ITP 20002-03 Discrete Mathematics, Fall 2021

Homework 2

Na Yea Won, Lee Hyo Ju

22000217@handong.edu , 22000603@handong.edu

1. Introduction

In this report, we solve the logical puzzles 'Anti-king sudoku' and 'Nondango' with a SAT solver algorithm to find answers that satisfy the rules of the game among all the cases that can be answered. To do this, the process we had to do is to understand the rules of the game, then find propositional formulas that satisfy those rules, and write them in c-language code correctly.

'Anti-king sudoku' is a puzzle that adds one additional rule to the existing sudoku rule. The existing rule of Sudoku is to fill in the nine sub-grid, nine rows, and nine columns with numbers from 1 to 9. There are already numbers in a few cells, and it is a game that requires solving by thinking about many cases so that the same numbers do not fill into the same row, column, and sub-grid. The Anti-king Sudoku's rule :

1. one cell's adjacent cells should not have the same number. The numbers from 1 to 9 must be entered in a row without overlapping.
2. Numbers from 1 to 9 must enter the same column without overlapping.
3. Numbers from 1-9 must enter the same sub-grid without overlapping.
4. The same number with the middle cell cannot come in the adjacent eight surrounding cells.

In 1,2, and 3 rules, it needs to be shown the existence and uniqueness of 1-9 numbers. In the 4th rule, the middle cell's up, down, left, and right cells already are satisfied according to the previous rules. Therefore, it only needs to make sure that diagonal cells are different from middle cells. We tried to find a more effective and simple way to approach diagonal cells, not the way that we learned in class when solving the nqueen-sat problem.

'Nondango' is a game of changing white circles to black circles while complying with two rules. This game initially presents a specific area and white circle in 10x10 space. The problem must be solved by meeting the condition that only one black circle must exist in one area and that three or more consecutive circles in the row, column, and diagonal directions should not have the same color. Because each region dynamically changes, unlike Sudoku's sub-grid, setting up a region and finding a way to access cells in this region were important points to solve the problem. This point is successfully conducted by reading information from the input.txt file.

2. Approach

2.1. Anti-king sudoku

Here, we denote our propositional variable p(i, j, n), where i represents a row, j for a column, and n for the number assigned in a range of 1 to 9.

As mentioned in the introduction earlier, there are four rules for anti-king-sudoku. If we translate these into propositional formulas:

1. The numbers from 1 to 9 must enter the same row without overlapping(C1).

$$\bigwedge_{i=1}^{9}\bigwedge_{n=1}^{9}\bigvee_{j=1}^{9} p(i,j,n)$$

2. Numbers from 1-9 must enter the same column without overlapping(C2).

$$\bigwedge_{j=1}^{9}\bigwedge_{n=1}^{9}\bigvee_{i=1}^{9} p(i,j,n)$$

3. Numbers from 1-9 must enter the same sub-grid without overlapping(C3).

$$\bigwedge_{r=0}^{2}\bigwedge_{s=0}^{2}\bigwedge_{n=1}^{9}\bigvee_{i=1}^{3}\bigvee_{j=1}^{3} p(3r+i,3s+j,n)$$

4. The same number with the middle cell cannot come in the adjacent eight surrounding cells((C4).

$$\bigwedge_{i=2}^{9}\bigwedge_{j=1}^{8}\bigwedge_{n=1}^{9} \neg(p(i,j,n)\wedge p(i-1,j+1,n))$$

$$\bigwedge_{i=1}^{8}\bigwedge_{j=1}^{8}\bigwedge_{n=1}^{9} \neg(p(i,j,n)\wedge p(i+1,j+1,n))$$

5. Each cell is assigned with exactly one number(C5).

$$\bigwedge_{i=1}^{9}\bigwedge_{j=1}^{9}\bigwedge_{n=1}^{8}\bigwedge_{m=n+1}^{9} \neg\big(p(i,j,n)\wedge p(i,j,m)\big)$$

$$\bigwedge_{i=1}^{9}\bigwedge_{j=1}^{9}\bigvee_{n=1}^{9} p(i,j,n)$$

To find answers that meet these conditions, we must first declare all possible cell variables. Since each cell can have numbers from 1 to 9 with rows and columns, we declare 9x9x9 Boolean type variables. Next, since each cell can have a number from 1-9, we have grouped p(i, j, n) into 'or' so that at least one number exists in the p(i, j) position, and we have grouped all the cells into 'and' so that this existing condition satisfy all 9x9 cells. 'Not (p(i,j,n) and p(i,j,m))' are grouped into 'and' in all cells so that 'and' of the two numbers that could come to p(i,j) always have false values to meet the uniqueness of not being able to put two numbers in the same place.

