Paint-By-Numbers Puzzle Game & Solver

Chanwit Suebsureekul

CS491B - Spring 2006

California State University, Los Angeles

June 6, 2006

Contents

Abstract

Paint by Numbers puzzle is a game of logic and deduction. The goal of solving the puzzle is to find a hidden image associated to the puzzle. To do so, a player uses a set of numbers attached to each puzzle as clues. Several techniques can be utilized to solve the puzzles. Some puzzles can be solved easily while the others might involve diverse advanced approaches. As a result, many people found Paint by Numbers puzzle attractive and challenging. Consequently, Paint by Numbers becomes prominent amongst puzzlers around the world.

The system proposed here employs Java 2 Platform, Standard Edition 5.0 to develop a software that can solve Paint by Numbers puzzles. It also provides the game interface allowing users to try solving the puzzles manually by interacting with the program. Furthermore, it offers various tools that can help users create their own puzzles, either from scratch or from regular image files.

1. Introduction

Many years ago, game industry shared only a slightly portion of software market. It was considered a child's play and not many companies seriously paid attention to it. For the past few years, however, it has become an attractive issue in software industry and gained much attention from software developers. Even Microsoft, the gigantic software company, has invested in the game market, venturing a lot of money on the Xbox project.

A game can be categorized into various genres: action, adventure, fighting, puzzle, RPG (role-playing game), shooting, simulation, etc. Normally, each type of game is apposite for a certain group of people. The puzzle game, on the other hand, is generally suitable for everybody: no matter how old or what gender they are. One of such puzzles is "Paint by Numbers."

Paint by Numbers is a picture logic puzzle. Each puzzle contains an empty grid and various sets of numbers associated to each row and column of the grid. Behind the grid lies a hidden monochrome picture. Hence, the goal of solving the puzzle, in addition to challenging the player's smartness, is to uncover the concealed image. The numbers on each row and column serve as "clues" telling the player how to paint the puzzle, which is discussed in more details in Section 1.2 (How to play). Figure 1.1 illustrates an example of a Paint by Numbers puzzle.

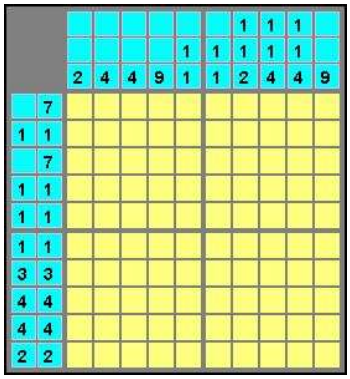

Figure 1.1 – Paint by Numbers puzzle [8]

1.1 History

The concept of Paint by Numbers puzzle was introduced in 1987 by two unrelated Japanese people: Non Ishida, a Japanese graphics editor, and Tetsuya Nishio, a professional Japanese puzzler. A few years later, magazines with Paint by Numbers puzzles started appearing in Japan, the United Kingdom, the United States, Sweden, South Africa, and some other countries. Since they have constantly gained popularity, Paint by Numbers puzzles are presently published worldwide in many magazine titles. In addition, some magazines have invented new names for Paint by Numbers puzzles to use in their magazines. Consequently, Paint by Numbers puzzles are also known by many other names, such as Nonograms, Griddlers, Edel, Tsunami, Pic-a-Pix, Japanese Crosswords, Illust-Logic, CrossPix, Picture Logic, etc. [9]

Not only were they published in magazines, but Paint by Numbers puzzles were also made into console games in 1995 by Nintendo under the name Mario's Picross (Picture Crossword). Nintendo released two Picross titles for Game Boy system and nine for Super Famicom, eight of which were released in a two-month interval! Additionally, in 1996 Deniam Corporation released the puzzles for Arcade game system using the name Logic Pro. Logic Pro was made into Play Station system in the later years as well. [9]

1.2 How to play

As illustrated in Figure 1.1 above, there is a set of numbers associated to each row and column of the puzzle. (For simplification, we will refer to a row or a column as a "line" onwards.) These numbers will help you draw the puzzle and reveal the obscured image. How? Each number tells how many contiguous black cells there are on that line. We also call a group of adjacent black cells a "block." In other words, each number represents a block of

black cells. Figure 1.2 shows the solution of the first column containing a 5, resulting in a block of five consecutive black boxes.

