Applying SAT Solving to Sudoku Puzzles

Radon Rosborough

Grade 8, Summit Charter Middle School
Boulder, Colorado

Abstract

A method is presented to solve Sudoku puzzles using a satisfiability solver. The latter is an engine used to study Boolean expressions, and can be applied to many different computer science applications, by reducing the problems to Boolean expressions. The method was programmed in Python. Both symmetric and non-symmetric Sudoku puzzles of varying perceived difficulties were tested using this method. Time to complete the solution and the number of calls to an internal function of the solver were recorded. Resulting analysis of the solutions showed that time and calls to the internal function were directly related to the difficulty of the puzzle.

Table of Contents

1. Introduction & Background

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

Figure 1. An unsolved Sudoku puzzle.

The first known Sudoku puzzles were popularized in Japan in the year 1986. They later became an international hit. The problem of filling in the 9x9 grid with the numbers one through nine so that the same number does not appear in any row, column, or 3x3 box more than once, is an interesting puzzle. It is a common problem in computer science to design algorithms to solve this puzzle.

For my science fair project, I coded a satisfiability (SAT) solver in the programming language Python and applied it to test how various types of Sudoku puzzles (based on difficulty and symmetry) affect two variables. The first is the time it takes to solve the puzzle. The second is the number of recursions the solver takes to solve it, as the program is inherently recursive (meaning that it calls itself). I had predicted as my hypothesis that both solve time and recursions would have a direct relationship with difficulty and no relationship with symmetry.

What is a SAT solver?

A SAT solver takes a Boolean expression, such as x AND NOT (x OR y), and checks whether, by assigning the variables to TRUE or FALSE, the expression can be made to equal TRUE. There are three binary operators that are mainly used: AND, OR, and NOT.

x	y	x AND y	x OR y	NOT x	NOT y
F	F	F	F	T	T
F	T	F	T	T	F
T	F	F	T	F	T
T	T	T	T	F	F

F stands for FALSE.
T stands for TRUE.

These operators, when combined with variables, make up Boolean expressions for the SAT solver to evaluate. For instance, take the expression x AND NOT (x OR y). For the entire expression to evaluate to TRUE, both sides of the AND must be TRUE; i. e. both x and NOT (x OR y) must be TRUE. But if x is TRUE, then x OR y evaluates to TRUE, so NOT (x OR y) is FALSE. This leaves TRUE AND FALSE, which is FALSE. Therefore, the expression x AND NOT (x OR y) cannot be made TRUE; in other words, it is *unsatisfiable*.

A Boolean expression can be written to represent the validity of a solved Sudoku puzzle; i.e. if the solution is valid, it will return TRUE, otherwise it will return FALSE. An unsolved Sudoku puzzle can be represented using that expression, with certain variables already assigned for the starting numbers. Then, this expression can be solved using a SAT solver. If the solver is adapted not only to return the satisfiability of the puzzle (hopefully TRUE), but also the variables that were assigned by the solver, then these variables can be decoded into the finished Sudoku. Of course, for this project, the actual solution is not the important part, but how long the solver takes to find it. By writing a script to automatically solve different types of puzzles, based on perceived difficulty and symmetry, and record the times taken, the different kinds of puzzles can be compared in "computational" difficulty.

As for the (Boolean) expression itself, its general form is of several conditions, such as the row and column rules, and some other properties of Sudoku puzzles (to aid the solver), connected with AND statements. However, a Sudoku puzzle is made of numbers, and the SAT solver works only with Boolean variables. This problem can be reconciled by using 729 variables - nine for each of the 81 cells of the grid. Each variable represents whether a certain cell is equal to a certain number (one to nine). The nomenclature that I will use for these variables is S_{rcn}, where r is the row, c the column, and n the proposed number in that cell. For instance if S_{473} is TRUE, then the cell in row four, column seven, is three; and if S_{315} is FALSE, then the cell in row three, column one, is *not* five.

Using this notation the rules for a valid Sudoku solution can be written as:

$$\left(\bigwedge_{r=1}^{9}\bigwedge_{n=1}^{9}\bigvee_{c=1}^{9} S_{rcn}\right)\wedge\left(\bigwedge_{c=1}^{9}\bigwedge_{n=1}^{9}\bigvee_{r=1}^{9} S_{rcn}\right)\wedge$$
$$\left(\bigwedge_{r=1}^{9}\bigwedge_{c=1}^{9}\bigvee_{n=1}^{9} S_{rcn}\right)\wedge\left(\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{n=1}^{9}\bigvee_{r=1}^{3}\bigvee_{c=1}^{3} S_{r+3i,c+3j,n}\right)$$

Each part of the expression represents a different rule. For instance, the first part represents the rule that each row must have the numbers one through nine. To be specific, the large symbols are, in essence, summations, except using Boolean operators (OR for the $\vee$ and AND for the $\wedge$) instead of addition. So, the first mini-expression means that *for all rows*, and *for every number one to nine*, that number appears in *at least one column* in that row. Note that all 729 variables appear in this expression, joined by AND's and OR's. This expression is then followed by expressions representing the column rule, the rule that each cell must have at least one number in it, and the sub-box (3x3) rule. This expression, however, is actually only part of the full expression that is used in the program, which includes extra rules added for efficiency, to aid deduction (but not needed for *correctness*). In total, the entire expression has 26,244 terms (there are only 729 S_{rcn} variables but they of course appear multiple times).

My implementation of the SAT solver in Python uses, as its main algorithm, the backtracking search. This algorithm is recursive, and I am counting, as a responding variable, the number of times that *satr*, the function that implements the recursion, is called. This is because program execution time is not a very accurate measure of the "computational difficulty" of the puzzle, as it could be affected by many outside factors, and will not be the same every time. This is not the case with the number of calls to *satr*.

The backtracking search algorithm consists of the program repeatedly simplifying the expression by assigning variables, and then automatically returning TRUE when it finds that it is possible to satisfy the expression, instead of going through all possible assignments of the variables (which, with 729 of them, could take a long time!) It also includes two optimizations:

simplifying the expression; i.e. x AND FALSE to FALSE and x AND TRUE to x. The other optimization is to deduce the values of variables: for instance, if the expression is x OR (x AND y), then x *must* be TRUE.

The main structure of the program is shown below, as a flow chart.

