Yajilin Puzzle Dissertation

Lizzie Vialls

May 5, 2016

Contents

1 Introduction **5**
- 1.1 What is the Yajilin Puzzle . . . 5
- 1.2 Aims of the Project . . . 6

2 Interface and File Format **8**
- 2.1 Structure of Program . . . 8
- 2.2 Graphics Package . . . 8
  - 2.2.1 TKinter . . . 8
  - 2.2.2 Making Canvas Objects . . . 9
  - 2.2.3 Canvas IDs . . . 9
  - 2.2.4 Mouse interaction . . . 10
- 2.3 Storing the Puzzle . . . 11
  - 2.3.1 Data format . . . 11
  - 2.3.2 File Format . . . 12

3 Heuristic Logic **13**
- 3.1 Building Blocks for Solving a Puzzle Through Heuristic Algorithms 13
  - 3.1.1 Is a solution correct? . . . 13
  - 3.1.2 Undo Function . . . 14
- 3.2 Heuristic Algorithms . . . 14
  - 3.2.1 Black Squares Must Have Lines or Clues Adjacent . . . 14
  - 3.2.2 Number Clues with Only One Option . . . 15
  - 3.2.3 Same Number Clues in Same Direction . . . 16
  - 3.2.4 Lines Must Leave in Two Directions . . . 17
  - 3.2.5 Only One Loop Allowed . . . 17

4 Solving the Puzzle **24**
- 4.1 Heuristic and "Brute Force" . . . 24
- 4.2 SAT Solvers . . . 25
  - 4.2.1 What are SAT Solvers? . . . 25
  - 4.2.2 Groundwork for SAT Solving Yajilin Puzzles . . . 26
  - 4.2.3 Generic Rules for Yajilin Puzzles . . . 26
  - 4.2.4 Rules for a Given Puzzle . . . 30
  - 4.2.5 Solving the Puzzle with the Rules in Place . . . 31

5 On Reflection **35**
5.1 Usability of Project 35
5.2 Heuristic or SAT? 35
5.3 Success of project 36
5.4 Further improvements that could be made 36

List of Figures

1.1 An example Yajilin Puzzle with labels 5

2.1 From above to below 10
2.2 Definitions for letters used in storing the puzzle state 11
2.3 An example .csv file 12

3.1 function BlacksCannotBeAdjacent 15
3.2 Example of Mutable Objects 16
3.3 An Example of a 0 Number Clue 16
3.4 How a Clue Can Be Filled - From Left to Right 17
3.5 OneOptionForNumberClues method 18
3.6 Same Number Clues 19
3.7 The method checking for the exact same clues 19
3.8 From left to right, lines have only one option to traverse out of squares. 20
3.9 From left to right, must be black where line cannot traverse through 20
3.10 Method for checking blocked squares 21
3.11 OneWayToGo method 22
3.12 From left to right, cannot make a small loop so lines extend in other directions 23
3.13 The stopSmallLoops method 23

4.1 A graphical demonstration of how Brute Force would work . . . 24
4.2 A set of example SAT rules with potential solution 25
4.3 Initial definitions for creating SAT rules 26
4.4 Truth Table for Ensuring a Square does not have Multiple States 27
4.5 Initial rules for setting square states 28
4.6 Code for defining what states a square can have 29
4.7 Truth Table for Ensuring a Black Square is not Adjacent to Another Black Square 29
4.8 Code to ensure black squares cannot be adjacent 30
4.9 Truth Table for Ensuring Lines Connect 30
4.10 Rules for ensuring lines connect 31
4.11 Rules for locations of Number Clues 31
4.12 Rules for 0 Clues 32

4.13 Number Clue Brainstorming 32
4.14 Defining number clue requirements 33
4.15 Rules Generated for a value 2 number clue over a space of length 4 34
4.16 Solving the Puzzle 34

Chapter 1

Introduction

1.1 What is the Yajilin Puzzle

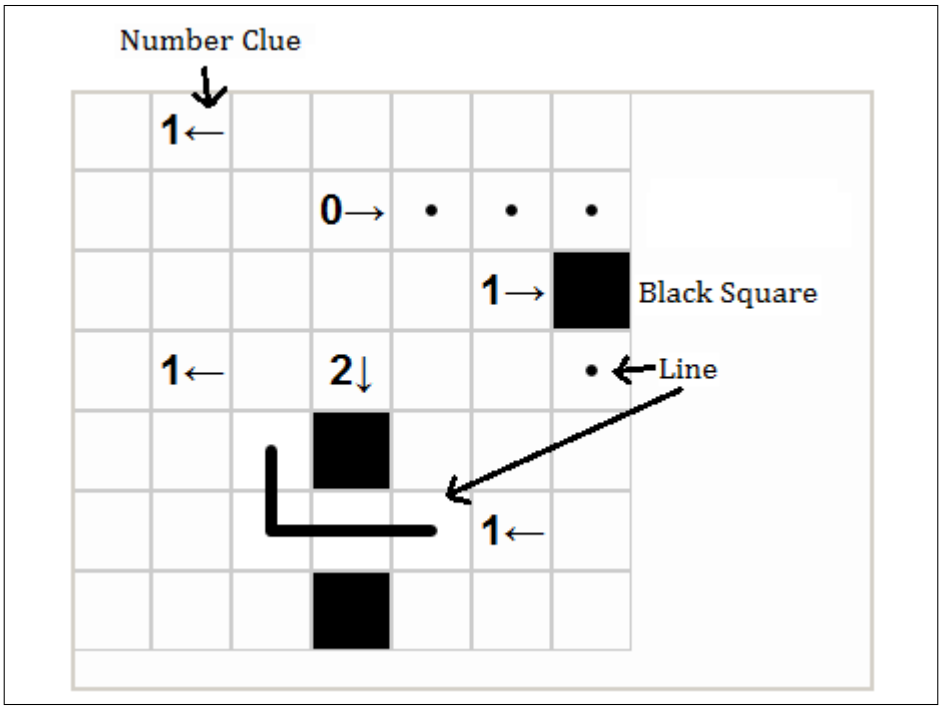

Figure 1.1: An example Yajilin Puzzle with labels

The Yajilin puzzle is a logic puzzle, believed to be NP complete, created by the company Nikoli, a Japanese company that popularised Sudoku[1]. It comprises of a rectangular grid of squares, where each square can be filled with different items, as seen in figure 1.1. The puzzle has a few rules to follow:

1. All squares must be filled with either a black square, a line, or a number clue.
2. Number clues show how many black squares are required in the direction indicated.
3. Black squares cannot be directly adjacent to another black square.
4. All lines must be part of one single loop.

The smallest of puzzles are generally at least ten squares in width and height, with puzzles of over twenty squares width and height existing. Simpler puzzles can be solved purely through looking for patterns in the puzzle, whereas harder puzzles require a certain amount of guesswork and back tracking.

1.2 Aims of the Project

The aim of the project is to create a piece of software that can assist in solving Yajilin Puzzles. To achieve this aim, the following objectives will need to be fulfilled:

1. To create a program that allows a user to import puzzles and then solve them.
   (a) Create a working user interface
   (b) Make an infrastructure for saved puzzles that allows saving, loading, importing.
   (c) Program some logic that allows the program to detect if a puzzle solution is valid.
2. To program the logic required for the program to find a solution to any Yajilin puzzle.
   (a) Implement an Undo button, which will in the process create a logging mechanism for actions.
   (b) Implement a Help button, which will in the process program heuristic methods of solving the puzzle
   (c) Create a solving method that combines the pattern recognition, "Brute force", and the logging from 2a.
   (d) Investigate the use of SAT solvers to find puzzle solutions.

