therefore, they are available for instantiation. A screenshot of the development screen is shown in figure 34.

As the figure shows, the library contains a large number of symbols. Some of these symbols are organized by folders as in accessories, Animals, background and others. To create an instance of any of these symbols, one needs to drag and drop into the work area. In the case of movies clips and buttons, it is necessary to give a name to the instance so that it can be referenced from ActionScript code. The top left area displays several layers that are used to organize the objects. In this way, some objects appear on top of others depending on the layer that they belong. Layers can also
be organized into folders. In the figure, the Store folder contains layers ActionScript, bUserAnswer, Buttons and others. The animation timeline displayed to the right of the layers section has only one active frame. This is because most of the animation is handled directly from ActionScript code.

The stage contains several objects. The textTool objects include tDialogText, tUserInput, tCurrentCredit and tEquationText. Since these text objects are dynamic, their content is controlled from the ActionScript code. Several graphic symbols such as pants, swords, gloves, jacket, hook, store clerk, parrot and others are displayed. These symbols are not referenced anywhere in the code since they are only used to provide a realistic pirate-like environment. Also, notice the movie clip above the store clerk. This movie clip is a window with a view of a sunset in the ocean. During gameplay, it shows a pirate ship sailing in the sea. This animation was incorporated at design time into the movie clip and therefore, its execution is independent from the main execution timeline.
The bMusic1 button switches the music on and off by calling the PlayBackground-Music function. Since there are several songs in the library, this function plays the next song in the sequence by creating an instance of a sound object at real-time and linking it to the selected song. The variable bPlayingMusic is a boolean variable used to determine whether music is currently playing. This variable is used in onPress event of the bMusic1 button to avoid playing a song if one is already playing. The onSoundComplete event is fired when the song has finished. The following is the ActionScript code:

```actionscript
bMusic1.onPress = function():Void {
    if (!bPlayingMusic) {
        PlayBackgroundMusic();
    }
};

function PlayBackgroundMusic():Void {
    BackgroundMusic = new Sound(this);
    if (nBackgroundMusicNumber<(nNumberOfSongs-1)) {
        nBackgroundMusicNumber++;
    } else {
        nBackgroundMusicNumber = 0;
    }
    BackgroundMusic.attachSound("PirateSong"+nBackgroundMusicNumber.toString());
    BackgroundMusic.onSoundComplete = function():Void {
        bPlayingMusic = false;
    };
    BackgroundMusic.start(0);
    bPlayingMusic = true;
}
```

The button bPirateStoreHelp displays a help screen by displaying a movie clip created at design time. The bExit button stops the music and any other activity, releases resources and exits the module.

When the module starts, it displays the main screen and executes ActionScript code. This ActionScript code begins by initializing several variables and objects that will later be used by functions and events. One such variables is eMyProblem which contains an instance of the WordProblem class described in section 4.2.5. The declaration is as follows:

```actionscript
var eMyProblem:WordProblem = new WordProblem();
```

This class creates a word problem based on a randomly generated equation. After the appropriate parameters of the class and methods for generation of the problem
have been called, the word problem is displayed in tDialogText1. The bVerify1 button checks whether the user’s solution is correct by calling the IsSolution method of the WordProblem class with the text in tUserInput as parameter. If the solution is correct, it increases several variables that keep track of the number of correct answers and calls function PlayRightAnswer which displays congratulation messages and plays a cheerful song. Otherwise, it proceeds to give hints by partially displaying the equation of the problem in tEquationText each time. When the full equation is displayed, a wrong answer leads to incrementing the variables that keep track of incorrect answers and a sad song is played. After each answer the score is updated. Notice that even though the score is not displayed during game play, the module still keeps track of it for metadata tracking purposes. Here is the source code:

```actionscript
bVerify1.onPress = function():void {
    VerifyAnswer();
};

function VerifyAnswer():void {
    if (eMyProblem.IsSolution(tUserInput.text)) {
        nTotalRightAnswers++;
        nTotalAnswers++;
        PlayRightAnswer(0);
    } else {
        nAttempts++;
        if (nAttempts<3) {
            GiveHint();
        } else {
            nTotalWrongAnswers++;
            nTotalAnswers++;
            PlayWrongAnswer(0);
        }
    }
    UpdateScore();
}
```

Along with the above activities, this module performs a variety of tasks such as metadata tracking, enabling/disabling buttons, calling other modules, etc. For the complete ActionScript code refer to appendix A.

### 4.2.5 Sample ActionScript Class - WordProblem.as

We now analyze the ActionScript code used in the WordProblem class, which is used by the Pirate Store module of Pirate Math to create mathematical word problems. As these problems are entirely random, a new problem is generated on every execution.
It is extremely unlikely that a problem repeats. Also, problems use humorous pirate vocabulary to engage students.

The first step in the problem generation is the initialization of data structures containing the words to be used. These structures include several arrays with pirate names, introductory phrases, mathematical operation words, verbs, relationships and items. The next step is the specification of an answer range and generation of a random equation. For instance, if the answer range is \([0, 100]\) then the solution of the equation will be between 0 and 100. Equations can either be of difficulty 0 or 1. An equation of difficulty 0 is a simple equation with \(X\) on only one side while equations of difficulty 1 contain \(X\) on both sides. The equation is then used in the creation of the word problem. In this process, the numbers and mathematical operations are converted to words.

To illustrate this process, consider the mathematical equation: \(10X - 1 = 109\). This is a type 0 equation since \(X\) appears on only one side. The mathematical operations are multiplication as in \(10X\) and subtraction as in \(10X - 1\). The price of a randomly chosen item is 109. This price will be used to compare to the price of another item, which corresponds to the solution of the equation. Suppose the randomly chosen item is a ring, then a possible word problem may read as follows:

*My fellow pirate Captain Thomas Thewood declared that a ring is one less than ten times the cost of a glove. If the cost of a ring is one hundred and nine, what is the cost of a glove?*

The first part reads *fellow pirate*, which is a relationship randomly taken from the relationship array, *Captain Thomas Thewood* is taken from the pirate names array and *declared* is from the verb array. The next part refers to the price of a *ring*, which is the randomly generated item. Also *one* is the number 1 written in words and *less* is a key word that indicates subtraction. Similarly, the word *times* indicates multiplication. Notice that the problem states that "*the cost of a ring is one hundred and
“nine”, which is the right side of the equation. The last part is just a question with a randomly generated item whose price is the solution. In this case, it is a glove. For complete source code please refer to appendix B.

4.3 Testing and Results

After completion of the development phase, the games were tested at Yorktown Middle School, Alexandria School and at the Indiana Academy for Math, Science and Humanities at Ball State University with approximately 200 middle-school students. Such tests utilized a technique based on pretest-posttest comparisons. Dr. James Jones, member of the research team stated that “although no change in pre to post test scores were found for Pirates or Chemistry, positive changes were evident for the cell divide game”. He also provided the following research summary.

“Cellular Divide and Conquer

Students from two sections within one school were given a 20 item pretest, allowed to play the game during a class session two weeks later, and then two weeks after game play were given the same 20 items in a posttest. The internal consistency as measured by Cronbach’s alpha was .49 at posttest.

Of the 29 students examined, 25 had complete data with 4 students absent the day of game play. A two-way, repeated measures ANOVA (gender by time) was run on these scores. A main effect for time ($F_{(1,23)} = 4.19, p = .052$) was found to approach statistical significance, but no other main effects or interaction were found to be statistically significant, so both boys and girls were impacted similarly by the game and had similar pre and posttest scores. Overall, the students increased their test score mean from 5.62(28.1%) at pretest to 6.88(34.4%) at posttest.

Zero order correlation coefficients between game play measures and the gain scores were calculated. A statistically significant correlation was found between the gain
score and time spent playing the game for the first session ($r = .41, p = .045$), and some correlations approached statistical significance, such as gain score and highest level reached during game play ($r = .38, p = .064$) and gain score and time spent playing the game during the second session ($r = -.39, p = .055$).

A regression was run for the overall gain score using as predictors the time taken for the first game, mean time taken per game session, and highest game level achieved. The resulting $R^2$ of .34 was statistically significant ($F_{(3,21)} = 3.53, p = .032$) with the time taken for the first game the most influential positive predictor ($\beta = .44, p = .022$) with the mean time taken per game having a negative relationship with the gain score ($\beta = -.38, p = .051$). The maximum level reached during game play was in a positive direction, but not statistically significant ($\beta = .29, p = .13$). These results would suggest that more time spent on the initial game coupled by less time on subsequent games produced higher gains on the posttest.

Pirate Math
No significant change in pre to post test scores ($t = .97, df = 27, p = .34$).

Chemistry Circus
No significant change in pre to post test scores ($t = 1.08, df = 35, p = .29$).

4.4 Pirate Math

The first educational video game we developed was Pirate Math. This game provides an introduction to variables, their graphic representation and how to solve simple equations. In particular, our goal is to eliminate common misconceptions that students have about variables. For instance, students often view variables as having a fixed value rather than varying in value. They also tend to believe that if two variables are represented by different symbols, they cannot have the same value. Some students even assign values to variables based on their position on the alphabet, so variable $c$ would be 3 and variable $f$ would 6. Based on these misconceptions,
we designed the game in such a way that students can clearly see that variables are like containers that can store a value and this value may change.

4.4.1 Introductory Screen

The game starts by showing an introductory screen with the main story line, which describes a pirate lost in an island, and with no possessions since someone has robbed him. The objective is to solve some math problems to earn enough doublons (pirate currency) to be able to purchase clothes, weapons and other merchandise. A screenshot of the introduction screen is shown in figure 35.

![Pirate Math Introduction Screen](image)

Figure 35: Pirate Math Introduction Screen

4.4.2 Main Menu

The game is divided into 4 modules. Each one requires the user to utilize a math skill so that every learning objective is covered. These modules are: coin toss, pirate gold, treasure map and pirate store. The main menu presents the user with an island where a pirate (main character) is shown together with some accessories and an animated ship in the background. It contains four buttons corresponding to each of the modules. Initially, only the coin toss button is enabled. Other buttons become
available as the user completes a module. A screenshot of the main menu is observed in Figure 36.

![Pirate Math Main Menu](image)

Figure 36: Pirate Math Main Menu

### 4.4.3 Coin Toss

The **coin toss** module aims to provide basic understanding of variables by presenting several chests containing gold coins. Initially, the user is presented with an introductory screen explaining the module’s objective. As seen in figure 37 this introduction explains that the value of a chest is given by the number of coins inside. Moreover, each chest is labeled by a letter so it is easy to see the relationship between chests, which are coin holders, and variables which store numbers.

When the module starts, several pirate chests appear on the upper half of the window. Each chest is named with a letter of the alphabet so that it relates to the value of a variable. The number on top of a chest represents the number of coins inside. On the bottom half, a pirate randomly throws coins to the chests. When a chest receives a coin, its number on top is enlarged so it is clear that its value is incremented by one. This is illustrated in figure 38 where the last chest to receive a
After a considerable number of coins have been thrown, the pirate temporarily stops and asks a math question based on the number of coins inside the chests. The difficulty of these questions depends on the level. At the beginning, only basic questions are asked (such as simple comparisons between variables). Complexity increases as the user progresses through different levels. A sample simple question is shown in figure 39. In this example, the question involves variables $G$, $C$ and $W$ which have values 5, 3 and 3 respectively. The correct answer is $< \text{ since } 3 - 3 = 0$ which is less than 5. Every correct answer increases the score until the user is promoted to a new level. After a promotion, the user is given doubloons. The number of doubloons depends on the level reached. Higher levels grant more doubloons than lower ones.