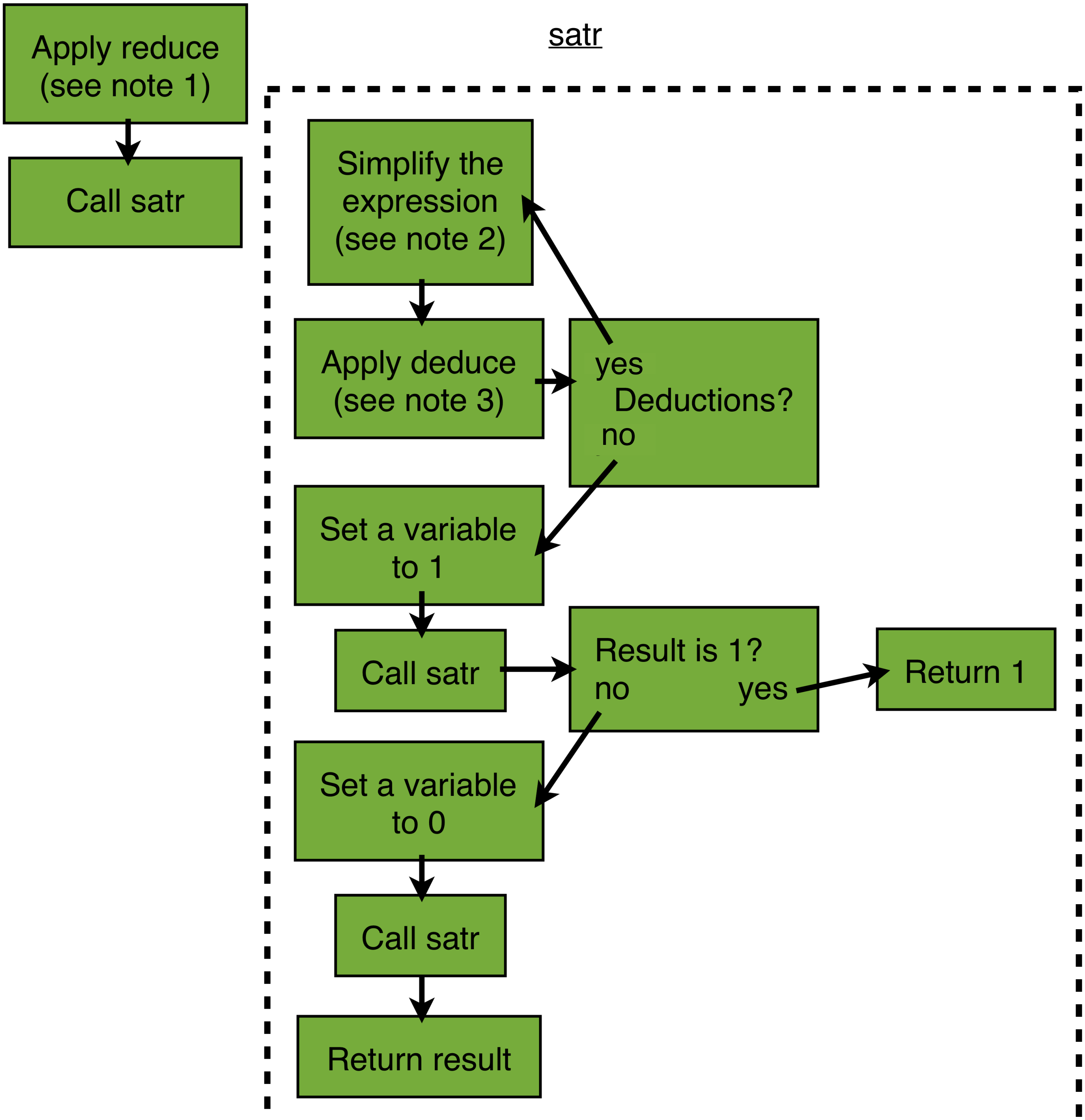

Figure 2. The flow chart of the program. See notes on the next page describing the separate functions.

Note 1.

reduce applies DeMorgan's Law. This pushes negations down to the variable level, making the rest of the solve more efficient. According to DeMorgan's Law,

NOT (x OR y) = NOT x AND NOT y
NOT (x AND y) = NOT x OR NOT y.

Therefore, when the law is applied to a sample expression,

NOT (x AND NOT (x OR y)) = (NOT x) OR x OR y.

Note 2.

simplify takes the variable assignments found in *deduce* and enters them into the expression. For instance, if the expression is x AND (y OR z) and y is TRUE, then

x AND (y OR z) = x AND (TRUE OR z) = x AND TRUE = x.

Note 3.

deduce looks at all the variables in the expression, and if there is only one value for them that could make the expression evaluate to TRUE, then it assigns them.

For instance, in x AND (NOT x) AND (NOT y), *deduce* finds that (1) x must be TRUE, (2) x must be FALSE, and (3) y must be FALSE. From this, a contradiction is reached, because x cannot be both TRUE and FALSE. Therefore, the expression simplifies to FALSE, which is returned. This expression would be immediately deemed unsatisfiable.

Let's look at another example: x AND (y OR z). First, *deduce* finds that x must be TRUE. (Both y and z individually don't have to be TRUE; only one or the other.) Then, in the next step (refer to the flow chart), *simplify* would do the assignment:

x AND (y OR z) = TRUE AND (y OR z) = y OR z.

2. Materials and Methods

Most of the time of the project was spent coding the SAT solver. After it was working, 480 Sudoku puzzles were generated using an online generator at http://www.jaapsch.net/sudoku.htm: 80 with no symmetry, 80 with diagonal symmetry, 80 with orthogonal (horizontal and vertical) symmetry, and 80 with both symmetries. For each puzzle, a difficulty rating was also gathered, ranging from 1 to ~850. The difficulties of the puzzles were fairly evenly distributed logarithmically; that is, there were approximately the same number at 1, from 2 to 10, from 11 to 100, and 101 to 1000. Therefore, on the graphs and data analysis, I used a logarithmic scaling for the difficulty (x) axis. There is not an agreed upon method for assigning the difficulty of a Sudoku puzzle. The method used by the online generator generally, but not exactly, reflect the difficulty of the given puzzles.

Then, a script was written to solve each puzzle automatically: first the expression representing the puzzle would be constructed, then given to the SAT solver, which would then return several variables. These variables were the solve time, calls to *satr*, calls to *simplify*, calls to *deduce*, and the solution to the puzzle. The solutions were not used further, but several were checked to verify the correctness of the algorithm.

After the data was recorded, a linear fit was estimated between the logarithm of difficulty and time or calls to the various functions (four data sets in total). Uncertainties were scaled by the scatter.

3. Results

I found that there was a general trend in my data of solve time and calls to *satr*, varying directly with difficulty. There was a direct relationship for both, and although there was significant scatter, these results were significant. The data for calls to *simplify* and calls to *deduce*, however, exhibited too much scatter to draw any trends or conclusions from.

One of the trends I did observe (easily visible in figures 3 and 4 on the next two pages) was that the symmetric puzzles were easier (both computation and time-wise) to solve. Among the different types of symmetry, there did not seem to be much difference.

This trend could have several reasons. First of all, according to the rules of Sudoku, if one cell can be deduced, then if the puzzle is symmetric that cell's counterparts can also be deduced. This would lead to symmetric puzzles taking less time and fewer calls to *satr* to be solved.

Another possible explanation is that the trend might be a result of the script used to generate the puzzles: it is possible that the extra requirements needed to generate symmetric puzzles caused the generator to create easier puzzles, and the built-in standard deviation of "perceived difficulty" caused the puzzles to be ranked somewhat higher than their actual values.

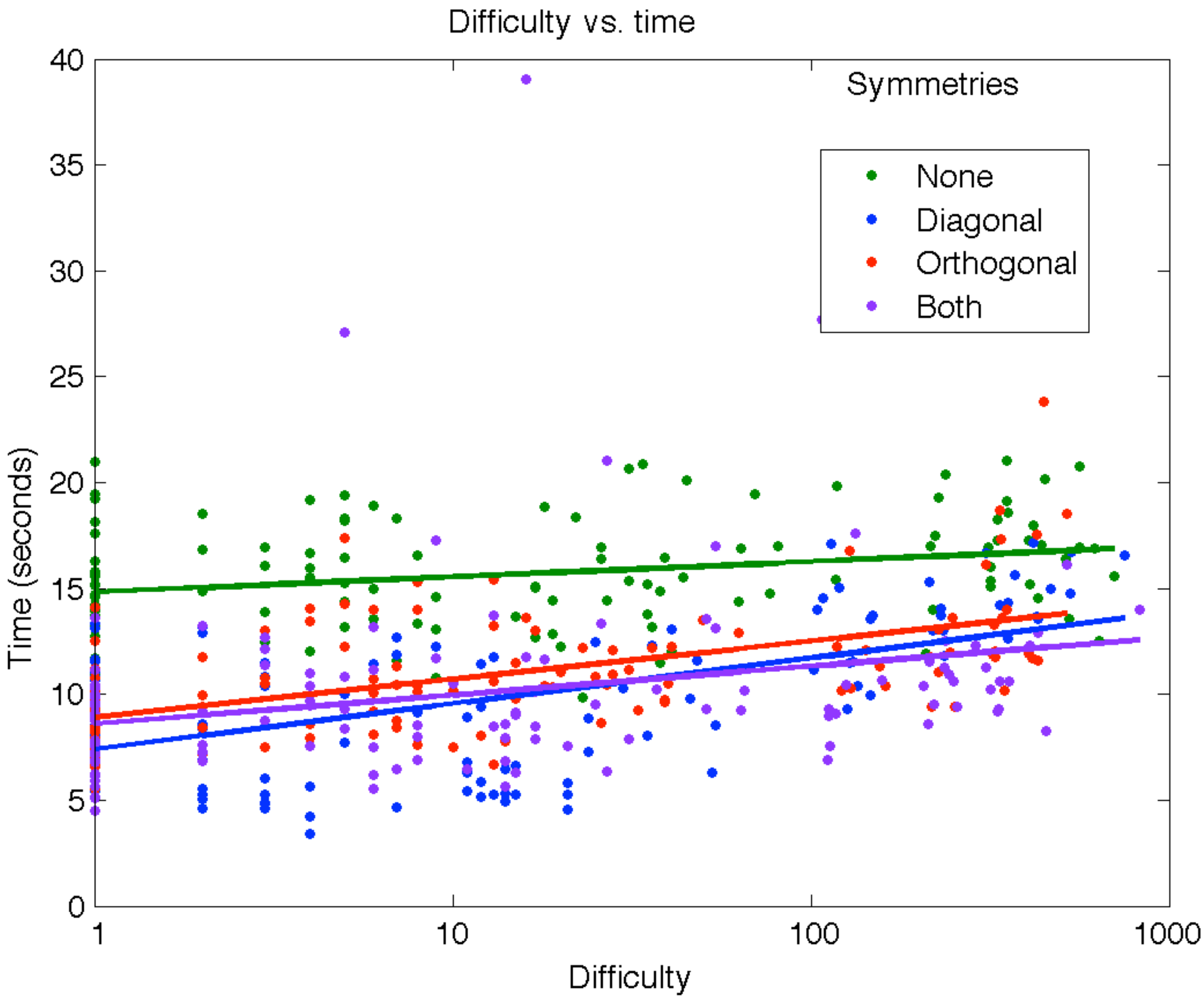

Figure 3. Difficulty vs. time

A logarithmic scaling for the x-axis on this graph is used because the slopes are very gradual, and the relative number of data points at any given difficulty is logarithmic.