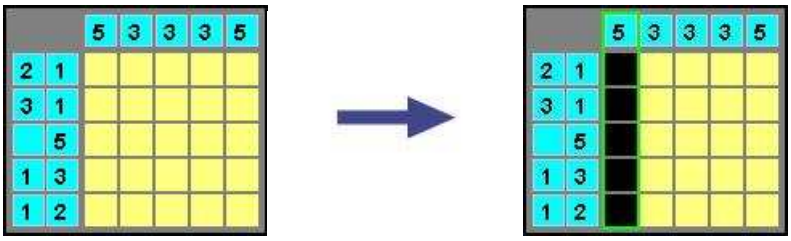

Figure 1.2 – Example of how to paint the puzzle (1)

If two or more numbers exist in the same line, resulting in two or more blocks, there must be "at least" one white cell between each block. Note that we do not know how many white cells exist between each block, but we "do" know that it must be at least one to break the black blocks apart from each other. Consider the second rows from the previous puzzle: it contains 3 and 1. Hence, the solution of this line has two blocks with the size of three and one separated by a white cell as shown in Figure 1.3. Also note that all the blocks in the same line must appear in the same sequence as the numbers. For instance, the second and the fourth rows have the same set of numbers, 1 and 3, but they come in the reverse order. Consequently, the solutions of these two lines are contradictory.

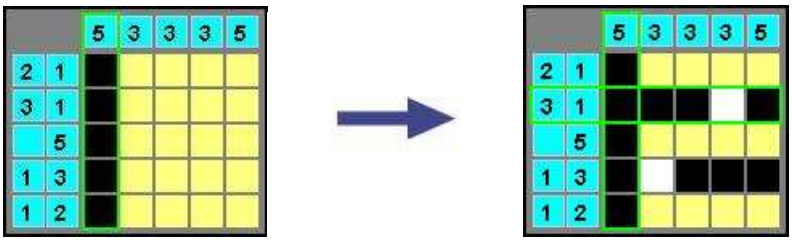

Figure 1.3 – Example of how to paint the puzzle (2)

Now that we have learned all the fundamental rules, if we continue solving the puzzle based on these bases, we will finally find the solution as illustrated in Figure 1.4. It's the letter N!

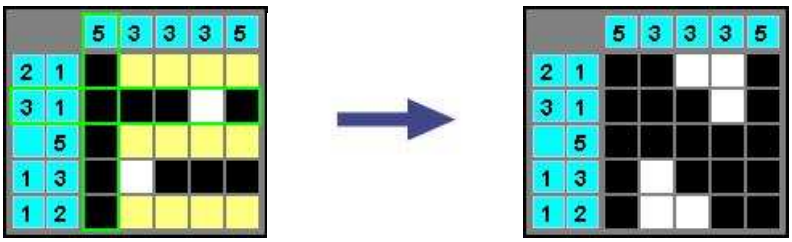

Figure 1.4 – The solution of the puzzle

1.3 Motivation

As stated before, Paint by Numbers puzzles were made into the Super Famicom console game system by Nintendo in 1990's. We instantly fell in love with these puzzles the first time we saw them, and that was the original incentive of this project.

Having been solving countless of puzzles so far, we have become familiar with the puzzles and developed various techniques that can be used to deal with them. Then, here come the interesting questions: "What about the computer?" "Can we make them solve the puzzles?" "Can we apply the same methods that we use into the programming language and make the computer think the same way that we 'humans' do?" "Is mathematics involves? Do we need any kind of arithmetic formula?"

2. Technological Background

The only major technology that we use in this project is the Java technology. The Java technology consists of both a programming language and a platform. The Java programming language is a high-level object oriented programming language used to create a program. The Java Platform, on the other hand, is an environment in which a program runs. [3]

In the Java programming language, a programmer writes the codes in plain text files. The Java compiler is then used to compile these files into Java bytecodes. The bytecodes are not native codes of the operating system (OS), but machine codes of the Java Virtual Machine (JVM). Thus, these bytecodes cannot be executed directly by the OS. The JVM, instead, will translate them into native codes, and pass them to the OS. Accordingly, we can simply write a program on one system, compile it into bytecodes, and then ideally run the program on any system provided that the JVM is available on that machine. This concept resulted in Sun Microsystems' prominent trademark: "Write once, run anywhere!"

The Java platform is a software-based platform running on top of other hardware-based platforms (Microsoft Windows, Linux, MacOS, etc.) It acts as an intermediate between the Java programming language and the hardware-based platform. The JVM is one of its components. Another component is the Java Application Programming Interface (API), which is basically a collection of ready-for-use software libraries. [1]

The primary reason that Java was chosen as the programming language for this project is that we need to practice and learn more about Java language. Also, Java's multi-platform capability provides us a broad area of customers. Furthermore, we can compile the Java source codes into Java applets easily, probably only a few lines of codes needed to be

modified. With Java applets, people around the world can access our program conveniently via their favorite web browsers.

