Sudoku (Propositional Logic)

Carter Kruse (October 24, 2023)

Instructions

The goal for this assignment is to write a few solvers for propositional logic satisfiability problems. As an example, we will use sets of sentences derived from Sudoku logic puzzles.

There is provided data and code to get started. The most important provided files end with the extension `.cnf`. These are representations of the problems we will solve, in conjunctive normal form.

Further, there are various Python files. `Sudoku.py` is the workhorse: it converts `.sud` data files to conjunctive normal form and may be used to display text representations of boards. `sudoku_to_cnf.py` is a utility that allows us to quickly do the conversion from the command line. Previously, `display.py` was included to display a `.sol` file, though this was implemented within the `solve.py` file.

The `solve.py` file solves and displays a `.cnf` problem (which may or may not be Sudoku). This is dependent on a `SAT.py` file containing a boolean satisfiability solver. This was the main objective of the assignment.

In this assignment, students were not required to use any of the provided code, though the provided Sudoku problems were already in `.cnf` format and the display and loading code is rather trivial.

CNF File Format

The format of the `.cnf` files is based loosely on that described by Ivor Spence. Each line of a `.cnf` file represents an *OR* clause in CNF. Thus, the line

```
111 112 113 114 115 116 117 118 119
```

indicates that at least one of the variables `111`, `112`, `113`, etc must have the value `TRUE`. Negative signs indicate negation. The line

```
-111 -112
```

indicates that either `111` or `112` must have the value `FALSE`. Since this is conjunctive normal form, every clause must be satisfied to solve the problem, but there are multiple variables in most clauses,

and thus multiple ways to satisfy each clause.

The variable names correspond to locations and values on the Sudoku board. `132` indicates that row 1, column 3 has a 2 in it. For a typical $ 9 \times 9 $ Sudoku board, there are therefore 729 variables.

The CNF Files

In order of increasing difficulty, the files are:

- `one_cell.cnf` : rules that ensure that the upper left cell has exactly one value between 1 and 9.
- `all_cells.cnf` : rules that ensure that all cells have exactly one value between 1 and 9.
- `rows.cnf` : rules that ensure that all cells have a value, and every row has nine unique values.
- `rows_and_cols.cnf` : rules that ensure that all cells have a value, every row has nine unique values, and every column has nine unique values.
- `rules.cnf` : adds block constraints, so each block has nine unique values, completing the rules for an empty Sudoku board.
- `puzzle1.cnf` : adds a few starting values to the game to make a puzzle.
- `puzzle2.cnf` : adds different starting values.
- `puzzle_bonus.cnf` : adds several starting values, which creates a difficult solution.

GSAT

The 'gsat' algorithm is described nicely on Wikipedia. It is quite simple.

1. Choose a random assignment (a model).
2. If the assignment satisfies all the clauses, stop.
3. Pick a number between 0 and 1. If the number is greater than some threshold `p_value`, choose a variable uniformly at random and flip it. Go back to step 2.
4. Otherwise, for each variable, score how many clauses would be satisfied if the variable value were flipped.
5. Uniformly at random choose one of the variables with the highest score. (There may be many tied variables.) Flip that variable. Go back to step 2.

The aim is to implement 'gsat', keeping the implementation notes in mind. This algorithm allos us to solve the first few `.cnf` problems, according to our implementation. The real Sudoku puzzles are too difficult to solve with 'gsat' in a reasonable time frame.

WalkSAT

The scoring step in 'gsat' can be slow.If there are 729 variables, and a few thousand clauses, the obvious scoring method loops more than a million times. And that only flips a single bit in the

assignment. Ouch.

'walksat' chooses a smaller number of candidate variables that might be flipped. For the current assignment, some clauses are unsatisfied. Choose one of these unsatisfied clauses uniformly at random. The resulting set will be your candidate set. Use these candidate variables when scoring. The aim is to implement 'walksat'.

In this case, 'walksat' needs quite a few iterations to solve `puzzle1.cnf` and `puzzle2.cnf`. The limit is set ot 100,000 iterations, with 0.3 as the threshold value for a random move.

There are many variations on 'walksat'. Some variations do a 'gsat' step occasionally to try to escape local minima. The implementation of the simple version described above is just a first step, as further exploration of variants is explained as an extension.

Implementation Notes

The variable names for Sudoku are things like `111`, `234`, etc. The SAT-solvers should be generic. They should accept any CNF problem, not just Sudoku problems. Thus, we might expect variable names like `VICTORIA_BLUE` for a map coloring problem for `MINE_32` for minesweeper. Thus, the code refers to the variables by numerical indices during calculation, with `111` as variable 0, `112` as variable 1, etc. Thus, the assignment may be represented using a simple list. (In this implementation, we use a map/dictionary for quick lookup.) Clauses are converted into lists (or perhaps sets) of integers.