As is seen on the graph, non-symmetric puzzles took significantly longer than symmetric puzzles. The uncertainties for the different types of symmetric (diagonal, orthogonal, both) were so large that no conclusion can be drawn from their data, other than the symmetric/non-symmetric trend.

Near the top of the graph, two outliers can be seen; these two puzzles took much longer. Coupled with the fact that there are also the same two outliers on the calls to *satr* data, this implies that much more data will be needed to get a complete picture of the trends for Sudoku puzzles.

These are the slope and uncertainty values for difficulty vs. time:

No symmetry	0.72	±	0.24
Diagonal symmetry	2.15	±	0.29
Orthogonal symmetry	1.81	±	0.25
Both symmetries	1.37	±	0.44

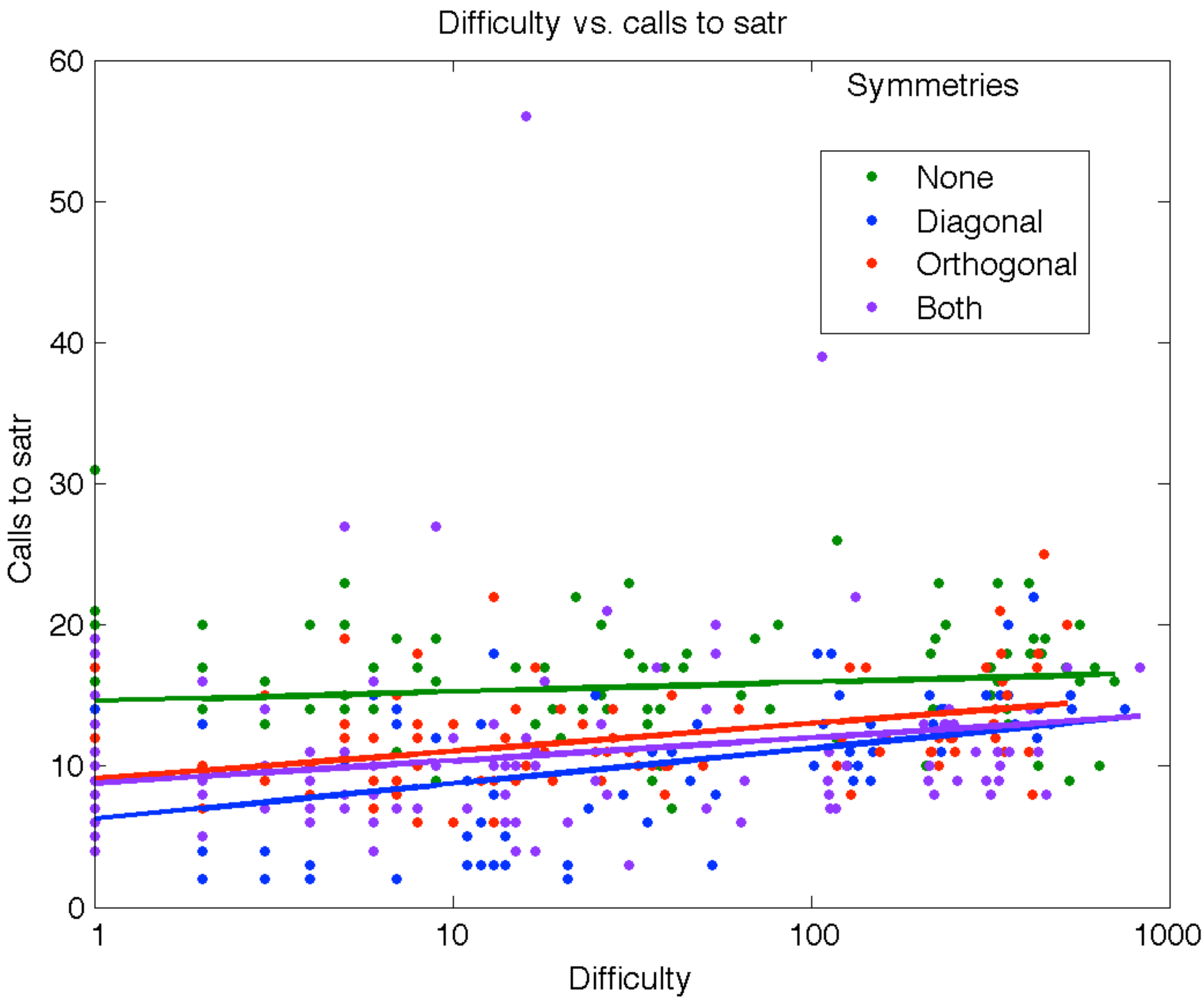

Figure 4. Difficulty vs. calls to *satr*

Easily visible on the graph above, the y-values for the points are much more well-defined than on the graph for difficulty vs. time. This is because calls to *satr* is always an integer.

The trend of symmetric puzzles of all types being easier to solve than non-symmetric puzzles is also present in this data set. As before, however, the different types of symmetries have too large of an uncertainty to compare to each other.

The same two outliers as on the previous graph are also seen here, at the top. They are a clear indicator that a much larger data set would need to be gathered in order for a complete trend to be observed.

These are the slope and uncertainty values for difficulty vs. time:

No symmetry	0.67	±	0.39
Diagonal symmetry	2.50	±	0.40
Orthogonal symmetry	1.96	±	0.33
Both symmetries	1.62	±	0.66