2.1 Java 2 Platform, Standard Edition 5.0

The Java platform is categorized into four different platforms: Java SE (Java Platform, Standard Edition), Java EE (Java Platform, Enterprise Edition), Java ME (Java Platform, Micro Edition), and Java Card. Each platform is specialized for particular development and deployment environments. Out of these platforms, the Java SE fits our project's requirements the most. We therefore employ Java 2 Platform, Standard Edition 5.0 (J2SE 5.0) as the developing environment for this project. Comparing to the previous version, many new features have been added into J2SE 5.0, such as Metadata, Generic Types, Autoboxing and Auto-Unboxing, Enhanced `for` Loop, Enumerated Types, Static Import, Formatted Input & Output, Varargs, Concurrency Utilities, and Core XML Support. [2]

The latest version of Java SE, as of today, is Java SE 6 Beta. Since it is still a beta version and has just been released for awhile, we do not know much about it, and, thus, do not have any plan to migrate our project to this new platform. However, theoretically, it should be backward compatible with J2SE 5.0 and our project should be able to be compiled and run on this new platform as well, hopefully without any major modification.

2.2 Javagram

Javagram is a Nonograms (one of Paint by Numbers' synonyms) puzzle solver Java package written by Steven Simpson. [4] Perhaps, the name Javagram is derived from the combination of the words Java and Nonograms. Not only did he write the puzzle solver program in Java, Steven Simpson coded the solver library in C language as well. All of these programs are open source software and are released free of charge under the GNU Lesser General Public License as published by the Free Software Foundation.

The Javagram package, however, cannot solve all puzzles. It might return an incomplete solution if the input puzzle belongs to one of the following circumstances: it has any conflict between the numbers on the rows and those on the columns, has more than one feasible solution, or has exactly one solution but require a guessing method. We discuss about the algorithm used in the Javagram and how to make the algorithm complete in section 4.3 (Puzzle Solver).

Apart from the solving algorithm, Javagram also provides two basic data types representing the key components in the puzzle: the `Grid` class represents the grid area of the puzzle while the `Puzzle` class represents all the clues associated to the puzzle. In order to utilize the solving algorithm of the Javagram package, we need to create objects of these two classes. Since they are pretty well designed, we then employ these two classes in our game system and other components throughout our project as well.

In addition, Javagram defines the format of a text file into which the puzzle information can be saved. We discuss about this file format in section 4.2 (Puzzle Editor).

2.3 Java Advanced Imaging

Java Advanced Imaging (JAI) is a Java API that provides rich functionalities on advanced image processing, allowing developers to perform sophisticated tasks on image data without the necessity of low-level background knowledge on image processing concepts. [10] JAI supports various standard image file formats including BMP, FlashPix, GIF, JPEG, PNG, PNM, TIFF, and WBMP. Currently, the latest version is JAI 1.1.3-beta while the version we use in our project is JAI 1.1.2_1.

In this project, we use JAI to read image data from files, resize the image so that it fits the puzzle's size, and then convert the image into monochrome. This feature helps users create a new puzzle from an image file within only a few mouse clicks.

3. System Overview

The primary goal of this project is to create a program that can solve any Paint by Numbers puzzle, or at least notify the users if the puzzle is invalid, meaning that either there are clue conflicts in the puzzle or the puzzle has more than one solution. The second objective is to provide the graphical game interface and tools for users who want to solve puzzles interactively through the program. The last goal is to provide tools that let users create their own puzzles and find the solutions as well.

Four main visible components exist in the system: the main menu, the puzzle editor, the game interface, and the puzzle browser. There is also another important component, the puzzle solver, which will not be seen by the user. The main menu asks the user whether he wants to create a new puzzle or load it from an existing file. The puzzle editor allows the user to create or edit the puzzle. Then the puzzle solver will be used to find the solution of the puzzle. If the puzzle has exactly one solution, the game interface will appear, and let the user

play the game, trying to solve the puzzle. In contrast, if the puzzle has more than one solution, the puzzle browser will show up instead. The puzzle browser tells the user how many solutions can be generated from this particular set of numbers and lets the user navigate through each of them one by one.

4. Design and Implementation

As mentioned in the previous section, there are five major components in this project. We discuss about each module in more details in the following sections.

