

Project 2- Solving Sudoku as Constraint Satisfaction Problem Using Propositional Logic

Dan Scarafoni

Overview

This project's main goal was to find a means of solving the sudoku puzzles. The program developed uses a java program and an instance of the MINISAT logic engine combined with several text files, all run by a shell program in order to solve the problem.

The Game

Standard sudoku is played on a 9x9 board. Every spot on the board is filled with an integer 1-9. The goal is to fill the entire board with integers, such that no spot is empty. Each board has nine columns, nine rows, and nine 3x3 subsections. A standard sudoku board is shown in figure 1.

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

figure 1: a sudoku board

<http://www.logicgamesonline.com/images/sudoku-puzzle-256.png>

Sudoku can be played on a larger or smaller board, so long as the length of the board has an integer square root. Figure two shows a 16x16 sudoku board.

1			2	3	4			12		6				7	
		8				7			3			9	10	6	11
	12			10			1		13		11				14
3			15	2			14				9				12
13				8			10		12	2		1	15		
	11	7	6				16				15			5	13
			10		5	15			4		8				11
16			5	9	12			1						8	
	2						13			12	5	8			3
	13			15		3			14	8		16			
5	8			1				2				13	9	15	
		12	4		6	16		13			7				5
	3			12				6			4	11			16
	7			16		5		14			1			2	
11	1	15	9			13			2				14		
	14				11		2			13	3	5			12

figure 2: a 16x16 board

<http://www.sudoku.4thewww.com/other/16x16-350.gif>

There are a number of constraints for the sudoku problem. There can only be one instance of each number one through 9 (or one through n on an nxn board). In a given row, column, and subsection of the board. Because of this, the game sudoku can be viewed as a constraint satisfaction problem, in which the individual squares are variables, the domains are the numbers one through nine inclusive, and the constraints are that no two spots in a given row, column, or subsection can contain the same value. This is shown more formally in figure 3.

$$\begin{aligned}
 \text{Variables} &= \{x_{11}, x_{12}, \dots, x_{99}\} \\
 \text{Domains} &= \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
 \text{Constraints} &= \{ \text{row constraints } (x_{11} \neq x_{12} \dots x_{11} \neq x_{19}), \\
 &\quad \text{column constraints } (x_{11} \neq x_{21} \dots x_{11} \neq x_{91}), \\
 &\quad \text{subsection constraints } (x_{11} \neq x_{12} \dots x_{11} \neq x_{33}) \}
 \end{aligned}$$

figure 3: the constraints for a 9x9 sudoku board

For an nxn sudoku board, the same constraints can be extrapolated. They are shown in figure 4

$$\begin{aligned}
Variables &= \{x_{11}, x_{12}, \dots, x_{nn}\} \\
Domains &= \{1, 2, 3, 4, 5, 6, 7, 8, 9 \dots n\} \\
Constraints &= \{ \text{row constraints } (x_{11} \neq x_{12} \dots x_{11} \neq x_{1n}), \\
&\quad \text{column constraints } (x_{11} \neq x_{21} \dots x_{11} \neq x_{n1}), \\
&\quad \text{subsection constraints } (x_{11} \neq x_{12} \dots x_{11} \neq x_{\sqrt{n}\sqrt{n}}) \}
\end{aligned}$$

figure 4: an extrapolation of the constraints of sudoku for an nxn board

the actual CSP implemented is more complex and contains more constraints than above, as will be discussed more in the algorithms section.

Algorithms

Then general rule for the program is as follows


1. *convert sudoku board into propositional logic*
2. *convert all constraints into propositional logic*
3. *send variables , constraints into logic engine*
4. *retrieve satisfactory variable values  engine*
5. *complete board with obtained values*

figure 5: the general algorithm for the solver

Converting the board into a form readable by the logical engine proved to be difficult, however.

The logical engine only understood propositional logic statements represented in conjunctive normal form. This presented a very narrow syntax, as all constraints had to be translated from a language that recognized integers, array's, and basic mathematical equivalence operations into one that only understood propositional literals, negations, logical or, and logical any (if used to join separate statements). In semantic terms, the logical engine had to be fed a very long and exhaustive amount of clauses specifying the exact truth state of the board.