Python was decided upon as the language of choice, for a couple of reasons: Firstly, it has many pre-build libraries that would assist in speeding up development - of note are PycoSat, a SAT solver library that is an implementation of the C library PicoSat[5]; and Tkinter, a pre-packaged graphics package. Secondly, Python has many functional aspects to its language, which helps in programming logic puzzles. Thirdly, it is cross-platform, which was a consideration in

the preliminary stages of planning due to ideas of an Android implementation of the puzzle. This was in the end discarded in favour of more features in the program itself.

Chapter 2

Interface and File Format

2.1 Structure of Program

For ease of programming and testing the project, the various aspects of the program were split into multiple files and classes. The following files were decided upon and created:

- YajilinBoard - a class that contained the state of the board and methods for editing the board state directly, or getting aspects of the state directly.
- YajilinGUI - contains all aspects of the GUI, and pulls all classes together into one.
- YajilinLogic - contains Heuristic Algorithms and solution checking methods.
- YajilinLogging - an object that stores the log of actions the user has done, so that the actions can potentially be undone.
- SatSolvingYajilin - all the SAT solving was put here

2.2 Graphics Package

2.2.1 TKinter

The GUI was created using a built in python graphical interface known as TKinter, which builds up user interfaces using a large number of "widgets", aka components. The initial configuration for frames and buttons was a fairly generic TKinter setup, using a frame to hold two other frames: one for the puzzle, one for the buttons.

The puzzle itself is created using a tkCanvas widget. The Canvas widget allows us to "draw" images upon the canvas of the widget, automatically giving them numbered references. This allows us to manipulate the images as soon

they are on the canvas, and potentially switch between various images. It also allows us to pick up on where the canvas is clicked on, and what has had a user interaction take place[2].

2.2.2 Making Canvas Objects

With the puzzle being composed of lines, squares and text, there were various ways that the puzzle could be built up on the canvas widget.

The first attempt involved drawing directly on the canvas - creating rectangles for the squares, creating text objects for the number clues, and drawing lines when lines were required. Building the puzzle this way had the advantage of the puzzle being able to change size with the window without losing definition. However, it was extremely complicated to implement, and was abandoned when an alternative was tried.

The alternative that then got implemented was using images. For each state the square could be in, there was an image created for that state. The puzzle was then build up using image objects. Building it this way meant that when a square had its state changed, all that needed to be done was changing the image's configuration and pointing to a different image.

2.2.3 Canvas IDs

An issue found with this early on involved the reference IDs TKinter assigned. When an object is created on the canvas, TKinter would assign it a new variable, value one higher than the previous. There was no easy way to adjust the values to be something more readable - e.g. coordinates of the square in the grid. As well as this, if there were objects deleted, their IDs were not freed up, so if objects are removed and re-added the IDs continue to go up and end up being incredibly messy to deal with.

To get around the problem, two methods were tried. Firstly, two functions were created - CanvastoXY and XYtoCanvas. These took the ID from the canvas and converted it to an x and y coordinate, and vice versa. This worked when objects were not being deleted from the canvas and the order in which the objects were created was known. In this case, the objects were created row by row, so the canvas ID was equivalent to $x_maxtimesy + x$.

However, if a new puzzle was being loaded, the entire canvas had to be broken down and remade to reset the IDs, otherwise the functions fell down.

The second method, which was the one chosen, was to use a python Dictionary - a Hash Map that could be modified on the fly and then used to get IDs. When an object is created, the dictionary is updated with the latest object ID mapped to a tuple of the (x,y) coordinates. If a new puzzle was loaded in, the dictionary could be wiped, and the process begun again. Doing it this way made the code as a whole more readable and have less code used, making it the preferred option.

2.2.4 Mouse interaction

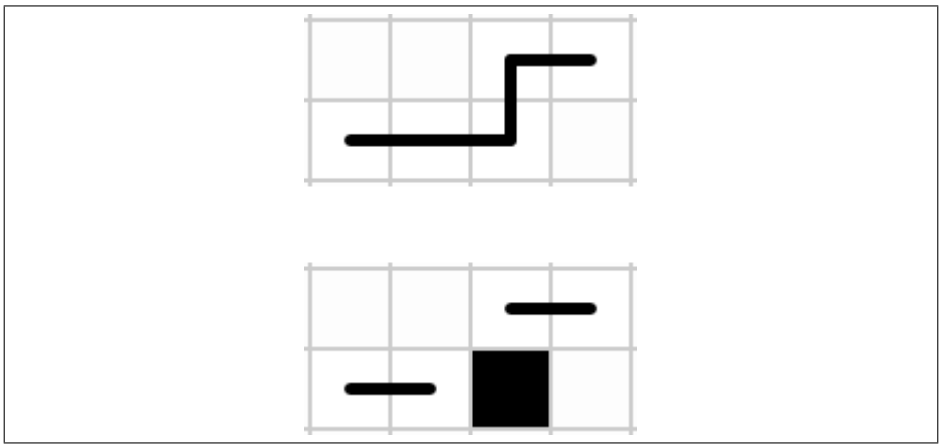

Figure 2.1: From above to below

The next step was to implement the ability for the user to modify the puzzle. It was decided early on to use the following assignments to mouse buttons, as it was felt to be the most natural for a logic puzzle:

Right Mouse Button(RMB) : Toggle black squares Left Mouse Button(LMB) : Draw a line

The Right Mouse button was fairly simple to implement. The Canvas widget would detect where the mouse event took place, and make a note of which object on the canvas created the event. With these factors, it was very simple to check if the square was black or not black - if it were black, the square would be made white, and if it was not black, it would be made black. A later addition also allowed a black square to be placed where there was a line currently. This required use of the dictionary of IDs to (x,y) coordinates, to detect where on the canvas the square was in relation to other squares, and to then update the adjacent squares to no longer path to the middle square, as seen in 2.1.

The Left Mouse button was more complicated. When a user manipulated lines, there were four options that could occur:

1. A dot was placed in a square, to indicate a line of unknown direction was in the square.

2. A dot was removed from the square.

3. Through clicking and dragging, a line was drawn where the mouse cursor indicated.

4. Through clicking and dragging, a line that was already there is erased.

Manipulation of a "dot" was simple. If the user clicks, then releases, the square's state is checked - if it's a dot, it gets removed. If it's not a line or a dot,

a dot is placed in the square. Click and drag required a few more interesting methods.

The main stumbling point was with TKinter's event handling. While it would indicate where the event took place, and what square a user clicks on with a "current" tag, the "current" tag would not update with a user dragging to a new square.

The first way that was used to solve this issue was to manually reapply the "current" tag as the user dragged. The second way was to ignore the "current" tag entirely and do everything with the coordinates given when an event is parsed by TKinter.

The latter option was used, and the coordinates of previous and future events were stored in the code. Through these coordinates, the program could extrapolate what cardinal direction the user was dragging in, and draw/remove lines in said direction depending on the current state of the squares.

2.3 Storing the Puzzle

2.3.1 Data format

V = Void, or blank square **B** = Black, or filled in square **L*dd*** = Line, potentially in the directions d, where $d \in \{N, E, S, W\}$. L can have 0-4 directions, representing a dot with no direction, or a line exiting the square in up to 4 directions. **N*nd*** = Number clue, where $n \geq 0$, indicating number of black squares in direction d, where $d \in \{N, E, S, W\}$

Figure 2.2: Definitions for letters used in storing the puzzle state

It was decided to store the state of the puzzle in as simple a format as possible. To this end, a list of lists was used as a 2d array to store the puzzle. This meant that the maximum y and x dimensions of the puzzle could be gotten through len(puzzleState) and len(puzzleState[0]) respectively. To store the state of each square, a string was set for each square. The contents of the string defined the square as shown in 2.2.

The directions in number clues and lines were set to be compass directions to remove any potential confusion when programming that may come about with using left and right.