4.1 Main Menu

The overall process of the program begins with the main menu, which is divided into two categories. The user can choose whether to create a new puzzle or open a puzzle from a saved file. To create a new puzzle, the users must specify the size of the newly being created puzzle too. Also note that, two methods can be used to create a puzzle, either from numbers or from a painting. Figure 4.1 illustrates the main menu GUI.

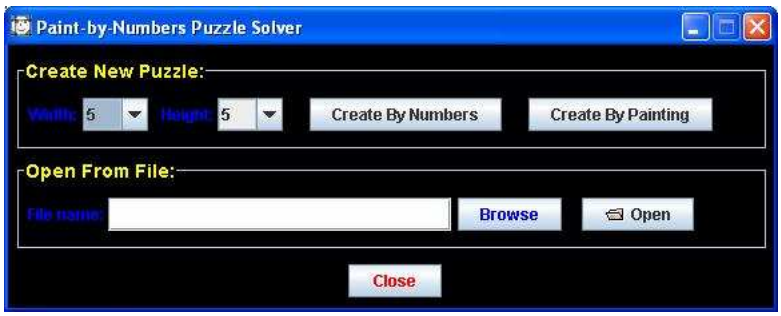

Figure 4.1 – The main menu GUI

Basically, the main menu acts as the gateway to the puzzle editor. It has basic GUI implemented by Swing components. The button "Browse" lets the user browse for a puzzle file using a `JFileChooser` object. We also create a `FileFilter` class, called `PBNFileFilter`, which will be used in the `JFileChooser` object to filter for only .pbn, .non, and .txt files since this will help the user browse for the puzzle files more quickly. We discuss about the format of the text file in more details in the next section. Note that pbn is short for Paint By Numbers, which is the name of this project, and that is why we add it in the list of supported file extensions. We also include .non in the list because it is originally supported in the Javagram package.

4.2 Puzzle Editor

If the user chooses to create a puzzle from numbers, a puzzle editor window will pop up and let the user fill in the numbers for each row and column. Creating a puzzle from a painting, in contrast, does not let the user input any number directly, but the user can paint an image in the grid area of the puzzle using the mouse, and the program will convert it into numbers automatically. Moreover, the user can import an image file into the grid area using the "Load Image" button, only provided in the "painting" mode. The program supports most of standard image file formats as stated in section 2.3 (Java Advanced Imaging).

We divide the GUI of the puzzle editor into two parts. The top part uses basic Swing components to create a collection of `JButtons` used as the controller of the puzzle editor window. In the middle, we use a `JPanel` object to paint the grid and the clues of the puzzle as shown in Figure 4.2. Basically, each grid is an instance of `JLabel` class, and each clue is an instance of `JTextField` class. We design our program this way because it is easy to implement. We can conveniently catch an event on each object and have it interact to that specific event. For example, if the user clicks the left button mouse on a grid cell, we will have that cell change the background color to black. This way we do not need to worry about the calculation of the mouse pointer position. We can also paint the background color of each box easily without the need to compute for the x and y coordination, the width, and the height of the area we want to paint. For the clue cells, since each cell is a `JTextField` object, we can obtain the capability of text editing automatically. Consequently, we can let the user edit the number on each cell, and simply call the method `getText()` on that object to read the user input. This helps us develop the program a lot more conveniently.

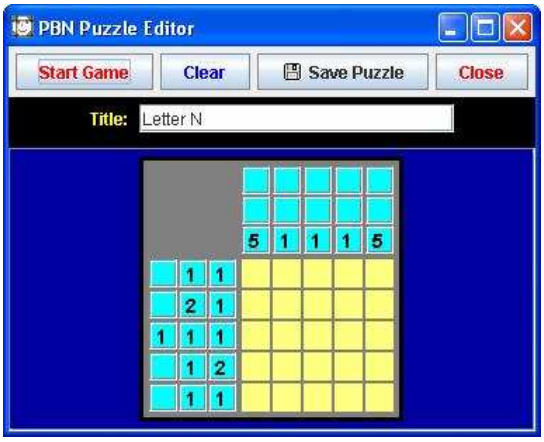

Figure 4.2 – The puzzle editor GUI (edit by numbers mode)

Figure 4.2 shows the GUI of the puzzle editor in the "numbers" mode. In the painting mode, however, is slightly different from the numbers mode. First of all, it does not let the user edit the clues. Instead, it lets the user paint in the grid area. Besides, it has one more button called "Load Image". This button lets the user browse for an image file using a `JFileChooser` object, which is similar to the browse button on the main menu. Nonetheless, this time we use a `PBNFileFilter` to filter for image files such as .bmp, .gif, .png, .jpg, and .jpeg. After the user have selected a file, we use JAI discussed in section 2.3 (Java Advanced Imaging) to read the image data, resize the image to fit the puzzle's size by processing those data, convert them into grayscale, convert the grayscale image into a monochrome, and have the program read the final data pixel by pixel and paint them on the grid area.

