SAT-based Sudoku Puzzle Solver (Programming Assignment #1)

Class: Discrete Mathematics (Handong Univ.)

Member: Jeonghwan Kim (21400181),

Chun Myung Park (21400337), Ho Yoon Choi (21600744)

1. Problem Definition

Sudoku, which requires logic and intuition, is a puzzle of 9-by-9 grid with 3-by-3 sub-grids within. Each cell has a number, which has a domain of 1 to 9, and for each cell the numbers are mutually exclusive; meaning that no two numbers can be placed in the same cell. This puzzle is solved by assigning a number to each cell, so it fills every row and column, and the 3-by-3 sub-grids. This puzzle can either be solved by strict logic or human intuition. In this assignment, only logic based on the compound propositions will be used to solve the puzzle.

2. Objective

The given task of the assignment requires us to construct a program that automatically builds a solution for a given 9-by-9 Sudoku puzzle. There are several constraints that must be considered:

- The problem must be represented in a propositional formula (in a DIMACS form)
- The given formula must be solved by an off-the-shelf SAT solver
- The SAT solver must be converted back to a Sudoku solution

The main objective of this assignment is to stick to the given constraints and use the given SAT solver, either z3 or MiniSAT, to solve the Sudoku puzzle.

3. Design of the program

3.1 Mathematical Design

Prior to constructing our program, the necessary compound propositions had to be analyzed and dissected to understand the logic and convert them into a proper DIMACS format.

We first discovered a formula to apply the index to each of the propositional variable constituting the 9-by-9 grid: "$p(i, j, n) = P[81(i-1) + 9(j-1) + n] = P_N$", where "n" is the value from 1 to 9 and "N" is an index for the propositional variable "P".

The compound propositions necessary to construct a proper DIMACS format are defined as CP1), CP2), CP3), CP4) and CP5), respectively:

CP1) : (i) There exists "true" statement among p(i, j, 1), p(i, j, 2), p(i, j, 3) , … , p(i, j, 9)

(ii) There exists a unique "true" statement among p(i, j, 1), p(i, j, 2), p(i, j, 3) , … , p(i, j, 9)

These propositions (F) can be rewritten as the following ($\neg$F):

(i) All the statements are "false"; p(i, j, 1), p(i, j, 2), p(i, j, 3) , … , p(i, j, 9)

(ii) More than one "true" statement exists among p(i, j, 1), p(i, j, 2), p(i, j, 3) , … , p(i, j, 9)

Another negation can be applied as such $\neg(\neg F)$ to get a CNF form of the compound proposition CP1).

$$F = (p1 \vee p2 \vee p3 \vee p4 \vee p5 \vee p6 \vee p7 \vee p8 \vee p9)$$

$$\wedge (\neg p1 \vee \neg p2) \wedge (\neg p1 \vee \neg p3) \wedge (\neg p1 \vee \neg p4) \wedge (\neg p1 \vee \neg p5) \wedge (\neg p1 \vee \neg p6) \wedge (\neg p1 \vee \neg p7) \wedge (\neg p1 \vee \neg p8) \wedge (\neg p1 \vee \neg p9)$$

$$\dots\dots\dots$$

$$\wedge (\neg p8 \vee \neg p9)$$

This is represented in a form of:

$$CP1) = \bigwedge_{i=1}^{9} \bigwedge_{j=1}^{9} CP1)'_{ij}$$

This result is the same as $\wedge_{i=1}^{9} \wedge_{j=1}^{9} \wedge_{n=1}^{8} \wedge_{m=n+1}^{9} \big(p(i,j,n) \rightarrow \neg p(i,j,m)\big) \wedge \big(p(i,j,m) \rightarrow \neg p(i,j,m)\big)$, which was written in the assignment instructions.