Representing Variables & Clauses In Code

In a clause, a negative integer is used to represent a negated atomic variable. So, the clause `-1 -2` might represent `-112 OR -113`. In the implementation, there is no issue with this clever trick, as assignment was handled only on the most outer parts of the problem.

That is the following issue did not persist: Since `-0` is the same as `+0`, the clauses involving the 0th variable might be misconstrued, which is tricky to correct. While the assignment recommends not 0-indexing the variables, the implementation here is appropriate. This has implications for assignments. If the assignment is a list, then the index 0 may refer to the value of the variable 1, so it's important to be careful with indexing.

Output `.sol` Files

It is wise to output a solution file when the solver is run, since the solver may take several minutes for a difficult problem. The files are outputted with `.sol` extensions that list the name of every variable in the assignment, with either no sign (for a `TRUE` value), or a negative sign (for a `FALSE`) value. The variable names are each on their own lines in the file.

While this is perhaps redundant, since knowing which variables are `TRUE` tells you that the others are `FALSE`, this is okay, since that way we construct a complete list of the variable names, and disk space is cheap.

The code for 'gsat' and 'walksat' directly follows from the pseudo-code for the algorithms presented in the book. As indicated, the algorithm stops making flips of boolean variables when the `max_flips` number is reached, or when a solution is found that satisfies the clauses of the conjunctive normal form.

The algorithm is implemented in `SAT.py` (in Python). While the file contains quite a few comments, we will quickly review the main components/structure here.

The constructor is responsible for the initialization of the clauses, alongside the map/dictionaries used to keep track of the relationship between the symbols/variables and integer values, given by an index. To determine the clauses, we read the lines of a given file, followed by an indexing/mapping of the symbols/variables within each clause to the dictionaries.

```
# Constructor
def __init__(self, file_name):
    # Instance Variables
    self.clauses = []
    self.variable_to_index = {}
    self.index_to_variable = {}
    self.index = 0
    self.model = {}

    # Open the file.
    file = open(file_name, 'r')

    # Cycle through the lines of the file.
    for line in file:
        # Add each line to the list of clauses.
        self.clauses.append(line.replace('\n', ''))

    # Close the file.
    file.close()

    # Cycle through the list of clauses.
    for clause in self.clauses:
        # Determine the specific variable/symbol in each clause.
        for variable in clause.split():
            # Update the variable/symbol to only be positive.
            var = variable.replace('-', '') if variable.startswith('-') else variable

            # If the variable is not already indexed...
```

```
            if var not in self.variable_to_index:
                # Update the mappings/dictionaries, along with the index.
                self.variable_to_index[var] = self.index
                self.index_to_variable[self.index] = var
                self.index += 1
```

Prior to looking into the main algorithm, let us consider the various methods that are used with the algorithm. The `satisfies()` method is responsible for determining if a given model satisfies a single clause. To do so, we consider the variables/symbols in the clause, only considering the positive value (not the negation). Using this information, we may update the result accordingly to provide an understanding of whether or not the model satisfies a given clause.

```
def satisfies(self, clause):
    result = False

    # Determine the specific variable/symbol in each clause.
    for variable in clause.split():
        # If the variable is negated...
        if variable.startswith('-'):
            # The index is determined by the positive variable.
            index = self.variable_to_index[variable.replace('-', '')]

            # The result is determined opposite of expected.
            result = (result or (not self.model[index]))
        # Otherwise...
        else:
            index = self.variable_to_index[variable]
            result = (result or self.model[index])

    return result
```

The `satisfies_clauses()` method is relatively straightforward, as we simply build upon the `satisfies()` method to consider not just a single clause, but a series of them. In this case, we aim to consider all of the clauses that are contained with the given problem. If a single clause is not satisfied, we return false.

```
def satisfies_clauses(self):
    # Cycle through the clauses.
    for clause in self.clauses:
        # If the clause is not satisfied, return false.
        if not self.satisfies(clause):
            return False

    return True
```

Similarly, the `false_clauses()` method is built upon the `satisfies()` method to construct a list of clauses that are not satisfied by the given model. The structure of this method is identical to that of the previous, with the exception of returning a different result.

```
  def false_clauses(self):
      false_clauses = []

      # Cycle through the clauses.
      for clause in self.clauses:
          # If the clause is not satisfied, add to the set of false clauses.
          if not self.satisfies(clause):
              false_clauses.append(clause)

      return false_clauses
```

Finally, the `score_model()` method follows a similar structure, and, once again, is built upon the `satisfies()` method. This method is responsible for determining the number of satisfied clauses for a given model, which acts as a heuristic (score) to determine the appropriate variables/symbols to flip in the 'walksat' implementation.

