

Andrew Davison

CSC 372: Survey of Artificial Intelligence

Dr. Thomas Allen

4/10/2023

A4: Solving Alphadoku Via SAT Reduction

Design, Data Structure, and Program Flow

My solver approaches Alphadoku using a SAT, or Boolean Satisfiability, approach in C++. This means that it turns the puzzle into a series of Boolean variables that are passed into a competition-grade SAT Solver, Slime, that returns a true or false assignment to those variables that it then maps back onto an Alphadoku board. Then then outputs that board to a file, and if a solution was found the first time, checks for a second to see if there are multiple valid solutions to the problem Alphadoku board.

Within the program, the Alphadoku puzzle is represented with a 25x25 char array that simply keeps track of what the literal problem board looks like. This makes it simple to access certain elements of the board when printing out the board and converting it into SAT.

Reduction to SAT

The bulk of the logic contained in the Alphadoku solver is in the transition from the internal data representation into DIMACS, which is the SAT file type that Slime reads. In order to do this, there are two separate phases: the game rules phase of clause generation and the puzzle board phase of clause generation. What this means is that there is a significant amount of information encoded into the DIMACS file that is the same for every instance of Alphadoku, since that information informs the SAT Solver of the rules of the game. Then, there are assertions that encode the actual information of the puzzle that is trying to be solved.

Both of these sets of clauses use variables. I mapped the variables such that the number 1, in SAT, meant that there was an A in position (1, 1). This was true for the first 25 numbers:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

These variables, in SAT, indicated that the corresponding numbers were in position (1, 1). Then, position (1, 2) used the numbers 26 – 50. This went on through the table so that a mapping of the variables to the positions they describe look something like this:

1-25	26-50	51-75	76-100	...
626-650	651-675	676-700	701-725	...
1251-1275	1276-1300	1301-1325	1326-1350	...
1901-1925	1926-1950	1951-1975	1976-2000	...
...

Mathematically, this meant that given some variable, that variable indicated:

$$row = \frac{variable}{25} \text{ (integer division)}$$

$$column = variable \% 25 \text{ (mod)}$$

$$symbol = symbol_table[(variable - 1) \% 25]$$

Where the symbol table is the same as the one shown above but indexed starting at 1.

The puzzle board series of clauses is simple. For every value given in the puzzle board to be solved, assert that the corresponding variable, found by doing the reverse of the mappings shown just above, must be true.

Additionally, if a solution has already been found, there is an accompanying encoding that a new solution must be different from the old one. To encode this into SAT, the program takes the assignment of every variable returned as a solution, negates them, and disjuncts all of the variables together. This is then asserting that at least one of the variables must have a different assignment than the assignment that it received in the first solution.

Finally, the game rules set of clauses is perhaps the most difficult. In order to properly encode the rules of Alphadoku, there are 4 distinct rules:

1. Each cell in the puzzle must have just one symbol
2. No symbols may repeat in each row of the puzzle.
3. No symbols may repeat in each column of the puzzle.
4. No symbols may repeat in each 5 x 5 subgrid of the puzzle (delineated by spaces in the puzzle format used for this solver).

The first rule can further be broken down into two rules: there must be at least one symbol in every cell, and every cell can have at most one symbol. To do these, the program first asserts, for a given cell, that at least one variable must be true that corresponds to that cell. For the first cell, this would look like this:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 0

And that, subsequently, it cannot be the case that any two of them could be true at the same time, so that there is at most one symbol in the cell. This is achieved by disjuncting the negation of

each pair of variables, so that if one is true (that is, its negation is false) the other's must be false (that is, its negation is true). That series of assertions would look something like this:

-1 -2 0
-1 -3 0
-1 -4 0
...
-24 -25 0

Next, the second rule requires that there cannot be more than one of each symbol in each row of the puzzle. To do this, there is a similar assertion made to the one just above, except that instead of between symbols within a cell, the assertions are made with variables that refer to the same symbol across different cells. For the first row, that set of assumptions looks like this:

-1 -26 0
-1 -51 0
...
-576 -601 0

This series of assertions makes it so that the letter 'A' does not repeat in the first row. These clauses are repeated for each of the 25 different possible symbols. This is done similarly for the columns, just that instead of finding the next variable by increasing by 25, the next number is found by increasing by 625 to go to the next row.

Finally, there must only be one of each symbol in the 5 x 5 subgrids. These assertions are made by looking at each entry in the 5x5 grids and asserting the same thing as above, that the disjunction of the negation of the variables must be true. The math behind this is more complicated, but it can be done by nesting loops so that 5x5 subgrids are considered like grids unto themselves until a larger outer loop is incremented (indicated by the loop variables *row_inc* and *col_inc* in the code). Then, those assertions are made for each variable with the corresponding variable in non-orthogonal cells in that grid:

-1 -651 0
-1 -676 0
...
