The order of the letters in the strings was important for parsing the puzzle - an "L" at the beginning of a square indicated it was a line, and an "N" a the beginning indicated a number clue. Keeping a set order made reading the puzzle easier as the program knew exactly where certain pieces of information were in a string.

2.3.2 File Format

```
V,N1W,V,V,V,V,V,
V,V,V,N0E,V,V,V,
V,V,V,V,V,N1E,V,
V,N1W,V,N2S,V,V,V,
V,V,V,B,V,V,V,
V,V,LE,LEW,LW,N1W,V,
V,V,V,B,V,V,V,
```

Figure 2.3: An example .csv file

When creating the load and save buttons, it was required to decide on a format for the puzzle. It was decided to make the puzzle be stored in a .csv format - a text based spreadsheet, such as Figure 2.3.

Each line represents the 7 rows in the puzzle, with the definitions for each letter shown in Figure 2.2.

Storing the puzzle in a text based spreadsheet format meant that the user had a way to import puzzles without the program having to have an option for writing the puzzles in. This meant priority could be given to the rest of the coding without the program losing features.

A user if they wished to import a puzzle could use a spreadsheet editing program, write in the puzzle manually, and then save it as a .csv file to be imported into the program through the load puzzle button.

With the format sorted, it was merely a matter of python's simple file reading format reading the file line by line into the puzzle state. This was made even easier by the python file reading format being able to split each line of the file by a set character (in the case of .csv, ","), creating an array the length of the puzzle that could be inserted into the puzzleState array.

Chapter 3

Heuristic Logic

3.1 Building Blocks for Solving a Puzzle Through Heuristic Algorithms

3.1.1 Is a solution correct?

The rules mentioned in the first chapter are what a potential solution is required to fulfil, and with the puzzle state stored a method was required to be made for parsing the puzzle and detecting any errors. The check solution method was created with multiple clauses, where if a clause fails, the method stops prematurely and returns the fail along with the reason why it failed.

The simplest error to check for was if there were any squares that were still blank. This clause goes through each square in the puzzle, and checks for any squares with value "V". If any were found, then the square was empty (void) and the solution was incorrect.

The next clause involved black squares. Black squares cannot be adjacent, so a clause was created that again looped through the puzzle. If it found a black square, it checked the squares adjacent to it ($\pm$ y, x) and (y, $\pm$ x). If any of the four adjacent squares were black, the solution was incorrect.

In relation to the previous clause, the third clause checked each number clue, and whether enough black squares exist to satisfy the clue. This was done through counting each of the squares in the direction. Thanks to a hash map of the four directions to the change in y/x each direction represents, it was a simple matter to count the black squares in a direction through a while loop. If the number of black squares were not equal to the number clue, then the solution was false.

The final and most complicated clause was checking that there was a single connected loop. The first part involved checking that every line in the puzzle left a square in two directions. If a line only left in one direction, or was a dot, the solution was incorrect. With only lines going in two directions, the next step was tracing from an arbitrary point until the line caught up with itself. If it did,

and the number of squares with lines in the puzzle were equal to the number of squares traced through, then the loop was indeed one single, unbroken loop. Otherwise, the solution would be incorrect.

At the end of the clauses, if all of them were satisfied, then the puzzle solution was correct and the program would return as such.

3.1.2 Undo Function

In preparation for the final implementation of heuristic algorithms to solve a given puzzle, a function that allowed back-tracking through a solution was required. To prepare for this, an undo button was implemented for the user to use when they made a mistake. Initially, the log was stored as a file, and the program read from the file. However, this was found to be clunky and often had mistakes appear that were hard to discern the source of.

In the end, a list of puzzle states was used (a list of (list of lists)), and a pointer variable that indicated where the undo log was currently "at" in terms of how far the undo was used. To implement this logging into the GUI, the black square method had an "Update log" function call added to it, and a left mouse button release method was made with the update log function call in. This meant that all the actions the user did were stored in the undo log.

3.2 Heuristic Algorithms

The meat of the assistance for the user was put in a "Help" button. This button was designed to detect any obvious patterns in a puzzle that could be solved, and fill them in for the user. To detect the patterns, various heuristic algorithms were created, and if a pattern was detected that could update the puzzle state, the puzzle state was updated. For simpler puzzles, with enough heuristic algorithms, the help button could solve the puzzle entirely on its own for the user.

3.2.1 Black Squares Must Have Lines or Clues Adjacent

The first pattern to be put in involved black squares. Because black squares cannot be directly adjacent, they must either have a number clue or line adjacent. This pattern looked at every black square, and filled any adjacent void squares with a dot, indicating a line of some kind exists. The pattern was implemented as shown in figure 3.1.

The top line involving dims is self explanatory - it takes the puzzle state being looked at and returns the dimensions of the puzzle. The line involving state2, however, is a bit more interesting. In python, list type objects are Mutable. Mutable objects work as shown in figure 3.2: Mutable objects, if defined as being equivalent to another object, are set as a pointer to the object rather than a new object in its own right with the value of the previous object. This mutability caused quite a few issues when programming. The solution was a

```
dims = [len(state), len(state[0])]
state2 = YajilinLogic.copyPuzzleState(state)

for y in range(0, dims[0]):
   for x in range(0, dims[1]):
      if state[y][x] == "B":
         if (y >= 1):
            if state[(y-1)][x] == "V":
               state2[(y-1)][x] = "L"
         if (y < dims[0]):
            if state[(y+1)][x] == "V":
               state2[(y+1)][x] = "L"
         if (x < dims[1]):
            if state[y][(x+1)] == "V":
               state2[y][(x+1)] = "L"
         if (x >= 1):
            if state[y][(x-1)] == "V":
               state2[y][(x-1)] = "L"
```

Figure 3.1: function BlacksCannotBeAdjacent

copyPuzzleState function, which created an empty list of the exact same size as the state, and filled in each square of the list manually. Doing this meant that the program could store both the state as it was, and the state after the pattern checking is applied.

Otherwise, the nested for loop is again straightforward. For each square, it checks if it's a black square. If it is a black square, it then checks if there's a square that exists in an adjacent direction (e.g. for the first clause in the if statement, if y¡1, then y = 0 and there does not exist a square with a smaller y value). If such a square exists, it checks if it's a void square. If it's a void square, then it changes it to be a line with no direction.

3.2.2 Number Clues with Only One Option

The next step was looking at number clues. Some number clues are very easy to fill, due to only having one option. The simplest of these is a clue indicating 0 squares in a direction(see figure 3.3). If there's 0 squares in a direction, then there must be only lines and number clues, and so can fill in any void squares with lines without direction.

Another easy to fill number clue involves a small amount of maths. If there exists a number clue of value n, if there are $2n - 1$ squares in the indicated direction there is only one way of filling the squares with black squares without black squares being adjacent, as shown in figure 3.4.

The final number clue that has only one option was a little more complicated to program. If a number clue of size n has in the indicated direction a number of void squares equal to n-(number of black squares), those void squares

```
#Immutable
a = 2
b = a
a = 4
output a, b: 4, 2

#Mutable
a = 2
b = a
a = 4
output a, b: 4, 4
```

Figure 3.2: Example of Mutable Objects

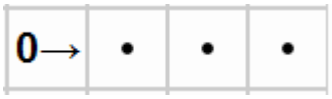

Figure 3.3: An Example of a 0 Number Clue

must be black squares. As well as this, if a number clue of size n has in the indicated direction a number of adjacent void squares equal to 2n-(number of black squares), then these can only be filled in one way.

Both of these clauses were programmed as shown in 3.5. The CountSquares method would return a list of 4 values, with each value representing the number of squares with a particular state, as mentioned in the comments. The first part of the if statement is for the harder part of the puzzle clause - if the count is equivalent to $2times n-1$, with n being the number clue, then it fills every other square with a black square. The latter part of the if statement checks if the number clue is 0, and if it is, fills each square in the direction with a dot. It also corrects any mistakes a user may have used, as it removes any black squares in the direction and replaces them with dots.