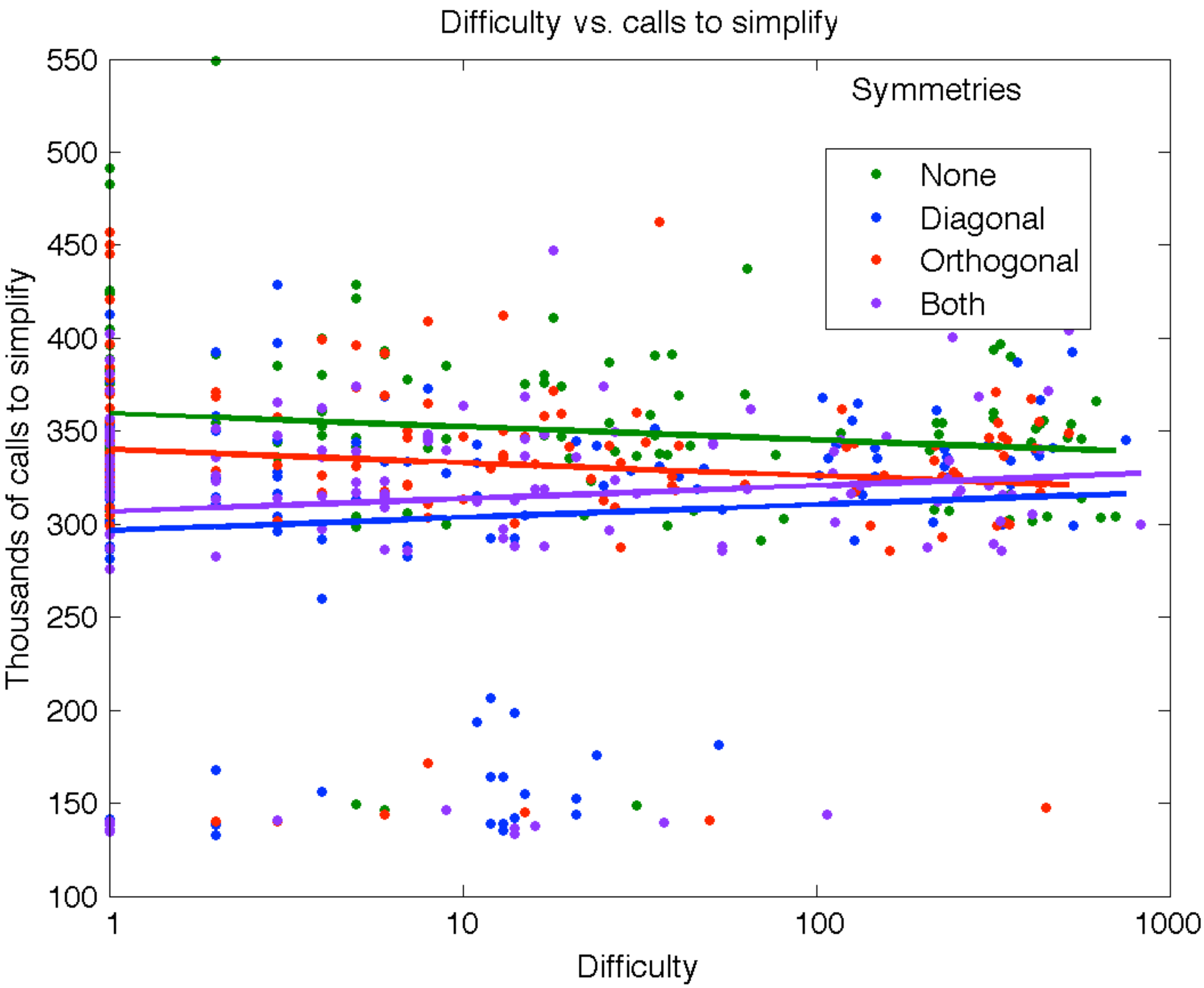

Figure 5. Difficulty vs. calls to *simplify*

The data for calls to *simplify* are not very significant. Since the uncertainties for the lines were scaled by the scatter, it stands to reason that part of the reason the slopes are so uncertain is the outliers at the bottom of the graph. There could be several causes for this, but there is one that I have concluded is the cause. Some puzzles can be solved using almost deduction alone. This means that the puzzle will take a smaller number of calls to *satr*, and, therefore, since *simplify* is called more each time *satr* repeats, *simplify* will be called much less.

Most likely, the reason that the same outliers do not appear on the calls to *satr* graph is that *simplify* is called much more than *satr* (on a scale of 500,000 to 50), so therefore the gap between the outliers and the main data set is scaled up on the *simplify* graph.

These are the slope and uncertainty values for difficulty vs. calls to *simplify*:

No symmetry	-7.06	±	5.00
Diagonal symmetry	6.96	±	6.87
Orthogonal symmetry	-7.04	±	5.76
Both symmetries	7.00	±	6.49

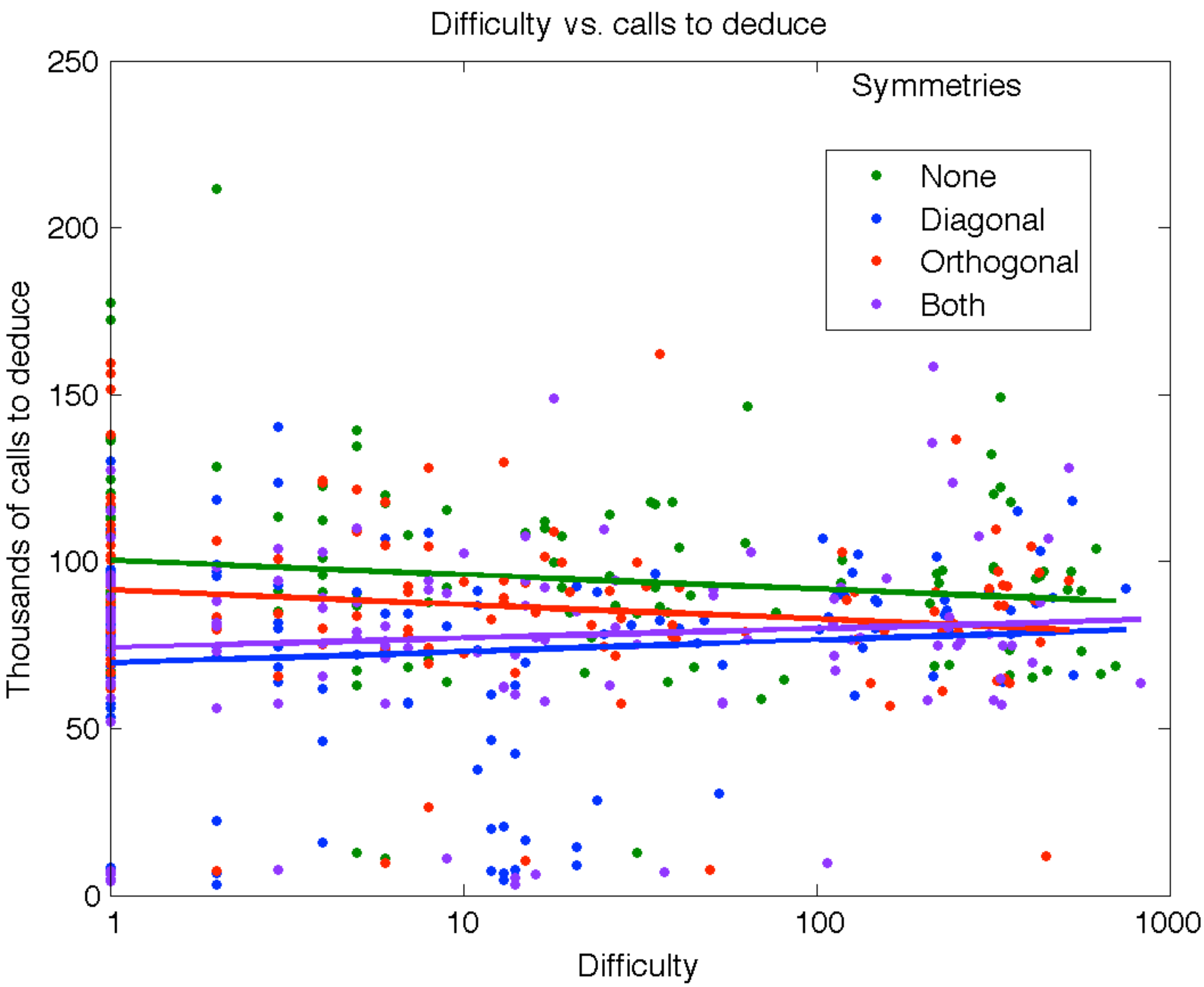

Figure 6. Difficulty vs. calls to *deduce*

Similarly to the data for calls to *simplify*, the calls to *deduce* data is not significant either. This is most likely due to the same reason that the *simplify* data is not significant. That is, some puzzles can be solved almost by deduction alone. This causes fewer calls to *satr* and the puzzle to be solved faster; therefore there are also fewer calls to *simplify*. Since the structure of the program dictates that *simplify* and *deduce* are called in the same method, the outliers on the *simplify* graph are also seen on the *deduce* graph. Note that this does not mean they are called the same amount of times, which is obvious on the graphs.

These are the slope and uncertainty values for difficulty vs. calls to *simplify*:

No symmetry	-4.27	±	2.63
Diagonal symmetry	3.45	±	2.87
Orthogonal symmetry	-4.31	±	2.76
Both symmetries	2.82	±	2.95

4. Conclusion

Sudoku puzzles can be reduced to large Boolean expressions representing their validity. This means that they can then be solved using a satisfiability (SAT) solver. I have implemented such a solver, as discussed in this paper, and have evaluated its efficiency.

From the data gathered in this investigation, I have concluded that there was a general trend in my data showing that both the time it took to solve the Sudoku puzzle and the recursions (calls to *satr*), increased with difficulty. The calls to *satr* and time data were both consistent with each other, while the calls to *simplify* and calls to *deduce* data had too much scatter to draw a conclusion.

I have also concluded that symmetric puzzles are easier, both computationally and time-wise, to solve. However, different types of symmetry were too close together and had large uncertainties, making it impossible to conclude whether diagonal, orthogonal, or puzzles with both symmetries were easier to solve.

As difficulty is a subjective term, it is hard to define quantitatively. As a result, the difficulty ratings received from the generator may be slightly inaccurate; this may have been what caused the large amount of scatter. However, I still observe a definite trend relating both solve time and calls to *satr* in a direct relationship with perceived difficulty.

Acknowledgements

I thank Dr. Aaron Bradley for his supervision on coding the SAT solver and advice on this paper. I also thank my parents for feedback on various drafts of the paper.

References

Bernal, Raoul A. "De Morgan's Laws Revisited: To Be AND/OR NOT To Be." SAS PharmaSUG Papers. Amgen, Inc., 2005. Web. 31 Dec. 2011. <http://www.lexjansen.com/pharmasug/2005/posters/po25.pdf>.

Gurevich, Yuri. "The Logic in Computer Science Column." Computing Science. Simon Fraser University. Web. 1 Jan. 2012. <http://www.cs.sfu.ca/~mitchell/papers/colLogCS85.pdf>.

Lewis, Rhyd. "Metaheuristics can Solve Sudoku Puzzles." *Journal of Heuristics* Volume 13 (2007): 387-401. Print.

Norvig, Peter. "Solving Every Sudoku Puzzle." *Peter Norvig*. Web. 18 Sept. 2011.