Of further significance is the `h_value` which is not primarily used in this method (as it truly serves as a probability measure in the advanced 'walksat' algorithm), though it is used to enhance the complexity of the model. In this case, if the `h_value` is zero, we simply maintain the original method of scoring the clauses, otherwise, we use a different scoring metric that takes into account the length of the clause. This method rewards satisfying clauses that have a shorter length.

IMPORTANT - This is part of the extra credit included for this assignment.

```
  def score_model(self, h_value):
      score = 0

      # Cycle through the clauses.
      for clause in self.clauses:
          # If the clause is satisfied, update the score.
          if self.satisfies(clause):
              # The 'h_value' is used only in an advanced algorithm.
              if h_value != 0:
                  score += 1 / len(clause)
              else:
                  score += 1

      return score
```

The `write_solution()` method is simply responsible for taking a model and writing it to a file, which allows us to store the results of the 'gsat' and 'walksat' methods. In this case, we consider if the model assigns the value of true or false to the variable, and adjust the output to the `.sol` file accordingly.

```
def write_solution(self, file_name):
    # Open the file to write to.
    file = open(file_name, 'w')

    # Cycle through the indices of the model.
    for index in self.model:
        # Determine the string based on true/false.
        string = self.index_to_variable[index] if self.model[index] else '-' + self.index

        # Write to the file.
        file.write(string + '\n')

    # Close the file.
    file.close()
```

Now, let us transition to the main algorithm, which is encompassed completely in the `sat()` method. Rather than distribute the 'gsat', 'walksat', and advanced/enhanced 'walksat' algorithms into different methods with much of the same code, there is consolidation using various parameters. The structure of the method is as follows:

```
def sat(self, p_value, max_flips, algo, h_value)
```

Initially, we construct a model with the random assignment of `TRUE` or `FALSE` for the variables/symbols, using the `random.choice()` method, alongside the indexing included in the model

```
# Create a model with a random assignment of true/false.
for index in range(self.index):
    # Each symbol in the clauses is represented.
    self.model[index] = random.choice([True, False])
```

Next we cycle through until `max_flips` is reached, or a solution (model that satisfies the clauses) is found. This is the most time/memory-consuming aspect, as it requires significant computation. If the model does indeed satisfy the clauses, we print out the number of flips (used for testing) and return the model.

```
# Cycle continuously until 'max_flips' is reached.
    for k in range(max_flips):
        # If the model satisfies the clauses...
        if self.satisfies_clauses():
            print("Flips:" + str(k))

            # Return the model.
            return self.model
```

Depending on the algorithm used, the random clause varies. With the 'walksat' algorithm, we specifically select a random clause that is false within the model constraints, which allows us to more quickly approach a solution. In this case, we must use our mapping/dictionaries to switch between the symbols/variables and the integers corresponding to them.

```
# Select a random clause that is false in the model.
if algo == 'walksat':
    # The 'random.choice()' is used to select from the false clauses.
    random_clause = random.choice(self.false_clauses())

    # The clause is mapped from the index to the appropriate variable and negation is dis
    random_clause = [self.variable_to_index[var.replace('-', '')] if var.startswith('-')
```

Now, we continue to the main part fo the algorithm, where with a given probability `p_value`, we flip a random variable/symbol in the model (or in the clause, if the algorithm is 'walksat').

```
# With a given probability, flip the value in the model.
if random.random() < p_value:
    # The selected symbol is randomly selected from all symbols ('gsat') or the random cl
    random_index = random.randrange(self.index) if algo == 'gsat' else random.choice(ranc
    self.model[random_index] = not self.model[random_index]
```

Otherwise, we flip whatever symbol in the model/clause maximizes the number of satisfied clauses. This implementation is slightly longer, though the commments guide the reader through it. In this case, we still consider whether or not the model is 'gsat' or 'walksat', the second of which is an optimal version.