3.2.3 Same Number Clues in Same Direction

In the case of two number clues being in the same direction and having identical value, there is only one way it can be interpreted - with the space between the two being lines, as shown in figure 3.6.

To program this pattern in, the number clues are looped through, like in previous clauses involving number clues. For each number clue reached, the program works its way along the puzzle in the direction of the number clue, and if it finds an identical clue, it notes the coordinates of both squares, and between these two fills any blank squares with lines.

The alternative method would be to identify equivalent number clues, and check if they appear in the same row or column, depending on direction of said clues. However, with how previous clauses have been implemented, it was easier

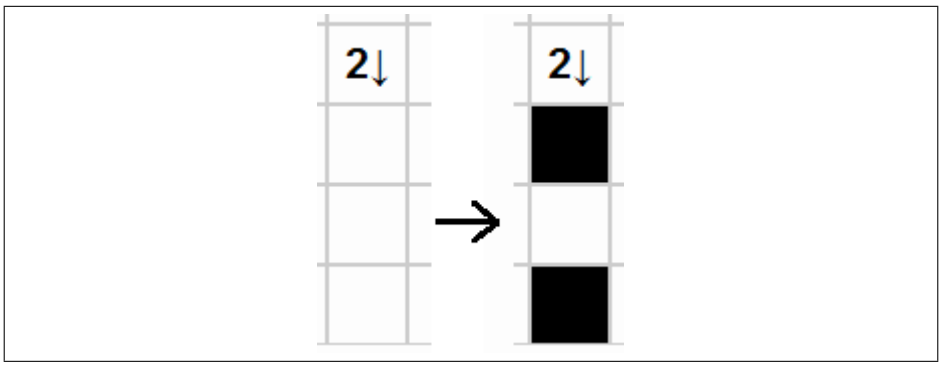

Figure 3.4: How a Clue Can Be Filled - From Left to Right

to reuse code from before and modify it to fit the new clause. This gives the annotated code fragment shown in figure 3.7

3.2.4 Lines Must Leave in Two Directions

As is mentioned earlier, all Yajilin Puzzles involve lines making up a single loop. To this end, every line must leave its square in two directions, otherwise it won't be part of the loop. This gives two clauses to add to the program: where a line has only two possible locations to traverse, it must traverse to said locations (figure 3.8; if a blank square has only one possible location to traverse, it must be a black square (figure 3.9).

To this end, a new method was created relating to the board - a "Check-Blocked" method (shown in figure 3.10). The purpose of this method is to return an array of values that indicate which directions a square is blocked from going to. A square is "blocked" if it is filled with a number clue, a black square, at a board edge, or a line that enters the middle square. A square is not blocked if it is blank, filled with a dot, or filled with a line that leaves in a direction that does not enter the middle square.

With this method, it's merely a case of checking how many squares are blocked and moving from there. If the square being checked contains a dot, and two adjacent squares are blocked, then it must leave in the two unblocked squares. If the square being checked contains a line with one direction, and three adjacent squares are blocked, then there's one direction it must leave through.

This gives the code shown in figure 3.11.

3.2.5 Only One Loop Allowed

The final clause that is implemented involves a clause that is easy for a human to notice, but harder for a computer to comprehend. Because there must be only one loop, if a line could go in two directions, and one of those directions would create a smaller loop separate from the rest of the puzzle, then it must go the other way. This is shown in figure 3.12.

```
                sq = board.getSquare(theBoard, i[0], i[1])
                count = board.countSquares(theBoard, i[0], i[1], sq[-1]) #
                    i is the coords of number clue, sq[-1] is the
                    direction to go in
                #Blank, Black, Line, Clue
                toBeFilled = int(sq[1:-1]) - count[1]
                dir = theBoard.numberChange[sq[-1]]
                if (count[0]+count[1]+count[2]+count[3]) ==
                    2*int(sq[1:-1]):
                  board.blackEveryOther(theBoard, i[0], i[1], sq[-1],
                      int(sq[1:-1]))
                elif int(sq[1:-1]) == 0:
                  y = i[0]
                  x = i[1]
                  while(y >= 0 and y < dims[0] and x >= 0 and x <
                      dims[1]):
                    s1 = board.getSquare(theBoard, y, x)
                    if s1 == "B" or s1 == "V":
                      board.addDot(theBoard, y, x)
                    y = y + dir[0]
                    x = x + dir[1]
```

Figure 3.5: OneOptionForNumberClues method

The easiest way found to program this was to create a line trace method - one that would return a list of all squares traversed through to go from one point to another, and then to check if the two points being checked are part of the same path. If this method is then the last one that the help button uses, earlier methods will catch the way to finish a puzzle, and this method would then be helpful with regards earlier stages of the puzzle solving. This gives the code shown in figure 3.13.

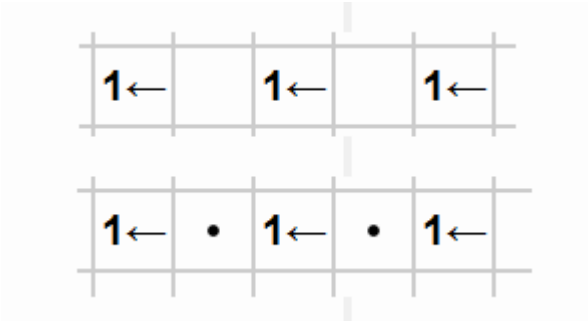

Figure 3.6: Same Number Clues

```
            for i in theBoard.numberClues:
               # number clues are stored as coordinates, so the value
                   of the clue is obtained
               sq = board.getSquare(theBoard, i[0], i[1])
               dir = sq[-1]
               dir = theBoard.numberChange[dir] # this dictionary
                   converts a direction into numerical values
                   indicating the change in y and x coordinates for
                   each direction
               dims = theBoard.dims # the dimensions of the board
               y = i[0] + dir[0] # get the first coordinates
               x = i[1] + dir[1] # to check for identical clues
               endpoint = i # current endpoint is the start point
               while(y >= 0 and y < dims[0] and x >= 0 and x <
                   dims[0]): # until we reach the board edge
                  if board.getSquare(theBoard, y, x) == sq:
                     endpoint = [y, x] # if the square currently being
                         looked at is the same as a number clue, end
                         point has been found
                  y = y+dir[0] # continue looking - if there's 3 or
                      more identical clues, all space will be filled
                      with lines
                  x = x+dir[1]
               if endpoint != i: # if we found one or more identical
                   number clues
                  board.fillFromTo(theBoard, i[0], i[1], endpoint[0],
                      endpoint[1], "L") # fill all squares between
                      start and end with lines if currently void
```