In the puzzle editor window, the user can also save the puzzle into a text file, which can be opened for a later use from the main menu. The text file format is originally defined in the Javagram package, and we simply adopt it into our system because it is simple and easy to understand. Figure 4.3 compares a puzzle file to its corresponding game interface.

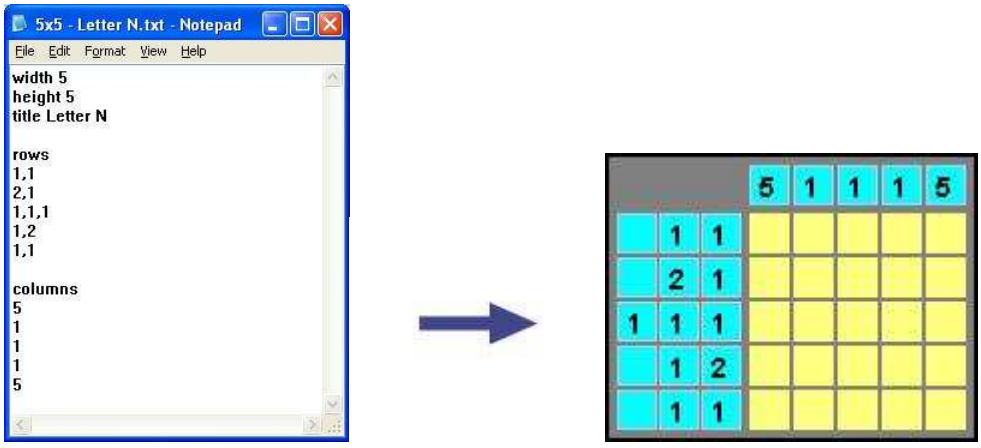

Figure 4.3 – A puzzle file and its corresponding game interface

As shown in Figure 4.3, the keyword "width" and "height" are followed by numbers used to specify the size of the puzzle. The keyword "title" is used to indicate the title of the puzzle. Then, the keyword "rows" tells the program that the following lines contain the clues of the puzzle on the left side of the grid. Each line represents the clues for one row starting from the top of the puzzle. The same thing applies to the keyword "columns", meaning that each line represents the clues for one column starting from the left of the puzzle. Two or more numbers in the same line are separated by commas.

4.3 Puzzle Solver

The puzzle solver is used to find the solution of the puzzle that the user creates or edits. We use the Javagram package discussed in section 2.2 (Javagram) as the primary algorithm to solve the puzzle. However, as stated before, the Javagram is incomplete. We, consequently, apply a technique called “recursive guessing” to make the solving algorithm perfect. The following sections discuss about the Javagram algorithm and the recursive guessing method in more details.

4.3.1 Javagram Algorithm

The Javagram does not solve the whole puzzle at once. Instead, it solves the puzzle one line at a time using an object of `LineSolver` class. The LineSolver class has two subclasses called `FastLineSolver` and `CompleteLineSolver`. The `FastLineSolver` solves a line by trying to push all blocks as far as possible to the left of the line, without opposing the rules of the game and the solved cells. It then does the same thing, but in the reverse direction. Finally, it looks for overlaps between both pushing actions. Figure 4.4 shows an example of this algorithm.

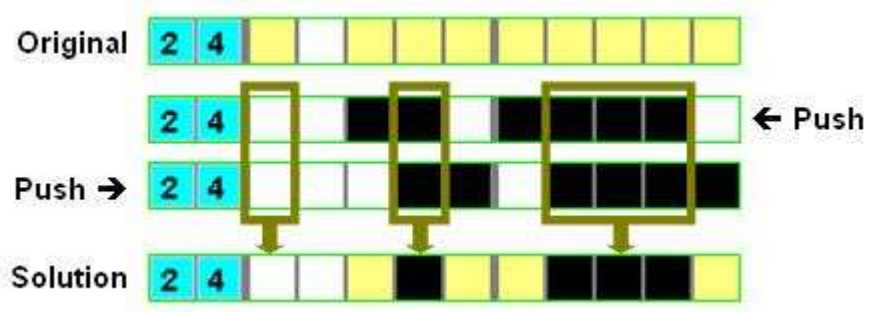

Figure 4.4 – An example of the FastLineSolver algorithm

The `CompleteLineSolver` thoroughly tries all possibilities to paint a line according to its clue numbers. If, for any combination, a cell can only be black, then it will be determined as black. If it can only be white, it will be resolved to white. We can think of this algorithm as a kind of brute-force algorithm.