2.1.1 Code Structure

C1, C2. To visit every cell, we use three for-loops. Two for-loops are used as row and column. The inner loop is used to assign the number. The number assigned between 1 and 9 is wrapped up with 'or', and each row and column is bound with 'and'. C1 and C2 have the same logical structure. Therefore, we can just switch the order of the for-loop in C2.

C3. Sudoku consists of a 9 by 9 grid where each subgrid is constructed with 3 rows by 3 columns. At most two outer for-loops are for 3 subgrids by 3 subgrids. Inner for-loops are used to visit every cell in a subgrid. At most the inner for-loop is to assign the number between 1 and 9. Due to the use of multiple for-loops, the code time complexity for C3 has the highest value.

C4. In C1 and C2, we already covered the constraints that each row and column cannot have the same number. Thus, we just add the constraint about the diagonal cases. We separate diagonal cases into two and used 3 for-loops each.

For the input data, we use a while-loop to read the file while it is not EOF(End Of File). To distinguish the value between number assigning and '?', we add the assigning number variable with row, column, and the value given by the input file by using function strcmp().

2.2. Nondango

As mentioned in the introduction earlier, there are four rules for 'Nondango'. We translate these into propositional as following formulas:

1. Every square is either a black circle('B') or a white circle('W') or nothing('X'). We represent this constraint by the following logic(P1).

$$\bigwedge_{i=1}^{10}\bigwedge_{j=1}^{10} p(i,j,B) \wedge \neg p(i,j,W) \wedge \neg p(i,j,X)$$

$$\bigwedge_{i=1}^{10}\bigwedge_{j=1}^{10} p(i,j,W) \wedge \neg p(i,j,X) \wedge \neg p(i,j,B)$$

$$\bigwedge_{i=1}^{10}\bigwedge_{j=1}^{10} p(i,j,X) \wedge \neg p(i,j,B) \wedge \neg p(i,j,W)$$

2. There must be no three circles of the same color placed on three consecutive squares. First, we conduct the constraint about the rows. There must be no three circles of the same color nearby in every row(P2).

$$\bigwedge_{r=1}^{10}\bigwedge_{c=2}^{9} \neg(p(r,c-1,W) \wedge p(r,c,W) \wedge p(r,c+1,W))$$

$$\bigwedge_{r=1}^{10}\bigwedge_{c=2}^{9} \neg(p(r,c-1,B) \wedge p(r,c,B) \wedge p(r,c+1,B))$$

3. Also, there must be no three circles of the same color, placed on three consecutive columns(P3).

$$\bigwedge_{c=1}^{10}\bigwedge_{r=2}^{9} \neg(p(r-1,c,W) \wedge p(r,c,W) \wedge p(r+1,c,W))$$

$$\bigwedge_{c=1}^{10}\bigwedge_{r=2}^{9} \neg(p(r-1,c,B) \wedge p(r,c,B) \wedge p(r+1,c,B))$$

4. Moreover, there must be no three circles of the same color, placed on three consecutive diagonals(P4).

$$\bigwedge_{r=2}^{9}\bigwedge_{c=2}^{9} \neg(p(r-1,c-1,W) \wedge p(r,c,W) \wedge p(r+1,c+1,W))$$

$$\bigwedge_{r=2}^{9}\bigwedge_{c=2}^{9} \neg(p(r-1,c-1,B) \wedge p(r,c,B) \wedge p(r+1,c+1,B))$$

$$\bigwedge_{r=2}^{9}\bigwedge_{c=2}^{9} \neg(p(r+1,c-1,W) \wedge p(r,c,W) \wedge p(r-1,c+1,W))$$

$$\bigwedge_{r=2}^{9}\bigwedge_{c=2}^{9} \neg(p(r+1,c-1,B) \wedge p(r,c,B) \wedge p(r-1,c+1,B))$$