Scherphius, Jaap. "Sudoku Generator, Version 3." Jaap's Scratch Pad. 2006. Web. 31 Dec. 2011. <http://www.jaapsch.net/sudoku.htm>.

Appendix A. Python Code

sat.py

```
class Expr():
    def __init__(self, oper, args=[]):
        self.oper = oper
        self.args = args
    def __str__(self):
        result = '(' + str(self.oper)
        if self.args:
            result += ', ['
            for arg in self.args:
                result += str(arg) + ', '
            result = result[:-2] + ']'
        result += ')'
        return result
    def __eq__(self, other):
        if type(self) == type(other):
            if self.oper == other.oper and self.args == other.args:
                return True
        return False
    def __ne__(self, other):
        return not self == other
    def __hash__(self):
        if self.oper in ['and', 'or', 'not']:
            num = {'and':2,'or':3,'not':5}[self.oper]
        if var(self.oper):
            num = self.oper
        for arg in self.args:
            num *= 7
            num += hash(arg)
        return num
    def simplify(self, vardict):
        if self.oper in ['false', 'true']:
            return self.oper
        if self.oper in vardict:
            return vardict[self.oper]
        if var(self.oper):
            return self
        a = []
        for arg in self.args:
            a.append(arg.simplify(vardict))
        if self.oper == 'not':
            if a[0] == 'false':
                return 'true'
            if a[0] == 'true':
                return 'false'
            else:
                return Expr('not', [a[0]])
        if self.oper == 'or':
            if 'true' in a:
                return 'true'
            false = True
            b = []
            for arg in a:
                if arg != 'false':
                    false = False
```

```
                b.append(arg)
        if false:
            return 'false'
        if len(b) == 1:
            return b[0]
        return Expr('or', b)
    if self.oper == 'and':
        if 'false' in a:
            return 'false'
        true = True
        b = []
        for arg in a:
            if arg != 'true':
                true = False
                b.append(arg)
        if true:
            return 'true'
        if len(b) == 1:
            return b[0]
        return Expr('and', b)
def deduce(self):
    if self.oper in ['false', 'true']:
        return set({}), set({})
    if var(self.oper):
        return set({self.oper}), set({})
    if self.oper == 'not' and var(self.args[0].oper):
        return set({}), set({self.args[0].oper})
    else:
        changeslistpos = []
        changeslistneg = []
        cd = False
        for i in range(len(self.args)):
            result = self.args[i].deduce()
            if self.args[i] == Expr('false'):
                return set({}), set({})
            changeslistpos.append(result[0])
            changeslistneg.append(result[1])
        changespos = changeslistpos[0]
        changesneg = changeslistneg[0]
        if self.oper == 'and':
            for i in range(1, len(self.args)):
                changespos = changespos.union(changeslistpos[i])
                changesneg = changesneg.union(changeslistneg[i])
            false = False
            for arg in self.args:
                if arg.oper == 'false':
                    false = True
                    break
            if false:
                cd = True
        if self.oper == 'or':
            for i in range(1, len(self.args)):
                if self.args[i] != 'false':
                    changespos = changespos.intersection(changeslistpos[i])
                    changesneg = changesneg.intersection(changeslistneg[i])
        if contradiction(changespos, changesneg) or cd:
            self = Expr('false')
            return set({}), set({})
```

```
            return changespos, changesneg
    def reduce(self, negate=False):
        if negate:
            if self.oper == 'and':
                self.oper = 'or'
                for i in range(len(self.args)):
                    self.args[i] = self.args[i].negate()
            elif self.oper == 'or':
                self.oper = 'and'
                for i in range(len(self.args)):
                    self.args[i] = self.args[i].negate()
            elif self.oper == 'not':
                self = self.args[0].reduce()
            elif var(self.oper):
                self = Expr('not', [Expr(self.oper)])
            elif self.oper == 'false':
                self.oper = 'true'
            elif self.oper == 'true':
                self.oper = 'false'
            else:
                print(self.oper)
                raise Exception('Unidentified operator')
        else:
            if self.oper == 'not':
                self = self.args[0].negate()
            elif self.oper in ['and', 'or']:
                for i in range(len(self.args)):
                    self.args[i] = self.args[i].reduce()
            elif var(self.oper) or self.oper in ['true', 'false']:
                pass
            else:
                print(self.oper)
                raise Exception('Unidentified operator')
        return self
    def negate(self):
        return self.reduce(negate=True)
    def getvar(self):
        if var(self.oper):
            return self.oper
        for arg in self.args:
            result = arg.getvar()
            if var(result):
                return result
        return None

def display(self):
    if type(self) == str:
        return self
    else:
        return self.display()

def var(self):
    return type(self) == int

def contradiction(changespos, changesneg):
    for change in changespos:
        if change in changesneg:
            return True
```

```
    return False

def adeduce(exp, vardict):
    while True:
        exp = exp.simplify(vardict)
        if type(exp) == str:
            break
        changespos, changesneg = exp.deduce()
        if not changespos and not changesneg:
            break
        for change in changespos:
            vardict[change] = 'true'
        for change in changesneg:
            vardict[change] = 'false'
        if type(exp) == str:
            exp = Expr(exp)
    return exp

def satr(exp, vardict):
    count = 1
    vardict = dict(vardict)
    exp = adeduce(exp, vardict)
    if exp == 'false' or exp == 'true':
        return [exp, vardict, count]
    var = exp.getvar()
    vardict[var] = 'true'
    result = satr(exp, vardict)
    count += result[2]
    if result[0] == 'true':
        result[2] = count
        return result
    vardict[var] = 'false'
    result = satr(exp, vardict)
    result[2] += count
    return satr(exp, vardict)

def sat(expr, vardict={}):
    expr = expr.reduce()
    expr = expr.simplify(vardict)
    if type(expr) == str:
        return expr, {}
    return satr(expr, vardict)

if __name__ == '__main__':
    print(sat(Expr('and', [Expr(1), Expr(2), Expr(3)])))
```