Other compound propositions are as follows:

CP2) Every row as every number:

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

CP3) Every column as every number:

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

CP4) Every 3-by-3 block as every number:

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

CP5) Each cell with an asterisk shares the same value "n". "XNOR" was used in this case to create a compound proposition that is true when the two propositional variables are true. Here is XNOR in a CNF form("⊙" this is XNOR sign):

$$(A \odot B) = (A \wedge B) \vee (\neg A \wedge \neg B) = \big(A \vee (\neg A \wedge \neg B)\big) \wedge \big(B \vee (\neg A \wedge \neg B)\big)$$

$$= (A \vee \neg B) \wedge (B \vee \neg A) \dots\dots\dots\dots\dots = \text{CP5)}$$

3.2 Program Structure and Flow of Execution

The program design is as written in the following as a pseudocode and they represent the previously elaborated compound propositions:

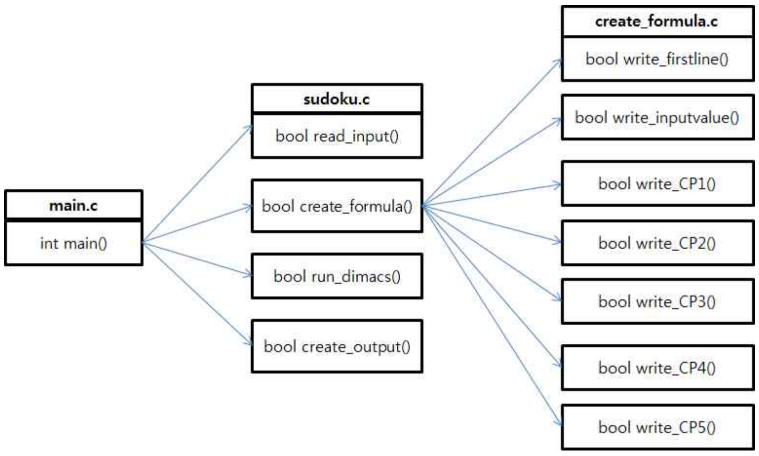


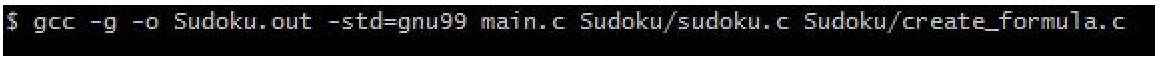
Figure 1: **The Program is made up of 3 files**. Above figure represent the include relationship between each other.

1) The main function(int main()) in "main.c" employs the functions in "sudoku.c" and "create_formula.c".
- It first reads the "input.txt" with "read_input()" function in sudoku.c
- Then, with "create_formula()" it creates DIMACS form compound propositions.
- With "run_dimacs()" and "create_output()", it runs the SAT solver and creates output.
2) read_input("input.txt", "input_array", FILE * input_stream)
- Receive the Sudoku problem from "input.txt" and stores each cell within an input array.
3) create_formula("input_array", FILE * formula_stream)
- creates "formula.txt" file to store DIMACS form of the compound propositions.
- returns "true" if successful.
- write_firstline("input_array", "formula.txt") – writes *"p cnf 729 N"*
- write_INPUT_VALUE("input_array", "formula.txt")
- write_CP1)("formula.txt"), write_CP2)("formula.txt"), write_CP3)("formula.txt"), write_CP4)("formula.txt"), write_CP5)("formula.txt"). (Refer to *Design of Program*)
4) run_dimacs("formula.txt", "SAT solver output_array")
- receives the "SAT solver output". If there is no solution, print "no solution" and exit the program.
- If there is a solution, interpret and convert the SAT solver output to 9-by-9 array.
5) create_output("output.txt", "SAT solver output_array")
- creates "output.txt" file and prints "output.txt" to compare with "input.txt"