Figure 3.7: The method checking for the exact same clues

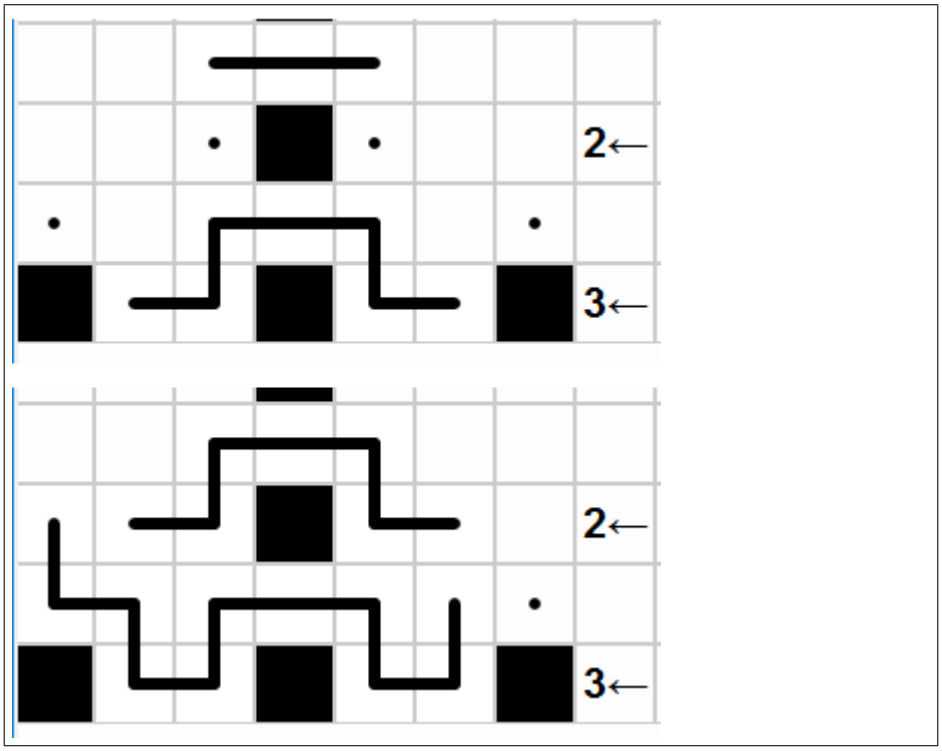

Figure 3.8: From left to right, lines have only one option to traverse out of squares.

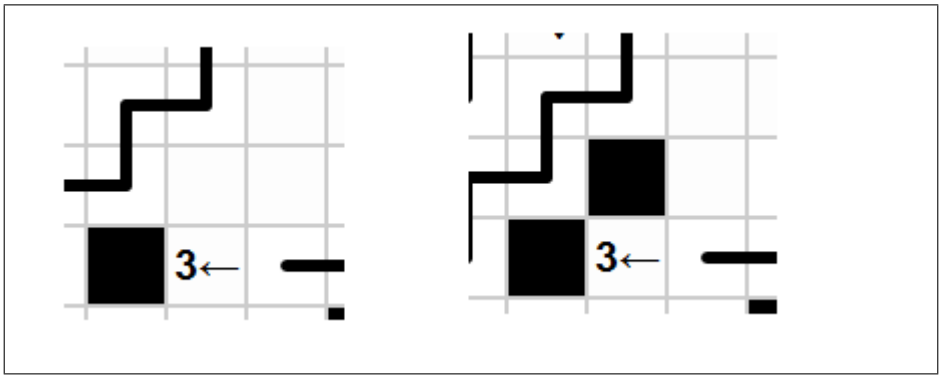

Figure 3.9: From left to right, must be black where line cannot traverse through

```
        to_return = [0, 0, 0, 0]

        #North
        if y - 1 < 0:
           to_return[0] = 1
        else:
           sq = self.getSquare(y-1, x)
        if sq == "B" or (sq[0] == "L" and len(sq) == 3) or sq[0] ==
            "N":
           to_return[0] = 1

        #South
        if y + 1 == self.dims[0]:
           to_return[2] = 1
        else:
           sq = self.getSquare(y+1, x)
        if sq == "B" or (sq[0] == "L" and len(sq) == 3) or sq[0] ==
            "N":
           to_return[2] = 1

        #West
        if x - 1 < 0:
           to_return[3] = 1
        else:
           sq = self.getSquare(y, x-1)
        if sq == "B" or (sq[0] == "L" and len(sq) == 3) or sq[0] ==
            "N":
           to_return[3] = 1

        #East
        if x + 1 == self.dims[1]:
           to_return[1] = 1
        else:
           sq = self.getSquare(y, x+1)
        if sq == "B" or (sq[0] == "L" and len(sq) == 3) or sq[0] ==
            "N":
           to_return[1] = 1

        return to_return
```

Figure 3.10: Method for checking blocked squares

```
            for y in range(0, dims[0]):
               for x in range(0, dims[1]):
                  sq = board.getSquare(theBoard, y, x)
                  if sq == "L":
                     sqs = board.checkBlocked(theBoard, y, x) # [N, E,
                         S, W]
                     if sqs[0] + sqs[1] + sqs[2] + sqs[3] == 2:
                        if sqs[0] == 0:
                           board.lineInDir(theBoard, "N", y, x)
                        if sqs[1] == 0:
                           board.lineInDir(theBoard, "E", y, x)
                        if sqs[2] == 0:
                           board.lineInDir(theBoard, "S", y, x)
                        if sqs[3] == 0:
                           board.lineInDir(theBoard, "W", y, x)

                  if sq[0] == "L" and len(sq) == 2:
                     sqs = board.checkBlocked(theBoard, y, x) # [N, E,
                         S, W]
                     if sqs[0] + sqs[1] + sqs[2] + sqs[3] == 3:
                        if sqs[0] == 0:
                           board.lineInDir(theBoard, "N", y, x)
                        if sqs[1] == 0:
                           board.lineInDir(theBoard, "E", y, x)
                        if sqs[2] == 0:
                           board.lineInDir(theBoard, "S", y, x)
                        if sqs[3] == 0:
                           board.lineInDir(theBoard, "W", y, x)

                  if sq == "V":
                     sqs = board.checkBlocked(theBoard, y, x) # [N, E,
                         S, W]
                     if sqs[0] + sqs[1] + sqs[2] + sqs[3] >= 3:
                        board.doBlack(theBoard, y, x)
```

Figure 3.11: OneWayToGo method

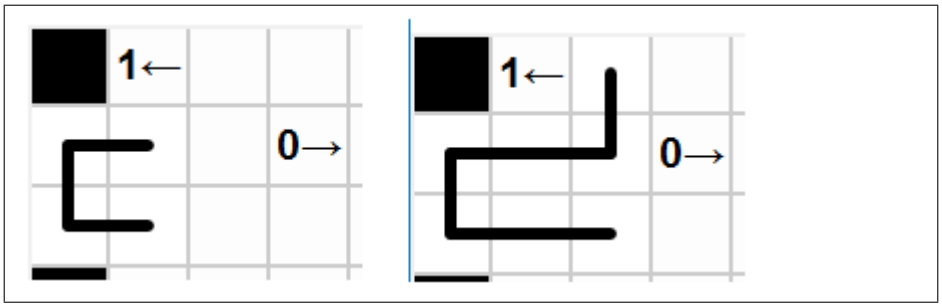

Figure 3.12: From left to right, cannot make a small loop so lines extend in other directions

```
            for y in range(0, theBoard.dims[0]):
               for x in range(0, theBoard.dims[1]):
                  sq = board.getSquare(theBoard, y, x)
                  if sq[0] == "L" and len(sq) == 2:
                     freeSquares = board.getFree(theBoard, y, x) #
                         check how many squares are free around a square
                     sqs = freeSquares[0] # number of free squares
                     coords = freeSquares[1] # coordinates of free
                         squares
                     if sqs[0] + sqs[1] + sqs[2] + sqs[3] == 2:
                        trace = board.traceLine(theBoard, y, x) #
                            trace[0] is coords, trace[1] is number of
                            coords
                        if coords[0] in trace[0]:
                           board.lineFromTo(theBoard, y, x,
                               coords[1][0], coords[1][1])
                        elif coords[1] in trace[0]:
                           board.lineFromTo(theBoard, y, x,
                               coords[0][0], coords[0][1])
```

Figure 3.13: The stopSmallLoops method

Chapter 4

Solving the Puzzle

4.1 Heuristic and "Brute Force"

With the pattern matching clauses implemented, the next step was to merge them together to enable the program to solve the puzzle automatically. For simple puzzles, this was very easy and quick - a continuous looping through all the heuristic algorithms returned the solution. However, for more complicated puzzles, the heuristic algorithms would reach a point where they cannot solve the puzzle further. In this case, "Brute Forcing" the puzzle is the next step - using trial and error to solve the puzzle, as shown in figure 4.1.