A key learning objective of this module is to show students how variables behave like pirate chests since variables store numbers and chests store gold coins. The purpose of this comparison is to explain that variables can change in value and that the letter used to identify a variable has no relationship to its current value.
Figure 38: Coin toss playing

Figure 39: Coin toss sample question
4.4.4 Pirate Gold

This module concentrates on basic understanding of simple arithmetic equations. To achieve this objective, it compares the process of solving an equation with that of balancing a seesaw. Each side of the seesaw corresponds to each side of the equation. Just as a seesaw is balanced only when the weight on each side is equal, a value that solves an equation makes both sides equal.

The module begins with an introductory screen explaining the main objective and how to achieve it as shown in figure 40. After the introductory screen, it presents a sea view of the pirate (main character), a dog next to him, a ship sailing on the ocean and a seesaw with a chest on one side as illustrated in figure 41.

![Figure 40: Pirate Gold introduction](image)

In addition, an equation is presented at the bottom of the screen. In this example, the equation is \( G + 3 = 6 \). The user is asked to give a value of \( G \) that will balance the equation. As this value corresponds to the weight of the dog, it is necessary to feed the dog with just the right amount of food so that its weight equals the solution. After this amount is entered, the dog starts walking and then jumps on top of the
free side of the seesaw causing the chest on the other side to be ejected. If the dog is too heavy, the chest will be thrown beyond the ship. On the other hand, if the dog is too light, the chest is also thrown, but it does not reach the ship and instead it sinks in the ocean. If the dog weight equals the solution, which in this case is 3, the chest will land in the ship. In this later scenario, the number of correct answers is updated. Otherwise, the user is warned that the given answer is incorrect. The complexity of the equations increases gradually as the user progresses through different levels. Completion of a level grants doubloons. The number of doubloons earned is based on the level completed as higher levels grant more doubloons than lower levels.

4.4.5 Treasure Map

The Treasure Map module consists of a pirate map with islands and treasures. The objective is to collect as many treasures as possible by sailing with a pirate ship. The ship sails by plotting a course in the map and solving a specific mathematical equation based on the current coordinates of the ship and the target location. This is explained in the introduction screen as shown in figure 42.

The map is divided by a grid that is used to indicate each location as in a Cartesian
coordinate system. The horizontal axe corresponds to the value of $x$ while the vertical axe references $y$. The lower left position $(x, y) = (0, 0)$ is the starting position of the ship.

Notice that in the screenshot of figure 43, five islands are shown and the top one contains a treasure. Therefore, the objective is to sail to the top island avoiding the others. To sail to a location, the user clicks on the desired position in the map locking that trajectory. In our example, the cross that represents the cursor is at position $(14, 1)$ and there is a dotted line between the ship and the cursor. This dotted line represents the path that the ship is attempting to sail. After the desired trajectory is locked, an equation with a question appears at the bottom. A correct answer causes the ship to sail to the target location (assuming there are no obstacles in the trajectory). Otherwise, it will go either before or after depending on the value entered by the user. If the ship hits an obstacle, the number of sank ships increments while an animated explosion indicates that the ship has been destroyed. Correct answers as well as treasures collected reward the player with points. When enough points are collected, doubloons are granted.
Figure 43: Pirate Math Treasure Map sample screenshot

The main educational value of this game is that it graphically demonstrates the relationship between dependent and independent variables by showing how the value of $x$ (independent variable) affects the value of $y$ (dependent variable). It also helps students understand how to use a Cartesian coordinate system to draw equations.

### 4.4.6 Pirate Store

This module becomes available after the player has completed all the previous ones. At this point, the player can purchase pirate merchandise by using the doubloons previously earned and by solving math problems. The introductory screen is displayed in figure 44.

In the Pirate Store, pirate merchandise is displayed inside a transparent counter and on the wall. A pirate acts as a store clerk and presents the player with a math problem. A correct answer grants access to the purchase area. An example problem is shown in figure 45.

All problems are randomly generated and therefore, it is extremely unlikely that a problem repeats. In our example, the problem reads “My brother Captain John Nutt declared that an eyepatch is five more than four multiplied by the cost of a hook. If the
CHAPTER 4. E-LEARNING

Figure 44: Pirate Store introduction

Figure 45: Pirate Store
cost of an eyepatch is sixty-five, what is the cost of a hook?” A key strategy to solve this problem is to transform it to a mathematical equation. For instance, suppose we use variable $Y$ to represent the value of an eyepatch and $X$ to represent the value of a hook. Based on the information provided, the equation would be $Y = 5 + 4X$ as $Y = 65$, the equation becomes $65 = 5 + 4X$. Solving the equation leads to $X = 15$.

When a correct answer is given, the player is taken to the purchase area where pirate merchandise can be bought by using doubloons earned in other sections. Only items whose price is equal or lower to the player’s credit can be purchased. Figure 46 shows a screenshot.

![Figure 46: Pirate Store shelf](image)

Every time an item is purchased, it becomes disabled so that it cannot be repurchased. Also, if the user runs out of credit, it is possible to go back to the previous modules to collect additional doubloons and revisit the Pirate Store. After all items are bought, the game displays a message congratulating the user.

The learning objective is to help students understand how math problems appear in everyday life situations and not be intimidated by them. Moreover, using pirate terminology and merchandise makes problems fun to solve.
4.5 Chemistry Circus

The second game we developed is Chemistry Circus. The learning objective is to help students understand the required steps necessary to balance chemical equations. To achieve this, the game presents an acrobat holding a bar on both hands and walking on a tightrope. The bar is linked to a chemical equation that needs to be balanced so that the acrobat can safely reach the destination.

4.5.1 Introductory Screen

The game begins with an introductory screen explaining the game objective with a storyline. This screen is depicted in figure 47. As this screen is optional, a player who wishes to skip it can do so by clicking on the start game button, in which case the game proceeds to the main menu.

Figure 47: Chemistry Circus introductory screen
4.5.2 Demo

After the introductory screen, the player is taken to the main menu. This menu provides two alternatives, to watch a demo or to directly play. The demo explains the process of balancing chemical equations by dragging and dropping coefficients from a pool to each side of the bar. Text boxes point to each element as illustrated in figure 48.

![Chemistry Circus demo, screen 1](image)

Figure 48: Chemistry Circus demo, screen 1

The next screen describes the items displayed on the upper right corner. These are the time remaining and the table showing the number of atoms on the reactants and on the products. The reactants appear to the left of the equation’s arrow, while products appear to the right of it. Figure 49 presents a screenshot of this help screen.

4.5.3 Game Play

The game starts by showing an acrobat walking on a tightrope and holding a bar. Above the bar, there are several boxes with molecules. Below the acrobat, there is a pool with nine balls floating. Each ball has a unique coefficient. The upper right corner displays the time remaining to balance the equation (given in seconds) along
with a table containing the number of reactants and products. Below the table, there is game information such as current level number, number of equations attempted, number of equations correctly balanced and score.

The player balances the equation by dragging coefficients (balls) from the pool and dropping them into the boxes above the acrobat’s bar. For instance, consider the sample screen of figure 50. On the reactants (left) side, there are 2 nitrogen and 2 hydrogen. On the products side (right), there are 2 nitrogen and 6 hydrogen. In order to balance the equation, the player should drag the ball with the coefficient 3 and drop it into the nitrogen box, which is the reactant’s leftmost box. This will cause the equation to be balanced and hence, the acrobat will walk straight to the destination. In the event that the coefficient selected does not balance the equation, the acrobat will move, but she will not reach her destination. In fact, as long as the equation is not balanced, the acrobat never reaches the destination regardless of how many times coefficients were dragged and dropped. Notice that below the acrobat’s bar, there are small boxes with little circles. Each circle represents the number of atoms for each element. Dragging and dropping coefficients will change the number of circles displayed. In this way, the player can clearly see how atoms are counted and easily compare these numbers between reactants and products. As the user balances
4.6 Cellular Divide and Conquer

The game **Cellular Divide and Conquer** deals with the biological process of cell division. The two processes addressed are mitosis and cytokinesis. Several learning objectives are targeted by this game including: recognition and understanding of the steps of cell division, awareness that the instructions for cells (genetic material) are found in chromosomes made of DNA and understanding that when cells reproduce, the genetic information is replicated and a copy is provided to each new cell. Also, elimination of misconceptions play an important role since students often believe that the steps of cell division are discontinuous rather than continuous, and that division of the nucleus and division of the cell are the same thing. This game attempts to
eliminate such misconceptions.

### 4.6.1 Main menu

The game begins by showing a screen with a space shuttle with the title **Cellular Divide and Conquer** as well as the **main menu**. The player can choose among three options: **controls**, **start** and **help** as shown in figure 51.

![Figure 51: Cellular Divide and Conquer main menu](image)

The **controls** option displays a keyboard layout screen explaining the use of arrow keys to guide the shuttle and the spacebar to shoot. This is observed in figure 52.

### 4.6.2 Help Screen

This first help screen gives an overview of the phases of mitosis and their purpose. The name of the phase appears on the left side inside boxes of different colors. The phases listed are: interphase, prophase, metaphase, anaphase and telophase. For each phase, there is a description and a small diagram giving an idea of how the nucleus looks on that phase as it is observed in figure 53.
 CHAPTER 4. E-LEARNING

Figure 52: Cellular Divide and Conquer main menu

Figure 53: Cellular Divide and Conquer help screen 1
The process of cytokinesis is described in the next help screen since it is not part of mitosis. Keeping the two processes separate helps to avoid the misconception that cytokinesis is part of mitosis. Figure 54 shows a screenshot.

![Cytokinesis help screen](image)

Figure 54: Cellular Divide and Conquer help screen 2

### 4.6.3 Game Play

*Cellular Divide and Conquer* is divided into three levels. Each level is linked to an animal that corresponds to the number of chromosomes for that level. Level 1 is *ant* with 2 chromosomes, level 2 is *mosquito* with 4 chromosomes and level 3 is *fruit fly* with 6 chromosomes. The objective is to save the animal by completing the phases of mitosis and cytokinesis.

The main screen is divided into two sections. To the right, there is a scoreboard which includes score, shots and lives. There is also a clock with all the phases of mitosis and cytokinesis as well as a menu to obtain help or exit the game. The left section is where chromosomes and the shuttle appear. A level starts by showing an image of its corresponding animal as shown in figure 55.

At the beginning of each phase, an introductory screen describes its *mission*. The introductory screen for *interphase* is observed in figure 56. When the game starts, the clock’s arm begins to move indicating the time remaining to complete the mission.
When time runs out, the player loses a life. Losing 3 lives in the same level leads to game over. The shuttle is controlled by the arrow keys and has a canon that can spin 360° allowing aim in any direction. The canon shoots when the space bar is pressed.

The interphase starts by showing the user’s shuttle and several chromosomes moving around. The number of chromosomes depends on the animal of the current level. In the first level, ant, two chromosomes are shown as illustrated in figure 57. The objective is to hit all the chromosomes by shooting from the shuttle. Initially, a chromosome is displayed as a long thick vertical segment. After it is hit, its shape resembles the letter X. In our example, the chromosome to the left has already been hit while the one to the right has not. Once the objective is completed, the player is promoted to prophase.

In prophase chromosomes are shown “X” shaped. The objective is to hit all the chromosomes. When a chromosomes is hit, it condenses into a smaller one. As in the previous level, the clock displays the time remaining to complete the mission. A screenshot of prophase is shown in figure 58.