4. Instruction

4.1 How to build and execute the program (refer to README in git repo)

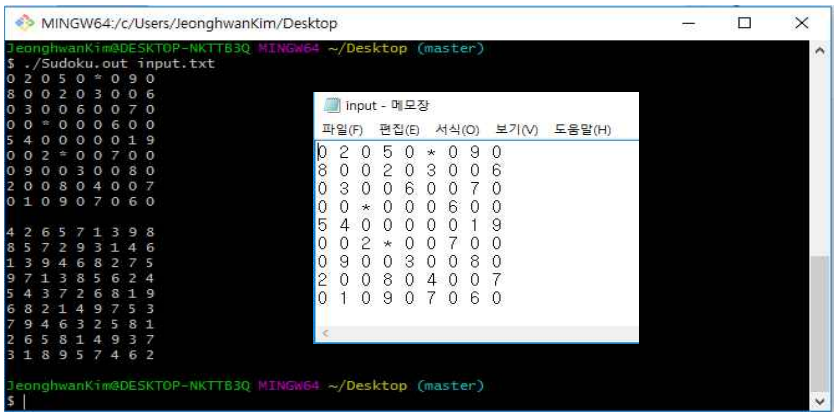
With "main.c" file in the main directory and the rest of other files (sudoku.c, sudoku.h, create_formula.c, create_formula.h) within "Sudoku" directory, input the following "gcc" instruction in the terminal(git bash):



```
$ gcc -g -o Sudoku.out -std=gnu99 main.c Sudoku/sudoku.c Sudoku/create_formula.c
```

Figure 2: This command line is defined in "makefile" also.

Then, it will create an executable file, "Sudoku.out.exe". The language standard is "gnu99", because "popen()" requires "gnu99" and it should be used to run the SAT solver, z3, within the program. Make sure to set "-std=gnu99" when compiling with gcc. We also made a "makefile", so Linux users can simply type "make" instruction to build the program. Refer to README in the git repository for more details about "make".
With a proper input file, "input.txt", in the same directory, we write "./Sudoku.out" followed by an input file name (i.e. "input.txt") to execute the program.
The execution gives both the input file and the output file, which contains the solution, to standard output and creates two text files: "output.txt" and "formula.txt"


MINGW64:/c/Users/JeonghwanKim/Desktop
input - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