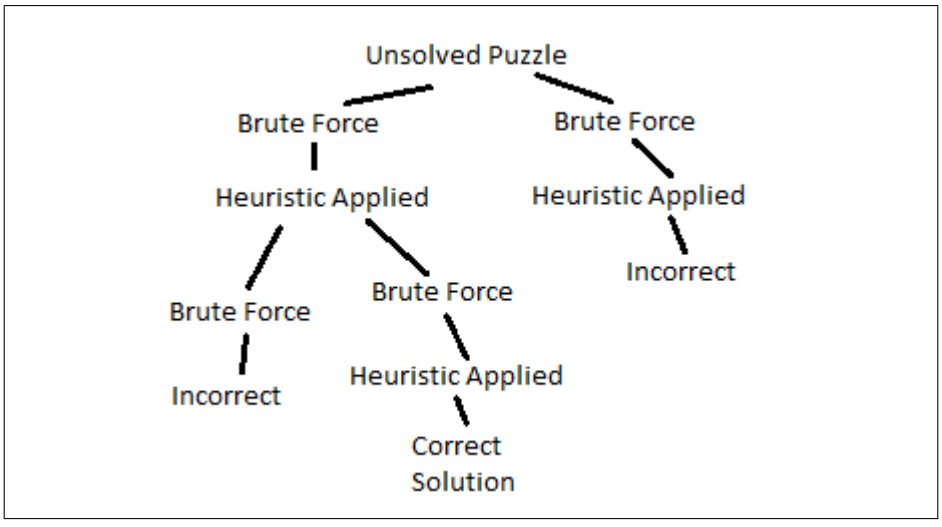

Figure 4.1: A graphical demonstration of how Brute Force would work

In theory, this would work for solving puzzles. However, in practical terms, it turned out to be suboptimal for harder puzzles that required the trial and

error solution, due to the time it takes and the number of options it has to parse through. With each leaf of the tree of possibilities having many more possibilities added through brute force, it takes a substantial amount of time to navigate the tree and find the solution. On top of this, the easiest way to navigate the tree is recursively - however, Python does not cope well with recursion. This was the first time in the entire project that there was mild regret for choosing Python as a language. Unfortunately, with time running out there was no chance for using Python to call a different language to solve the puzzle. So it was then that other options had to be explored.

4.2 SAT Solvers

4.2.1 What are SAT Solvers?

SAT solvers, or Boolean satisfiability problem solvers, involve returning potential solutions to a list of variables that can be true or false. In the case of programming, these variables are generally numbers where a positive value indicates it's true, and a negative value indicates it's false. The variables will also have attributed to them a set of rules that a solution must fulfil, in the form of "or" statements that must all be satisfied - in practical terms, a long line of or statements anded together. Figure 4.2 shows an example set of rules and a potential solution for a SAT problem.

```
Rules:
[1,2]
[3,4]
[-2]
[-5,3]

Solutions:
[1,-2,3,4,-5]
[1,-2,3,4,5]
[1,-2,3,-4,-5]
[1,-2,3,-4,5]
[1,-2,-3,4,-5]
```

Figure 4.2: A set of example SAT rules with potential solution

SAT solvers are programs or libraries that will take a set of rules and return solutions that satisfy the rules. In the case of Python, PycoSat is an example SAT solver library, and the one used in the project to solve the Yajilin Puzzles. Using PycoSat meant that there was no need to reinvent the wheel and write a SAT solver from scratch, and allowed for the only problem being how to put the Yajilin Puzzle into terms that a SAT solver could solve it[5].

4.2.2 Groundwork for SAT Solving Yajilin Puzzles

To this end, a few ground rules were laid out:

1. Each square could be black, a clue, or a line in two directions
2. Lines have a choice of four directions to go in
3. For each square, there are a potential eight states it could be in
4. The eight states could be reduced to six if squares are allowed to have two line states; the two states being the two directions the line goes in.

This creates the framework for what values the SAT solver will be working through: six values for each square in a puzzle with dimensions (y, x). In a purely arbitrary fashion, the numbers one to six were assigned square states. To help with readability of the code, a function v(y, x, square) was created to convert a square into its respective boolean value (4.3), and six variables were created, named after the square states, and assigned the six values. A Rules object was added also, which was defined as an empty list. Using this object, each new rule can be appended to it so as to build up the rules.

```
def v(y, x, sq):
   return y*6*xMax + x*6 + sq
black = 1
clue = 2
north = 3
east = 4
south = 5
west = 6
rules = []
```

Figure 4.3: Initial definitions for creating SAT rules

Using this as our framework for creating the boolean values, each square now has six unique values assigned to it. This allows the rules to be created. The majority of the rules were valid for any puzzle, and so a basePuzzleRules method was implemented to separate puzzle specific rules from generic rules.

4.2.3 Generic Rules for Yajilin Puzzles

The first rule implemented involving forcing a square to only have one state, or two if it were a line. To this end, a truth table was created for this, with the top part of it shown in 4.4. The bottom half was left out as the entirety of it is False and did not add much to the table.

This visualisation was useful to help solidify what was required. With that in mind, the first two rules for a square were written, as shown in figure 4.5

Black	Clue	North	East	South	West	Result
0	0	0	0	0	0	0
1	0	0	0	0	0	1
0	1	0	0	0	0	1
0	0	1	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	1	0
1	1	0	0	0	0	0
1	0	1	0	0	0	0
1	0	0	1	0	0	0
1	0	0	0	1	0	0
1	0	0	0	0	1	0
0	1	1	0	0	0	0
0	1	0	1	0	0	0
0	1	0	0	1	0	0
0	1	0	0	0	1	0
0	0	1	1	0	0	1
0	1	1	0	1	0	1
0	1	1	0	0	1	1
0	1	0	1	1	0	1
0	1	0	1	0	1	1
0	1	0	0	1	1	1

Figure 4.4: Truth Table for Ensuring a Square does not have Multiple States

These nine rules indicate that a square can either have the first state be false, and/or the second state be false. In particular, this relates to only allowing Clue and Black to be valid options if all other options are false.

After this, there's matter of allowing two cardinal directions at once. As such, the next few rules added are:

```
black, clue, north, east, south, west = 1, 2, 3, 4, 5, 6
rules.append[black, clue, north, east, south]
rules.append[black, clue, north, east, west]
rules.append[black, clue, north, west, south]
rules.append[black, clue, west, east, south]
```

These four rules require that at least one of each state listed in a row be True. Combined with the previous rules, this forces a square to be either Black, a Clue, or a combination of the four directions. However, currently a square could have all four directions be true and still be labeled as valid by the SAT Solver. To deal with this, a final four rules are added:

```
black, clue, north, east, south, west = 1, 2, 3, 4, 5, 6
rules.append[-north, -east, -south]
rules.append[-north, -east, -west]
```

```
black, clue, north, east, south, west = 1, 2, 3, 4, 5, 6

# for the first square, which has values 1,2,3,4,5 and 6
    associated with it

rules.append[-black, -clue]
rules.append[-black, -north]
rules.append[-black, -east]
rules.append[-black, -south]
rules.append[-black, -west]
rules.append[-clue, -north]
rules.append[-clue, -east]
rules.append[-clue, -south]
rules.append[-clue, -west]
```

Figure 4.5: Initial rules for setting square states

```
rules.append[-north, -west, -south]
rules.append[-west, -east, -south]
```

This forces the requirement that at least two of the directions must be False - for instance, if North is false, the final rule would require that one of East, South and West be false also.

In combination, these rules force a square to be Black, a Clue, or any two directions, which are the only valid states for a solved puzzle, which gives us the code shown in figure 4.6 Using the v function allows us to loop through every square and assign the unique rules for each square. In total, it gives $6timesyMaxtimesxMax$ rules.