After completion of prophase, the player is promoted to metaphase. The objective of this phase is similar to the previous ones as chromosomes have to be hit. When this happens, the chromosome will no longer move, and instead it will be aligned as the one to the left in figure 59. After all chromosomes have been aligned, the game progresses to anaphase.
Figure 56: Interphase introduction

Figure 57: Interphase game play

Figure 58: Prophase game play
The objective of anaphase is to split chromosomes. The middle part of a chromosome is called centrome, and if it is hit, the chromosome splits. When this occurs, an animation shows the two parts moving away from each other as it is illustrated by the chromosome to the left in the sample screenshot of figure 60.

The final phase of mitosis is telophase. This phase begins by displaying a screen based on the completion of the previous phase. As chromosomes are split, the screen shows the upper and lower parts away from each other. The player needs to hit all the chromosomes parts to complete the level as shown in figure 61.

The last phase of the game is cytokinesis. This phase is not part of mitosis, but
it is necessary for cell division. The objective is to divide the cell by shooting to its nucleus as illustrated in figure 62.
Chapter 5

Model Checking

Given a model of a system, Model Checking is a method for automatic verification of that model. This model may refer to software, hardware, reactive systems or others. In other words, given a real-life system, create a simplified model and then automatically test it to find out if it meets certain specifications. The specifications usually refer to safety requirements. As an example, suppose that we want to show that a certain concurrent program never deadlocks. We construct a formal model of the system which captures that property and excludes details that do not affect the correctness of the checked property. Model checking was originally developed by research teams in academia, and nowadays it is also used in industrial applications [22, 28, 35]. Model checking is part of the discipline of formal methods described in section 5.2.

Because critical systems often deal with life-threatening or very dangerous situations, they are a primary target of model checking. For such systems, a system failure is unacceptable. Due to the increasing number of systems that are being considered critical, model checking is becoming an essential method for verification of large and complex systems.

In this chapter, we present an assembly line simulator that we have modeled using SPIN, a popular open-source tool mainly used in the formal verification of distributed
software systems. This simulator is part of a collaborative research project with Dr. Jay Bagga. This model and its results were presented at the International Colloquium on Theoretical Aspects of Computing (ICTACS 2005), held in Hanoi, Vietnam [10].

We have used Holzmann’s book [35] and the SPIN’s website [36] as references for SPIN and PROMELA. The definitions and examples of critical systems are taken from [42]. The discussion of formal methods is primarily based on [65] and [19]. The definition of assembly line was obtained from [50] and the discussion on its historical perspective is based on [63].

In the first section, we discuss critical systems. We start with an overview of the characteristics that differentiate these systems from others. We then study their significance and the need to eliminate defects due to the dangerous risks involved.

We continue our discussion by analyzing formal methods. We define their characteristics and explain how they are used in the verification of critical systems.

In the following section we give an overview of SPIN and its applications. We examine the difference between simulation and verification run explaining the purpose of each one. We also provide a brief description of its specification language called PROMELA by examining the use of processes, communication channels, variables, atomic statements, etc. The main purpose of this section is to serve as an introduction to our the assembly line simulator rather than to provide a comprehensive tutorial on SPIN and PROMELA. The interested reader is referred to Holzmann’s book [35].

In the last section, we study an assembly line simulator. We give a description of an assembly line and explain its advantages in the manufacturing process by providing a classic case study. This helps us gain a better insight about the building blocks of the actual machine that we modeled. We study each one of its parts and how they all work together to process machinery parts. We conclude by presenting
the actual SPIN model and explaining how Model Checking was used to verify certain safety properties.

5.1 Critical systems

As explained by Knight [42], systems whose failure could result in major property damage, damage to the environment and even loss of life are referred to as safety-critical systems. Classic examples in application areas include medical devices, aircraft flight control, weapons and nuclear systems. In addition, many other information systems are starting to be considered safety-critical in a broad sense since their failure may cause significant financial losses. As systems become larger and more complex, the likelihood of subtle errors increases and traditional testing techniques may not be sufficient to detect them. Due to the critical nature of these systems, it is imperative to produce reliable software in spite of its complexity. Moreover, due to the fact that system failures are unacceptable, it is essential to have proof of system correctness to ensure that specifications are met.

Examples of common methods to improve reliability are testing, simulation and formal verification. As previously discussed, testing and simulation may not be able to find all the errors. For this reason, formal methods are used in critical systems since they guarantee correctness.

5.2 Formal methods

Development of computer systems usually involves the use of several techniques. One of these techniques is called formal methods. These are mathematically based techniques that are used to describe system properties. Such methods serve as a framework for specification, development and verification [65]. Formal methods techniques are based on mathematical logic. A formal verification involves strict deductions of logical statements using laws of mathematical logic. The significance of these methods is that they provide a vehicle to symbolically examine the complete state space of
a hardware or software design, establish correctness and safety properties that hold for all inputs \cite{19}.

5.2.1 SPIN

We have used the tool SPIN to create, test and verify our assembly line model. The name SPIN \cite{35} is an acronym for Simple PROMELA Interpreter where PROMELA is the name of the language accepted by SPIN. This tool is a powerful open-source software primarily used for the formal verification of distributed software systems. SPIN was conceived by Gerard Holzmann while working in the Computing Sciences Research Center at Bell Labs back in the early '80s. The software became freely available in 1991. Moreover, SPIN was awarded the prestigious 2001 System Software Award by the ACM. Nowadays, it is a very popular tool used by thousands of individuals around the world \cite{36}.

SPIN has two modes: simulator and verifier. The simulator mode is used to get an idea of the kind of behavior captured by the model. It is important to understand that regardless of the number of simulations performed, only a verification can prove that the model is correct. However, if the verification finds a counterexample to a correctness claim, it is possible to use the simulator to perform a guided simulation based on the error trace. Refer to \cite{35} for further details and an excellent tutorial to SPIN.

PROMELA

PROMELA is an acronym for Process Meta-Language and it is the language accepted by SPIN. PROMELA was designed to be a system description language rather than an implementation language. Hence, the stress is placed on the modeling of process synchronization and coordination instead of computation. The main elements of SPIN models are asynchronous processes, message channels, synchronizing statements and structured data. There is no notion of time, no floating point numbers and only few computational functions \cite{35}. 

Processes are used to define behavior and are declared by using the `proctype` statement. They can be instantiated by using the keyword `active`. It is also possible to create several instances of a specific `proctype` by including the desired number in square brackets next to the `active` keyword. For instance, consider the following code:

```c
active [5] proctype sample_process() {
    printf("Process id is: %d\n", _pid)
}
```

The `proctype` keyword is used to declare the process `sample_process`, the `active` keyword indicates that the process will be instantiated and the number 5 inside the square brackets indicates that there will be five instances of `sample_process`. The predefined variable `_pid` contains the process instantiation number. In this particular example, 5 processes are instantiated and each one prints the value of its instantiation number and terminate. A simulation run of this example produces the output

```
$ spin sample_process
Process id is: 2
    Process id is: 0
        Process id is: 4
            Process id is: 1
                 Process id is: 3
5 processes created
```

By default SPIN organizes the output of each active process to appear in a different column. Notice that the output is not printed sequentially. This is because execution is asynchronous, and therefore the five lines of output may appear in any order. For this reason, other simulations may show a different sequence.

Processes exchange data by the use of `message channels`. These channels can be declared locally or globally. The typename `chan` is used to declare a channel. A message can contain any finite number of fields. An example from Holzmann’s book [35]
illustrates this concept. Consider the following statement, which declares a message name \texttt{qname}.

\begin{verbatim}
chan qname = [16] of \{short, byte, bool\}
\end{verbatim}

This declaration indicates that the channel \texttt{qname} contains three fields. The first one is of type \texttt{short} and the other two are of type \texttt{byte} and \texttt{bool}. The following statement illustrates the syntax used to send a message.

\begin{verbatim}
qname!expr1,expr2,expr3
\end{verbatim}

Using this declaration, a message is sent with the values of \texttt{expr1, expr2, expr3}. As it was shown in the declaration, the types declared were \texttt{short, byte} and \texttt{bool} and therefore, each expression value is cast to the type of the message field corresponding to the position in the declaration. The statement shown below retrieves a message and stores the values from the \texttt{expr1, expr2, expr3} into \texttt{var1, var2, var3}.

\begin{verbatim}
qname?var1,var2,var3
\end{verbatim}

The receive statement is executable only if the source channel is not empty. It is an error to send or receive either more or fewer messages fields than were declared.

An \textbf{atomic} sequence defines a series of statements that are to be executed with no interruptions. Hence, since the beginning of the execution of the first statement in the atomic sequence until the last statement has finished executing, no other process can execute statements.

\section{5.3 Assembly Line Simulator}

According to Encyclopaedia Britannica [50], an \textbf{assembly line} is “an industrial arrangement of machines, equipment, and workers for continuous flow of workpieces in
mass-production operations”. For instance, consider a car manufacturing company that wants to introduce a new car model A. In this case, the final product (output) will be a car model A and the objects required to build it (input) will be machinery parts. The design of this assembly line contains the sequence of steps necessary to process each part and manufacture each product component as well as the final product. The movement of material is reduced, with no backtracking, cross flow or repetitive operations, and all operations are to be compatible.

An article published on the Public Broadcasting Service (PBS)’s website [63] presents a classic example of an assembly line at the beginning of the 20th century when Ford Motor Company was able to produce the Model T in a more efficient way and at a lower the price. The mass-production of this model made it possible for the average person to afford a car. The principles used by Ford were: interchangeable parts, continuous flow, division of labor, and reduced wasted effort.

In this section, we present a model to simulate an assembly line. We first describe our model, give details on its components and how they work. We then specify some properties that it has to meet. Finally, we develop formal specifications using temporal logic and use a model checking tool to verify these properties.

5.3.1 Machine description

The machine we studied is used to process machinery parts. It contains several components that work simultaneously to produce the final product. Each component is assigned only one task and is dependent on the successful completion of the previous one. These components are: an input chute, which provides the raw parts; a linear actuator, that transports the parts along a beam; a raw part gripper, which picks the raw part up and puts it into the machine; a machine that processes the parts and finally, a finished part gripper that places the finished part back on the linear actuator. Figure 63 shows a diagram of these components.
The actual sequence of steps to produce a finished part from a raw part is as follows. A new raw part is brought by the input chute. This new part is placed on the linear actuator, which will transport it until the raw part gripper picks it up. The raw part gripper gives the raw part to the machine. The machine processes the part and when it is ready, the finished part gripper retrieves it from the machine and places it back on the linear actuator.

5.3.2 SPIN model

In order to perform testing and verification, we created a simplified model of the actual machine. In this way, we isolated the essential components and left out the irrelevant ones. Our actual model contains several components. A walking input beam, a load arm, a face and center machine, an unload arm and a walking output beam. For simplicity, we refer to the face and center machine as oven. In this work, both terms will be used interchangeably. A close look at figure 64 will help us understand the functioning of this model.

A raw part is placed on the walking input beam. The walking input beam transports the part until it is picked up by the load arm. The load arm grabs the part and puts it inside the oven. The oven process the part. The finished part is picked up by arm B. This arm grabs the finished part and places it on the walking output beam. The walking output beam transports the finished part to its final destination.
The oven has a door and jaws. When a part is being processed, the door is closed and the jaws hold the part. The part is then processed and when it is ready, the jaws release it and the door opens. The states that are defined to be \texttt{mtype} are detailed in table 3 and the global variables are listed in table 4.

