# CSE 30151 Theory of Computing
# Fall 2017
# Project 1-SAT Solving

Version 3: Sept. 21, 2017

The purpose of this project is to gain an understanding of one of the most central problems of computing: Boolean Satisfiability or **SAT**[1]. Such problems determine whether or not there is an assignment of values (true or false) to a set of Boolean variables that makes some Boolean expression true. If an assignment exists, the expression is called "**Satisfiable**"; if no such assignment exists, the expression is said to be "**Unsatisfiable**."

## 1 Overview

The particular form of Boolean expressions of interest here is called **Conjunctive Normal Form (CNF)**, and defined as

- Each expression is the logical AND ($\wedge$) of a set of **clauses**.
- Each clause is the logical OR ($\vee$) of a set of **literals**.
- Each literal is a variable or its negation (prefix $\sim$).

We will call properly formatted expressions of this form as **wffs** (**well formed formulas**). When the clause with the largest number of distinct literals has k such literals, the wff is said to be in the problem class "**kSAT**". An example[2] of a wff from 3SAT is

$$(x_1 \ \vee \ \sim x_2) \ \wedge \ (x_2 \ \vee \ x_3) \ \wedge \ (\sim x_1 \ \vee \ \sim x_3) \ \wedge \ (\sim x_1 \ \vee \ \sim x_2 \ \vee \ x_3) \qquad (1)$$

This 3SAT wff has 4 clauses and 3 variables, and is satisfiable (try $x_1 = 0,\ x_2 = 0,\ x_3 = 1$), but if we add the clause $(x_1 \ \vee \ x_2 \ \vee \ \sim x_3)$, the wff is then unsatisfiable - no possible assignment makes all clauses true.

The specific goal of the project is to build enough of some simple SAT solvers to understand that something strange happens to the time complexity of algorithms that solve SAT problems when the structure of the problem reaches a certain point. The goal is understand that this complexity inflection point occurs when wffs are members of kSAT, where k is 3 or greater. To the best of our current knowledge, the best possible algorithm to solve such 3SAT or above problems is exponential in the number of variables, in the worst case. In contrast you will find that 1SAT or 2SAT problems are not exponential (in fact they have linear or low polynomial algorithms). If by some chance you find an algorithm for 3SAT or

[1] See https://en.wikipedia.org/wiki/Boolean_satisfiability_problem or section 7.4 of Sipser.
[2] Taken from D. Knuth, The Art of Computer Programming, Volume 4 Satisfiability.

above whose worst case complexity is polynomial, then be prepared for very great fame and glory!

As an aside, if you google "sudoku to sat" you will find a long list of ways to convert sudoku puzzles to SAT problems. Feel free to try some on your solvers and let me know what happens.

## 2 Project Structure

Your project will consist of a series of simple programs that accept sample wffs represented as files in DIMACS CNF format (discussed in Section 4.1), and determine if they are satisfiable or not. In addition to this determination, you will also measure your execution time for each solution, and graph that time as a function of the number of variables in the wff. In particular, the project includes developing the following programs:

1. A solver that determines satisfiability by simply generating all possible assignments ($2^V$ of them where V is the number of variables), trying each assignment against the wff, and stopping when either a solution is found, or when all possible assignments have been tried.
2. A solver that uses simple backtracking that has much better average performance.
3. A solver for just 2SAT that does better than $O(2^V)$.

We are not after world-class performance here, just some relatively simple code that is functional and from which you can draw insight. You are free to go beyond the minimal requirements and produce enhanced code, and if in the instructor's view it truly represents valuable extensions, then it will be considered for extra credit.

You are free to use C, C++, or Python. The latter in particular has proven in the past as perhaps the most efficient way to get decent working code (the overall goal). If you use Python, feel free to use the *time* module. Use of any other modules requires preapproval of the instructor.

### 2.1 Exhaustive Search

The first program, to be called **brute-*team***, where *team* is your team name (all members in the should use the same team name), determines satisfiability by trying all combinations of assignments until either one is found that satisfies, or they all fail. The program should have one command line argument, the name of the input file that holds the wffs to test, and a binary argument where "1" triggers some optional tracing that you may use for debugging, and "0" suppresses such intermediate output. Performance runs should be done using the latter.