Because the logic engine requires all boolean clauses (in conjunctive normal form) and variables as input, it would be impossible to feed the integer value of given states as expressed in the more general formula in the previous section. Because of this, every integer value had to be converted into boolean form. Each possible space could have the values one through nine. Thus, there were a total of $9^3 = 729$ total variables that needed to be represented for the complete 9x9 board.

This, in turn, added some extra constraints to the system. Now, along with the previous constraints, only one variable (out of nine) in a given square could be set to true. The remainder had to be false. This represents the fact that no square could contain multiple values. This constraint is shown in figure 6

$$x_{111} \vee x_{112} \vee x_{113} \dots x_{118} \vee x_{119},$$

$$x_{111} \neq x_{112} \wedge x_{111} \neq x_{113} \dots,$$

$$x_{112} \neq x_{113} \dots,$$

$$x_{118} \neq x_{119}$$

figure 6: the additional constraint

The “not equals” shown above can be reduced to the propositional logical form below.

$$P \neq Q = \neg(P \wedge Q)$$

This, in turn can be reduced to conjunctive normal form via DeMorgan's laws.

$$\neg(P \wedge Q) = \neg P \vee \neg Q$$

Thus, the additional constraint shown in figure 6 can be reduced to conjunctive normal form, shown in figure 7.

$$x_{111} \vee x_{112} \vee x_{113} \dots x_{118} \vee x_{119},$$

$$\neg x_{111} \vee \neg x_{112} \wedge \neg x_{111} \vee \neg x_{113} \dots,$$

$$\neg x_{112} \vee \neg x_{113} \dots,$$

$$\neg x_{118} \vee \neg x_{119}$$

figure 7: conjunctive normal form

For added consistency, the previously mentioned constraints could also be thought of in this way. Because each row, column, and subsection had exactly one of each number at a given depth, we could imagine grouping the individual squares in these constructs (each an array of booleans) and, for each layer (number), performing the operation in figure seven. This is based on the principle that each construct has exactly one value of one (the boolean at depth 1 is true) one two (boolean at depth two is true) etc... This enabled the above algorithm (which shall be referred to as `onlyOne()`) to be reused for efficiency's sake. This algorithm took several arrays of variables and outputs a series of logical clauses that can only be satisfied if exactly one variable in the array is true. The pseudo code for this function is shown in figure 8 for clarification purposes.

```
onlyOne (boolean array[] x) returns boolean array[] constraints {
constraints.add(toAdd)
for (i=0; i<x.length; i++)
    for (j=i+1; j<x.length; j++){
        constraints.add( $\neg x[i] \vee \neg x[j]$ )
    }
return constraints
}
```

figure 8: the `onlyOne` function

The core of the function was the logical engine, the code that took in a series of propositional logic statements and used constraint propagation to determine the correct value for each variable. The engine used for most of this project was the MINISAT engine produced by Niklas Een and Niklas Sorensson. This program used the DPLL algorithm, backtracking (through analysis of conflicts), and clause recording in order to solve the CSP.

One of the core components of the algorithm is constraint propagation. From the various clauses fed into the solver, inferences about the state space were made. The various constraints imposed were used to narrow down the number of possible worlds that could constitute the solution to the problem. The primary algorithm for this in MINISAT is the DPLL algorithm, shown in figure 9.

```

function DPLL-SATISFIABLE?(s) returns true or false
  inputs: s, a sentence in propositional logic

  clauses  $\leftarrow$  the set of clauses in the CNF representation of s
  symbols  $\leftarrow$  a list of the proposition symbols in s
  return DPLL(clauses, symbols, { })

```

```

function DPLL(clauses, symbols, model) returns true or false
  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  P, value  $\leftarrow$  FIND-PURE-SYMBOL(symbols, clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model  $\cup$  {P=value})
  P, value  $\leftarrow$  FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model  $\cup$  {P=value})
  P  $\leftarrow$  FIRST(symbols); rest  $\leftarrow$  REST(symbols)
  return DPLL(clauses, rest, model  $\cup$  {P=true}) or
    DPLL(clauses, rest, model  $\cup$  {P=false})

```

figure 9: the pseudo code for the DPLL algorithm

This algorithm not only contains the code necessary for constraint propagation, but also contains a number of heuristics, backtracking, and a depth-first enumeration of possible worlds in order to solve the problem. It uses a “pure symbol” heuristic, which follows the principle that a symbol that is always listed as true in the constraints given can never be false in order to narrow down options.