Comparing to the `CompleteLineSolver`, the `FastLineSolver` algorithm is fast, but it might not “best” extract information from a given line. This means that it might leave some cells that should be solved unsolved as illustrated in Figure 4.5. The `CompleteLineSolver` algorithm, in contrast, guarantees that a line will be solved as best

as possible according to the available clues, but it is rather slow. [5] Therefore, to solve a puzzle, the Javagram first applies the `FastLineSolver` to solve as many possible cells as it can. Then it uses the `CompleteLineSolver` to solve the remaining cells. This is to minimize the use of the `CompleteLineSolver` algorithm since it consumes quite a large amount of time.

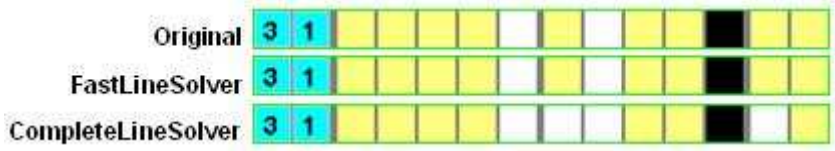

Figure 4.5 – Compare the solutions from both algorithms on a sample line

4.3.2 Recursive Guessing Algorithm

As mentioned earlier, the Javagram algorithm might return an incomplete solution, leaving some cells unsolved, if the puzzle has any contradiction, has more than one solution, or has one solution but required a guessing technique. We use a technique called recursive guessing to solve these problems.

The recursive guessing acts as an intermediate between the main program and the Javagram. Instead of returning the solution directly from the Javagram to the program, the recursive guessing takes the returned solution from the Javagram, evaluate the solution, decides whether the solving process is complete or not, and then takes an appropriate action. If the Javagram return an incomplete solution, leaving some cells unsolved, and there is no contradiction in the solving process, the recursive guessing will randomly pick an unsolved cell, make a guess, and send the solution with the new information back to the Javagram to continue solving the puzzle. Note that, in this project, we always look for an unsolved cell starting from the first row, and from the left column to the right column. Also note that, we always guess “black” rather than “white” because, empirically, a black cell usually gives more information to continue solving the puzzle than a white cell.

Apparently, after making the first guess, it is possible that the Javagram might return an incomplete solution again. If this happens, the recursive guessing will make another guess on another cell and follow the same algorithm. If, finally, the Javagram returns a complete solution, the returned solution will be added into an `ArrayList`, which is used to store all feasible solutions of the puzzle. At this point, the recursive guessing algorithm will roll back to the state before it made the latest guess, change the guessed value, and have the Javagram solve this new information as well. In the case that there is more than one solution, a

particular cell could be either black or white, and that is why we need to roll back and try another guess even though we have already found a solution from the previous guessed value. In other words, we must exhaust all possible paths in order to achieve all feasible solutions of the puzzle. When it rolls back, if the latest guessed cell has used up all the guessed value (black and white), the algorithm will roll back to the guessed cell before the most recently guessed cell, and try another guess from that cell instead.

If the Javagram finds a contradiction, the recursive guessing algorithm will also roll back, change the guessed value, and try another path in the same manner as when it finds a complete solution. The only difference is that no solution will be inserted into the solution `ArrayList`.

After the recursive guessing algorithm exhausts all possible paths, it will return the control to the main program. Then the main program can determine the solution of the puzzle from the `ArrayList`. If the size of the solution `ArrayList` is zero, this means that the puzzle doesn't have a valid solution. In other words, at least one clue conflict occurs in the puzzle. If there is exactly one solution in the `ArrayList`, the puzzle is valid and the game interface will be started. However, if the size of the `ArrayList` is greater than one, the puzzle has multiple solutions and the puzzle browser will show up instead.