While not required, it is highly suggested that you write **brute-*team*** from 4 functions for reusability:

1. A function which reads in the next wff from a specified input file.
2. A function that generates the next possible assignment for the current wff you are working with (remember there are $2^V$ of them where $V$ is the number of variables.

3. A function notionally called **verify** that takes a wff and an assignment and returns whether or not the assignment satisfied the wff.

4. A function that creates the output line for the wff in the desired format.

**brute-*team*** should time the execution time taken for each wff starting with the first call to the assignment generator to the completion of the call to **verify**, and avoiding the time to read and parse the wff from the given file, and the time to generate the output line. This time should be in microseconds. If you are using Python, the package "time" has a function "time" within it which if you call as "time.time()" you get the current clock as a float in seconds. Multiplying this by 1E6 gives microseconds.

Given that in some languages like Python, this program can begin to take a significant amount of time (up to minutes per clause) once the number of variables exceeds around 20 (which corresponds to $2^{20}$ cases to try or about a million). Thus it may make sense in your program to include an extra call parameter that specifies a number of variables above which the program will skip any input clause. This allows you to get the program working correctly before trying the bigger cases.

### 2.2 Backtrack

The second program, to be called **backtrack-*team***, where *team* is your team name, determines satisfiability by building an assignment piecemeal as discussed in class. You choose a variable to guess a value, and record the variable name and value on a stack, along with whether or not there is another value to try for that variable. You then see which clauses have now been satisfied. If all clauses have been satisfied, then you can declare the wff "Satisfiable".

If in the checking you can't satisfy all clauses, but no clause has been unsatisfied, then you need to make another choice for a different variable. You can use any heuristic (such as try a value for the next variable), but a particularly good one is to try a variable that is part of one of the clauses you haven't satisfied yet.

If in the checking you find a clause that has all its literals with values that are all false, then you can erase the last assignment as specified on the top of the stack, and if the flag associated with that entry says that the other value has not been tried, then you can flip the assignment for the variable and try again. If both values have been tried, then pop the stack and repeat on the prior one. If you empty the stack declare "Unsatisfiable."

Again you should time from the start of the first push to the response.

The verify function may be handy here when debugging test cases that do not include the answer.

### 2.3 2sat

The third program, to be called **2sat-*team***, where *team* is your team name, determines satisfiability by observing that for 2SAT, if a value is assigned to a variable then any clause that has a literal that requires the negation of that value to be true will automatically force an additional assignment to the variable associated with the second literal in the clause. For example in $(\sim x_1 \vee \sim x_2)$, if you tentatively assign a value of 1 to $x_1$ then you must assign "0" to $x_2$ to make the second literal true. This new assignment can then make other clauses true, and also force additional assignments to other variables. In many wffs the result is like

a long string of dominos arranged in tree-like structures - drop one and others fall, perhaps on multiple branches.

If in the process of making such chained assignments, you ever find a case where you are trying to assign both true and false to the same variable, then you have reached a contradiction, and you must erase all these intermediate assignments and restart the process with a different value to the original variable. If there are no alternatives to look at then it is unsatisfiable. This is essentially the same as the backtracking in the prior program.

If these chained assignments do not result in a conflict, then you know you have a satisfying assignment for the set of clauses that have been made true so far, and that all remaining clauses do not depend on any of the variables assigned so far. You can thus freeze the assignment and start working on the clauses that have not yet been affected as essentially a separate, and smaller, problem.

An interesting observation is that you can actually build this trick into the prior backtrack program. During the testing of an assignment, if you reach a clause where all but one literal is false, and the final literal's value does not have a value yet, you can force a new assignment to make the literal true.

### 2.4 Test WFFs

Under the Projects tab of the class website https://www3.nd.edu/ kogge/courses/cse30151-fa17/index.html there is a directory called Project1. Within this directory there are several test files, each containing multiple wffs:

- **kSAT.cnf** has a series of wffs of different numbers of variables, clauses, and literals per clause for which the correct answers are included in the first comment line.

  This file should be used with **brute** and **backtrack** but not with **2sat**.

- **kSATu.cnf** has a series of wffs of different numbers of variables, clauses, and literals per clause for which the correct answers are NOT included in the first comment line.

  This file should be used with **brute** and **backtrack** but not with **2sat**.