Another implemented heuristic, the unit clause heuristic, any clause in which all but one of the variables are assigned false is treated as a single unit clause. This helps narrow down the possible states of the program considerably.

Backtracking is also an integral part of the algorithm. When contradictory state is reached (one that violates constraints given), the program backtracks through variable assignment and chooses a different path through the state space. This form of simulated annealing provides a search algorithm which is effective at finding the solution. MINISAT in particular comes with learning algorithms, which it uses to document variable assignment combinations that cause violations of the constraints; this in turn allows the program to avoid making the same mistake twice, and cuts down considerably on run time.

Program

Implementing these algorithms into a machine computable form was not easy. The main program was written in Java. Other language were up for consideration, including Python and Prolog. Python was considered because it allowed more forms of string manipulation than Java, which was theorized to be very useful as the main file would be reading from and writing to text files. However, because this language was much slower than Java, and in the end only basic reading and writing functions were necessary, Python was not chosen.

Prolog was also considered as it is very effective for propositional logic. However, because a pre-made logical engine was to be used, Prolog's advantages were quickly minimized. In the end, Java was chosen because of it's large library (which provided a wealth of data structures and built in functions that would be of great use), and because it runs faster than Python. We were also provided with starting code in this language.

The Java program consisted of three separate classes. The first class, Sudoku, contained the data for the Sudoku board, as well as methods for accessing elements therein. It was also the driver for the Java program. There were methods for obtaining the values of individual cells, the values of each row, column, and subsection of the board. The board was stored as a 2d array of integers. There was some debate as to whether or not a three dimensional array of booleans would be more optimal (to more closely simulate the data that would be passed to the logical engine). In the end, however, this was scrapped. The two dimensional array of integers was much smaller (by an order of magnitude) which greatly decreased the space complexity of the program. Although the entire board was converted into 729 variables, this was done in a separate program as the logical clauses were being generated.

The Solver class contained all of the methods necessary for converting the sudoku board (and

all of its constraints) into a form readable by the logic engine. This means that this board contained, for example, the onlyOne function mentioned above. This form also contained the methods necessary for reading the variable assignments back into the sudoku puzzle after the logic engine had solved the problem.

The third class, Translator, contained only two methods and was used to read data from and write data to two text files which interacted with the logical engine for the program.

The logical engine was a compiled binary file using the MINISAT solver. This program was fed in a text file (in our case toTranslate.txt) and output the result into another text file (translated.txt).

Run.sh was a shell script that coordinated and ran the overall program. It took input from the user specifying the puzzle the user wished to solve, and the logical engine the user wished to use.

Figure 10 below shows the general flow of information for the overall project.