Since the recursive guessing algorithm has to remember the previous data so that it can roll back when it finds a solution or a contradiction, we implement the algorithm using a `Stack` and `SolvingThreadMove` objects. `SolvingThreadMove` is an object representing a movement occurring during the guessing process. It contains all data necessary for each roll back step. Each instance of this object will be created and pushed onto the top of the `Stack` when the algorithm can determine a solution and, consequently, change the value of a cell. When the algorithm needs to roll back, a `SolvingThreadMove` object will be popped out of the top of the `Stack` and the program will change back to the previous state according to this object.

4.4 Game Interface

The game interface will show up when the user clicks the "Start Game" button in the puzzle editor window. Actually, some behind-the-scene tasks will be performed before the game interface appears. Firstly, the program will collect all the user inputs and prepare them for the solving algorithm. If the user is in the painting mode, the painting will be converted into numbers first. During the process of collecting user inputs, the program will do some

basic checking on the inputs as well. This is to prevent the user from passing invalid inputs into the program. Secondly, the puzzle solver will be used to solve the puzzle and return the solution `ArrayList` to the program. Finally, if the puzzle has only one solution, the game interface can start, using the solution from the previous step as a parameter.

The GUI of the game interface window is similar to those of the puzzle editor window. The top area contains a set of controller buttons while in the middle lies a `JPanel` object representing the grid and clues of the puzzle. If the user moves the mouse pointer onto the grid or the clues areas, the program will draw focus bars covering the whole lines of that row and column. This will be extremely helpful while the user is solving a large size puzzle. Figure 4.6 shows an example of the game interface window.

Figure 4.6 – An example of the game interface [6]

In the game interface window, the user can paint the puzzle using the mouse. Each cell has three different states resulting in three different background colors including black, white, and green. The user can paint each of these colors using the left, right, and middle mouse buttons respectively. Additionally, the user can drag the mouse while pressing one of its buttons to continuously paint the corresponding color throughout the grid area. You may ask “What if I do not have a three-buttons mouse?” That is alright. We also offer another method to paint the puzzle. If the user hold a shift key on the keyboard while making the left click, the program will change the grid color from black to white to green and back to black again. The color will change in the reverse order if we do the same thing using the right click instead. In addition, if the user clicks the mouse on a clue cell, the font color of that clue will turn gray, making it less distinguishable. This does not affect the solution of the puzzle. It

simply helps the user eliminate the clues that have already been solved so that they will not catch the user's attention when he is trying to solve the neighboring clues. The user can also turn it back to black color by clicking on that clue again.

If the user realizes that he has made some mistaken moves, he can use the "Undo" button to remove the latest action he made. The program will keep the history of the movements in the memory up to one hundred steps. We implement this functionality using a `Stack` and `Move` objects. A `Move` object consists of four attributes: `x`, `y`, `previousData`, and `newData`. "`x`" and "`y`" represent the row and the column on the grid respectively. "`previousData`" is the status of that grid cell before it was changed while "`newData`" is the new status. A new `Move` object will be created when the user clicks the mouse on the grid area resulting in the change of color on a particular grid cell. This means that no movement exists if the color does not change. The new `Move` object then will be pushed onto the top of the `Stack` of `Move`. If the user clicks undo button, the program pops the last `Move` object out of the top of the `Stack` and paint the grid cell back to its previous state.

This technique is similar to those used in the recursive guessing algorithm discussed in the previous section with a slightly difference. The movement object used to implement the undo button is the `Move` object while the one used in the recursive guessing algorithm is the `SolvingThreadMove` object. In fact, `SolvingThreadMove` is a subclass of `Move` with a few additional attributes in order to comply with the Javagram solving algorithm.

We define the size of the `Stack` for the undo button to one hundred. The purpose of this is to limit the resource used by the `Stack` since a new `Move` object will be created constantly. If we let that happen, sooner or later our program will run out of memory, probably resulting in the system crash. To prevent this casualty, before pushing a new `Move` object onto the `Stack`, we check whether the size of the `Stack` is equal to one hundred or not. If it is, the program will remove the `Move` object at the bottom of the `Stack` before adding the new one.

While playing the game, the user can restart the game simply by clicking on the "Clear" button. If the user can paint exactly all the required black cells in the puzzle, the congratulation screen will appear. The user can also click the button "Solution" to see the solution of the puzzle directly. To check whether the player has found the solution or not, the program has to go through all the cells in the grid and compare the value of each cell to those of the solution. If the program has to do this checking every time the user clicks the mouse, it

will take so much CPU time. We solve this problem by having the program count the number of black cells of the solution before the game is started. Then the program will need to do the checking only when the number of black cells that the user paints matches those of the solution.