In addition, there are three constants that determine the maximum number of parts that the input beam can have at any given time, the maximum number of parts that the output beam can have at any given time and the number of parts to be processed; these constants are \texttt{MAX\_INPUT\_BEAM}, \texttt{MAX\_OUTPUT\_BEAM} and \texttt{TOT\_NUM\_PIECES} respectively. The main diagram of our model is divided into two parts. At the beginning, all variables are initialized as shown in figure 65.

After initialization, the main process executes as it is illustrated in figure 66. At the beginning, it attempts to place a part on the input beam. This can be accomplished only if the input beam has not reached the maximum number of parts allowed.
Table 4: Global variables definitions

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>door</td>
<td>mtype</td>
<td>oven's door</td>
<td>up</td>
</tr>
<tr>
<td>jaws</td>
<td>mtype</td>
<td>oven's jaws</td>
<td>release</td>
</tr>
<tr>
<td>inputarm</td>
<td>mtype</td>
<td>load arm</td>
<td>withoutpiece</td>
</tr>
<tr>
<td>outputarm</td>
<td>mtype</td>
<td>unload arm</td>
<td>withoutpiece</td>
</tr>
<tr>
<td>oven</td>
<td>mtype</td>
<td>face and center machine</td>
<td>idle</td>
</tr>
<tr>
<td>ovenstate</td>
<td>mtype</td>
<td>state of the oven</td>
<td>withoutpiece</td>
</tr>
<tr>
<td>NumPiecesInputBeam</td>
<td>int</td>
<td>current number of pieces in input beam</td>
<td>0</td>
</tr>
<tr>
<td>NumPiecesOutputBeam</td>
<td>int</td>
<td>current number of pieces in output beam</td>
<td>0</td>
</tr>
<tr>
<td>TotNumPiecesProcessed</td>
<td>int</td>
<td>total number of pieces finished</td>
<td>0</td>
</tr>
<tr>
<td>StopInputBeam</td>
<td>bit</td>
<td>when 1, Input Beam stops</td>
<td>0</td>
</tr>
<tr>
<td>StopOutputBeam</td>
<td>bit</td>
<td>when 1, Output Beam stops</td>
<td>0</td>
</tr>
<tr>
<td>StopArm1</td>
<td>bit</td>
<td>when 1, load arm stops</td>
<td>0</td>
</tr>
<tr>
<td>StopArm2</td>
<td>bit</td>
<td>when 1, unload arm stops</td>
<td>0</td>
</tr>
<tr>
<td>StopOven</td>
<td>bit</td>
<td>when 1, oven stops</td>
<td>0</td>
</tr>
<tr>
<td>StopMain</td>
<td>bit</td>
<td>when 1, main stops</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 65: Variables initialization

at a given time and also, if the total number of parts processed is less than the maximum allowed. If these conditions are met, the part is placed on the input beam. This task continues as long as conditions are met. If the maximum number of parts processed is reached, then execution stops. Otherwise, the part is processed.

The PROMELA code of the main process starts by initializing variables. As the initialization statements are inside the atomic clause, execution will proceed sequentially and no other processes can be executed until the last statement in the atomic clause has completed. After this, it proceeds to run the processes of each component of the machine asynchronously. Next, it remains in the process state until the end conditions force the main process to exit the loop. In this case, it sends a termination
message to the *walking input* beam indicating that it should stop after processing its current part. This will cause the *walking input beam* to also send termination messages to the next component of the machine. Each component will terminate and send a message to the next until all components have stopped processing parts. In this last scenario, the main process will exit the waiting loop and print some text indicating successful completion and the number of parts processed. Here is the PROMELA code for the main process.

```prome
active proctype main()
{
    atomic {
        /* Initialize variables */
        door=up;
        oven=idle;
        jaws=release;
        inputarm=withoutpiece;
        outputarm=withoutpiece;
        NumPiecesInputBeam=0;
        NumPiecesOutputBeam=0;
        NumPiecesProcessed=0;
        StopInputBeam=0;
        StopOutputBeam=0;
        StopArm1=0;
        StopArm2=0;
        StopOven=0;
    }

    run doWalkingInputBeam();
    run doArm1();
    run doOven();
    run doArm2();
    run doWalkingOutputBeam();
}
```
CHAPTER 5. MODEL CHECKING

The walking input beam can be in one of two states. In the *idle* state, the beam is not moving. This occurs if there are no parts to be transported, but the StopInputBeam variable is set to 0 (false). On the other hand, if there is at least a part to be transported and the load arm is empty, then a message is sent to the load arm to receive the part. Another scenario is that a part is ready to be loaded, in which case, a *load* message is received and then it goes to *Movebeam*. A flow diagram of this process is detailed in figure 67.

In *Movebeam*, the input beam is no longer in the *idle* state, but it is loading parts and giving parts to the load arm as long as preconditions for those two events are met. For instance, a part can be picked up by the load arm only if there are parts in the input beam and also the load arm is free (without a part). Figure 68 illustrates this in a flow diagram.

Here is the PROMELA code for the walking input beam.

```proctype dowalkinginputbeam()
```
Figure 67: Walking input beam idle state

Figure 68: Walking input beam move state
When the model executes, each component performs a task asynchronously. As each task is interconnected, it is reasonable to expect that some components may need to wait for other components to finish before proceeding. For instance, consider the load arm of figure 64. Its work consists of picking up a part and loading it into the oven. However, as the oven can only process one part at a time, the load arm has to check that the oven is in *idle* state before attempting to load it. At the same time, the unload arm has to make sure that the oven has indeed finished processing a part before attempting to unloaded it from the oven. The following is a simulation run for $MAX_INPUT_BEAM = 3, MAX_OUTPUT_BEAM =$
3. \text{TOT\_NUM\_PIECES} = 5.

Notice that interconnection does not only occur among different components, but
it can also happen inside a single component. Consider the oven component. The oven is “free” when it is in \textit{idle} state, its door is up (open) and the jaws are on \textit{release} state. When processing a part, the part is inside, the state is \textit{processing}, the door is down (closed) and the jaws are on \textit{hold} state. Notice that the case that a part is been processed and the door is up, should never occur. These are just few examples of properties that can be verified. For complete diagrams refer to appendix C and for the entire PROMELA source code see appendix D.

In this work, we verified three safety properties. These properties state that the following conditions should never occur.

1. (FaceAndCenterMachine==busy) And (door==up)
2. (jaws==hold) And (door==up)
3. NumPiecesInputBeam$>$MAX\_INPUT\_BEAM

We use values of MAX\_INPUT\_BEAM=5, MAX\_OUTPUT\_BEAM=5 and TOT\_NUM\_PIECES=8. By running Spin in verification mode, we were able to prove that under any circumstances the above conditions become true and therefore, our model is verified. The following output generated by Spin shows this result.

(Spin Version 5.2.0 -- 2 May 2009)

Full statespace search for:

\begin{verbatim}
never claim +
assertion violations + (if within scope of claim)
acceptance cycles - (not selected)
invalid end states - (disabled by never claim)
\end{verbatim}

State-vector 0 byte, depth reached 0, errors: 0
0 states, stored
0 states, matched
0 transitions (= stored\+matched)
0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):

\begin{verbatim}
0.000 equivalent memory usage for states (stored\+State-vector + overhead)
0.000 actual memory usage for states (less than 1k)
\end{verbatim}
The never claims and assertion violations are checked as indicated by the + sign. The output also indicates that 0 errors were found.
Chapter 6

Conclusion

We have shown the importance of software systems for teaching and research by presenting several computer applications that we have developed. We demonstrated their exceptional educational value by showing the enormous leverage they offer to researchers.

The three areas covered were Graph Theory, E-Learning and Model Checking. We provided an overview on each topic and illustrated the benefits of using our software systems on all these areas.

We described how our software system JEdit provides students with hands-on experience on graph theory since they can manipulate graphs and perform graph operations. They can also execute algorithms with a step-by-step feature that allows them to follow the execution flow which gives them an idea of the steps involved. We also showed how students implement new algorithms by writing the appropriate Java source code and easily adding them to JEdit without the need to recompile the entire system. This teaching technique is consistent with the constructivism theory of learning as students are not directly taught the material but instead, they “discover” the new knowledge.

We also presented three educational video games in the areas of mathematics,
chemistry and biology and showed how they provide an enriching learning experience to middle school students by using graphics, animations, sounds and humorous vocabulary. Furthermore, we discussed how the games combine teaching techniques based on operant conditioning of the behaviorism theory of learning and use concepts familiar to students so that students can easily relate to the new material.

6.1 Research

By using our software systems we generated large amounts of data which led us to discover important properties of graphs leading to the publication of two papers [14, 15]. In some instances, the data included over one million graceful labelings which shows clearly how manual analysis is impractical. In addition, we also mentioned conferences where our software systems have been presented [9, 11, 12, 13].

6.2 Future directions

As part of the ongoing work on graph theory, a valuable contribution would be the completion of our system Graph Algorithm Constructor. This would allow the creation and execution of graph algorithms without the need to write computer source code.

Concerning our system Manohar, it would be beneficial to incorporate new algorithms including those to enumerate graceful labelings of caterpillars and lobsters. Another important feature would be the computation of graceful labelings for any graph G. In this way, it would display a graceful labeling if G is graceful or inform the user that no graceful labeling was found.

In regards to our educational games, the results presented by Dr. James Jones provide encouraging results for our game Cellular Divide and Conquer. It would be beneficial to find out if some modifications to our games could improve students’
general understanding of the topic and elimination of misconceptions.
Appendices
Appendix A

Pirate Store ActionScript Code

this.stop();
Mouse.show();
var sMyText:String = "";
var sMyWordProblem:WordProblem = new WordProblem();
var ParrotSound1:Sound;
var BackgroundMusic:Sound;
var aDiaLogSound:Array;
var bPlayingMusic:Boolean = false;
var nBackgroundMusicNumber:Number = 2;
var nNumberOfSongs:Number = 4;
var nNumberOfRightQuestionsInLevel2 = 6;
var nTotalRightAnswers:Number = 0;
var nTotalWrongAnswers:Number = 0;
var nTotalAnswers:Number = 0;
var nAttempts:Number = 0;
var sIncorrectSound:Sound;
var CorrectSound:Sound;
var sSceneCalledID:Number;
var tNewFormat:TextFormat = new TextFormat();
var sDialog:String = "";
var aDialog = new Array();
var aDialogCallID = new Array();
tEquationText.text = "";
tEquationText.selectable=false;
tDialogText.text = "";
tDialogText.selectable=false;
tUserInput.text = "";
tUserInput.enabled = false;
tTitleAnswerWindow.text = "";
bVerify1._alpha = 0;
bVerify1.enabled = false;
bBackAnswer1.enabled = false;
bBackAnswer1._alpha = 0;
mcDialogCloud1._alpha = 0;
mcDialogCloud2._alpha = 0;
mcDialogCloudParrot._alpha = 0;
EnableAnswerWindow();
GetNextProblem();
UpdateScore();
var nTimeToAnswer:Number=0;
var nStartTimeToAnswer:Number=GetInstant();
var nTotalTimeUsedInLevel:Number=0;
function EnableAnswerWindow():Void {
    bBackAnswer1._alpha = 60;
    tTitleAnswerWindow.text = "Enter Answer";
    tUserInput.text = "0";
    bVerify1._alpha = 100;
    bVerify1.enabled = true;
    tUserInput.selectable = true;
}

function DisableAnswerWindow():Void {
    bBackAnswer1._alpha = 0;
    tTitleAnswerWindow.text = "";
    tUserInput.text = "";
    bVerify1._alpha = 0;
    bVerify1.enabled = false;
    tUserInput.enabled = true;
    mcDialogCloudParrot._alpha = 0;
    tEquationText.text = "";
    tUserInput.selectable = false;
}

bVerify1.onPress = function():Void {
    VerifyAnswer();
};

var oStoreKeyListener:Object = new Object();
oStoreKeyListener.onKeyDown = function():Void {
    if (Key.getCode() == Key.ENTER) {
        VerifyAnswer();
    }
};