The next set of SAT Solver rules are quite a bit smaller and more simple than the first set of rules. For a given puzzle, lines cannot path outside of the puzzle, and so squares with y value 0 cannot have north be true, squares with x value 0 cannot have west be true, etc. To this end, a very small fragment adds in the $2timesyMax + xMax$ rules to the Solver:

```
for y in range(0, yMax):
   rules.append([-v(y,0,west)])
   rules.append([-v(y,xMax-1,east)])
for x in range(0, xMax):
   rules.append([-v(0,x,north)])
   rules.append([-v(yMax-1,x,south)])
```

Creating a rule with only one variable adds in a required variable for the SAT Solver to implement.

The next important rule to add is the requirement that black squares cannot be adjacent. This produces a much simpler truth table (shown in 4.7), which has a very strong similarity to a common logic function: NAND, aka Not And.

Unfortunately, the SAT Solver won't allow an and statement in place of an

```
def v(y, x, sq):
   return y*6*xMax + x*6 + sq

black, clue, north, east, south, west = 1, 2, 3, 4, 5, 6
rules = []

for y in range(0, yMax):
   for x in range(0, xMax):
      for i in range(1, 7):
         if i != 1:
            rules.append([-v(y,x,black), -v(y,x,i)])
         if i != 2 and i != 1:
            rules.append([-v(y,x,clue), -v(y,x,i)])
      rules.append([v(y,x,black),v(y,x,clue),v(y,x,north),v(y,x,east),v(y,x,south)])
      rules.append([v(y,x,black),v(y,x,clue),v(y,x,west),v(y,x,east),v(y,x,south)])
      rules.append([v(y,x,black),v(y,x,clue),v(y,x,north),v(y,x,west),v(y,x,south)])
      rules.append([v(y,x,black),v(y,x,clue),v(y,x,north),v(y,x,east),v(y,x,west)])
      rules.append([-v(y,x,north),-v(y,x,east),-v(y,x,south)])
      rules.append([-v(y,x,west),-v(y,x,east),-v(y,x,south)])
      rules.append([-v(y,x,north),-v(y,x,west),-v(y,x,south)])
      rules.append([-v(y,x,north),-v(y,x,east),-v(y,x,west)])
```

Figure 4.6: Code for defining what states a square can have

Square 1	Square 2	Result
0	0	1
1	0	1
0	1	1
1	1	0

Figure 4.7: Truth Table for Ensuring a Black Square is not Adjacent to Another Black Square

or statement. To solve this problem, NAND is rewritten as $(A \vee B)$, giving us the code fragment in figure 4.8

The final rule that was implemented was ensuring that lines must connect. The truth table for two example squares adjacent vertically is shown in figure 4.9.

To get this in the form required, the logic is converted to $(A \vee B) \wedge (A \vee B))$, which gives the above truth table. Converted into code, this gives figure 4.10

The one base rule that was not implemented was checking that all lines make a single loop. This was due to this being the most complicated of the rules, and while it is almost certainly possible, was not able to be worked out in the time period the project was being worked on.

```
for y in range(1, yMax):
   for x in range(0, xMax):
      rules.append([-v(y,x,black), -v(y-1,x,black)])

for y in range(0, yMax):
   for x in range(1, xMax):
      rules.append([-v(y,x,black), -v(y,x-1,black)])
```

Figure 4.8: Code to ensure black squares cannot be adjacent

Square 1 North	Square 2 South	Result
0	0	1
1	0	0
0	1	0
1	1	1

Figure 4.9: Truth Table for Ensuring Lines Connect

4.2.4 Rules for a Given Puzzle

With the base rules added, the next step in making a SAT solver solve the puzzle is to input the clues that a puzzle currently has.

The simplest part to this is informing the SAT Solver where the number clues are. As mentioned earlier, when saving the puzzle state, the number clues were saved as a separate array of their coordinates, which makes this part of the puzzle very simple to add, as seen in figure 4.11 This forces the SAT Solver to have a clue where this is a clue, and force there to not be a clue where clues don't exist.

The next step is to implement a way for the SAT Solver to know where black squares can go in relation to the clues. The simplest of these is the zero number clue - if there's a zero number clue, then all squares in that direction cannot be black.

To make this able to be looped through no matter the direction, the dictionary of directions to change in values of y and x was borrowed from the board class, and figure 4.12 was created.

Comments have been added to help with readability of the code.

Number clues of value one and above, however, are a little more complicated to deal with. Initially, it was thought that number clues of value one could be dealt with by creating a rule that looks something like

```
[sqBlack1, sqBlack2, sqBlack3, sqBlack4, sqBlack5]
```

However, this would be satisfied if there were multiple black squares spaced correctly, as well as only one square being correct - with an Or statement, it still returns true if multiple of the options being decided between are correct.

The solution is to look at the squares that can't be black. The table in figure 4.13 shows the number of possibilities for various sizes of clue and squares for

```
        for y in range(0, yMax):
           for x in range(1, xMax):
              rules.append([v(y,x, west), -v(y,x-1,east)])
              rules.append([-v(y,x, west), v(y,x-1,east)])

        for y in range(1, yMax):
           for x in range(0, xMax):
              rules.append([v(y,x, north), -v(y-1,x,south)])
              rules.append([-v(y,x, north), v(y-1,x,south)])
```

Figure 4.10: Rules for ensuring lines connect

```
     clues = theBoard.numberClues
     for y in range(0, yMax):
        for x in range(0, xMax):
           if [y, x] in clues:
              rules.append([v(y, x, clue)])
           else:
              rules.append([-v(y, x, clue)])
```

Figure 4.11: Rules for locations of Number Clues

the clues to be fulfilled in, ignoring the requirement for black squares to not be adjacent.

With the possibilities being of the form nCr, rules can be created relating to these that are extendible for any size number clue in any length space. To this end, the library IterTools is borrowed from Python, which has the capacity to calculate nCr for us over a list of variables.

The code given in figure 4.14 fulfils the requirements required. However, the main point of contention is how to choose the numbers to iterate over and create combinations. The solution was found to set n to be the length of the space, and r to be $n - clue + 1$ for rules involving a black square being true, and r to be $clue + 1$ for a black square being false. An example of these values in action is shown in figure 4.15.

4.2.5 Solving the Puzzle with the Rules in Place

With the rule clauses created, the SAT Solver library can now solve the puzzle. As mentioned above, there is no clause forcing there to be a single unbroken loop, so the SAT Solver solution had to be combined with the CheckSolution method created for previous objectives of the project. This gives the code in figure 4.16 Pycosat has two functions relating to solving SAT problems: solve, and Itersolve. Solve returns the first solution the solver finds, and Itersolve returns every single solution it finds. Using Itersolve, all solutions can be looped through until a correct one is found, and then the correct solution can be returned.

```
    for i in clues:
        y, x = i[0], i[1] # get the coordinates of the number clue
        dir = state[y][x][-1] # get the direction
        change = theBoard.numberChange[dir] # get the values for
            changing the x/y depending on the direction
        yi, xi = y+change[0], x+change[1] # set the variables for the
            square being looked at
        if int(state[y][x][1:-1]) == 0: # if it's a 0 number clue
            while(yi >= 0 and yi < yMax and xi >= 0 and xi < xMax): #
                while it hasn't reached the board edge
                rules.append([-v(yi, xi, black)]) # cannot be black
                yi, xi = yi+change[0], xi+change[1] # move onwards one
                    square in the direction required
```

Figure 4.12: Rules for 0 Clues

Clue Value	Squares in Location	Possibilities
1	4	4
1	5	5
2	4	6
2	5	10
3	4	4
3	5	10
n	m	mCn

Figure 4.13: Number Clue Brainstorming

ParsePuzzle is a function that reverses the conversion to SAT Solver format, and allows the program to check the solution using the previous method.