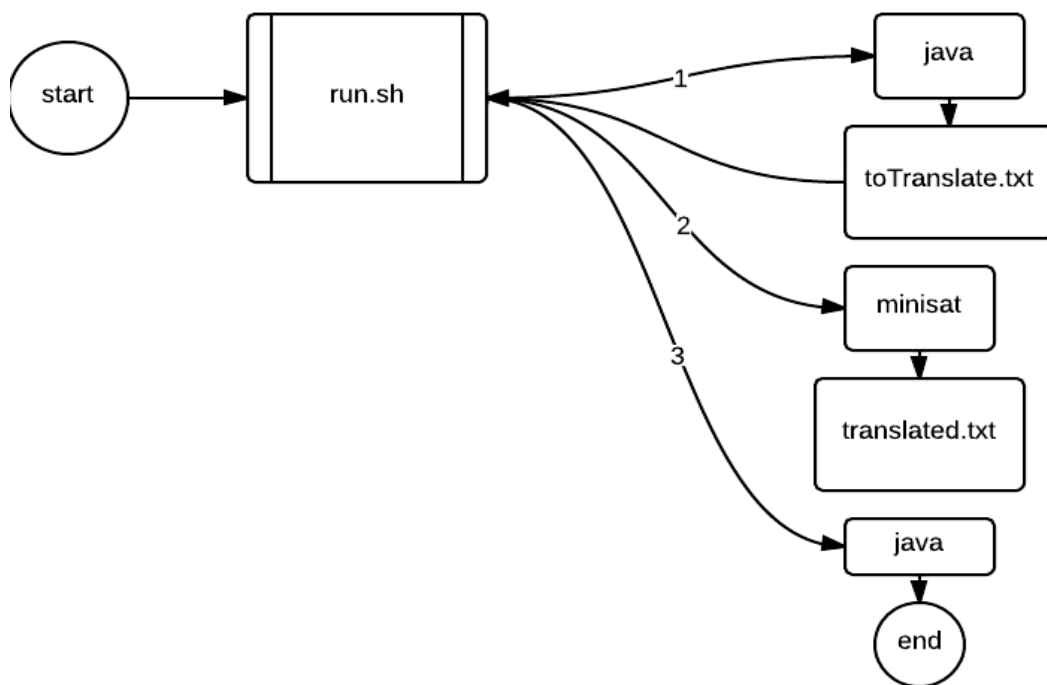


Figure 10: the flow of information of the program

As is shown by the diagram, run.sh controls all major functions of the program; it calls the java functions and MINISAT. The java function was called twice, in each instance doing a different task. The first time the program was called, it read in the puzzle file, produced all constraints for the puzzle, and printed them out in conjunctive normal form into toTranslate.txt. The flow of information for this program is shown in figure 11.