- **2SAT.cnf** has a series of wffs of 2 literals per clause for which the answers are NOT given.

  This file should be used with **2sat**, and may optionally be used with the other two solvers, but you may find that execution time gets excessive.

In addition to the above files, each team should create a .cnf file for the trivial 2x2 sudoku discussed in the class notes, and show that you get the correct assignment. The 2sat program is sufficient for this.

If you are brave, try creating a wff for a 3x3 or 4x4.

## 3 Documentation

Two pieces of documentation, both in PDF format, are required. One (entitled **readme-*team*.pdf** is submitted once by the entire team; the other **teamwork-*netid*.pdf** is submitted separately by each member of the team.

### 3.1 readme-team

A key part of what your teams submit is a **readme-*team*.pdf** that includes the following in this order:

1. The members of the team.
2. Approximately how much time was spent in total on the project, and how much by each student.
3. A description of how you managed the code development and testing. Use of github in particular is a particularly strong suggestion to simplifying this process, as is some sort of organized code review.
4. The language you used, and a list of libraries you invoked.
5. A description of the key data structures you used, especially for the internal representation of wffs, assignments, and choice point stacks.
6. For each of the programs and each of the provided test files a chart of execution time versus number of variables. The points from the chart should come from your output run, with wffs that were unsatisfiable shown as red points, and wffs that were satisfiable shown as green. You may also want to use different symbols for wffs of different kSAT subsets (esp. 2SAT).

   The representation of the output files as .csv was done to make it easy to import into a spreadsheet where graph generation is easy.
7. From this data a curve fit to the worst case times, again as a function of V. Here it may be useful as was done in class to divide the time by the total number of literals in each wff first.
8. A description of what you learned in terms of the relative complexity of the different solvers, especially as a function of the number of literals per clause. Especially important is what you observed in the transition from 2SAT to 3SAT and above.
9. If you did any extra programs, or attempted any extra test cases, describe them separately.

Only one member of the team need submit this report to their dropbox.

### 3.2 teamwork-netid

In addition, each team member should prepare a brief discussion of their own personal view of the team dynamics, and stored in a PDF called **teamwork-*netid*.pdf**, where netid is your netid. The contents should include:

1. Who were the other team members.
2. Under whose netid is the **readme-*team*.pdf**, code, and other material saved.
3. How much time did you personally spend on the project, and what did you do?
4. What did you personally learn from the project, both about the topic, above programming and code development techniques, and about algorithms.

5. In your own words, how did the team dynamics work? What could be improved? (e.g. did you use github and if so did it help, did you meet frequently enough, etc.)

6. From your own perspective, what was the role of each team member, and did any member exceed expectations, or vice versa.

Each student's submission here should be in their own words and SHOULD NOT be a copy of any other team members submission, nor should they be shared with the other team members. These reports will be kept private by the graders and instructor, and will be used to ensure healthy team dynamics. The instructor retains the right to adjust the score of an individual team member from the base score (both up and down) on the basis of these reports. Also, a composite of all such reports from all projects will be used to create an overall "lessons learned" at the end of the project in what techniques seemed to work better, and where problems arose. These hopefully will be of use for the next project.

# 4 File Formats

## 4.1 CNF Format

The DIMACS CNF format[3] is a standard text file format used to describe Boolean expressions in CNF, and is the format of the input files your programs should accept. A description of a wff in this format is a series of text lines formatted as follows:

- An arbitrary number of lines where the first character is "c". These are comment lines.

  For this project we assume that there will be only one such comment line, and following the "c" it has a space and then an integer representing a problem number, then a space and an integer representing the maximum number of literals in any clause, and then optionally a space followed by a "S" or a "U". This last character is present in some test files to allow you to determine if your code is correct. It will not be present when you run a test case with an "unknown" (at least to you) answer.

- A line, denoted the "problem line". that begins with a "p,"followed by a space and then the string "cnf" (to denote the file format), followed by another space and the number of variables, and finally a space and the number of clauses.

- Each of the following lines represents a single clause as a series of space separated integers representing literals, and ending with a "0". In for full CNF spec, it is permitted for a clause to stretch over one line; the "0" literal is the key that a clause is complete, and the next line starts a new clause. For the test files you will deal with, you may assume a single clause always fits on one line.