sudoku.py

```
from sat import *
from time import *

def mkexpr():
    SIZE = 3
    rows = []
    for r in range(1, SIZE**2+1):
        for n in range(1, SIZE**2+1):
            columns = []
```

```
        for c in range(1, SIZE**2+1):
            exprs = [Expr(r*100+c*10+n)]
            for c2 in range(1, c):
                exprs.append(Expr('not', [Expr(r*100+c2*10+n)]))
            for c2 in range(c+1, SIZE**2+1):
                exprs.append(Expr('not', [Expr(r*100+c2*10+n)]))
            columns.append(Expr('and', exprs))
        rows.append(Expr('or', columns))
rexpr = Expr('and', rows)

columns = []
for c in range(1, SIZE**2+1):
    for n in range(1, SIZE**2+1):
        rows = []
        for r in range(1, SIZE**2+1):
            exprs = [Expr(r*100+c*10+n)]
            for r2 in range(1, r):
                exprs.append(Expr('not', [Expr(r2*100+c*10+n)]))
            for r2 in range(r+1, SIZE**2+1):
                exprs.append(Expr('not', [Expr(r2*100+c*10+n)]))
            rows.append(Expr('and', exprs))
        columns.append(Expr('or', rows))
cexpr = Expr('and', columns)

boxes = []
for br in range(SIZE):
    for bc in range(SIZE):
        for n in range(1, SIZE**2+1):
            cells = []
            for r in range(1, SIZE+1):
                for c in range(1, SIZE+1):
                    exprs = [Expr((SIZE*br+r)*100+(SIZE*bc+c)*10+n)]
                    for r2 in range(1, SIZE+1):
                        for c2 in range(1, SIZE+1):
                            if r2 < r or c2 < c or r2 > r or c2 > c:
                                exprs.append(Expr('not',
                                     [Expr((SIZE*br+r2)*100+(SIZE*bc+c2)*10+n)]))
                    cells.append(Expr('and', exprs))
            boxes.append(Expr('or', cells))
bexpr = Expr('and', boxes)

cells = []
for r in range(1, SIZE**2+1):
    for c in range(1, SIZE**2+1):
        nums = []
        for n in range(1, SIZE**2+1):
            nums.append(Expr(r*100+c*10+n))
        cells.append(Expr('or', nums))
nexpr = Expr('and', cells)

rows = []
for r in range(1, SIZE**2+1):
    for c in range(1, SIZE**2+1):
        for n1 in range(1, SIZE**2+1):
            for n2 in range(1, SIZE**2+1):
                if n1 != n2:
                    rows.append(Expr('or', [Expr('not', [Expr(r*100+c*10+n1)]),
                                            Expr('not', [Expr(r*100+c*10+n2)])]))
```

```
    rexpr2 = Expr('and', rows)

    columns = []
    for c in range(1, SIZE**2+1):
        for r in range(1, SIZE**2+1):
            for n1 in range(1, SIZE**2+1):
                for n2 in range(1, SIZE**2+1):
                    if n1 != n2:
                        columns.append(Expr('or',
                                            [Expr('not', [Expr(r*100+c*10+n1)]),
                                             Expr('not', [Expr(r*100+c*10+n2)])]))
    cexpr2 = Expr('and', columns)

    boxes = []
    for br in range(SIZE):
        for bc in range(SIZE):
            for r in range(1, SIZE+1):
                for c in range(1, SIZE+1):
                    for n1 in range(1, SIZE**2+1):
                        for n2 in range(1, SIZE**2+1):
                            if n1 != n2:
                                boxes.append(Expr('or',
                                                [Expr('not', [Expr(r*100+c*10+n1)]),
                                                 Expr('not', [Expr(r*100+c*10+n2)])]))
    bexpr2 = Expr('and', boxes)

    cells = []
    for r in range(1, SIZE**2+1):
        for c in range(1, SIZE**2+1):
            nums = []
            for n1 in range(1, SIZE**2+1):
                for n2 in range(1, SIZE**2+1):
                    if n1 != n2:
                        nums.append(Expr('or', [Expr('not', [Expr(r*100+c*10+n1)]),
                                                Expr('not', [Expr(r*100+c*10+n2)])]))
            cells.append(Expr('or', nums))
    nexpr2 = Expr('and', cells)

    expr = Expr('and', [rexpr, cexpr, bexpr, nexpr, rexpr2, cexpr2, nexpr2])

    return expr

def clean(sudoku):
    return sudoku.replace(' ', '').replace('_', '.').replace('\n\n', '\n')

def solve(sudoku, expr):
    SIZE = 3
    sudoku = clean(sudoku)
    rows = sudoku.split('\n')
    for i in range(len(rows)):
        rows[i] = list(rows[i])

    vardict = {}

    for i in range(SIZE**2):
        for j in range(SIZE**2):
            if rows[i][j] != '.':
                for n in range(1, SIZE**2+1):
```

```
                    vardict[(i+1)*100+(j+1)*10+n] = str(rows[i][j] == str(n)).lower()

    t = time()
    solution, vardict, count = sat(expr, vardict)
    t = time() - t

    solved = [['0','0','0','0','0','0','0','0','0'],
              ['0','0','0','0','0','0','0','0','0'],
              ['0','0','0','0','0','0','0','0','0'],
              ['0','0','0','0','0','0','0','0','0'],
              ['0','0','0','0','0','0','0','0','0'],
              ['0','0','0','0','0','0','0','0','0'],
              ['0','0','0','0','0','0','0','0','0'],
              ['0','0','0','0','0','0','0','0','0'],
              ['0','0','0','0','0','0','0','0','0']]

    for k in vardict:
        if vardict[k] == 'true':
            solved[int(str(k)[0])-1][int(str(k)[1])-1] = str(k)[2]

    for i in range(len(solved)):
        solved[i] = ''.join(solved[i])
    solved = '\n'.join(solved)

    return solved, t, count
```

sciencefair.py

```
from sudoku import *

sudokus = open('./sudokus.txt', 'r+')
ratings = open('./ratings.txt', 'r+')
times = open('./times.txt', 'w+')
counts = open('./counts.txt', 'w+')

sudokulist = sudokus.read().replace('\n\n', '\n').split('\n')
ratinglist = ratings.read().replace('\n\n', '\n').split('\n')

sudokus.close()
ratings.close()

sudokulist = sudokulist[:-1]

for i in range(len(sudokulist)):
    sudoku = list(sudokulist[i])
    for j in tuple(range(81, 0, -9)):
        sudoku.insert(j, '\n')
    sudokulist[i] = ''.join(sudoku)[:-1]

expr = mkexpr()

for i in range(len(sudokulist)):
    solved, t, count = solve(sudokulist[i], expr)
    times.write(str(t) + '\n')
    counts.write(str(count) + '\n')
    print(i+1, 'puzzles solved.')
```

```
times.close()
counts.close()
```

Appendix B. Raw Data

symmetry	rating	time	satr	simplify	deduce
None	1	18.135	21	353997	94694
None	1	14.052	14	339811	87464
None	1	11.724	9	395976	120584
None	1	15.783	12	382135	112812
None	1	13.743	13	333795	84070
None	1	14.093	16	423698	136117
None	1	20.978	31	383029	113520
None	1	17.577	17	355281	96770
None	1	14.711	15	339933	88790
None	1	15.571	19	336628	86751
None	1	16.283	13	343827	90408
None	1	8.826	8	377018	107445
None	1	12.752	11	491564	177553
None	1	15.194	13	354134	96087
None	1	15.632	14	348211	93129
None	1	13.174	9	482685	172312
None	1	11.532	10	388971	116244
None	1	10.993	10	331177	83382
None	1	13.637	13	341601	88427
None	1	17.615	15	404632	124550
None	1	13.949	12	425930	137165
None	1	14.605	14	337542	87292
None	1	19.261	20	307048	67509
None	1	13.919	12	339665	87344
None	1	14.148	18	335680	84764
None	1	15.717	19	344699	91332
None	1	15.164	16	300116	65618
None	1	19.441	19	375416	108426
None	1	13.241	15	341860	88564
None	1	14.191	15	302430	65152
None	6	14.965	14	391327	117491
None	3	13.875	16	384938	113368
None	4	19.18	20	360808	101119
None	2	14.845	14	354315	95607
None	6	13.581	8	392915	119933
None	2	18.519	20	548799	211622
None	4	12.011	7	380018	112232
None	9	10.774	9	385081	115413
None	5	15.228	15	421423	134632
None	7	18.326	19	306248	68546
None	7	11.592	11	377677	107882
None	9	14.623	19	345860	92261
None	5	18.197	20	149116	12737
None	8	16.574	17	311210	70656
None	5	18.281	23	303822	67451
None	9	13.045	16	299612	63856
None	4	15.533	14	347443	91061
None	5	15.267	13	346655	90572
None	4	15.953	11	352476	95906
None	4	16.675	10	399624	122594
None	3	16.052	16	300548	65513

None	3	12.466	13	335986	84973
None	8	13.328	13	340813	87680
None	5	19.394	19	428648	139469
None	6	18.912	17	146155	11052
None	3	16.968	15	344183	91103
None	5	14.366	14	341462	88358
None	5	13.159	14	339095	86888
None	5	16.426	19	298884	62983
None	2	16.844	17	391002	128299
None	18	18.864	17	410838	99721
None	34	20.884	17	358787	117853
None	19	14.414	14	373850	107498
None	31	20.656	23	149073	12738
None	38	11.479	10	337451	84750
None	22	18.372	22	304510	66776
None	15	13.669	17	375500	108494
None	31	15.369	18	336866	84529
None	81	16.981	20	303046	64637
None	26	16.936	15	354261	95505
None	17	15.096	13	380210	112049
None	20	12.25	12	335521	84905
None	35	13.8	14	347945	92217
None	17	15.04	13	347462	92321
None	64	16.899	15	437259	146326
None	23	9.875	14	323204	77144
None	35	15.205	13	390451	117072
None	26	16.391	20	386973	114084
None	77	14.748	14	337181	84811
None	41	12.013	7	369493	104166
None	38	14.846	14	299204	63928
None	63	14.353	14	369908	105710
None	36	13.158	9	337935	86375
None	45	20.111	18	307160	68530
None	70	19.434	19	290928	58851
None	44	15.52	17	342396	89718
None	17	12.708	11	375678	110073
None	19	12.843	9	347320	92248
None	27	14.459	14	339156	86950
None	39	16.445	17	391002	117853
None	527	13.574	9	353951	96989
None	318	15.366	15	394039	120387
None	118	19.855	26	361994	100393
None	331	18.235	23	443692	149137
None	210	11.948	10	339682	87535
None	312	16.931	17	417338	132249
None	318	15.109	17	356851	97649
None	621	16.883	17	365858	103878
None	419	17.981	19	351627	95147
None	430	14.556	10	354152	95811
None	219	14.003	14	354454	96479
None	216	16.989	18	307984	68888
None	514	16.371	17	346247	91699
None	450	20.188	19	304414	67427
None	353	21.032	14	315556	73477
None	117	15.399	12	348744	93680
None	317	16.041	15	359756	98316
None	355	18.581	13	390051	117967
None	353	19.137	18	302590	65831