The main idea is that we make a change to the model, evaluate the number of clauses that would be satisfied, and then revert the model, similar to the process used in minimax for Chess boards. The maximum score is updated appropriately, alongside the variables/symbols that when flipped, attain the maximum score.

```
# Otherwise, flip whichever symbol in the clause maximizes the number of satisfied clause
else:
    max_score = 0
    best_variables = []

    # Determine the set of symbols to iterate over, according to 'gsat' vs 'walksat'.
    loop_set = range(self.index) if algo == 'gsat' else random_clause
    for i in loop_set:
        # Flip the symbol within the model.
        self.model[i] = not self.model[i]

        # Determine the score of the model.
        current_score = self.score_model(h_value)

        # Update the max score if appropriate, along with the best variables/symbols.
        if current_score > max_score:
            best_variables.clear()
            max_score = current_score
            best_variables.append(i)

        # Otherwise, simply update the best variables/symbols.
        elif current_score == max_score:
            best_variables.append(i)

        # Flip the symbol (back) within the model.
        self.model[i] = not self.model[i]
```

After this is accomplished, we flip the variable/symbol with the highest score. While there may be many, we simply use a random choice to select which to flip, before returning to the top of the loop.

```
# Flip the variable/symbol with the highest score (using a random choice).
highest_variable = random.choice(best_variables)
self.model[highest_variable] = not self.model[highest_variable]
```

There is an element included within the `sat()` method that will addressed later on in the extra credit, revolving around the idea of increasing the complexity of the 'walksat' algorithm by introducing further randomness, which actually serves to speed up the computation of the solution.

Extra Credit/Bonus

The following implementations are designed to enhance the problem, thus meriting "extra credit" as defined by the outlines of the assignment. By going above and beyond to consider extensions to the problem, we consider a greater level of complexity for the problem. Due to the structure of our code, we are able to maintain much of the structure of the previously written code.

According to the instructions, any "scientifically interesting extension related to the theme of the assignment is welcome." The following represent implementation of a few of the ideas that were encouraged.

Resolution - Bonus #1

Resolution is the process of taking a complex set of constraints (perhaps written in conjunctive normal form, CNF) and reducing the set of constraints down, to be able to prove aspects of a given problem. As the assignment alluded to, we are able to set up a small Sudoku problem (with just the upper left block), and initialize it with a few values. From this, we are able to prove information about the other values. This is the process that humans use when trying to solve Sudoku, alongisde random walks.

In the `Simplify.py` file, the `resolution()` method is implemented, which serves to reduce the CNF of the `rules.cnf` to only include the upper left block of information. From this, we may further reduce the CNF according to the `simplify()` function, which will be discussed later on, as it involves a different aspect of extra credit.

The `resolution()` method opens a file, specifically the `rules.cnf` file, and cycles through the lines to determine what should be modified or discluded. In this sense, we only aim to consider Sudoku constraints of the form `abc`, where `a` and `b` are numbers from 1 to 3 (representing the first three rows and columns of Sudoku), and `c` is a number from 1 to 9, indicating the value of a particular location.

By using a tricky mathematical technique to determine the hundreds and tens digit of a various CNF clause variable/symbol, we may eliminate lines that are not required to be included.

```
def resolution(self):
    # Open the 'infile' and 'outfile' with the appropriate names.
    with open(self.file_name, 'r') as infile, open('data/resolution.cnf', 'w') as outfile
        # Cycle through the lines of the 'infile'
        for line in infile:
            # Strip each line of any whitespace.
            line = line.strip()

            # Boolean: Determines if this line is written in the 'outfile'.
            write_line = True

            # Cycle through the elements of each line.
            for element in line.split():
                # Check if the hundreds digit is okay.
                if (abs(int(element)) % 1000) // 100 > 3:
                    write_line = False
                # Check if the tens digit is okay
                if (abs(int(element)) % 100) // 10 > 3:
                    write_line = False
```

```
            # If the line is to be written, update the 'outfile'.
            if write_line:
                outfile.write(line + '\n')
```

Following this, we may simply use our `SAT.py` to solve the problem. By inputting further constraints, as in specifying the exact location of certain values, we may further reduce the CNF file to reflect a smaller subset of constraints that must be satisfied. As aforementioned, this will be discussed later.

In this case, the process is relatively trivial, as the constraints simply require that the numbers 1 through 9 are placed in the $ 3 \times 3 $ square. This is the idea of resolution. By considering the placement of values, along with reduction in the complexity of the problem, we are able to essentially eliminate the constraints on the rows and columns, as the box constraints for the Sudoku problem already satisfy the previous.

In this particular scenario, the problem is underconstrained, indicating that there are many solutions, which results in a relatively easy solving process.