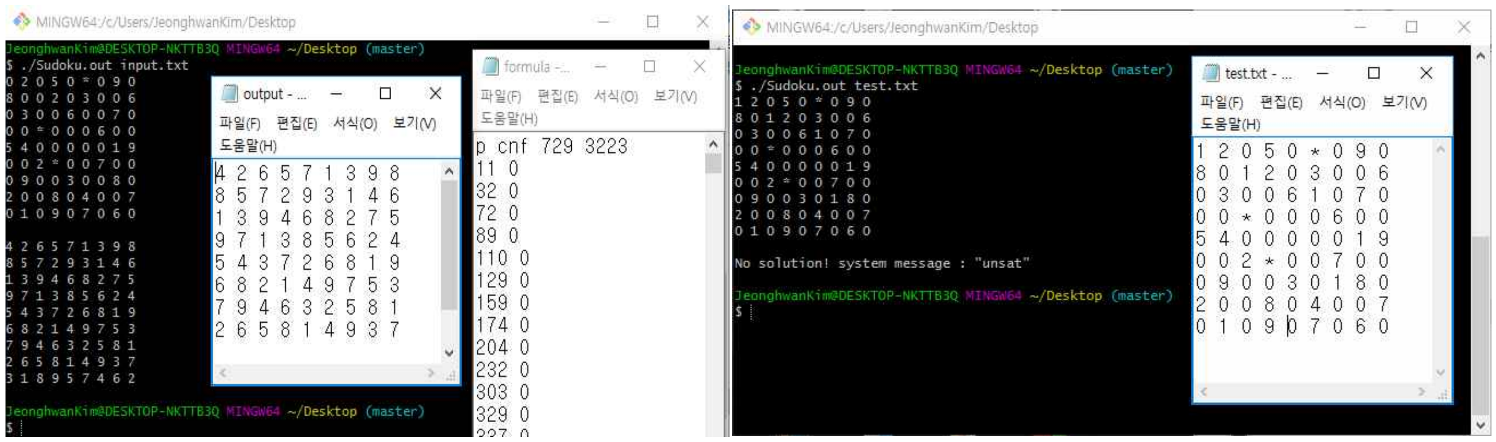
```
JeonghwanKim@DESKTOP-NKTTB3Q MINGW64 ~/Desktop (master)
$ ./Sudoku.out input.txt
0 2 0 5 0 * 0 9 0
8 0 0 2 0 3 0 0 6
0 3 0 0 6 0 0 7 0
0 0 * 0 0 0 6 0 0
5 4 0 0 0 0 0 1 9
0 0 2 * 0 0 7 0 0
0 9 0 0 3 0 0 8 0
2 0 0 8 0 4 0 0 7
0 1 0 9 0 7 0 6 0

4 2 6 5 7 1 3 9 8
8 5 7 2 9 3 1 4 6
1 3 9 4 6 8 2 7 5
9 7 1 3 8 5 6 2 4
5 4 3 7 2 6 8 1 9
6 8 2 1 4 9 7 5 3
7 9 4 6 3 2 5 8 1
2 6 5 8 1 4 9 3 7
3 1 8 9 5 7 4 6 2

JeonghwanKim@DESKTOP-NKTTB3Q MINGW64 ~/Desktop (master)
$ |
```

```
0 2 0 5 0 * 0 9 0
8 0 0 2 0 3 0 0 6
0 3 0 0 6 0 0 7 0
0 0 * 0 0 0 6 0 0
5 4 0 0 0 0 0 1 9
0 0 2 * 0 0 7 0 0
0 9 0 0 3 0 0 8 0
2 0 0 8 0 4 0 0 7
0 1 0 9 0 7 0 6 0
```

Figure 3: **Commonly get result** like above figure
when program successfully execution on bash shell

5. Demonstration

As shown here, the execution of the program:


MINGW64:/c/Users/JeonghwanKim/Desktop
output - ...
formula - ...
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
MINGW64:/c/Users/JeonghwanKim/Desktop
test.txt - ...
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
JeonghwanKim@DESKTOP-NKTTB3Q MINGW64 ~/Desktop (master)
$ ./Sudoku.out input.txt
0 2 0 5 0 * 0 9 0
8 0 0 2 0 3 0 0 6
0 3 0 0 6 0 0 7 0
0 0 * 0 0 0 6 0 0
5 4 0 0 0 0 0 1 9
0 0 2 * 0 0 7 0 0
0 9 0 0 3 0 0 8 0
2 0 0 8 0 4 0 0 7
0 1 0 9 0 7 0 6 0

4 2 6 5 7 1 3 9 8
8 5 7 2 9 3 1 4 6
1 3 9 4 6 8 2 7 5
9 7 1 3 8 5 6 2 4
5 4 3 7 2 6 8 1 9
6 8 2 1 4 9 7 5 3
7 9 4 6 3 2 5 8 1
2 6 5 8 1 4 9 3 7
3 1 8 9 5 7 4 6 2

JeonghwanKim@DESKTOP-NKTTB3Q MINGW64 ~/Desktop (master)
$
```

```
4 2 6 5 7 1 3 9 8
8 5 7 2 9 3 1 4 6
1 3 9 4 6 8 2 7 5
9 7 1 3 8 5 6 2 4
5 4 3 7 2 6 8 1 9
6 8 2 1 4 9 7 5 3
7 9 4 6 3 2 5 8 1
2 6 5 8 1 4 9 3 7
```