Pre-assigned cells are given in the input data. In the input data file, the number represents which grid the cell is in. 'W' represents that the cell located in the same row and column is either a white circle or a black circle. We represent this constraint as the following:

$$\bigwedge_{r=1}^{10}\bigwedge_{c=1}^{10} (E(r,c,W) \wedge \neg p(r,c,X)) \vee (\neg E(r,c,W) \\ \wedge \big(p(r,c,X) \wedge \neg p(r,c,B) \wedge \neg p(r,c,W)\big))$$

Here, we denote our propositional variable E(r, c, Q), where r represents a row, c for a column, and Q for a character value either 'W' or 'X' or 'B'.

In a C program, we implement the constraint by using if-statements.

2.2.1 Code Structure

Here, we denote our propositional variable p(i, j, c), where i represents a row, j for a column, and c for the character value ‘B’, or ‘X’, or ‘W’.

P1. We use two for-loops to enter every square row by column. The square can have either ‘B’ or ‘W’ or ‘X’. We consider every case the square can have. For instance, if the square has ‘X’ in it, it cannot have ‘B’ and ‘W’ at the same time. Likewise, the square cannot have ‘X’ and ‘B’ while containing ‘W’. It can be represented as the combination of three to one. Thus, we think of three cases where elements are combined with ‘and’ and wrapped these up with ‘or’. In this way, we can assign a character value (‘B’, ‘X’, ‘W’) to every square.

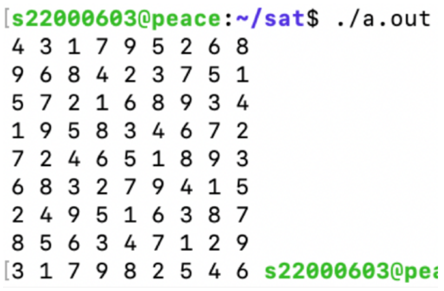
P2, P3, P4. To check three consecutive cells if they are not containing the same-colored circles, we do not consider the cases where one of those cells contains ‘X’. Instead, we consider the cases that should not be true. All the three consecutive cells cannot contain ‘W’ or ‘B’. We conduct this logic by using two for-loops to visit every cell and use ‘not’ and ‘and’ to express that there no exist three white colored consecutive cells and three black colored consecutive cells.

For the input data, we use a while-loop to read the file. While looping one character at a time, we extract the information about a grid number that the cell is in and the existence of a circle. We add an integer array with the size of 32 indexes to store the number of the assigned grid number. The index of the array represents the grid number, and its value represents how many there are. To find out the existence of a circle, we conduct the conditional statement. If the last word of the data, read in the input file, is ‘W’, we add constraints that there is either ‘W’ or ‘B’. Instead of using ‘B’ and ‘W’ and ‘or’, we use ‘not’ and ‘X’ to represent it. This can be rewritten using propositional variable E (r, c, W) where ‘r’ represents row, ‘c’ represents column as following:
[logic]

3. Evaluation

3.1. Anti-king Sudoku

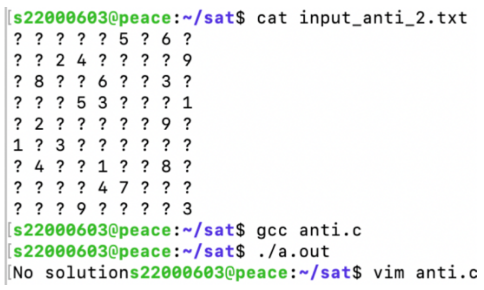
From the given input data, we were able to obtain the same result as the expected output as the following:



```
[s22000603@peace:~/sat$ ./a.out
 4 3 1 7 9 5 2 6 8
 9 6 8 4 2 3 7 5 1
 5 7 2 1 6 8 9 3 4
 1 9 5 8 3 4 6 7 2
 7 2 4 6 5 1 8 9 3
 6 8 3 2 7 9 4 1 5
 2 4 9 5 1 6 3 8 7
 8 5 6 3 4 7 1 2 9
[3 1 7 9 8 2 5 4 6 s22000603@pea
```