Other Satisfiability Problems - Bonus #2

As indicated, Sudoku is certainly not the only satisfiability problem that may be solved using the form of CNF. For this extension, there were two further boolean satisfiability problems: map-coloring and 4-queens. While these problems are both relatively simplistic, they indicate the general aspects of SAT solvers.

The CNF file for the map-coloring is `map_coloring.cnf` . This encodes all of the relevant constraints for the map-coloring of Australia, as given in the textbook. In this file, the symbols/variables are not given as integers, but rather as strings, similar to the following:

```
WA_R WA_G WA_B
NT_R NT_G NT_B
SA_R SA_G SA_B
Q_R Q_G Q_B
NSW_R NSW_G NSW_B
V_R V_G V_B
T_R T_G T_B
```

This simply represents each of the territories of Australia, along with the corresponding color. The CNF clauses above state that each of the territories must be encoded with a color: red, green, or blue. Further constraints are provided in the CNF file, including the constraint that a territory cannot be two colors, and that territories that share a border may not be the same color. An example of each of these is given as follows, respectively:

```
-WA_R -WA_G
-WA_R -NT_R
```

To solve the map-coloring satisfiability problem (as determined), we use the following parameters.

```
sat.sat(p_value = 0.3, max_flips = 100000, algo = 'walksat', h_value = 0)
```

This produces a solution with `59` flips in a time of `0:00:00.009370`. The solution, which is given in `map_coloring.sol` is provided as follows:

```
-WA_R
WA_G
-WA_B
-NT_R
-NT_G
NT_B
SA_R
-SA_G
-SA_B
-Q_R
Q_G
-Q_B
-NSW_R
-NSW_G
NSW_B
-V_R
V_G
-V_B
T_R
-T_G
-T_B
```

This indicates that the following colors should be assigned to the territories:

```
Western Australia - Green
Northern Terrority - Blue
South Australia - Red
Queensland - Green
New South Wales - Blue
Victoria - Green
Tasmania - Red
```

While other options certainly would satisfy the problem, specifically the clauses given in the CNF file, this assignment seems appropriate.

Similarly, the CNF file for the 4-queens problem is `queens.cnf`. This encodes all of the relevant constraints for the 4-queens problem, in which 4 queens are placed on a $ 4 \times 4 $ chessboard, and are not allowed to be on the same row, column, or diagonal. This is a simple form of the higher-dimensional n-queens problem, but serves as a proof of concept.

To solve the 4-queens satisfiability problem (as determined), we use the following parameters.

```
sat.sat(p_value = 0.3, max_flips = 100000, algo = 'gsat', h_value = 0)
```

This produces a solution with `5` flips in a time of `0:00:00.004189`. The solution is provided as follows:

```
-11
12
-13
-14
-21
-22
-23
24
31
-32
-33
-34
-41
-42
43
-44
```

In this case, it turns out that running the 'walksat' algorithm actually results in increased complexity, likely due to the simplicity of the 4-queens problem itself. To solve the 4-queens satisfiability problem (as determined), we use the following parameters.

```
sat.sat(p_value = 0.3, max_flips = 100000, algo = 'walksat', h_value = 0)
```

This produces a solution with `87` flips in a time of `0:00:00.021081`. The solution, which is given in `queens.sol` is provided as follows:

```
-11
-12
13
-14
21
-22
-23
-24
-31
-32
-33
34
-41
42
-43
-44
```

These results indicate that there are multiple ways to solve the problem, and the result varies depending on the algorithm used. This is the result of the problem being underdetermined.

In summary, in cases where the set of clauses is not significantly great, the 'gsat' algorithm functions appropriately, as there is no need to speed up the SAT solver. The results may differ slightly, but that is not the objective of solving a boolean satisfiability problem. As we will see later, the benefits of the 'walksat' algorithm are evident in problems with significantly more constraints, resulting in more CNF clauses.

Advanced Random Walk Algorithm - Bonus #3

While the 'walksat' algorithm is certainly an improvement on the 'gsat' algorithm, there are certainly more sophisticated random walk algorithms.

As previously alluded to, the following introduces a further element of randomness. This section of code is located within the `sat()` method, and serves to prevent hitting local minima by selecting multiple variables/symbols to flip in the model, with a given probability. This only occurs if there are at least two variables/symbols with similar scores.

```
# BONUS: To enhance the walksat algorithm, we introduce a further aspect of randomness.
if h_value != 0:
    # If there are at least two 'best' variables/symbols...
    if len(best_variables) >= 2:
        # With a given probability, we make an additional flip in the model (at random).
        if random.random() < h_value:
            highest_variable = random.choice(best_variables)
            self.model[highest_variable] = not self.model[highest_variable]
```

Further, the scoring method is modified, according to the `h_value` not being equal to zero, in which case the enhanced 'walksat' algorithm would never take effect. As highlighted by the following code, by adjusting the scoring of clauses to account for the length of each clause, we incentive the model to take steps that will quickly solve the shortest clauses before continuing to the longer ones.