```
p cnf 729 3223
11 0
32 0
72 0
89 0
110 0
129 0
159 0
174 0
204 0
232 0
303 0
329 0
```

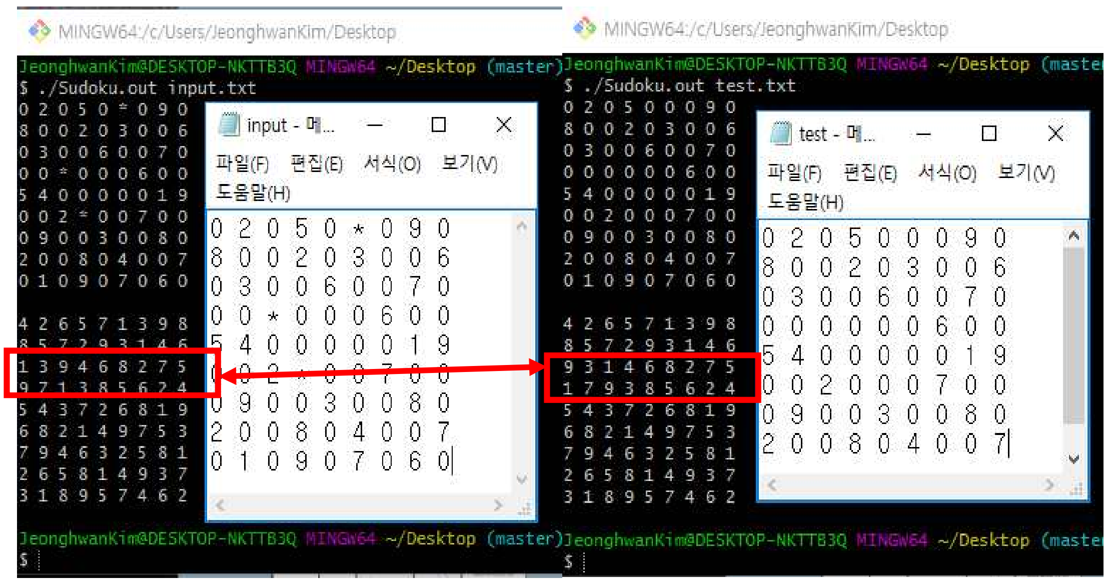
```
JeonghwanKim@DESKTOP-NKTTB3Q MINGW64 ~/Desktop (master)
$ ./Sudoku.out test.txt
1 2 0 5 0 * 0 9 0
8 0 1 2 0 3 0 0 6
0 3 0 0 6 1 0 7 0
0 0 * 0 0 0 6 0 0
5 4 0 0 0 0 0 1 9
0 0 2 * 0 0 7 0 0
0 9 0 0 3 0 1 8 0
2 0 0 8 0 4 0 0 7
0 1 0 9 0 7 0 6 0

No solution! system message : "unsat"

JeonghwanKim@DESKTOP-NKTTB3Q MINGW64 ~/Desktop (master)
$
```

```
1 2 0 5 0 * 0 9 0
8 0 1 2 0 3 0 0 6
0 3 0 0 6 1 0 7 0
0 0 * 0 0 0 6 0 0
5 4 0 0 0 0 0 1 9
0 0 2 * 0 0 7 0 0
0 9 0 0 3 0 1 8 0
2 0 0 8 0 4 0 0 7
0 1 0 9 0 7 0 6 0
```

Figure 4: *Left:* Commonly get printed result on bash shell and formula, output .txt file like above figure when the Sudoku problem is solvable. *Right:* Common get results when the Sudoku problem is unsolvable.

"Sudoku.exe" gives the solution and the input file to both the standard output and the two files. "output.txt" has the solution for the given 9-by-9 Sudoku, and "formula.txt" has the compound proposition solution of the given Sudoku in DIMACS format. (see Figure 4, left)

If the given input file contains a sudoku that is unsolvable, the program returns a message "No solution! system message: "unsat". (see Figure 4, right)