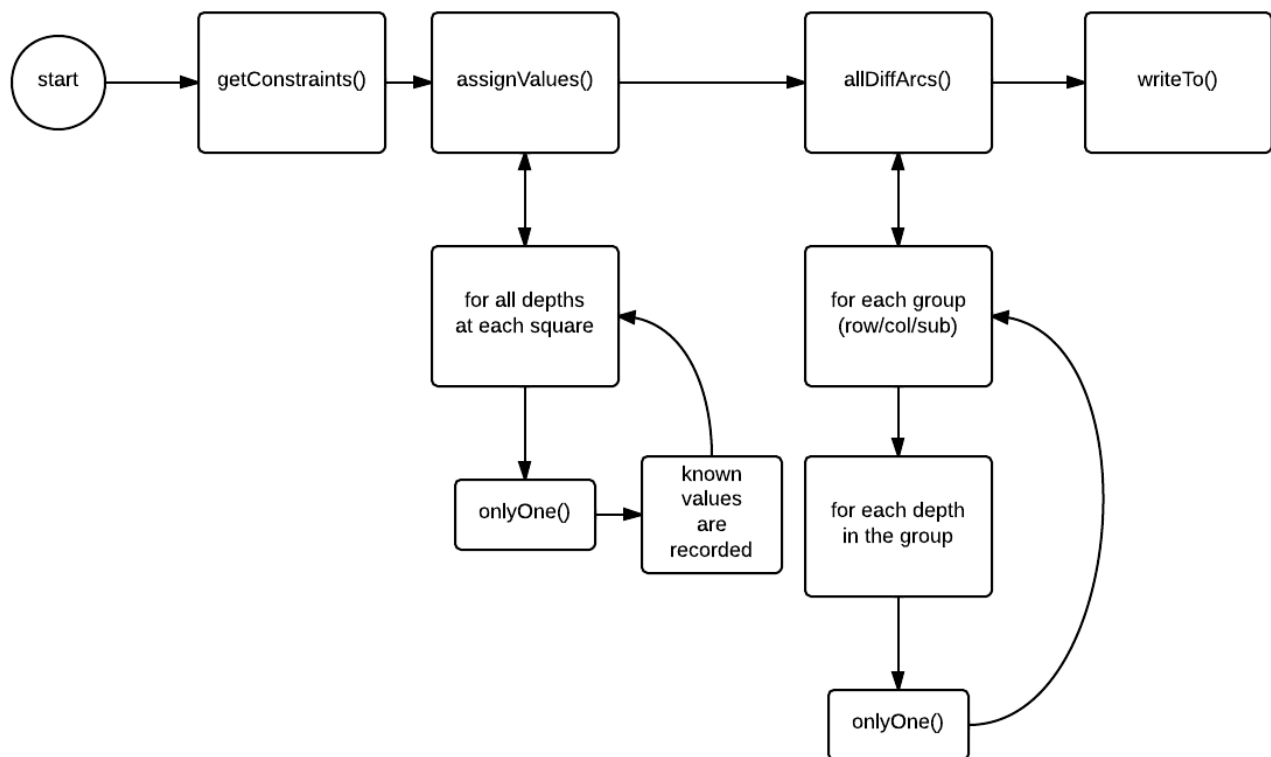


figure 11: generation of constraints in java

In the above diagram, onlyOne() refers to the function described previously, which takes in a list of variables and outputs a series of logical statements that can only be satisfied if exactly one of those variables is true. AssignValue() does this for each square; it runs onlyOne() on the list of all possible variables (1-9). AllDiffArcs() does the same function for each variable in each group (row,

col, or subsection) of the board. These variables are differentiated by their “depth” because all of the variables for the board are stored conceptually as a three dimensional array (as described in algorithms). The second part of the program is shown in figure 12.

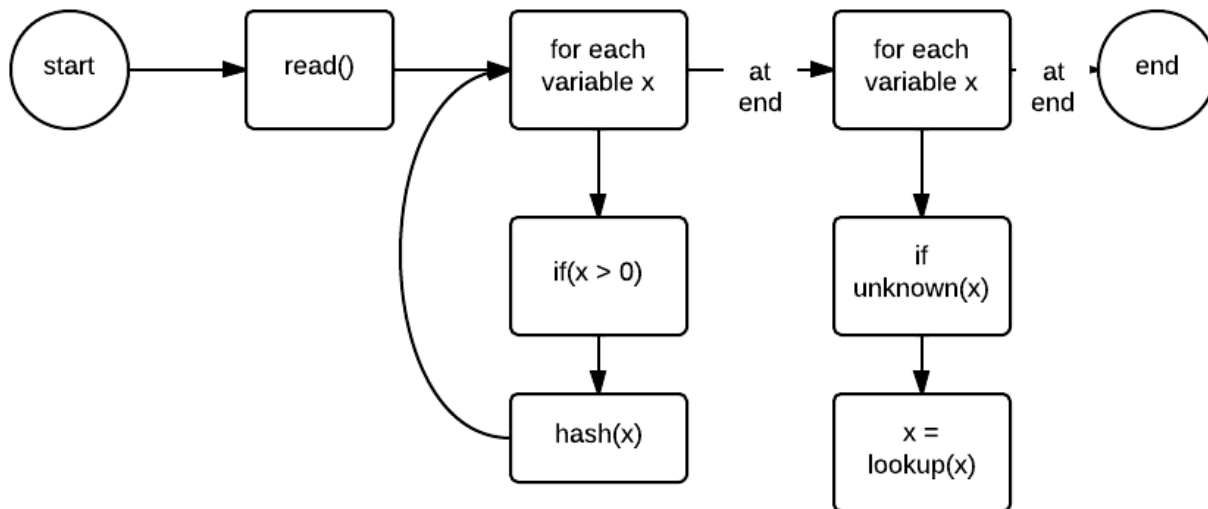


Figure 12: reading in the solved problem

This program takes in translated.txt, and hashed all positive values. Because a positive variable translated to a numerical assignment in the project, only the positive values needed to be read in. Once all relevant data was fed in, this Hash table was fed into another method which searched through all spots in the board. If that spot had not value, each variable for that square (representing the numbers 1-9) as searched for in the hash table, and the corresponding variable representing the value of the square was returned.

Results

Because our the most complex algorithm in the program ran in cubic time, the program is deduced to have a $O(n^3)$. Run time tests were done using the “time” command found in linux shell on CSUG computers. A variety of puzzles of different sizes were tested (including a custom made 16x16 puzzle. Puzzles of varying completion were also tested, the data is below in figure 13. Each puzzle was run ten times each, and the average time was recorded for each.