None	439	17.068	18	355435	97067
None	238	20.363	20	307208	69224
None	563	16.961	16	346016	91380
None	638	12.541	10	303390	66266
None	410	15.183	18	301790	65470
None	228	19.291	23	354365	97389
None	223	17.5	19	348610	93591
None	561	20.765	20	314116	73305
None	405	17.26	23	344127	89177
None	331	17.249	16	396573	122300
None	705	15.597	16	303870	68574
Diagonal	1	11.556	12	323383	77709
Diagonal	1	10.85	11	377302	109804
Diagonal	1	9.915	7	288366	57479
Diagonal	1	5.412	5	301213	66632
Diagonal	1	9.14	10	307992	68872
Diagonal	1	7.557	6	330588	83932
Diagonal	1	9.464	10	320378	77877
Diagonal	1	9.043	7	281242	53470
Diagonal	1	7.87	9	309777	70709
Diagonal	1	9.434	7	313368	74293
Diagonal	1	6.828	8	329617	81792
Diagonal	1	11.045	13	316961	75792
Diagonal	1	10.649	10	315668	73981
Diagonal	1	8.018	6	413094	130147
Diagonal	1	13.336	10	354217	96957
Diagonal	1	7.652	8	313791	72057
Diagonal	1	6.916	7	337328	87041
Diagonal	1	7.2	5	315715	74644
Diagonal	1	11.672	13	327526	80288
Diagonal	1	13.351	8	336597	86286
Diagonal	1	11.098	14	286606	55979
Diagonal	1	7.814	6	352879	95787
Diagonal	1	9.561	7	314270	74223
Diagonal	1	13.13	12	336106	85324
Diagonal	1	8.543	7	141319	8190
Diagonal	1	7.932	7	325818	81237
Diagonal	1	6.203	4	301655	65463
Diagonal	1	8.142	5	336322	86504
Diagonal	1	8.1	9	356930	97705
Diagonal	1	6.751	7	333018	84398
Diagonal	7	11.868	13	287876	57340
Diagonal	8	9.133	7	372753	108607
Diagonal	6	11.441	15	368797	106983
Diagonal	6	8.122	7	333337	84371
Diagonal	2	13.199	13	314665	74678
Diagonal	5	10.009	10	313062	72197
Diagonal	3	4.817	2	295995	63860
Diagonal	3	4.627	2	304234	68256
Diagonal	2	7.221	8	133081	3228
Diagonal	4	4.212	2	291700	61898
Diagonal	3	4.873	2	325844	80086
Diagonal	4	5.631	3	156293	15889
Diagonal	2	5.544	2	350333	95641
Diagonal	2	5.042	4	138587	6773
Diagonal	2	5.249	2	350795	97131
Diagonal	2	4.627	2	324515	83212
Diagonal	7	4.646	2	282721	57925

Diagonal	3	5.287	2	316585	74442
Diagonal	4	3.422	2	259979	46154
Diagonal	2	6.904	5	167696	22301
Diagonal	2	8.593	8	310853	71328
Diagonal	5	7.728	7	343798	90850
Diagonal	9	12.278	12	327405	80816
Diagonal	3	11.498	10	428683	140454
Diagonal	3	6.033	4	327843	81853
Diagonal	3	10.393	9	345220	92843
Diagonal	3	10.817	9	397614	123575
Diagonal	7	12.67	14	333428	84537
Diagonal	2	8.395	8	392517	118472
Diagonal	2	12.907	13	357935	99003
Diagonal	35	8.028	6	351359	96385
Diagonal	21	5.284	2	344489	92658
Diagonal	12	5.152	6	138953	7184
Diagonal	12	9.4	9	206571	46471
Diagonal	11	6.802	5	193429	37754
Diagonal	13	5.276	3	164262	20719
Diagonal	13	5.268	8	138851	6821
Diagonal	21	4.547	3	152793	14521
Diagonal	14	5.322	3	292260	62816
Diagonal	24	7.309	7	176119	28350
Diagonal	11	6.314	5	342958	91138
Diagonal	53	6.329	3	181239	30403
Diagonal	21	5.818	3	143920	8989
Diagonal	15	5.26	4	304963	69876
Diagonal	15	6.623	4	154723	16392
Diagonal	12	5.841	3	164403	20045
Diagonal	13	11.765	18	135072	4461
Diagonal	14	6.457	5	198639	42400
Diagonal	14	4.943	3	141900	7560
Diagonal	11	5.456	3	332853	86746
Diagonal	24	8.874	7	342166	91009
Diagonal	41	13.054	11	325598	80013
Diagonal	25	12.467	15	320493	78281
Diagonal	48	11.604	13	330198	82313
Diagonal	12	11.409	13	292432	60060
Diagonal	30	10.311	8	328614	81129
Diagonal	11	8.94	9	315468	73619
Diagonal	46	9.796	9	318921	75624
Diagonal	36	12.302	11	331244	82502
Diagonal	54	8.525	8	307753	69209
Diagonal	428	13.675	12	336743	88204
Diagonal	102	11.151	10	326277	79755
Diagonal	214	15.294	15	300844	65642
Diagonal	147	9.938	9	325889	79969
Diagonal	307	16.678	15	343262	90985
Diagonal	149	13.747	11	335246	87828
Diagonal	355	12.614	20	321420	78264
Diagonal	430	13.555	14	366842	103230
Diagonal	371	15.643	13	387046	115247
Diagonal	128	11.464	11	291384	60001
Diagonal	120	15.04	15	343465	89930
Diagonal	417	17.161	22	338882	87621
Diagonal	337	14.198	15	299668	63909
Diagonal	234	13.029	14	336345	85458
Diagonal	231	13.737	14	331239	84069

Diagonal	231	14.063	11	340095	88564
Diagonal	135	10.416	10	315842	74045
Diagonal	114	17.127	18	342555	90020
Diagonal	750	16.576	14	345205	91753
Diagonal	147	13.549	13	340848	88623
Diagonal	104	13.977	18	367709	106849
Diagonal	131	11.557	9	364805	102290
Diagonal	108	14.562	13	335226	83448
Diagonal	126	9.321	10	355665	96804
Diagonal	356	14.316	15	334129	85314
Diagonal	530	14.769	15	392724	118149
Diagonal	219	13.037	13	361297	101363
Diagonal	534	16.725	14	299252	65983
Diagonal	467	14.971	13	340734	89104
Diagonal	236	11.796	14	325174	79439
Orthogonal	1	9.947	17	372079	108522
Orthogonal	1	10.799	12	378152	111142
Orthogonal	1	14.103	17	445573	151594
Orthogonal	1	12.528	11	456897	159369
Orthogonal	1	7.779	8	299241	66672
Orthogonal	1	11.212	11	332387	83955
Orthogonal	1	6.614	4	335815	85535
Orthogonal	1	7.266	8	329178	83116
Orthogonal	1	6.986	5	334223	87256
Orthogonal	1	10.463	10	325860	79577
Orthogonal	1	8.213	8	308360	70879
Orthogonal	1	8.346	8	338616	88658
Orthogonal	1	9.802	12	420880	137935
Orthogonal	1	7.473	11	330342	82555
Orthogonal	1	9.432	9	369526	104981
Orthogonal	1	6.609	5	319976	76011
Orthogonal	1	6.55	6	309078	73294
Orthogonal	1	8.302	8	362294	101671
Orthogonal	1	8.48	6	450265	156286
Orthogonal	1	8.462	7	396560	119327
Orthogonal	1	9.121	10	355291	100591
Orthogonal	1	5.557	6	294407	61953
Orthogonal	1	9.238	10	304656	69418
Orthogonal	1	9.375	8	349927	94703
Orthogonal	1	7.632	7	315427	76932
Orthogonal	1	7.084	9	345108	92490
Orthogonal	1	10.15	9	384842	117256
Orthogonal	1	6.836	7	349957	94355
Orthogonal	1	7.967	12	322573	79044
Orthogonal	1	8.383	9	315650	75953
Orthogonal	6	10.724	10	317458	76353
Orthogonal	7	8.771	8	350239	92580
Orthogonal	3	13.032	15	140468	7645
Orthogonal	6	14.014	16	143920	9659
Orthogonal	5	12.254	12	395953	121466
Orthogonal	4	13.462	10	399091	123516
Orthogonal	5	14.288	13	331425	83726
Orthogonal	2	9.949	8	368875	106148
Orthogonal	3	10.504	9	357755	100616
Orthogonal	2	9.405	10	328712	83284
Orthogonal	8	14.011	18	310808	74325
Orthogonal	3	7.507	9	331933	84424
Orthogonal	5	17.396	19	373502	109013