Using the Solver

Currently, the solver is hardcoded to iterate through the 30 problems presented in the puzzles subfolder and solve each of them, putting the output into another subdirectory titled solutions with numbers that correspond to the puzzle solved. This can easily be changed by removing the

loop in the main function and changing the filename strings, also easily found in the main function.

Results

This program is not verbose. Instead, a subfolder is filled with solutions:

 alphadoku_solution_0	4/9/2023 11:48 PM	Text Document	3 KB
 alphadoku_solution_1	4/9/2023 11:48 PM	Text Document	2 KB
 alphadoku_solution_2	4/9/2023 11:48 PM	Text Document	2 KB
 alphadoku_solution_3	4/9/2023 11:48 PM	Text Document	2 KB
 alphadoku_solution_4	4/9/2023 11:48 PM	Text Document	2 KB
 alphadoku_solution_5	4/9/2023 11:48 PM	Text Document	2 KB
 alphadoku_solution_6	4/9/2023 11:48 PM	Text Document	2 KB
 alphadoku_solution_7	4/9/2023 11:48 PM	Text Document	2 KB
 alphadoku_solution_8	4/9/2023 11:48 PM	Text Document	2 KB
 alphadoku_solution_9	4/9/2023 11:48 PM	Text Document	2 KB
 alphadoku_solution_10	4/9/2023 11:48 PM	Text Document	2 KB
 alphadoku_solution_11	4/9/2023 11:48 PM	Text Document	1 KB
 alphadoku_solution_12	4/9/2023 11:48 PM	Text Document	1 KB
 alphadoku_solution_13	4/9/2023 11:48 PM	Text Document	3 KB
 alphadoku_solution_14	4/9/2023 11:48 PM	Text Document	1 KB
 alphadoku_solution_15	4/9/2023 11:48 PM	Text Document	1 KB
 alphadoku_solution_16	4/9/2023 11:48 PM	Text Document	1 KB
 alphadoku_solution_17	4/9/2023 11:48 PM	Text Document	1 KB
 alphadoku_solution_18	4/9/2023 11:48 PM	Text Document	1 KB
 alphadoku_solution_19	4/9/2023 11:48 PM	Text Document	1 KB
 alphadoku_solution_20	4/9/2023 11:48 PM	Text Document	1 KB
 alphadoku_solution_21	4/9/2023 11:48 PM	Text Document	3 KB
 alphadoku_solution_22	4/9/2023 11:48 PM	Text Document	3 KB
 alphadoku_solution_23	4/9/2023 11:49 PM	Text Document	3 KB
 alphadoku_solution_24	4/9/2023 11:49 PM	Text Document	3 KB
 alphadoku_solution_25	4/9/2023 11:49 PM	Text Document	3 KB
 alphadoku_solution_26	4/9/2023 11:49 PM	Text Document	3 KB
 alphadoku_solution_27	4/9/2023 11:49 PM	Text Document	3 KB
 alphadoku_solution_28	4/9/2023 11:49 PM	Text Document	3 KB
 alphadoku_solution_29	4/9/2023 11:49 PM	Text Document	3 KB
 alphadoku_solution_30	4/9/2023 11:49 PM	Text Document	3 KB

Interestingly, you can see the solution found by looking at the file size: a size of 1 KB indicates that no solution was found, a size of 2 KB indicates that one solution was found, and a size of 3 KB indicates that more than one solution was found. Here are some sample outputs:

alphadoku_solution_0 - Notepad

```
File Edit Format View Help
PKDLB GHUEW JAOFI RTNQX MCVVS
UQCWM LBNJK VEGDP AIHYS FXTRO
NJVAI QDCRX SULTY BGOFM KWPHE
EHTFX YPMSO QCNRB LUWVK JDAIG
GYRSO VAIIFT MWKHX EJCDP ULQNB
```

```
YVQCF MNBKJ LIEXA GDPRT WUSOH
LBMNH UWQCE KJFGT IAVSO DPRYX
WUKJE DIHGR YOSPN QBMXC AVLTF
AIGRP STXOY UQD VW NLFKH EBM CJ
DTXOS FLVAP CHBMR JWYUE QNKG I
```

```
QMUKN WJLDC EVAYG FHRTI BSOXP
BDAIJ NUEMG RLQCF PSXOW THVKY
TFSXL RVAYH BPIUO MKEND CGJQW
OCWPG KQFTI NDXSH VY LJB RMUEA
VREHY PXOBS TMJWK UQACG LIFDN
```

```
CGLUQ TEWVN DBMOJ HPSIR YFXAK
IWNVD CGKUM PRYQE XFJLA SOHBT
MEPTA JORXL FNHKS DVBWY IQGUC
FSOBK HYDPA GXUIC TMQEN VRWJL
JXH YR BFSIQ ATWLV OCKGU NEDPM
```

```
KL BQU XMJNF WGTED YOIHV PACSR
RNJMC EKGHU ISVAQ WXTPF OYBLD
HAFEW ISTLD OYPJU CRGBQ XKNMV
SOYGV ARPQB XFCNL KEDMJ HTIWU
XPIDT OCYVW HKRBM SNUAL GJEFQ
```

```
PKDLB GHUEW JAOFI RTNQX MCVVS
UQCWM LBNJK VEGDP AIHYS FXTRO
NJVAI QDCRX SULTY BGOFM KWPHE
EHTFX YPMSO QCNRB LUWVK JDAIG
GYRSO VAIIFT MWKHX EJCDP ULQNB
```

```
YUQCF MNBKJ LIEXA GDPRT WVS OH
LBMNH UWQCE KJFGT IAVSO DPRYX
WVKJE DIHGR YOSPN QBMXC AULTF
AIGRP STXOY UQD VW NLFKH EBM CJ
DTXOS FLVAP CHBMR JWYUE QNKG I
```

```
QMUKN WJLDC EVAYG FHRTI BSOXP
BDAIJ NUEMG RLQCF PSXOW THVKY
TFSXL RVAYH BPIUO MKEND CGJQW
OCWPG KQFTI NDXSH VY LJB RMUEA
VREHY PXOBS TMJWK UQACG LIFDN
```

```
CGLUQ TEWVN DBMOJ HPSIR YFXAK
IWNVD CGKUM PRYQE XFJLA SOHBT
MEPTA JORXL FNHKS DVBWY IQGUC
FSOBK HYDPA GXUIC TMQEN VRWJL
JXH YR BFSIQ ATWLV OCKGU NEDPM
```

```
KL BQU XMJNF WGTED YOIHV PACSR
RNJMC EKGHU ISVAQ WXTPF OYBLD
HAFEW ISTLD OYPJU CRGBQ XKNMV
SOYGV ARPQB XFCNL KEDMJ HTIWU
XPIDT OCYVW HKRBM SNUAL GJEFQ
```

alphadoku_solution_1 - Notepad

```
File Edit Format View Help
BUICKG QWMDC FNOPR TLVHJ SAYXE
QCLWM FJUBN VKEAY GDSOX HTRPI
REXSA KYVOP THJDI QUF CW BGMNL
JVHPT EXAIR UQSLG BMNKY CDFWO
NDFYO HLGST CBMXW IREAP UKQVJ
```

```
UQCB L WMNJK DEIOA VTPYR GXHSF
WNKED UCBAS QTLGF HJIXO RVP MY
ATVOF DQLEG HPYRX MNKSB ICWJU
MIPHY VROXF BJWKS DCUGQ TNLEA
GSRJX YTPHI MCNUV EFWLA KQODB
```

```
OWUFQ CVJME NDXBP KIYRG ALSHT
CBJLN XK FUY WMGIH SATED POVQR
IMEVH BDQWA KURST LPONF JYCGX
TAYGK NPSRL OVCFJ UWXQH DEBIM
PXDRS IGH TO ALQYE JBCMV NFUKW
```

```
LKQUC MEXGW JSVTN POBFI YRDAH
DOWMJ LFTPU IAHEK RYQVS XBNCG
EGATP JNKQH RYFMB CXDWL VUIOS
YRN XVS BICD GWUQO AKHTM LJEF P
HFSIB AORYV PXDCL NGJUE WMKTQ
```

```
KJBCW GAENQ LOPHM YSRDT FIXUV
VYOQR PIWKB XGANC FHMJU ESTLD
FPGDE THYVM SRKJU XQLIC OWABN
SHMAI RUCLX EFTWD OVBGN QPJYK
XLTNU OSD FJ YIBVQ WEAPK MHGRC
```

Solution is unique.

alphadoku_solution_11 - Notepad

File Edit Format View Help

No solution

Lessons Learned from this Assignment

In this assignment, I learned how valuable it can be to reduce a problem to SAT or to a problem instance of something that has already been solved and has had very intelligent and efficient solvers created already that can do this work very well. Additionally, I learned more about what that process of reducing a problem to SAT actually looks like outside of theory. Finally, I've learned more about C++, particularly how to manage external processes and some threading -- which is technically how Slime is called -- which I had never done before.

Acknowledgements and References

This is the SAT solver that I used. I chose to use this one over MiniSAT because of Windows compatibility issues. There is a slime executable file included in my code zip folder that should be able to work for any testing.

[SLIME | A Free World Class High Performance SAT Solver \(maxtuno.github.io\)](https://maxtuno.github.io/SLIME/)

This is a stack overflow post that I used to determine the best way to call Slime and capture its output.

[process - How do I execute a command and get the output of the command within C++ using POSIX? - Stack Overflow](#)