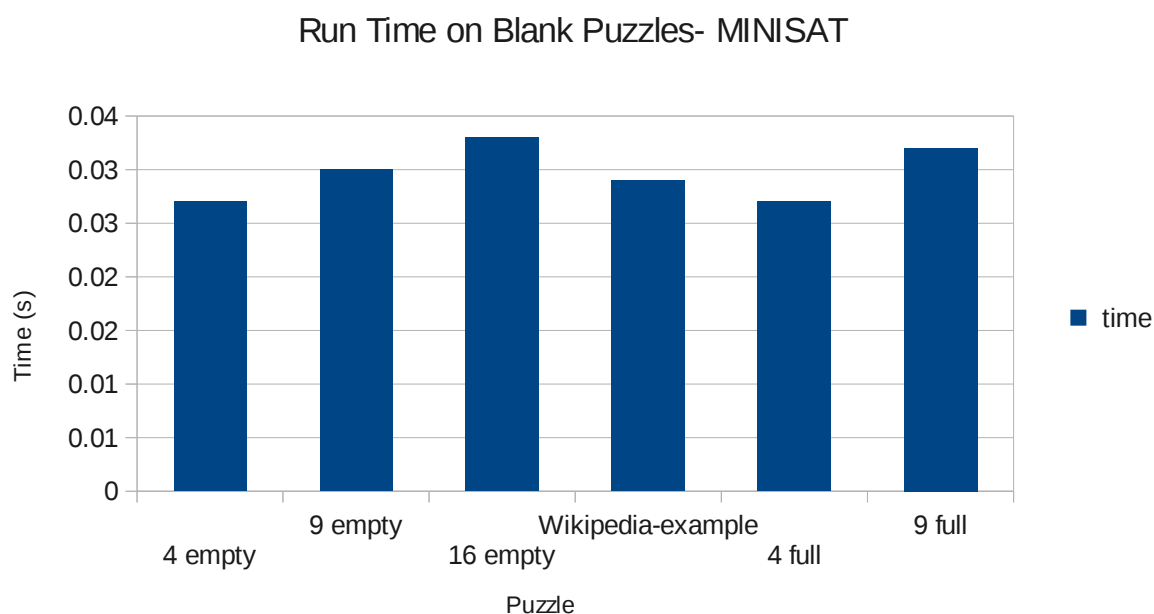


figure 13: run time of various programs

While the program does seem to take longer amounts of time on larger boards, the amount of filled spaces in the grid does not seem to alter the time needed to solve the problem. The code solved an empty 9x9 board almost as fast as a full 9x9 board, even though the latter is trivially easy to solve. Paradoxically, it solved the empty 4x4 board faster than the full version.

Other logic engines were also tested. WALKSAT, a program developed in part by University of

Rochester's own Henry Kautz, is a logical engine that solves conjunctive normal form logical statements, just like MINISAT. The above tests were done on WALKSAT, and the results are displayed in figure 14.