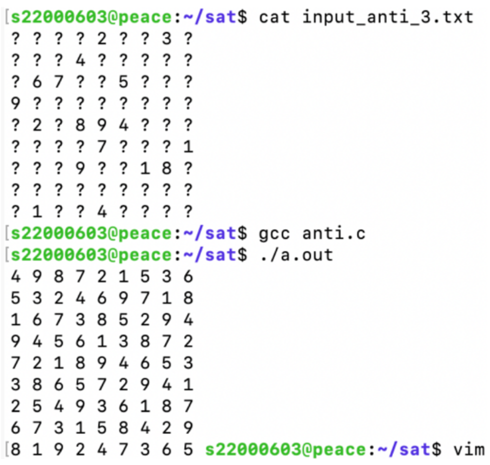
From the various test cases, we obtained several different following results:

1)



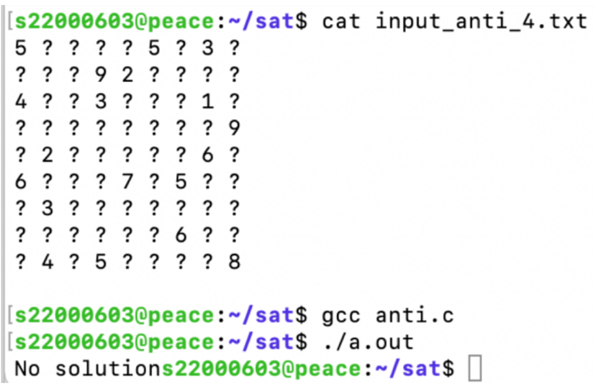
```
[s22000603@peace:~/sat$ cat input_anti_2.txt
 ? ? ? ? ? ? 5 ? 6 ?
 ? ? 2 4 ? ? ? ? 9
 ? 8 ? ? 6 ? ? 3 ?
 ? ? ? 5 3 ? ? ? 1
 ? 2 ? ? ? ? ? 9 ?
 1 ? 3 ? ? ? ? ? ?
 ? 4 ? ? 1 ? ? 8 ?
 ? ? ? ? 4 7 ? ? ?
 ? ? ? 9 ? ? ? ? 3
[s22000603@peace:~/sat$ gcc anti.c
[s22000603@peace:~/sat$ ./a.out
[No solutions22000603@peace:~/sat$ vim anti.c
```

2)



```
[s22000603@peace:~/sat$ cat input_anti_3.txt
 ? ? ? ? 2 ? ? 3 ?
 ? ? ? 4 ? ? ? ? ?
 ? 6 7 ? ? 5 ? ? ?
 9 ? ? ? ? ? ? ? ?
 ? 2 ? 8 9 4 ? ? ?
 ? ? ? ? 7 ? ? ? 1
 ? ? ? 9 ? ? 1 8 ?
 ? ? ? ? ? ? ? ? ?
 ? 1 ? ? 4 ? ? ? ?
[s22000603@peace:~/sat$ gcc anti.c
[s22000603@peace:~/sat$ ./a.out
 4 9 8 7 2 1 5 3 6
 5 3 2 4 6 9 7 1 8
 1 6 7 3 8 5 2 9 4
 9 4 5 6 1 3 8 7 2
 7 2 1 8 9 4 6 5 3
 3 8 6 5 7 2 9 4 1
 2 5 4 9 3 6 1 8 7
 6 7 3 1 5 8 4 2 9
[8 1 9 2 4 7 3 6 5 s22000603@peace:~/sat$ vim
```

3)

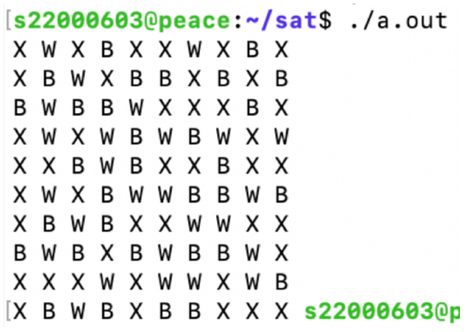


```
[s22000603@peace:~/sat$ cat input_anti_4.txt
 5 ? ? ? ? 5 ? 3 ?
 ? ? ? 9 2 ? ? ? ?
 4 ? ? 3 ? ? ? 1 ?
 ? ? ? ? ? ? ? ? 9
 ? 2 ? ? ? ? ? 6 ?
 6 ? ? ? 7 ? 5 ? ?
 ? 3 ? ? ? ? ? ? ?
 ? ? ? ? ? ? 6 ? ?
 ? 4 ? 5 ? ? ? ? 8

[s22000603@peace:~/sat$ gcc anti.c
[s22000603@peace:~/sat$ ./a.out
 No solutions22000603@peace:~/sat$
```