Key.addListener(oStoreKeyListener);

function VerifyAnswer():Void {
    if (eMyProblem.IsSolution(tUserInput.text)) {
        nTotalRightAnswers++;
        nTotalAnswers++;
        PlayRightAnswer(0);
    } else {
        nAttempts++;
        if (nAttempts<3) {
            GiveHint();
        } else {
            nTotalWrongAnswers++;
            nTotalAnswers++;
            PlayWrongAnswer(0);
        }
    }
    UpdateScore();
}

function PlayWrongAnswer(nSceneNumber:Number):Void {
    if (nSceneNumber>1) {
        tDialogText.text = "Arrg..Wrong Answer my fellow Pirate! The correct answer is "+eMyProblem.GetSolution();
        tfNewFormat.bold = true;
        tfNewFormat.color = 0xFF0000;
        tfNewFormat.size = 30;
        tDialogText.setTextFormat(tfNewFormat);
        IncorrectSound = new Sound(this);
        IncorrectSound.attachSound("IncorrectFanfare");
        IncorrectSound.start();
        nSceneCalledID = setInterval(PlayWrongAnswer, 5000, nSceneNumber+1);
        AddCurrentData(false);
    } else {
        tDialogText.text = "Arrg..Wrong Answer my fellow Pirate! The correct answer is "+eMyProblem.GetSolution();
        tfNewFormat.bold = true;
        tfNewFormat.color = 0xFF0000;
        tfNewFormat.size = 30;
        tDialogText.setTextFormat(tfNewFormat);
        IncorrectSound = new Sound(this);
        IncorrectSound.attachSound("IncorrectFanfare");
        IncorrectSound.start();
        nSceneCalledID = setInterval(PlayWrongAnswer, 5000, nSceneNumber+1);
        AddCurrentData(false);
    }
APPENDIX A. PIRATE STORE ACTIONSCRIPT CODE

function PlayRightAnswer(nSceneNumber:Number):Void {
  if (nSceneNumber<1) {
    tDialogText.text = "Great answer Pirate!";
    tfNewFormat.bold = true;
    tfNewFormat.color = 0xFF0000;
    tfNewFormat.size = 30;
    tDialogText.setTextFormat(tfNewFormat);
    CorrectSound = new Sound(this);
    CorrectSound.attachSound("CorrectFanfare");
    CorrectSound.start();
    nSceneCalledID = setInterval(PlayRightAnswer, 5000, nSceneNumber+1);
  } else {
    clearInterval(nSceneCalledID);
    CorrectSound.onSoundComplete = EndFunctionCorrect();
    BackgroundMusic.stop();
    this.stop();
    mcPurchasedItems1._x=-20;
    mcPurchasedItems1._y=0;
  }
}

function GiveHint():Void {
  var sEquation:String = eMyProblem.GetEquation();
  var aEqSides = new Array();
  aEqSides = sEquation.split("=");
  ParrotSound1 = new Sound(this);
  ParrotSound1.attachSound("ParrotSound");
  ParrotSound1.start();
  mcDialogCloudParrot._alpha = 80;
  if (nAttempts == 1) {
    tEquationText.text = "Brr..One side of the equation " + aEqSides[0];
  } else {
    tEquationText.text = "Complete equation " + sEquation;
  }
}

bMusic1.onPress = function():Void {
  if (!bPlayingMusic) {
    PlayBackgroundMusic();
  }
};

function PlayBackgroundMusic():Void {
  BackgroundMusic = new Sound(this);
  if (nBackgroundMusicNumber<(nNumberOfSongs-1)) {
    nBackgroundMusicNumber++;
  } else {
    nBackgroundMusicNumber = 0;
  }
  BackgroundMusic.attachSound("PirateSong"+nBackgroundMusicNumber.toString());
  BackgroundMusic.onSoundComplete = function():Void {
    bPlayingMusic = false;
  };
  BackgroundMusic.start(0);
  bPlayingMusic = true;
}

function EndFunctionCorrect() {
AddCurrentData(true);
GetNextProblem();
}

function EndFunctionIncorrect() {
GetNextProblem();
}

function UpdateScore():Void {
  if(_global.nDubloons==1) tCurrentCredit.text = "Credit: "+_global.nDubloons+" doubloon"
  else tCurrentCredit.text = "Credit: "+_global.nDubloons+" doubloons";
}

function GetNextProblem():Void {
  if (nTotalRightAnswers<nNumRightQuestionsBefLevel2) {
    eMyProblem.GenerateWordProblem(0);
  } else {
    var nKindOfProblem:Number = Math.floor(Math.random()*2);
    eMyProblem.GenerateWordProblem(nKindOfProblem);
  }
  tDialogText.text = eMyProblem.GetProblem();
  if (length(tDialogText.text)<200) {
    mcDialogCloud2._alpha = 90;
    mcDialogCloud1._alpha = 0;
  } else {
    mcDialogCloud1._alpha = 90;
    mcDialogCloud2._alpha = 0;
  }
  nAttempts = 0;
  tUserInput.text = "0";
  tUserInput.setFocus();
  mcDialogCloudParrot._alpha = 0;
  tEquationText.text = "";
}

bReadWordProblem1.onPress = function():Void {
  var i:Number;
  sDialog = eMyProblem.GetDialog();
  aDialog = sDialog.split("|");
  PlayDialog(0);
};

var testSound:Sound;
function PlayDialog(nPartNumber:Number):Void {
  var CurSound:String;
  if (nPartNumber<aDialog.length) {
    if ((aDialog[nPartNumber] != ".") && (aDialog[nPartNumber] != "")) {
      CurSound = aDialog[nPartNumber].toLowerCase()+".mp3";
      testSound = new Sound();
      testSound.attachSound(CurSound);
      testSound.start(0);
      aDialogCallID[nPartNumber] = setInterval(PlayDialog, testSound.duration-100, nPartNumber+1);
    } else {
      aDialogCallID[nPartNumber] = setInterval(PlayDialog, 500, nPartNumber+1);
    }
  } else {
    clearInterval(aDialogCallID[nPartNumber-1]);
  }
}

bPirateStoreHelp.onPress=function():Void {
APPENDIX A. PIRATE STORE ACTIONSCRIPT CODE

bExit1._y+=3000;
bPirateStoreHelp._y+=3000;
bMusic1._y+=3000;
tUserInput.enabled=false;
bVerify1.enabled=false;
mcPirateStoreHelp1._y=400;
mcPirateStoreHelp1._x=400;
}

bExit1.onPress = function():Void {
    BackgroundMusic.stop();
    this.stop();
    nAttempts = 0;
    bBackAnswer1._alpha = 0;
    tTitleAnswerWindow.text = "";
    tUserInput.text = "";
    bVerify1._alpha = 0;
    bVerify1.enabled = false;
    tUserInput.enabled = true;
    mcDialogCloudParrot._alpha = 0;
    tEquationText.text = "";
    _root.gotoAndPlay("MainMenu");
};

function AddCurrentData(bPassed:Boolean):Void {
    nTimeToAnswer=GetInstant()-nStartTimeToAnswer;
    _global.udo.AddAttempt(bPassed, "SectionPirateMath", undefined, 8, nTimeToAnswer, 0,0,[]);
    nStartTimeToAnswer=GetInstant();
    _global.udo.GamePassed = true;
    _global.udo.SendData("anyFunction", callbackObj);
}

function GetInstant():Number {
    var myDate = new Date();
    return myDate.getTime();
}
Appendix B

Word Problem Class

class WordProblem {
    private var _nRange: Array = [-100, 100];
    private var _bRandomVariableNames: Boolean;
    private var _nEquation: String = null;
    private var _nSolutionValue: Number;
    private var _nLeftTerm: Number;
    private var _nRightTerm: Number;
    private var _sWordProblem: String = "";
    private var _sVowels: String = "aiueo";
    private var _sDialog: String = ""
    private var _aItems: Array = [
        "sword", "hat", "boot", "shirt", "knife", "pirate flag",
        "bandanna", "coat", "glove", "belt", "ring", "earring", "necklace",
        "pendant", "book", "eyepatch", "dagger"];
    private var _aAdditionWordsAfter: Array = [
        "plus", "more", "in addition to"];
    private var _aAdditionWordsBetween: Array = [
        "addition"];
    private var _aSubstractionWordsAfter: Array = [
        "minus", "less"];
    private var _aSubstractionWordsBetween: Array = [
        "difference"];
    private var _aMultiplicationWordsAfter: Array = [
        "times", "multiplied by"];
    private var _aMultiplicationWordsBetween: Array = [
        "product of"];
    private var _aDivisionWordsAfter: Array = [
        "divided by", "distributed among"];
    private var _aDivisionWordsBetween: Array = [
        "quotient", "ratio"];
    private var _aSayVerb: Array = [
        "used to say", "mentioned", "told me", "yelled", "complained",
        "realized", "stated", "announced", "declared", "explained", "mandated",
        "ordered", "requested", "shoved"];
    private var _aIntroductionPhrase: Array = [
        "A long time ago", "During the battle of Pearl Island",
        "While eating live snakes in the ship",
        "While some monkeys were stealing our food",
        "In a lost island", "Under the shade of a palm tree"];
    private var _aPirateAdjectives: Array = [
        "crazy", "mad", "hostile", "abominable", "dangerous",
        "fearless", "ruthless", "brutal", "savage", "wild",
        "cruel", "cold-blooded", "merciless", "bloodthirsty", "sanguinary"];
    private var _aRelationship: Array = [
        "friend", "partner", "fellow pirate", "monkey", "treasure hunter friend",
        "old friend", "pirate chum", "pirate buddy", "brother",
        "comrade", "pirate ally"];
    private var _aPirateNames: Array = new Array;
    private function SetRange(nLow: Number, nHigh: Number): Void {
        _nRange[0] = nLow;
        _nRange[1] = nHigh;
    }
}
public function GetProblem():String { return _sWordProblem; }

public function GetDialog():String { return _sDialog; }

public function GetEquation():String { return _sEquation; }

private function ComputeEquation(nEquationType:Number):Void {
    var na:Number = _nRange[0], nb:Number = _nRange[1];

    switch (nEquationType) {
        case 0:
            _nRightTerm2=-999.99;
            _nSolutionValue = na + Math.floor(Math.random() * (nb - na + 1));
            _nSolCoefficient = Math.floor(Math.random() * 10);
            if (_nSolCoefficient == 0) _nSolCoefficient = 10;
            _nLeftTerm = 5-Math.floor(Math.random() * 10);
            _nRightTerm = _nSolCoefficient * _nSolutionValue + _nLeftTerm;
            _sEquation = _nSolCoefficient.toString() + "X";
            if (_nLeftTerm < 0) _sEquation += _nLeftTerm.toString();
            else _sEquation += " + " + _nLeftTerm.toString();
            _sEquation += " = " + _nRightTerm.toString();
            break;
        case 1:
            _nSolutionValue = na + Math.floor(Math.random() * (nb - na + 1));
            _nSolCoefficient = Math.floor(Math.random() * 10);
            if (_nSolCoefficient == 0) _nSolCoefficient = 10;
            _nLeftTerm = 5-Math.floor(Math.random() * 10);
            _nRightTerm = _nSolCoefficient * _nSolutionValue + _nLeftTerm;
            _nRightTerm2 = Math.floor(Math.random() * 10)+1;
            _nSolCoefficient += _nRightTerm2;
            _sEquation = _nSolCoefficient.toString() + "X";
            if (_nLeftTerm < 0) _sEquation += _nLeftTerm.toString();
            else _sEquation += " + " + _nLeftTerm.toString();
            _sEquation += " = " + _nRightTerm.toString() + " + _nRightTerm2*X";
            break;
    }
}

public function GenerateWordProblem(nEquationType:Number):Void {
    var nRandomNum:Number;
    var sItemX: String;
    var sItemY: String;
    var sRandomPirateName: String;
    var sRandomRelationship: String;
    var sSayVerb: String;