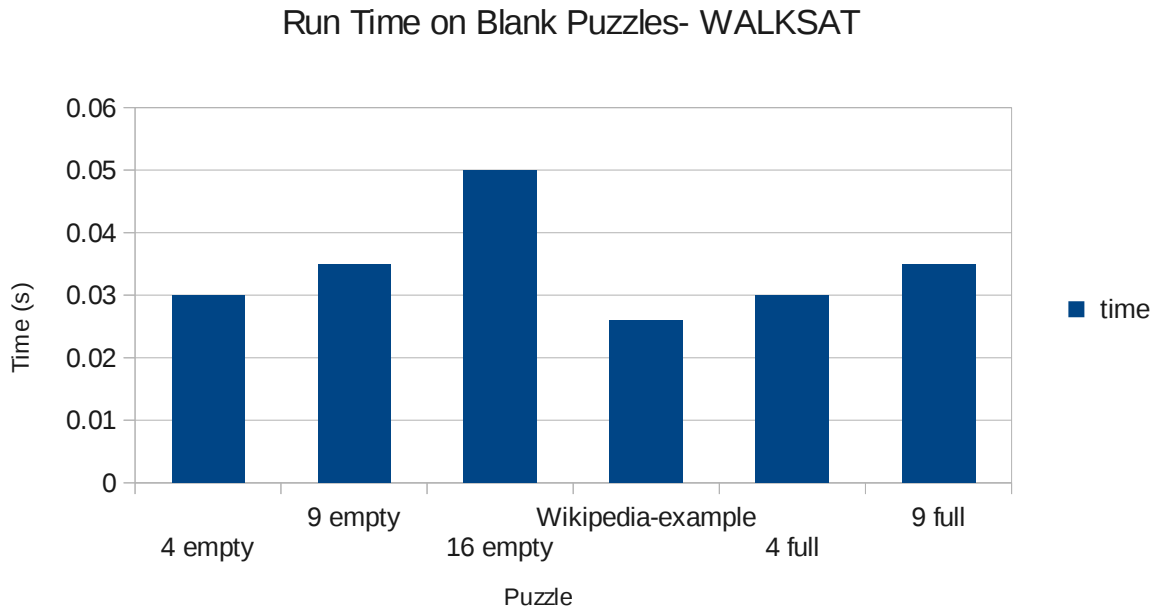


figure 14: run time analysis of WALKSAT

Like MINISAT, WALKSAT seems to take increased time for larger input. It differs, however, in that it is slower. Every puzzle took longer to solve in WALKSAT, and the 16x16 puzzle took the longest of any puzzle tested. Further, WALKSAT would often not solve the puzzles given, which implies that there may be some bug in the code. For example, the following screen shot was taken shortly after running the WALKSAT on the wikipedia-example puzzle.

```
dscarafo@colden:bin
File Edit View Search Terminal Help
    successful ratio mean numbad to mean std dev = 1.925964
statistics on unsuccessful runs:
    unsuccessful mean average numbad level = 0.000000
    unsuccessful mean numbad std deviation = 0.000000
    unsuccessful ratio mean numbad to mean std dev = 0.000000
ASSIGNMENT FOUND
9
+---+---+---+
|53.|374|678|
|691|195|267|
|298|195|364|
+---+---+---+
|881|968|343|
|425|863|781|
|759|726|146|
+---+---+---+
|263|426|288|
|537|419|915|
|713|982|479|
+---+---+---+
0.265u 0.029s 0:00.26 107.6%    0+0k 0+432io 0pf+0w
[dscarafo@colden bin]$ print screen
print: Command not found.
[dscarafo@colden bin]$
```

As is apparent, one of the spots in the puzzle is blank. Two slots are left blank on the blank 9x9 puzzle, as shown below.

```
dscarafo@colden:bin
File Edit View Search Terminal Help
successful mean average numbad = 16.783454
successful mean numbad std deviation = 5.969191
successful ratio mean numbad to mean std dev = 2.811680
statistics on nonsuccessful runs:
nonsuccessful mean average numbad level = 0.000000
nonsuccessful mean numbad std deviation = 0.000000
nonsuccessful ratio mean numbad to mean std dev = 0.000000
ASSIGNMENT FOUND
9
+---+---+---+
|.7|892|365|
|983|654|712|
|652|731|849|
+---+---+---+
|271|465|938|
|345|987|126|
|869|213|457|
+---+---+---+
|794|126|583|
|136|578|294|
|528|349|671|
+---+---+---+
0.200u 0.029s 0:00.20 110.0% 0+0k 0+432io 0pf+0w
[dscarafo@colden bin]$
```

Conclusions and Discussion

There could be more effort of research done into the optimization of the java code. Although a number of improvements were made (for example, using a hash table instead of a linked list in part two so that the values of variables could be more easily looked up), however, there are certainly more that could be done in order to fully optimize the code used.

The MINISAT solver provided a very effective means of solving the problem, so much so that I believe it may be worth researching using it to solve puzzles of very very large size (for instance,

125x125 puzzle). The program used can already handle puzzles of arbitrary size, and it would be interesting to see the effect of very large constraint sets on the performance of MINISAT. Further, there could be more research done into the bugs encountered when using WALKSAT. As noted above, the program occasionally failed to solve certain puzzles that MINISAT had no trouble with. This is almost certainly not due to an error in file formatting as both MINISAT and WALKSAT take in the same formatting of files.