3.2. Nondango

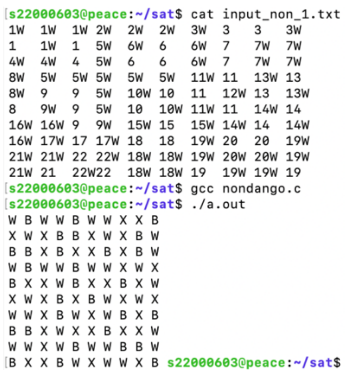
From the given input data, we were able to obtain the same result as the expected output as the following:



```
[s22000603@peace:~/sat$ ./a.out
 X W X B X X W X B X
 X B W X B B X B X B
 B W B B W X X X B X
 X W X W B W B W X W
 X X B W B X X B X X
 X W X B W W B B W B
 X B W B X X W W X X
 B W B X B W B B W X
 X X X W X W W X W B
[X B W B X B B X X X s22000603@p
```

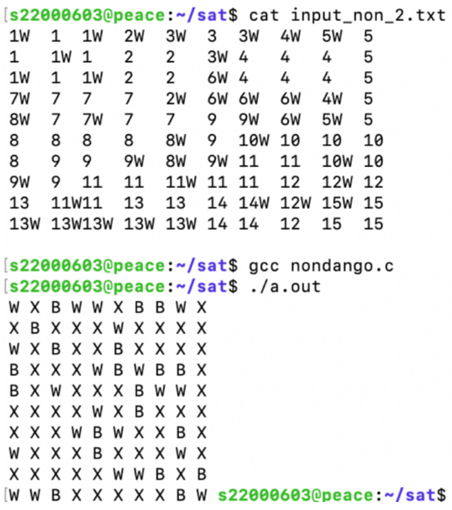
From the various test cases, we obtained several different following results:

1)



```
[s22000603@peace:~/sat$ cat input_non_1.txt
 1W  1W  1W 2W  2W  2W  3W  3   3   3W
 1   1W  1  5W  6W  6   6W  7   7W  7W
 4W  4W  4  5W  6   6   6W  7   7W  7W
 8W  5W  5W 5W  5W  5W  11W 11  13W 13
 8W  9   9  5W  10W 10  11  12W 13  13W
 8   9W  9  5W  10  10W 11W 11  14W 14
 16W 16W 9  9W  15W 15  15W 14W 14  14W
 16W 17W 17 17W 18  18  19W 20  20  19W
 21W 21W 22 22W 18W 18W 19W 20W 20W 19W
 21W 21  22W22  18W 18W 19  19W 19W 19
[s22000603@peace:~/sat$ gcc nondango.c
[s22000603@peace:~/sat$ ./a.out
 W B W W B W W X X B
 X W X B B X W X B W
 B B X B X X B X B W
 W B W W B W W X W X
 B X X W B X X B X W
 X W X B X B W X W X
 W W X B W X W B X B
 B B X W X X B X X W
 W W X W B W W B B W
[B X X B W X W W X B s22000603@peace:~/sat$
```