Orthogonal	6	9.177	9	369332	104740
Orthogonal	8	10.13	10	365048	104463
Orthogonal	4	14.069	11	399585	124174
Orthogonal	2	8.436	9	139985	7372
Orthogonal	8	7.632	6	303294	69390
Orthogonal	3	10.775	14	301549	65768
Orthogonal	7	11.329	15	346316	90979
Orthogonal	8	10.412	12	409152	128010
Orthogonal	8	15.318	13	171361	26452
Orthogonal	6	8.088	7	391944	117982
Orthogonal	4	7.919	10	326273	79868
Orthogonal	2	11.77	7	371312	106315
Orthogonal	6	10.075	12	315052	73831
Orthogonal	4	8.607	8	317316	75271
Orthogonal	7	8.432	9	321505	79609
Orthogonal	7	10.457	7	320935	77931
Orthogonal	2	7.193	10	326252	79479
Orthogonal	50	13.505	10	140772	7837
Orthogonal	36	12.21	10	462646	162213
Orthogonal	17	13.03	17	358083	101381
Orthogonal	15	11.488	14	145293	10512
Orthogonal	10	7.499	6	313609	72324
Orthogonal	41	12.275	15	341982	92206
Orthogonal	16	13.627	10	332454	84637
Orthogonal	20	11.038	14	341658	90783
Orthogonal	12	8.038	9	330106	82694
Orthogonal	25	10.858	11	312758	74383
Orthogonal	27	10.46	11	308846	71836
Orthogonal	63	12.891	14	321561	78878
Orthogonal	10	10.176	13	347156	94059
Orthogonal	18	10.371	11	371552	108843
Orthogonal	39	9.607	8	325723	80927
Orthogonal	14	7.79	12	300510	66619
Orthogonal	19	10.319	9	359276	99895
Orthogonal	23	12.22	13	324451	80981
Orthogonal	15	9.803	9	347116	93482
Orthogonal	13	10.634	9	349977	94182
Orthogonal	13	6.686	6	335440	87720
Orthogonal	13	13.208	13	411875	129683
Orthogonal	28	10.931	14	287836	57638
Orthogonal	31	11.18	11	360154	99727
Orthogonal	26	8.667	9	342493	91382
Orthogonal	33	9.276	10	344081	92622
Orthogonal	13	15.391	22	337126	89173
Orthogonal	40	10.485	10	318169	76995
Orthogonal	28	12.062	12	332919	83002
Orthogonal	39	9.739	10	320920	77762
Orthogonal	228	13.087	12	293072	61304
Orthogonal	446	23.831	25	147653	11912
Orthogonal	337	18.676	21	347327	92777
Orthogonal	307	16.106	17	346697	92015
Orthogonal	343	13.607	16	301116	64924
Orthogonal	216	11.787	11	334035	84970
Orthogonal	346	10.174	11	345045	92731
Orthogonal	162	10.386	12	285644	56762
Orthogonal	142	12.063	17	299116	63452
Orthogonal	155	11.323	11	326354	79161
Orthogonal	405	11.918	11	367357	104681

Orthogonal	323	11.876	13	370749	109571
Orthogonal	248	13.615	12	420121	136670
Orthogonal	415	11.697	8	340989	88495
Orthogonal	324	13.286	13	299068	64385
Orthogonal	118	12.267	10	361630	102942
Orthogonal	327	12.031	14	354176	96912
Orthogonal	129	10.286	8	317713	76521
Orthogonal	430	11.579	18	316721	75831
Orthogonal	340	17.333	18	336876	86772
Orthogonal	244	11.983	12	327776	81412
Orthogonal	427	17.546	17	355027	96800
Orthogonal	128	16.794	17	343204	90972
Orthogonal	218	9.393	11	342185	90867
Orthogonal	520	18.516	20	348699	94275
Orthogonal	252	9.42	11	325323	78990
Orthogonal	122	10.183	12	341342	88506
Orthogonal	353	13.981	15	299572	63492
Orthogonal	228	11.047	10	325511	79163
Orthogonal	328	11.761	12	341463	86665
Both	1	7.794	6	380665	115328
Both	1	10.286	19	333756	83685
Both	1	9.826	9	138441	6446
Both	1	9.62	11	348247	93051
Both	1	9.447	11	387994	115252
Both	1	6.249	8	347725	94494
Both	1	5.9	6	326920	82049
Both	1	4.507	4	319014	75440
Both	1	6.784	7	134766	4237
Both	1	7.747	7	335533	84546
Both	1	6.137	4	323706	80292
Both	1	5.09	5	287095	59114
Both	1	8.742	9	315663	76458
Both	1	6.184	6	348772	94285
Both	1	7.823	7	350937	95767
Both	1	5.627	6	321048	76841
Both	1	11.146	15	371780	107413
Both	1	13.638	18	139951	7368
Both	1	6.819	4	135736	4829
Both	1	6.889	4	356627	95860
Both	1	10.265	10	334126	85516
Both	1	7.1	6	346332	93953
Both	1	8.921	5	402494	127370
Both	1	9.992	13	275972	51970
Both	1	9.091	6	314837	72902
Both	1	7.272	7	343222	89479
Both	1	5.108	5	294557	63066
Both	1	7.558	9	294765	63998
Both	1	5.152	5	331017	82892
Both	1	10.461	9	347156	93910
Both	3	12.695	14	140643	7752
Both	5	8.384	7	317297	77200
Both	2	7.235	9	323268	80767
Both	8	6.916	7	346275	94286
Both	6	7.514	8	316238	76205
Both	6	5.554	4	309207	71020
Both	5	27.077	27	339140	87794
Both	4	10.989	11	315440	75416
Both	2	9.177	9	351650	98292

Both	9	17.269	27	146322	11105
Both	6	6.22	6	314790	74704
Both	2	7.265	5	313265	73325
Both	8	8.546	7	348506	94306
Both	4	9.671	7	335796	86226
Both	2	13.246	16	282618	56205
Both	3	8.735	7	365336	104020
Both	5	9.282	8	322634	78971
Both	4	10.382	11	362506	102760
Both	5	10.818	11	374077	110162
Both	8	8.015	7	344567	91651
Both	4	7.542	6	297202	65618
Both	6	11.182	10	286332	57500
Both	9	11.684	10	339753	90517
Both	3	12.163	14	347921	94341
Both	6	13.196	16	323128	80758
Both	2	7.597	8	336212	88167
Both	4	9.451	9	339648	89092
Both	2	6.858	8	326193	81706
Both	7	6.489	7	285804	74158
Both	3	11.374	10	313991	57461
Both	25	9.54	9	374058	109807
Both	18	11.643	16	446884	149032
Both	26	13.358	13	296482	63075
Both	13	8.469	10	292769	62402
Both	15	9.119	6	336969	86892
Both	14	8.609	8	312972	72278
Both	54	17.004	20	285804	57862
Both	13	13.706	13	297666	62512
Both	65	10.185	9	362133	102852
Both	31	7.904	3	316422	75075
Both	27	6.342	8	323655	80215
Both	14	6.84	6	136448	5265
Both	51	9.288	7	342842	91608
Both	17	7.899	4	318701	76604
Both	15	6.294	4	368321	107652
Both	14	5.626	10	133189	3300
Both	16	11.764	12	318815	77237
Both	14	7.95	8	288178	60251
Both	17	10.101	11	288268	58076
Both	51	13.549	14	343287	89916
Both	27	21.048	21	349416	94284
Both	54	13.099	18	288348	57478
Both	10	10.482	12	363497	102382
Both	11	6.44	7	311870	72896
Both	15	9.043	10	346030	94445
Both	64	9.243	6	319135	76399
Both	21	7.539	6	336115	84989
Both	17	8.499	10	348135	92241
Both	16	39.057	56	137861	6152
Both	37	10.245	17	139505	7088
Both	289	12.284	11	368763	107443
Both	307	11.264	9	320603	78438
Both	519	16.12	17	404195	127991
Both	158	10.676	12	346927	95030
Both	117	9.083	7	345230	92350
Both	243	10.95	14	400374	123494
Both	331	9.187	11	301604	64948

Both	357	10.603	11	315903	75019
Both	213	8.614	9	423577	135730
Both	112	8.958	11	311916	71871
Both	255	9.438	9	318116	76072
Both	206	10.412	13	287596	58401
Both	454	8.293	8	371369	106757
Both	334	9.32	9	285804	57096
Both	113	7.55	7	301260	67235
Both	318	10.243	8	289340	58618
Both	221	9.503	8	312320	74999
Both	408	12.278	14	305481	69741
Both	107	27.693	39	144049	9796
Both	336	10.611	10	316090	75023
Both	214	11.547	10	456014	158507
Both	250	10.607	13	315654	74970
Both	829	14.012	17	299796	63523
Both	112	9.31	8	339241	88887
Both	125	10.427	10	316408	76458
Both	431	12.93	11	340482	87975
Both	236	11.27	13	324681	80390
Both	111	6.93	9	327419	81490
Both	238	12.519	13	334004	83341
Both	133	17.586	22	320617	77297