Return None at the end of the code is a fail-safe - if there is no solution found, then the method will return no solution, allowing the program to pick up on this and inform the user of the puzzle not being solvable.

```
    # first two lists, one of blacksquare = true, one of blacksquare =
        false
    while(yi >= 0 and yi < yMax and xi >= 0 and xi < xMax):
       toIter1.append(v(yi, xi, black))
       toIter2.append(-v(yi, xi, black))
       yi, xi = yi+change[0], xi+change[1]
    # calculate using the number clue and the length of space the
        number of squares to use in each positive black square rule
    combNum = len(toIter1)-int(state[y][x][1:-1])+1
    temp = itertools.combinations(toIter1, combNum) # get combinations
    toIter1 = [] # itertools returns tuples
    for i in temp: # so turn tuples into list
       toIter1.append(list(i))
    combNum = int(state[y][x][1:-1])+1 # number of squares to use in
        each negative black square rule
    temp = itertools.combinations(toIter2, combNum)
    toIter2 = []
    for i in temp:
       toIter2.append(list(i))
    #now have lists of rules to be added to the main rules
    for i in toIter1:
       rules.append(i)
    for i in toIter2:
       rules.append(i)
```

Figure 4.14: Defining number clue requirements

```
1 : [black1, black2, black3]
2 : [black1, black2, black4]
3 : [black1, black4, black3]
4 : [black4, black2, black3]
5 : [-black1,-black2,-black3]
6 : [-black1,-black2,-black4]
7 : [-black1,-black4,-black3]
8 : [-black4,-black2,-black3]

Let black1 be True. This gives:
1 : black1
2 : black1
3 : black1
4 : [black4, black2, black3]
5 : [-black2, -black3]
6 : [-black2, -black4]
7 : [-black4, -black3]
8 : [-black4, -black2, -black3]

Let black4 also be True, giving:

1 : black1
2 : [black1, black4]
3 : [black1, black4]
4 : black4
5 : [-black3, -black2]
6 : -black2
7 : -black3
8 : [-black2, -black3]

For rules 1-8 to be true, there MUST be black1, -black2,
-black3, black4
```

Figure 4.15: Rules Generated for a value 2 number clue over a space of length 4

```
        for i in pycosat.itersolve(rules):
           solution = SatSolvingYajilin.parsePuzzle(i, theBoard, yMax,
               xMax)
           if logic.checkSolution(solution)[0] == True:
              return solution
        return None
```

Figure 4.16: Solving the Puzzle

Chapter 5

On Reflection

5.1 Usability of Project

Using the result of the Yajilin Project, a user can solve almost any Yajilin Puzzle, with the limiting factor being the size of the screen compared to the size of the puzzle. On a 1366x768 monitor, a puzzle taller than 16 squares would start to have issues with readability, with the user having to move the program around the screen to attempt to view all the puzzle. This is the downside of using images to create the GUI, and could be fixed if an alternative solution for drawing the puzzle is found through drawing objects directly onto a canvas rather than displaying fixed size images. It could also be fixed by using smaller squares, or implementing horizontal and vertical scrollbars in the program for puzzles larger than a given size.

Other than this, a user can easily solve a puzzle using the program, and with the addition of the help button, larger puzzles can have assistance in solving when it comes to more obvious clues.

5.2 Heuristic or SAT?

On testing, the SAT Solver solved puzzles with large amounts of clues much quicker than puzzles with fewer clues and more blank space. This is an interesting contrast to the difference in speed the Heuristic Algorithms had when finding a solution, as the Heuristic Algorithms worked incredibly quickly on simple puzzles, and more complicated ones gave the SAT solver an edge on speed. However, even with the SAT Solver taking longer on more sparse puzzles, the time saved through using the SAT Solver over Heuristic Algorithms for more complicated puzzles is huge.

With these points in mind, it is recommended to the user of the Yajilin Puzzle program to use the Heuristic Algorithm to find solutions to simpler puzzles, and the SAT Solver to be used on more complicated puzzles.

5.3 Success of project

To recap, the objectives as stated at the start of this document:

1. To create a program that allows a user to import puzzles and then solve them.
   (a) Create a working user interface
   (b) Make an infrastructure for saved puzzles that allows saving, loading, importing.
   (c) Program some logic that allows the program to detect if a puzzle solution is valid.
2. To program the logic required for the program to find a solution to any Yajilin puzzle.
   (a) Implement an Undo button, which will in the process create a logging mechanism for actions.
   (b) Implement a Help button, which will in the process program heuristic methods of solving the puzzle
   (c) Create a solving method that combines the pattern recognition, "Brute force", and the logging from 2a.
   (d) Investigate the use of SAT solvers to find puzzle solutions.

Objective one was completed successfully. There is a working user interface, and the puzzle states can be saved, loaded, and imported through writing .csv documents for the puzzles. In addition, there is infrastructure in place to check if a solution is correct.

Objective two was also completed successfully. The Undo button works, as does the Help button, and both were combined to solve a puzzle with heurisitc algorithms. Finally, after heuristic algorithms were found to take an incredibly long time on harder puzzles, SAT Solvers were both investigated and implemented successfully to find solutions to puzzles.

So overall, the project was a success with all aims being achieved. However, there were parts that could have been done better, and other parts that could have had better solutions created for them.

5.4 Further improvements that could be made

The most obvious improvement that could be made if there was more time, would be implementing into the program a way to input puzzles. As it stands, it's possible to import puzzles through .csv files, however is not the most user friendly of methods. Creating an interface that allows the user to input number clues, and save the puzzle to then be solved, would further improve the usability of the program.

The second improvement would be to convert the Help button into a Hint button - instead of filling in the puzzle for the user, it would search for a recognised pattern, and then highlight this clue to the user. The only question that would have to be answered would be what to do if the pattern matching found no patterns. And for that, the third improvement would come in handy.

The third improvement would be to implement for the user a way to draw "notes" and "temporary" lines in a different colour. This would allow the user, on reaching a point where they're unsure where to go, to experiment and see what happens if a choice is made. The improvement would entail the ability to draw in items into the puzzle in a separate colour, the ability to "apply" the temporary items, and the ability to remove all the coloured lines with a single click. In essence, this would allow the user to backtrack manually, like the "brute force" method does.

A fourth improvement that would be to improve the SAT Solver. Currently, there is no implementation for checking for a single continuous loop, and the looping through all potential solutions takes up time that could be improved. The easiest way to help improve the time is to combine the Heuristic algorithms with the SAT solver, and input into the SAT solver the values the Heuristic algorithms return. This would reduce the number of potential solutions and increase the speed of the SAT Solver solution finding. The more complicated option would be to work out how to write the requirement for a single connected loop in SAT format.

Finally, a difficult but potentially very interesting improvement would be to implement a random puzzle generator. To this end, a SAT solver could have some parameters input - number of number clues, size of puzzle - and then one of the solutions generated by the SAT solver could be selected and placed into the interface unsolved. However, this would require a solution to the current issue of the SAT solver algorithms - a way to get it to discount solutions with more than one loop - as otherwise it would take a huge amount of processing power and take a while to get a puzzle.

Bibliography

[1] *Rules of Yajilin Puzzle*, last visited (10/12/15)
`http://www.nikoli.com/en/puzzles/yajilin/rule.html`

[2] *Tkinter Overview*, last visited (10/12/15)
`http://tkinter.unpythonic.net/wiki/Tkinter`

[3] *Stack Overflow*, last visited (10/12/15)
`http://stackoverflow.com`

General Coding Resource.

[4] Mark Lutz. *Programming Python*. O'Reilly Media Inc., Sebastopol, CA 95472.

General Coding Resource

[5] *Python Package Index: pycosat 0.6.1*, last visited (05/05/15)
`http://https://pypi.python.org/pypi/pycosat`