    SetRange(1,20);
    ComputeEquation(nEquationType);
    _sDialog = "";

    if (_nRightTerm2<0) {
        // First kind of problem. X is only on left side
        _sWordProblem = "My ";
    }
APPENDIX B. WORD PROBLEM CLASS

```javascript
_sDialog = "My|"

nRandomNum = Math.floor(Math.random() * _aRelationship.length);
sRandomRelationship = _aRelationship[nRandomNum];
_sWordProblem += sRandomRelationship + " ";
_sDialog += sRandomRelationship + "|"

nRandomNum = Math.floor(Math.random() * _aPirateNames.length);
sRandomPirateName = _aPirateNames[nRandomNum];
_sWordProblem += sRandomPirateName + " ";
_sDialog += sRandomPirateName + "|"

nRandomNum = Math.floor(Math.random() * _aSayVerb.length);
sSayVerb = _aSayVerb[nRandomNum];
_sWordProblem += sSayVerb + " that "
_sDialog += sSayVerb + "|that|"

nRandomNum = Math.floor(Math.random() * _aItems.length);
sItemY = _aItems[nRandomNum];
if (_sVowels.indexOf(sItemY.charAt(0)) > -1) { _sWordProblem+="an "; _sDialog+="an|"; } else { _sWordProblem+="a "; _sDialog+="a|"; }
_sWordProblem += sItemY + " is " +ConvertNumber(_nSolCoefficient)+" ";
_sDialog += sItemY + "|is|"+ConvertNumberDash(_nSolCoefficient)+"| |

nRandomNum = Math.floor(Math.random() * _aMultiplicationWordsAfter.length);
_sWordProblem += _aMultiplicationWordsAfter[nRandomNum]+" the cost of ";
_sDialog += _aMultiplicationWordsAfter[nRandomNum]+"|the cost of| 

nRandomNum = Math.floor(Math.random() * _aItems.length);
sItemX = _aItems[nRandomNum];
//Make sure sItemX is not equal to sItemY
while (sItemY==sItemX) {
  nRandomNum = Math.floor(Math.random() * _aItems.length);
  sItemX = _aItems[nRandomNum];
}
if (_sVowels.indexOf(sItemX.charAt(0)) > -1) { _sWordProblem+="an "; _sDialog+="an|"; } else { _sWordProblem+="a "; _sDialog+="a|"; }
_sWordProblem += sItemX;
_sDialog += sItemX+"| ";

if (_nLeftTerm>0) {
  nRandomNum = Math.floor(Math.random() * _aAdditionWordsAfter.length);
  if (_aAdditionWordsAfter[nRandomNum]!="more") {
    _sWordProblem += " +" +ConvertNumber(_nAdditionWordsAfter[nRandomNum]+" ConvertNumber(_nLeftTerm)+" ";
    _sDialog += _aAdditionWordsAfter[nRandomNum]+"|+"+ConvertNumberDash(_nLeftTerm)+"| |
  } else {
    _sWordProblem=InsertTextAfter(_sWordProblem," is ",ConvertNumber(_nLeftTerm)+" more than ");
    _sWordProblem+=" ";
    _sDialog=InsertTextAfter(_sDialog,"is|",ConvertNumberDash(_nLeftTerm)+"|more than| ");
  }
  _sDialog+= "|
}
else
```
if (_nLeftTerm<0) {
    nRandomNum = Math.floor(Math.random() * _aSubstractionWordsAfter.length);
    if (_aSubstractionWordsAfter[nRandomNum] == "less") {
        _sWordProblem += " "+_aSubstractionWordsAfter[nRandomNum] + "\n"+ConvertNumber(Math.abs(_nLeftTerm)) + " \n";  
        _sDialog += _aSubstractionWordsAfter[nRandomNum] + "\n"+ConvertNumberDash(Math.abs(_nLeftTerm)) + "|\n";  
    } else {
        _sWordProblem += _aSubstractionWordsAfter[nRandomNum] + "\n" + ConvertNumber(Math.abs(_nLeftTerm)) + " \n";  
        _sDialog += _aSubstractionWordsAfter[nRandomNum] + "\n" + ConvertNumberDash(Math.abs(_nLeftTerm)) + "|\n";  
    }
    else { _sWordProblem += " "; _sDialog += " ";  
        _sWordProblem += "If the cost of ";  
        _sDialog += "If the cost of " ;
        if (_sVowels.indexOf(sItemY.charAt(0)) > -1) { _sWordProblem += "an "; _sDialog += "an " ; } else { _sWordProblem += "a "; _sDialog += "a " ; }  
        _sWordProblem += sItemY + " is " + ConvertNumber(_nRightTerm) + " \n";  
        _sDialog += sItemY + " is " + ConvertNumberDash(_nRightTerm) + "|\n";  
        if (_sVowels.indexOf(sItemX.charAt(0)) > -1) { _sWordProblem += "an " ; _sDialog += "an " ;  
            _sWordProblem += sItemX + "?" ;  
            _sDialog += sItemX + "?" ;  
        } else { _sWordProblem += " a " ; _sDialog += " a " ;  
            _sWordProblem += sItemX ;  
            _sDialog += sItemX ;  
    }
} else {  
    // Second kind of problem. X is on both sides  
    nRandomNum = Math.floor(Math.random() * _aIntroductionPhrase.length);  
    _sWordProblem += _aIntroductionPhrase[nRandomNum] + " ";  
    nRandomNum = Math.floor(Math.random() * _aPirateAdjectives.length);  
    _sWordProblem += _aPirateAdjectives[nRandomNum] + " ";  
    nRandomNum = Math.floor(Math.random() * _aPirateNames.length);  
    sRandomPirateName = _aPirateNames[nRandomNum];  
    _sWordProblem += sRandomPirateName + " had " ;  
    _sWordProblem += ConvertNumber(_nSolCoefficient) + " \n";  
    nRandomNum = Math.floor(Math.random() * _aMultiplicationWordsAfter.length);  
    _sWordProblem += _aMultiplicationWordsAfter[nRandomNum] + " the cost of "  
    nRandomNum = Math.floor(Math.random() * _aItems.length);  
    sItemX = _aItems[nRandomNum];  
    if (_sVowels.indexOf(sItemX.charAt(0)) > -1) { _sWordProblem += "an " ; 
        else _sWordProblem = "a " ;  
        _sWordProblem += sItemX ;  
        if (_nLeftTerm>0) {  
            nRandomNum = Math.floor(Math.random() * _aAdditionWordsAfter.length);
if (_aAdditionWordsAfter[nRandomNum]!="more")
    _sWordProblem += " + _aAdditionWordsAfter[nRandomNum] + " +ConvertNumber(_nLeftTerm) +" gold coins. ";
else {
    _sWordProblem=InsertTextAfter(_sWordProblem," had ",ConvertNumber(_nLeftTerm) +" more than ");
    _sWordProblem=" gold coins. ";
}
else
else if (_nLeftTerm<0) {
    nRandomNum = Math.floor(Math.random() * _aSubstractionWordsAfter.length);
    if (_aSubstractionWordsAfter[nRandomNum]!="less")
        _sWordProblem += " + _aSubstractionWordsAfter[nRandomNum] + " +ConvertNumber(Math.abs(_nLeftTerm)) +" gold coins. ";
    else {
        _sWordProblem=InsertTextAfter(_sWordProblem," had ",ConvertNumber(Math.abs(_nLeftTerm)) +" less than ");
        _sWordProblem=" gold coins. ";
    }
}
else _sWordProblem += " gold coins. ";

nRandomNum = Math.floor(Math.random() * _aPirateNames.length);
_nRandomPirateName = _aPirateNames[nRandomNum];
_sWordProblem += _nRandomPirateName + " possessed ";
_sWordProblem += ConvertNumber(_nRightTerm2) +" ";

nRandomNum = Math.floor(Math.random() * _aMultiplicationWordsAfter.length);
_sWordProblem += _aMultiplicationWordsAfter[nRandomNum] +" the cost of "
_sWordProblem += sItemX +" ";
//----------
if (_nRightTerm>0) {
    nRandomNum = Math.floor(Math.random() * _aAdditionWordsAfter.length);
    if (_aAdditionWordsAfter[nRandomNum]!="more")
        _sWordProblem += " + _aAdditionWordsAfter[nRandomNum] + " +ConvertNumber(_nRightTerm) +" gold coins. ";
    else {
        _sWordProblem=InsertTextAfter(_sWordProblem," possessed ",ConvertNumber(Math.abs(_nRightTerm)) +" more than ");
        _sWordProblem=" gold coins. ";
    }
}
else
else if (_nLeftTerm<0) {
    nRandomNum = Math.floor(Math.random() * _aSubstractionWordsAfter.length);
    if (_aSubstractionWordsAfter[nRandomNum]!="less")
        _sWordProblem += " + _aSubstractionWordsAfter[nRandomNum] + " +ConvertNumber(Math.abs(_nRightTerm)) +" gold coins. ";
    else {
        _sWordProblem=InsertTextAfter(_sWordProblem," possessed ",ConvertNumber(Math.abs(_nRightTerm)) +" less than ");
        _sWordProblem=" gold coins. ";
    }
}
else _sWordProblem += " gold coins. ";
else _sWordProblem += " gold coins. ";
//----------------
_sWordProblem += "If both have the same number of gold coins, what is the cost of 
if (_sVowels.indexOf(sItemX.charAt(0)) > -1) _sWordProblem="an "
else _sWordProblem="a "

_sWordProblem += sItemX+"?"
}
}

public function IsSolution(nSol:Number):Boolean { return nSol==_nSolutionValue; }

public function DiffToSolution(nSol:Number):Number { return _nSolutionValue-nSol; }

public function GetSolution():Number { return _nSolutionValue; }

public function WordProblem() {
    _sEquation = "";
    InitializePirateNames();
}

private function InsertTextAfter(sText:String, sDivisorText:String, sTextToInsert:String):String {
    var aTextArray = new Array;
    aTextArray=sText.split(sDivisorText);
    return aTextArray[0]+sDivisorText+sTextToInsert+aTextArray[1];
}

private function InsertTextBefore(sText:String, sDivisorText:String, sTextToInsert:String):String {
    var aTextArray = new Array;
    aTextArray=sText.split(sDivisorText);
    return aTextArray[0]+sTextToInsert+sDivisorText+aTextArray[1];
}