4.5 Puzzle Browser

The puzzle browser will appear when the user clicks the "Start Game" button and the puzzle solver finds more than one possible solution. Figure 4.7 shows an example of the puzzle browser window.

Figure 4.7 – An example of the puzzle browser

The puzzle browser tells the user how many solutions can be found from the input puzzle. The user can select the solution number from the drop-down list and then click the "Start" button to see the solution of the puzzle with the specified number. Then a game interface window will appear and the solution will be painted onto the grid of the game interface. If the user clicks the "Start All" button, all solutions will be painted onto separated game interface windows at once.

5. System Evaluation

We test our software on two machines as follow.

	Computer I	**Computer II**
Type	Laptop	Desktop
CPU	Intel Pentium Centrino	Intel Pentium 4
CPU Speed	1.3 GHz	2.93 GHz
RAM	480 MB	760 MB
OS	Microsoft Windows XP Professional SP1	Microsoft Windows XP Professional SP1
JDK	J2SE 5.0	J2SE 5.0

Both machines run the puzzle editor and the game interface properly. On the first machine, if the size of the puzzle becomes bigger, probably around 30-40, depending on the available system resources at that moment, the program will start having difficulty in catching the mouse events while the second machine can handle it normally. This shows that the GUI part of the program works fine comparing to the puzzle solving part.

The puzzle solving part, which is based mostly on the Javagram package, also works fine when the size of the puzzle is not too big. Obviously, the time need to solve the puzzle is likely to increase if the size of the input puzzle becomes bigger. However, what is more interesting is that the size of the puzzle is not the only factor that affects the solving time. The pattern of the numbers plays an important role as well. On the same test machine, two same-size puzzles might require different amount of time that the program needs to find the solutions. The difference could be in milliseconds, or even minutes. This is because the `CompleteLineSolver` algorithm used in the Javagram package requires a large amount of time, and we do not know how often it will be used to solve a particular puzzle. On one hand, only the `FastLineSolver` algorithm is enough to solve the entire puzzle. On the other hand, the program might need to employ the `CompleteLineSolver` algorithm several times. Therefore, the pattern of the input numbers significantly affects the solving time.

There is also an article discussing about Paint by Numbers puzzle and NP-Completeness problem written by Nobuhisa Ueda and Tadaaki Nagao. [7] Being an NP-Complete problem means there is no algorithm that can solve Paint by Numbers puzzle in polynomial time. Thus, although we concentrate on puzzles with a specific size, we still cannot determine the exact time that the program needs to solve Paint by Numbers puzzles.

6. Conclusion

For this project, we create a program that can solve Paint by Numbers puzzles. It provides tools that can help the user create puzzles as well. We also provide the game interface for the user who enjoys solving the puzzles. The algorithms used to solve the puzzles in this project work pretty well. However, there might be some other techniques that can be used to solve the puzzles. Accordingly, in the future, we wish to find or develop new algorithms that will improve the solving time. Finally, our ultimate goal is to make the program support multi-colors puzzles, not only black and white as in this project.

7. References

[1] About the Java Technology. Retrieved May 3, 2006 from http://java.sun.com/docs/books/tutorial/getStarted/intro/definition.html

[2] J2SE 5.0 in a Nutshell. Retrieved May 3, 2006 from http://java.sun.com/developer/technicalArticles/releases/j2se15/index.html

[3] Java Technology Overview. Retrieved May 3, 2006 from http://java.sun.com/overview.html

[4] Nonogram Solver. Retrieved May 3, 2006 from http://www.comp.lancs.ac.uk/computing/users/ss/nonogram/index.html.en-GB

[5] Nonogram Solver: FAQ. Retrieved May 23, 2006 from http://www.comp.lancs.ac.uk/computing/users/ss/nonogram/faq.html.en-GB

[6] Nonogram Solver: Local puzzle collection. Retrieved May 29, 2006 from http://www.comp.lancs.ac.uk/computing/users/ss/nonogram/puzzles.html.en-GB

[7] NP-Completeness Results for Nonogram via Parsimonious Reductions. Retrieved May 29, 2006 from http://www.cs.titech.ac.jp/~tr/reports/1996/TR96-0008.ps.gz

[8] Online Tsunami. Retrieved April 25, 2006 from http://www.playtsunami.com/

[9] Paint by Numbers – Wikipedia, the free encyclopedia. Retrieved April 24, 2006 from http://en.wikipedia.org/wiki/Paint_by_numbers

[10] What is Java Advanced Imaging? Retrieved May 3, 2006 from http://java.sun.com/products/java-media/jai/whatis.html