In other words, we use a different scoring metric that takes into account the length of the clause. This method rewards satisfying clauses that have a shorter length, though may or may not prove effective.

```
def score_model(self, h_value):
    score = 0

    # Cycle through the clauses.
    for clause in self.clauses:
        # If the clause is satisfied, update the score.
        if self.satisfies(clause):
            # The 'h_value' is used only in an advanced algorithm.
            if h_value != 0:
                score += 1 / len(clause)
            else:
                score += 1

    return score
```

To experimentally test these modifications, we may solve the satisfiability problem for `rules.cnf` using the following parameters, to determine if there is an actual difference, contributing to a faster solution.

`h_value = 0`

Parameters

```
sat.sat(p_value = 0.3, max_flips = 100000, algo = 'walksat', h_value = 0)
```

Results

```
Flips: 2188
6 9 4 | 1 8 7 | 3 2 5
2 1 7 | 5 3 9 | 4 6 8
5 8 3 | 2 4 6 | 7 9 1
---------------------
7 4 1 | 6 5 8 | 9 3 2
8 2 9 | 7 1 3 | 6 5 4
3 6 5 | 9 2 4 | 8 1 7
---------------------
```

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

Time: 0:00:18.653374
```

h_value = 0.8

Parameters

```
sat.sat(p_value = 0.3, max_flips = 100000, algo = 'walksat', h_value = 0.8)
```

Results

```
Flips: 1716
8 9 3 | 6 4 7 | 2 1 5
5 6 2 | 9 1 3 | 8 4 7
7 1 4 | 2 8 5 | 9 6 3
---------------------
6 5 8 | 1 7 9 | 4 3 2
3 7 1 | 4 5 2 | 6 8 9
2 4 9 | 3 6 8 | 7 5 1
---------------------
9 3 5 | 8 2 6 | 1 7 4
1 2 6 | 7 3 4 | 5 9 8
4 8 7 | 5 9 1 | 3 2 6

Time: 0:00:14.456477
```

As highlighted, the enhanced/advanced 'walksat' method converges to a solution faster (less flips and time), which supports the hypothesis that increasing the randomness of the flips avoids local minima and allows us to determine a solution to the CNF clauses more quickly. This indicates that it is beneficial to consider the implementation of various models for the Sudoku boolean satisfiability problem.

Known Cell Values - Bonus #4

In applying the 'walksat' algorithm, there is violation of constraints at every step, including the known cell values of a Sudoku problem. The question we aim to answer is as follows: Is the idea that the SAT solver ignores the fact that some variables have known values a good thing or a bad thing?

In this analysis, arguments will be made on both sides, and there will be experimental data created by modifying the CNF files to eliminate the known variables, followed by running the 'walksat' algorithm.

That is, if `342` is known to be true, we will treat it as a constant, not a variable.

In the `Simplify.py` file, the `simplify()` method is implemented, which served to reduce the CNF of any particular CNF file to only include the relevant information/clauses, according to known values. Accordingly, with knowledge of the values in certain cells, we are able to reduce the number of clauses, and thus reduce the complexity of the SAT problem through modifying the CNF.

The `simplify()` method opens a file, and cycles through the lines to determine what should be modified or discluded, according to the known values. These known values are given as an instance variable, and are hard-coded for `puzzle1.cnf` and `puzzle2.cnf`. In this sense, we aim to eliminate many of the lines that contain redundant information about the known cell values.

By modifying the lines containing the known cell values (or simply not including them at all), we are able to prevent the SAT solver from considering the clauses and treating the known cell values for the Sudoku problem as variables. This means that when using the solver, the known cell values are never encoded in the mapping/dictionaries used to represent the variables, and are simply set in the models.