//Converts a given integer (in range [0..1T-1], inclusive) into alphabetical format ("one", "two", etc.).
private function ConvertNumber(nSourceNumber:Number) {
    var nGn:Number, nKn:Number, nHn:Number, nDn:Number, nOne:Number;
    var nRes:String;
    var aOnes:Array = ["", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten", "eleven", "twelve", "thirteen", "fourteen", "fifteen", "sixteen", "seventeen", "eighteen", "nineteen"];
    var aTens:Array = ["", "", "twenty", "thirty", "fourty", "fifty", "sixty", "seventy", "eighty", "ninety"];

    if ((nSourceNumber < 0) || (nSourceNumber > 999999999))
    {
        return "nSourceNumber";
    }

    nGn = Math.floor(nSourceNumber / 1000000); /* Millions (giga) */
    nSourceNumber -= nGn * 1000000;
    nKn = Math.floor(nSourceNumber / 1000); /* Thousands (kilo) */
    nSourceNumber -= nKn * 1000;
    nHn = Math.floor(nSourceNumber / 100); /* Hundreds (hecto) */
    nSourceNumber -= nHn * 100;
    nDn = Math.floor(nSourceNumber / 10); /* Tens (deca) */
nOne = nSourceNumber % 10;  /* Ones */  
nRes = "";
if (nGn)  
{  
    nRes += ConvertNumber(nGn) + " million";
}
if (nKn)  
{  
    nRes += ( nRes ? " " : "") + ConvertNumber(nKn) + " thousand";
}
if (nHn)  
{  
    nRes += ( nRes ? " " : "") + ConvertNumber(nHn) + " hundred";
}
if (nDn || nOne)  
{  
    if (nRes != "")  
    {  
        nRes += " and ";
    }
    if (nDn < 2)  
    {  
        nRes += aOnes[nDn * 10 + nOne];
    }
    else  
    {  
        nRes += aTens[nDn];
        if (nOne)  
        {  
            nRes += "-" + aOnes[nOne];
        }
    }
}
if (nRes == "") nRes = "zero";
return nRes  

private function ConvertNumberDash(nSourceNumber:Number) {  
    var sCurNum:String = ConvertNumber(nSourceNumber);  
    var aNum = new Array;  
    var sReturn:String="";  
    var sCurElem: String;  
    var i:Number;
    aNum=sCurNum.split(" ");  
    for(i=0;i<sNum.length;i++)  
    {  
        sCurElem=aNum[i];
        if(sCurElem==")  
        {  
            if (sCurElem.indexOf("-",0)<=0) sReturn+=aNum[i]+"|");
        else  
        {  
            var aTemp=new Array;  
            return sReturn;
        }
    }
}
APPENDIX B. WORD PROBLEM CLASS

```javascript
private function InitializePirateNames():Void {
    _aPirateNames[0]="Captain Thomas Anstis"
    _aPirateNames[1]="Barbarossa Brothers"
    _aPirateNames[2]="Bartolomeo"
    _aPirateNames[3]="Count Maurycy Beniovski"
    _aPirateNames[4]="Black Bart"
    _aPirateNames[5]="Black Beard"
    _aPirateNames[6]="Black Bellamy"
    _aPirateNames[7]="Stede Bonnet"
    _aPirateNames[8]="Anne Bonney"
    _aPirateNames[9]="Captain George Booth"
    _aPirateNames[10]="Captain Jan de Bofff"
    _aPirateNames[12]="Roche Brasiliano"
    _aPirateNames[13]="Captain Tobias Bridge"
    _aPirateNames[14]="Captain Enrique Browes"
    _aPirateNames[15]="Captain Nathaniel Butler"
    _aPirateNames[16]="Black Caesar"
    _aPirateNames[17]="Captain John Calles"
    _aPirateNames[18]="Roberto Coffea"
    _aPirateNames[19]="William Dampier"
    _aPirateNames[20]="Hovell Davis"
    _aPirateNames[21]="Benito de Soto"
    _aPirateNames[22]="Diabolitto"
    _aPirateNames[23]="Captain Peter Easton"
    _aPirateNames[24]="Captain Edward England"
    _aPirateNames[25]="Captain Henry Every"
    _aPirateNames[26]="Red Legs Greaves"
    _aPirateNames[27]="Henriques the Englishman"
    _aPirateNames[28]="Captain Benjamin Hornigold"
    _aPirateNames[29]="Calico Jack"
    _aPirateNames[30]="Captain Jackson"
    _aPirateNames[31]="Captain James Kelley"
    _aPirateNames[32]="Captain William Kidd"
    _aPirateNames[33]="Oliver le Bouch"
    _aPirateNames[34]="Francisco le Clerc"
    _aPirateNames[35]="Captain George Lourher"
    _aPirateNames[36]="David Harteen"
    _aPirateNames[37]="Captain Christopher Minga"
    _aPirateNames[38]="Sir Henry Morgan"
    _aPirateNames[39]="Captain John Morris"
    _aPirateNames[40]="Captain John Rut"n
    _aPirateNames[41]="Grace O'Malley"
    _aPirateNames[42]="Captain Thomas Paine"
    _aPirateNames[43]="Captain Lawrence Prince"
    _aPirateNames[44]="Captain Jack Rackam"
    _aPirateNames[45]="Mary Read"
    _aPirateNames[46]="Barbarossa"
    _aPirateNames[47]="Captain Manuel Pardel Rivero"
    _aPirateNames[48]="Captain Simon Simonson"
    _aPirateNames[49]="Captain Thomas Thewood"
    _aPirateNames[50]="Captain Charles Vane"
    _aPirateNames[51]="Captain John Yard"
    _aPirateNames[52]="Captain Richard Worley"
    _aPirateNames[53]="Guybrush Threepwood"
```
_aPirateNames[54]="Captain Hector Barbossa"
_aPirateNames[55]="Captain Jack Sparrow"
Appendix C

Assembly line simulator flow diagrams

Figure 69: Initialization
Figure 70: Main process

Figure 71: Walking input beam idle

Figure 72: Walking input beam move
Figure 73: Load arm idle

Figure 74: Load arm ready-to-load

Figure 75: Oven idle

Figure 76: Oven ready-to-load
APPENDIX C. ASSEMBLY LINE SIMULATOR FLOW DIAGRAMS

Figure 77: Oven process

Figure 78: Unload arm idle

Figure 79: Unload arm ready-to-unload
Figure 80: Walking output beam idle

Figure 81: Walking output beam move
Appendix D

Assembly line simulator source code

```
#define MAX_INPUT_BEAM 5
#define MAX_OUTPUT_BEAM 5
#define TOT_NUMPieces 8

/****** mtypes declarations **************/
mtype = { up, down }; /* door state */
mtype = { hold, release }; /* jaws state */
mtype = { idle, busy }; /* oven state */
mtype = { withpiece, withoutpiece, onhold }; /* arm1, arm2 states */
mtype = { load, unload }; /* beam messages */
mtype = { receivepiece, loadpiece, unloadpiece } /* arm messages */
mtype = { ready, notready }

/****** global variables **************/
mtype door = up;
mtype jaws = release;
mtype inputarm = withoutpiece;
mtype outputarm = withoutpiece;
mtype oven = idle;
mtype ovenstate = withoutpiece;

int NumPiecesInputBeam; /* current number of pieces in input beam */
int NumPiecesOutputBeam; /* current number of pieces in output beam */
int TotNumPiecesProcessed;

bit StopInputBeam; /* when 1, Input Beam stops */
bit StopOutputBeam; /* when 1, Output Beam stops */
bit StopArm1; /* when 1, Arm1 stops */
bit StopArm2; /* when 1, Arm2 stops */
bit StopOven; /* when 1, Oven stops */
bit StopMain; /* when 1, main stops */

chan loadarm1 = [0] of { mtype }; /* channel for message to arm1 */
chan loadarm2 = [0] of { mtype }; /* channel for message to arm2 */
chan loadbeam1 = [0] of { mtype }; /* channel for message to beam1 */
chan loadbeam2 = [0] of { mtype }; /* channel for message to beam2 */
chan loadoven = [0] of { mtype }; /* channel for message to oven */
```
never
{
  true;
  do
  :: (oven==busy) && (door==up) -> break
  :: (jaws==hold) && (door==up) -> break
  :: (NumPiecesInputBeam>MAX_INPUT_BEAM) -> break
  :: else
  od
}

proctype dowalkinginputbeam()
{
  int firstpiece=0;
  int lastpiece=0;

  Idle:
  if
  :: loadbeam1?load -> goto Movebeam
  :: (inputarm==withoutpiece) && (NumPiecesInputBeam>0) ->
      atomic {
        NumPiecesInputBeam--;
        firstpiece++;
        printf("<INPBEAM> Piece \[%d\] sent to ARM\n",firstpiece);
        loadarm1!receivepiece
      }
      goto Idle
  :: (1) && (!StopInputBeam) ->
      goto Idle
  :: (StopInputBeam) && (NumPiecesInputBeam==0) ->
      goto Completed
  fi;

  Movebeam:
  if
  :: (NumPiecesInputBeam<MAX_INPUT_BEAM) ->
    NumPiecesInputBeam++;
    lastpiece++;
    printf("<INPBEAM> Received new piece \[%d\]. Load is \%
",lastpiece,NumPiecesInputBeam);
    goto Idle
  :: (inputarm==withoutpiece) && (NumPiecesInputBeam>0) ->
      atomic {
        NumPiecesInputBeam--;
        firstpiece++;
        printf("<INPBEAM> Piece \[%d\] unloaded and sent to ARM\n",firstpiece);
        loadarm1!receivepiece
      }
      goto Movebeam
  :: else ->
      goto Movebeam
  fi;

  Completed:
  printf("<INPBEAM> Stopped. ***");
  StopArm1=1;
}

proctype dowalkingoutputbeam()
{
  int firstpiece=0;
  int lastpiece=0;

  Idle:
  if
  :: loadbeam2?receivepiece -> goto Movebeam
  :: (NumPiecesOutputBeam>0) ->
APPENDIX D. ASSEMBLY LINE SIMULATOR SOURCE CODE

```
atomic {
  firstpiece++;
  NumPiecesOutputBeam--;  
  printf("<OUTBEAM> Piece [%d] unloaded\n",firstpiece);
}
goto Idle
:: (1) && (!StopOutputBeam) ->
goto Idle
:: (StopOutputBeam) ->
goto Completed
fi;
Movebeam:
if
:: (NumPiecesOutputBeam<MAX_INPUT_BEAM) ->
  NumPiecesOutputBeam++;
  lastpiece++;
  outputarm=withoutpiece;
  printf("<OUTBEAM> Piece [%d] received from arm2. OUTBEAM load is %d piece(s)\n",lastpiece,NumPiecesOutputBeam);
  goto Idle
:: (NumPiecesOutputBeam>0) ->
  atomic {
    firstpiece++;
    NumPiecesOutputBeam--;
    printf("<OUTBEAM> Piece [%d] unloaded\n",firstpiece)
  }

  if
:: (!StopOutputBeam) || (NumPiecesOutputBeam>0) ->
goto Movebeam
:: else
  goto Completed
fi
:: else ->
goto Movebeam
fi;
Completed:
printf("<OUTBEAM> Stopped **\n");
StopMain=1
}

proctype doarm1()
{
  int piece=0;

  Idle:
  if
:: loadarm1?receivepiece ->
  atomic {
    inputarm=withpiece;
    piece= piece++;
    printf("<ARM1> Received piece: [%d]\n",piece);
    goto ReadyToLoad
  }
:: (1) && (!StopArm1) ->
goto Idle;
:: StopArm1 ->
goto Completed
fi;
ReadyToLoad:
if
:: (door=up) && (oven=idle) && (ovenstate=withoutpiece) && (jaws=release) -->
  atomic {
    printf("<ARM1> Loaded piece [%d] into oven\n",piece);
```
ovenstate=withpiece;
inputarm=onhold;
loadoven!load;
goto Idle
}
:: else ->
goto ReadyToLoad
fi;
Completed:
printf("<ARM1> Stopped \n");
StopOven=1
}
proctype doarm2()
{
int piecenumber=0;