```
MINGW64:/c/Users/JeonghwanKim/Desktop
JeonghwanKim@DESKTOP-NKTTB3Q MINGW64 ~/Desktop (master)
$ ./Sudoku.out input.txt
0 2 0 5 0 * 0 9 0
8 0 0 2 0 3 0 0 6
0 3 0 0 6 0 0 7 0
0 0 * 0 0 0 6 0 0
5 4 0 0 0 0 0 1 9
0 0 2 * 0 0 7 0 0
0 9 0 0 3 0 0 8 0
2 0 0 8 0 4 0 0 7
0 1 0 9 0 7 0 6 0

4 2 6 5 7 1 3 9 8
8 5 7 2 9 3 1 4 6
1 3 9 4 6 8 2 7 5
9 7 1 3 8 5 6 2 4
5 4 3 7 2 6 8 1 9
6 8 2 1 4 9 7 5 3
7 9 4 6 3 2 5 8 1
2 6 5 8 1 4 9 3 7
3 1 8 9 5 7 4 6 2

JeonghwanKim@DESKTOP-NKTTB3Q MINGW64 ~/Desktop (master)
$
```

```
input - 메...
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
0 2 0 5 0 * 0 9 0
8 0 0 2 0 3 0 0 6
0 3 0 0 6 0 0 7 0
0 0 * 0 0 0 6 0 0
5 4 0 0 0 0 0 1 9
0 0 2 * 0 0 7 0 0
0 9 0 0 3 0 0 8 0
2 0 0 8 0 4 0 0 7
0 1 0 9 0 7 0 6 0
```

```
MINGW64:/c/Users/JeonghwanKim/Desktop
JeonghwanKim@DESKTOP-NKTTB3Q MINGW64 ~/Desktop (master)
$ ./Sudoku.out test.txt
0 2 0 5 0 0 0 9 0
8 0 0 2 0 3 0 0 6
0 3 0 0 6 0 0 7 0
0 0 0 0 0 0 6 0 0
5 4 0 0 0 0 0 1 9
0 0 2 0 0 0 7 0 0
0 9 0 0 3 0 0 8 0
2 0 0 8 0 4 0 0 7
0 1 0 9 0 7 0 6 0

4 2 6 5 7 1 3 9 8
8 5 7 2 9 3 1 4 6
9 3 1 4 6 8 2 7 5
1 7 9 3 8 5 6 2 4
5 4 3 7 2 6 8 1 9
6 8 2 1 4 9 7 5 3
7 9 4 6 3 2 5 8 1
2 6 5 8 1 4 9 3 7
3 1 8 9 5 7 4 6 2

JeonghwanKim@DESKTOP-NKTTB3Q MINGW64 ~/Desktop (master
$
```

```
test - 메...
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
0 2 0 5 0 0 0 9 0
8 0 0 2 0 3 0 0 6
0 3 0 0 6 0 0 7 0
0 0 0 0 0 0 6 0 0
5 4 0 0 0 0 0 1 9
0 0 2 0 0 0 7 0 0
0 9 0 0 3 0 0 8 0
2 0 0 8 0 4 0 0 7
```

Figure 5 **Depending on whether asterisk(*) is used or not,**

each red boxes shows different results on same Sudoku problem.

When there are "asterisks" (*) present within the given input file (i.e. test.txt), the output is different from the case when there are asterisks (*) within. The illustration shows the output when the asterisks are present within the input file (i.e. input.txt). It is evident that the two outputs are different.

6. Interesting Thoughts

Can we configure a compound proposition that predetermines the locations of the asterisks (*) that will give "unsat" as an output? Intuitively, it is apparent that asterisks in a given Sudoku increases the possibility of getting "unsat" as an output. The locations of the asterisks can turn a solvable Sudoku into a puzzle without an answer; which gives "unsat" as output. If we can find a mathematical formula (a compound proposition) that can figure out if the Sudoku is solvable or not based on the locations of the asterisks, we can effectively decrease the overall cost of the operation.

If the Sudoku turns out unsolvable with our "proposed" compound proposition, then there is no need for the SAT solver to go through every possible truth value for each cell.