```
def simplify(self):
    # Open the 'infile' and 'outfile' with the appropriate names.
    with open(self.file_name, 'r') as infile, open(self.file_name[:-4] + '_modified.cnf',
        # Cycle through the lines of the 'infile'.
        for line in infile:
            # Strip each line of any whitespace.
            line = line.strip()

            # Boolean: Determines if this line is written in the 'outfile'.
            write_line = True

            # Cycle through the list of known values (constants).
            for cell in self.known_values:
                # Determine the index of the known value.
                index = line.find(str(cell))

                # If the value is found, modify the results.
                if index != -1:
                    # Check the boundary condition.
                    if index != 0:
                        # Check if the value is negative in the line.
                        if line[index - 1] == '-':
                            # If so, update the line by removing it.
                            line = (line[:index - 1] + line[index + 3:]).strip()
                        # Otherwise, do not write the line.
                        else:
                            write_line = False
                    else:
```

```
                    write_line = False

            # If the line is to be written, update the 'outfile'.
            if write_line:
                outfile.write(line + '\n')

        # Include the known values back into the CNF file.
        for cell in self.known_values:
            outfile.write(str(cell) + '\n')
```

As indicated, following our reconstruction of various Sudoku puzzles, we may simply use our `SAT.py` to solve the problem. By inputting the constraints, as in specifying the exact location of certain values, we are able to reduce the CNF file to reflect a smaller subset of constraints that must be satisfied.

By treating known variables as constains instead of variables in 'walksat', there are various potential advantages and disadvantages:

Advantages:

- Reduces the search space by eliminating known variables from consideration, whcih prunes parts of the search space.
- Avoids flipping known vairables, so the search is more "directed".
- May speed up convergence by only focusing on unknown variables.

Disadvantages:

- May become stuck in local optima more easily without the ability to violate all constraints.
- Known variables provide a "gravitational pull" that guides the search, and removing them may hurt.
- Implementation requires modifying the CNF generation to remove known variables, which is tricky.

To experimentally test this, we implement the `simplify()` method to construct the `puzzle1_modified.cnf` and `puzzle2_modified.cnf` files. From this point, we are able to solve the satisfiability problems using the following parameters, to determine if the pruning of the CNF did indeed contribute to a faster solution.

```
sat.sat(p_value = 0.3, max_flips = 100000, algo = 'walksat', h_value = 0)
```

puzzle1.cnf

```
Flips: 26146
5 7 8 | 4 2 6 | 1 3 9
6 1 4 | 7 9 3 | 5 2 8
9 2 3 | 1 8 5 | 7 6 4
---------------------
8 3 2 | 5 6 1 | 4 9 7
1 6 9 | 8 7 4 | 3 5 2
7 4 5 | 2 3 9 | 8 1 6
---------------------
4 9 1 | 6 5 8 | 2 7 3
2 8 6 | 3 1 7 | 9 4 5
3 5 7 | 9 4 2 | 6 8 1

Time: 0:03:56.675008
```

puzzle1_modified.cnf

```
Flips: 6836
5 1 9 | 6 3 2 | 8 7 4
6 8 7 | 5 9 4 | 1 3 2
2 4 3 | 1 8 7 | 5 6 9
---------------------
8 9 2 | 4 6 5 | 3 1 7
1 5 4 | 8 7 3 | 9 2 6
7 3 6 | 9 2 1 | 4 5 8
---------------------
9 7 5 | 3 4 6 | 2 8 1
3 2 8 | 7 1 9 | 6 4 5
4 6 1 | 2 5 8 | 7 9 3

Time: 0:01:01.426320
```

puzzle2.cnf

```
Flips: 14438
5 3 6 | 9 8 2 | 4 1 7
2 7 8 | 1 4 5 | 6 3 9
4 9 1 | 6 7 3 | 5 2 8
---------------------
8 4 9 | 2 6 1 | 7 5 3
6 1 3 | 8 5 7 | 9 4 2
7 5 2 | 3 9 4 | 1 8 6
---------------------
9 6 5 | 4 2 8 | 3 7 1
1 2 4 | 7 3 9 | 8 6 5
3 8 7 | 5 1 6 | 2 9 4
```

```
Time: 0:02:11.589662
```

puzzle2_modified.cnf

```
Flips: 8490
2 3 7 | 4 9 6 | 8 5 1
5 6 8 | 1 2 7 | 9 3 4
4 9 1 | 3 5 8 | 2 6 7
---------------------
8 2 9 | 7 6 5 | 1 4 3
6 4 5 | 8 1 3 | 7 9 2
7 1 3 | 9 4 2 | 5 8 6
---------------------
3 5 6 | 2 8 1 | 4 7 9
1 8 4 | 6 7 9 | 3 2 5
9 7 2 | 5 3 4 | 6 1 8

Time: 0:01:14.481192
```

As highlighted, the simplified encoding converges to a solution faster (less flips and time), which supports the hypothesis that treating the known cell values as constants is beneficial. While local optima issues may still exist, for the Sudoku problems, the modification of the CNF files effectively reduced the complexity.

Puzzle Bonus - Bonus #5

As indicated by the assignment instructions, the `puzzle_bonus.cnf` file contains a problem that is difficult to complete, and thus a solution is welcome, though certainly not required. Thus, considering that the solution found is relatively efficient, the time/effort put into solving this problem merits extra credit.

Given that there are several starting values, it was effective to first use the `simplify()` method to construct a CNF file with a reduced number of constraints (`puzzle_bonus_modified.cnf`), without loss of generality. To this end, we were able to effectively reduce the complexity of the problem, while still ensuring that a solution would be found.

From this point, we are able to solve the satisfiability problem using the following parameters, which is the optimal way to solve this problem.