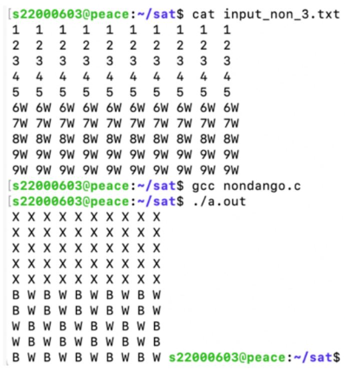
2)



```
[s22000603@peace:~/sat$ cat input_non_2.txt
 1W   1   1W   2W   3W   3   3W   4W   5W   5
 1    1W  1    2    2    3W  4    4    4    5
 1W   1   1W   2    2    6W  4    4    4    5
 7W   7   7    7    2W   6W  6W   6W   4W   5
 8W   7   7W   7    7    9   9W   6W   5W   5
 8    8   8    8    8W   9   10W  10   10   10
 8    9   9    9W   8W   9W  11   11   10W  10
 9W   9   11   11   11W  11  11   12   12W  12
 13   11W11    13   13   14  14W  12W  15W  15
 13W  13W13W   13W  13W  14  14   12   15   15

[s22000603@peace:~/sat$ gcc nondango.c
[s22000603@peace:~/sat$ ./a.out
 W X B W W X B B W X
 X B X X X W X X X X
 W X B X X B X X X X
 B X X X W B W B B X
 B X W X X X B W W X
 X X X X W X B X X X
 X X X W B W X X B X
 W X X X B X X X W X
 X X X X X W W B X B
[W W B X X X X X B W s22000603@peace:~/sat$
```

3)



```
[s22000603@peace:~/sat$ cat input_non_3.txt
 1  1  1  1  1  1  1  1  1  1
 2  2  2  2  2  2  2  2  2  2
 3  3  3  3  3  3  3  3  3  3
 4  4  4  4  4  4  4  4  4  4
 5  5  5  5  5  5  5  5  5  5
 6W 6W 6W 6W 6W 6W 6W 6W 6W 6W
 7W 7W 7W 7W 7W 7W 7W 7W 7W 7W
 8W 8W 8W 8W 8W 8W 8W 8W 8W 8W
 9W 9W 9W 9W 9W 9W 9W 9W 9W 9W
 9W 9W 9W 9W 9W 9W 9W 9W 9W 9W
[s22000603@peace:~/sat$ gcc nondango.c
[s22000603@peace:~/sat$ ./a.out
 X X X X X X X X X X
 X X X X X X X X X X
 X X X X X X X X X X
 X X X X X X X X X X
 X X X X X X X X X X
 B W B W B W B W B W
 B W B W B W B W B W
 W B W B W B W B W B
 W B W B W B W B W B
 B W B W B W B W B W s22000603@peace:~/sat$
```

4. Discussion

4.1. Anti-King Sudoku

While implementing the constraint, there cannot be the same number assigned in adjacent cells, we took a long time figuring out how to design the diagonal part. In the beginning, left top to right bottom, right top to left bottom, we designed 4 sections of for loops for left bottom to right top, and right bottom to left top, total 4 cases. It was intuitive but the number of lines of codes were too long, and there were too many propositional logics that it was difficult to debug. Therefore, we changed our logic to ease the pain. We looped twice from 1 to 9 for left bottom to right top and left top to right bottom, 2 cases. We were able to reduce almost 30 lines of code implementing such logic to cover the whole grid, and it was also easier to debug.

4.2. Nondango

The biggest challenge we faced was extracting two types of information from the input text files. We first opened the file twice. First, we checked the existence of a circle in each cell, and next we checked each grid number. This method, however, led the code to open the file 200 times, 100 times into two-character values. To minimize the time complexity, we tried to read two separate pieces of information from the first queue. As a result, we figured out how to build an array storing 32 integer indexes. It was also intuitive and saved in terms of complexity.

5. Conclusion

In this report, we solve the logical puzzles ‘anti-king sudoku’ and ‘Nondango’ with a SAT solver algorithm, and a C program code. By specifying the conditions that an expected output must satisfy, SAT solver algorithm finds answers among all the cases that can be answered. We use C program source code to generate propositional formulas in an effective logical way.

The process we had to do was to understand the rules of the game, then define the propositional variables, find propositional logics, and write them in c-language code correctly to generate a file that SAT solver can use.

To solve the puzzles, we first came up with the propositional logics, or specified conditions that an expected output specifies. We then translate these logics into propositional formulas by executing C program code designed to produce z3 requirements. We can get either the specified solution or ‘unsat’ which indicates the model is unavailable.

We successfully designed our C program in such a way that it does not only produce requirements for z3 but also produce the human-readable results of the puzzles provided by z3. To verify our program, we ran various test cases to each puzzle where some cases expected to have no solution and the other cases expected to have a solution. We were successfully able to get the solution as we expected.