Idle:
if
:: loadarm2?receivepiece -> goto ReadyToUnload
:: (outputarm==withpiece) && (NumPiecesOutputBeam<MAX_OUTPUT_BEAM) ->
    atomic {
        printf("<ARM2> Loading piece [%d] into output beam\n",piecenumber);
        loadbeam2!receivepiece;
        goto Idle;
    }
:: (1) && (!StopArm2) ->
goto Idle
:: StopArm2 ->
goto Completed
fi;
ReadyToUnload:
if
:: (outputarm==withoutpiece) ->
    atomic {
        ovenstate=withoutpiece;
        outputarm=withpiece;
        jaws=release;
        oven=idle;
        door=up;
        piecenumber++;
        printf("<ARM2> Received piece [%d] from oven\n",piecenumber);
        goto ReadyToUnload
    }
:: (outputarm==withpiece) && (NumPiecesOutputBeam<MAX_OUTPUT_BEAM) ->
    atomic {
        outputarm=onhold;
        printf("<ARM2> Loading piece [%d] into output beam\n",piecenumber);
        loadbeam2!receivepiece;
        goto Idle;
    }
:: else ->
goto ReadyToUnload
fi;
Completed:
printf("<ARM2> Stopped \n");
StopOutputBeam=1
}
proctype dooven()
{
int piecenumber=0;

Idle:
if
  :: loadoven?load ->
    goto ReadyToLoad
  :: StopOven ->
    goto Completed
fi;

ReadyToLoad:
if
  :: (ovenstate=withpiece) && (oven=idle) && (door=up) ->
    atomic {
      inputarm=withoutpiece;
      door=down;
      oven=busy;
      piecenumber++;
      printf("<OVEN> Ready to Process piece \[%d\]", piecenumber);
      goto Process
    }
  :: (!StopOven) ->
    goto ReadyToLoad
  :: (StopOven) ->
    goto Completed
fi;

Process:
if
  :: (jaws=release) ->
    jaws=hold;
    goto Process;
  :: (jaws=hold) ->
    atomic {
      printf("<OVEN> Processing piece \[%d\]", piecenumber);
      printf("<OVEN> Piece \[%d\] processed\n", piecenumber);
      loadarm2!receivepiece;
      goto Idle
    }
  :: else ->
    goto Process
fi;

Completed:
printf("<OVEN> Stopped **\n");
StopArm2=1

active proctype main() {
  atomic {
    /* Initialize variables */
    door=up;
    oven=idle;
    jaws=release;
    inputarm=withoutpiece;
    outputarm=withoutpiece;
    NumPiecesInputBeam=0;
    NumPiecesOutputBeam=0;
    TotNumPiecesProcessed=0;
    StopInputBeam=0;
    StopOutputBeam=0;
    StopArm1=0;
    StopArm2=0;
    StopMain=0;
StopOven=0
}

run dowalkinginputbeam();
run doarm1();
run dooven();
run doarm2();
run dowalkingoutputbeam();

Process:
if
:: (NumPiecesInputBeam<MAX_INPUT_BEAM) && (TotNumPiecesProcessed<TOT_NUM_PIECES) ->
   loadbeam1!load;
   TotNumPiecesProcessed**;
   goto Process
:: (TotNumPiecesProcessed=TOT_NUM_PIECES) ->
   goto EndExecution
:: (TotNumPiecesProcessed>TOT_NUM_PIECES) ->
   goto Process
fi;
EndExecution:

printf("<MAIN> Sending end signal to processes...\n");
StopInputBeam=1;

Waiting:
if
:: (StopMain) ->
   goto Completed
:: else ->
   goto Waiting
fi;
Completed:
printf("<Main> Successful Completion. Total %d piece(s) ** \n",TotNumPiecesProcessed);
Appendix E

Source code and executable files

The entire source code and executable files of the software systems described in this work are contained in the companion CD.
List of Tables and Figures
List of Tables

1  Number of graceful labeling of cycles for different size \((n)\) and missing label \((M)\) ................................. 23
2  Performance comparison .................................................. 23
3  States of mtype ............................................................... 93
4  Global variables definitions .............................................. 94
List of Figures

1 Seven Bridges of Königsberg ............................................. 13
2 Graph representation of the problem .................................... 13
3 Sample $SUN(5, 2)$ ........................................................... 15
4 Molecular graphs ............................................................ 16
5 Planar graph example ....................................................... 16
6 Three houses and three utilities problem ............................... 17
7 Example polygons ............................................................ 17
8 A graceful labeling of $C_8$ ................................................ 20
9 Manohar’s GUI Sections .................................................... 24
10 Manohar buttons ............................................................ 25
11 Algorithm menu ............................................................. 26
12 Path enumeration sample output ....................................... 27
13 Cycle enumeration sample output .................................... 27
14 Sun enumeration sample output ....................................... 28
15 Execution branches ......................................................... 30
16 Execution Tree ............................................................. 33
17 Sublabeling $S_7$ of $f$ ...................................................... 33
18 Case i ......................................................................... 34
19 Case ii ....................................................................... 35
20 Case iii ...................................................................... 36
21 Colossus screenshot ......................................................... 38
22 Colossus buttons ............................................................ 38
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>Resulting effect after clicking on create polygon button for a polygon of 20 endpoints</td>
<td>39</td>
</tr>
<tr>
<td>24</td>
<td>Moving endpoints when visibility mode is enabled</td>
<td>40</td>
</tr>
<tr>
<td>25</td>
<td>Visibility in orthogonal mode</td>
<td>40</td>
</tr>
<tr>
<td>26</td>
<td>Graph algorithm constructor's main window</td>
<td>42</td>
</tr>
<tr>
<td>27</td>
<td>Graph algorithm constructor's runner window</td>
<td>43</td>
</tr>
<tr>
<td>28</td>
<td>Prim minimum spanning tree</td>
<td>45</td>
</tr>
<tr>
<td>29</td>
<td>Before execution of planarity drawing</td>
<td>46</td>
</tr>
<tr>
<td>30</td>
<td>After execution of planarity drawing</td>
<td>46</td>
</tr>
<tr>
<td>31</td>
<td>Document window</td>
<td>54</td>
</tr>
<tr>
<td>32</td>
<td>A movie clip of ball and a button</td>
<td>56</td>
</tr>
<tr>
<td>33</td>
<td>Ball after button is clicked</td>
<td>57</td>
</tr>
<tr>
<td>34</td>
<td>Pirate Store Development Environment</td>
<td>58</td>
</tr>
<tr>
<td>35</td>
<td>Pirate Math Introduction Screen</td>
<td>64</td>
</tr>
<tr>
<td>36</td>
<td>Pirate Math Main Menu</td>
<td>65</td>
</tr>
<tr>
<td>37</td>
<td>Coin toss intro</td>
<td>66</td>
</tr>
<tr>
<td>38</td>
<td>Coin toss playing</td>
<td>67</td>
</tr>
<tr>
<td>39</td>
<td>Coin toss sample question</td>
<td>67</td>
</tr>
<tr>
<td>40</td>
<td>Pirate Gold introduction</td>
<td>68</td>
</tr>
<tr>
<td>41</td>
<td>Pirate Gold sample screenshot</td>
<td>69</td>
</tr>
<tr>
<td>42</td>
<td>Pirate Math Treasure Map introduction</td>
<td>70</td>
</tr>
<tr>
<td>43</td>
<td>Pirate Math Treasure Map sample screenshot</td>
<td>71</td>
</tr>
<tr>
<td>44</td>
<td>Pirate Store introduction</td>
<td>72</td>
</tr>
<tr>
<td>45</td>
<td>Pirate Store</td>
<td>72</td>
</tr>
<tr>
<td>46</td>
<td>Pirate Store shelf</td>
<td>73</td>
</tr>
<tr>
<td>47</td>
<td>Chemistry Circus introductory screen</td>
<td>74</td>
</tr>
<tr>
<td>48</td>
<td>Chemistry Circus demo, screen 1</td>
<td>75</td>
</tr>
<tr>
<td>49</td>
<td>Chemistry Circus demo, screen 2</td>
<td>76</td>
</tr>
<tr>
<td>50</td>
<td>Chemistry Circus play screen</td>
<td>77</td>
</tr>
<tr>
<td>51</td>
<td>Cellular Divide and Conquer main menu</td>
<td>78</td>
</tr>
<tr>
<td>52</td>
<td>Cellular Divide and Conquer main menu</td>
<td>79</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>53</td>
<td>Cellular Divide and Conquer help screen 1</td>
<td>79</td>
</tr>
<tr>
<td>54</td>
<td>Cellular Divide and Conquer help screen 2</td>
<td>80</td>
</tr>
<tr>
<td>55</td>
<td>Introductory screen</td>
<td>81</td>
</tr>
<tr>
<td>56</td>
<td>Interphase introduction</td>
<td>82</td>
</tr>
<tr>
<td>57</td>
<td>Interphase game play</td>
<td>82</td>
</tr>
<tr>
<td>58</td>
<td>Prophase game play</td>
<td>82</td>
</tr>
<tr>
<td>59</td>
<td>Metaphase game play</td>
<td>83</td>
</tr>
<tr>
<td>60</td>
<td>Anaphase game play</td>
<td>83</td>
</tr>
<tr>
<td>61</td>
<td>Telophase game play</td>
<td>84</td>
</tr>
<tr>
<td>62</td>
<td>Telophase game play</td>
<td>84</td>
</tr>
<tr>
<td>63</td>
<td>Assembly line machine</td>
<td>92</td>
</tr>
<tr>
<td>64</td>
<td>Assembly line model</td>
<td>93</td>
</tr>
<tr>
<td>65</td>
<td>Variables initialization</td>
<td>94</td>
</tr>
<tr>
<td>66</td>
<td>Main Process</td>
<td>95</td>
</tr>
<tr>
<td>67</td>
<td>Walking input beam idle state</td>
<td>97</td>
</tr>
<tr>
<td>68</td>
<td>Walking input beam move state</td>
<td>97</td>
</tr>
<tr>
<td>69</td>
<td>Initialization</td>
<td>120</td>
</tr>
<tr>
<td>70</td>
<td>Main process</td>
<td>121</td>
</tr>
<tr>
<td>71</td>
<td>Walking input beam idle</td>
<td>121</td>
</tr>
<tr>
<td>72</td>
<td>Walking input beam move</td>
<td>121</td>
</tr>
<tr>
<td>73</td>
<td>Load arm idle</td>
<td>122</td>
</tr>
<tr>
<td>74</td>
<td>Load arm ready-to-load</td>
<td>122</td>
</tr>
<tr>
<td>75</td>
<td>Oven idle</td>
<td>122</td>
</tr>
<tr>
<td>76</td>
<td>Oven ready-to-load</td>
<td>122</td>
</tr>
<tr>
<td>77</td>
<td>Oven process</td>
<td>123</td>
</tr>
<tr>
<td>78</td>
<td>Unload arm idle</td>
<td>123</td>
</tr>
<tr>
<td>79</td>
<td>Unload arm ready-to-unload</td>
<td>123</td>
</tr>
<tr>
<td>80</td>
<td>Walking output beam idle</td>
<td>124</td>
</tr>
<tr>
<td>81</td>
<td>Walking output beam move</td>
<td>124</td>
</tr>
</tbody>
</table>
Bibliography