```
sat.sat(p_value = 0.3, max_flips = 100000, algo = 'walksat', h_value = 0.8)
```

The results of using the advanced/enhanced 'walksat' algorithm, with a `p_value` of 0.3 and further randomization (along with a better scoring method) introduced by an `h_value` of 0.8, are provided as follows:

```
Flips: 5076
5 3 4 | 6 7 8 | 1 9 2
6 7 2 | 1 9 5 | 3 4 8
1 9 8 | 3 4 2 | 5 6 7
---------------------
8 5 9 | 7 6 1 | 4 2 3
4 2 6 | 8 5 3 | 9 7 1
7 1 3 | 9 2 4 | 8 5 6
---------------------
9 6 1 | 5 3 7 | 2 8 4
2 8 7 | 4 1 9 | 6 3 5
3 4 5 | 2 8 6 | 7 1 9

Time: 0:00:40.762464
```

Evidently, despite the fact that the problem is more computationally difficult to solve, we are able to use simplified encoding (CNF), along with an enhanced algorithm and better scoring to benefit from the additional constraints placed by the known cell values. In this case, we converge to a solution faster (less flips and time), which reflects the culmination of this assignment. While further improvement may be found in implementing even more advanced algorithms, for this Sudoku problem, we are able to effectively reduce the complexity and find a solution.

Literature Review - Bonus #6

The Sudoku AI problem from this assignment is a well-known one, and it has been studied (relatively) extensively in the AI reseaerch community (alongside problems involving map coloring and other games). While there is no required additional part of the assignment, I found it relevant to complete a brief review of a somewhat-relate paper on SAT problems (and CNF).

Thus, a paper was chosen and read through enough to get a sense of the approach and major findings. In this report, we briefly discuss the paper. The discussion should describe the problem attacked by the paper, give a quick summary of the main result(s), and discuss the basic approach of the paper.

Paper: *Optimized CNF Encoding for Sudoku Puzzles* by Gihwon Kwon, Himanshu Jain

This paper provides an overview of an optimized encoding method to represent Sudoku puzzles in conjunctive normal form (CNF), focusing primarily on reducing complexity by enhancing the encoding. As we know, Sudoku (a logic-based number placement puzzle) may be represented by CNF (a standard format to represent boolean satisfiability problems in logic). In this paper, the authors hint at

an improved way to encode the rules/constraints of the Sudoku puzzle into CNF clauses, alloiwng Sudoku to be solved using SAT solvers.

Abstract "A Sudoku puzzle can be regarded as a propositional SAT problem. Various encodings are known for encoding Sudoku as a Conjunctive Normal Form (CNF) formula. Using these encodings for large Sudoku puzzles, however, generates too many clauses, which impede the performance of state-of-the-art SAT solvers. This paper presents an optimized CNF encoding in order to deal with large instances of Sudoku puzzles. We use fixed cells in Sudoku to remove obvious redundancies during the encoding phase. This results in reducing the number of clauses in the CNF encoding and a significant speedup in the SAT solving time."

The key idea is that an optimized encoding removes the obvious redundancies when encoding a given problem in CNF, which creates an optimized encodign that is used to speed up SAT solving to a significant degree.

The authors allude to the idea that focusing on the data, rather than the algorithm, may be the best way to improve the SAT solving method, as reducing the number of constraints significantly speeds up the process of satisfies the clauses. By using the "zChaff" encoding system, many Sudoku puzzles are reduced in complexity, allowing them to be solved in less than a tenth of a second.

In summary, the paper argues that optimizing the encoding method to produce fewer clauses and variables compared to the basic CNF encodings of Sudoku is rather beneficial. The key process to doing so is using unary constraints that reflect the value at a singular cell to update the other clauses within the CNF file. By analyzing the performance of the "zChaff" encoding, it is evident that it is able to solve Sudoku problem faster compared to naive approaches in a computational setting.