Each of the integers in a clause line represents a literal based on the variable whose name is associated with the absolute value of of the literal. A positive integer is a literal whose value is that of the associated variable. A negative integer is a literal whose value is the logical negation of the associated variable. A "0" is an invalid representation of a variable and may only be used to end a clause line.

For this project a single test file may have multiple wffs within it, with each such wff formatted as above.

[3] see http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html for more detail

An example of the CNF format for the problem mentioned in Section 1 is:

```
c 17 3 S
p cnf 3 4
1 -2 0
2 3 0
-1 -3 0
-1 -2 3 0
```

The website http://www.satcompetition.org/ has a rich description of a series of challenge problems in this format that have been studied in the past.

### 4.2 Output Format

Each of your programs for this project should generate an output file in a ".csv" format, with one line for each wff in the input file. Each line has the following items in text format separated by ",".

- The problem number of the wff (from the input).
- The number of variables in the problem (from the input).
- The number of clauses in the problem (from the input).
- The maximum number of literals in any clause (which kSAT the wff is in).
- The total number of literals in the problem wff (You must compute).
- An "S" if you predict satisfiability or a "U" if you predict unsatisfiability.
- A "1" if the test code provided the answer and you agreed, or a "-1" if you got the wrong answer, or a "0" if the test file did not have an answer provided.
- The execution time in microseconds.
- If you predict satisfiability, the values you got for the assignment as "1" for true and "0" for false, again comma-separated one entry per variable.

  Note that it is possible, especially for backtrack, to find a satisfying solution without giving a value to some variables. In such cases it may be interesting to output a "-1" for such variable values.

The last line in the file should then have the following numbers, again comma separated:

- The name of the input file you processed (without the extension).
- Your team name.
- The number of wffs in the file.
- The number of those wffs that you declared satisfiable.
- The number of those wffs that you declared unsatisfiable.
- The number of those wffs that had answers provided.

- The number of those wffs with answers that you got correctly.

As an example the output line for the sample wff above would be:

```
17,3,4,3,9,S,47.6,0,0,1
```

## 5 Submission

- Each team member should create a directory in their own class dropbox called **Project1-*team***, where *team* is your team name (all members in the should use the same team name).
- When the team is ready to submit, one (and only one) team member will place copies of all code in your directory.
  - Your code should be runnable on any of the studentnn.cse.nd.edu machines so that if there is an issue the graders can run the code themselves.
  - If you wrote in a compiled language like C++, include all needed source files (excepting standard libraries), a make file, and a compiled executable. The source code is there to allow the graders to look at the code for comments and to resolve any discrepancies that may arise in looking at your results.
  - If you wrote in a language like Python make sure your code is compatible with one of the versions supported on the studentnn.cse.nd.edu machines, again to allow the graders to check something if there is an issue.
- Also in the same dropbox as the code the team should place an output file for each of the test files that you ran. The format should be as described in Section 4.2, and the name should be the same name as the test file but with a ".csv" rather than a ".cnf" extension. For this project there should be 5 of them.
- Finally include the .cnf file for the 2x2 puzzle and the output you got from that (hopefully showing the correct assignment).

In addition, every team member should include in their own Project1 directory a copy of their **readme-*team*** file, again in pdf. For the team members who did not upload code and readme's, this should be the only thing in their own dropbox directory, unless they did some solitary extra credit.

## 6 Grading

Grading of the project will be based on a 100 points, divided up as the following

- Points off for late submissions (10 points per day).
- 5 points for following naming and submission conventions.
- 10 points for "reasonably" commented source code.
- 50 points: 10 points each based on the percent of cases you got correct for running **brute** against **kSAT.cnf** and **kSATu.cnf**, the same for **backtrack**, and 10 points for running **2sat** against **2SAT.cnf**.

- 5 points for the conversion and execution of the 2x2 Sudoku problem.
- 25 points for completeness and quality of the readme file.
- 5 points for the teamwork report.

All but the last item will be common to all members of a team. The last entry is specific to each member.

Special consideration will be given for successful projects done by 1 or 2 person teams, and/or for exceptional algorithm performance (as judged by the instructor). If someone can actually solve a Sudoku puzzle, for example, that would be neat! Even better, if you get a provably correct polynomial kSAT solver for $k > 2$. you get an automatic class grade of "A" along with your Turing Award (look it up).