A Graphical 3-SAT Program for Research and Instruction

Thomas E. O'Neil
Computer Science Department
University of North Dakota
oneil@cs.und.edu

Abstract

This paper describes a program that provides a graphical user interface for experimenting with the satisfiability problem on 3-CNF Boolean expressions. The program contains a random expression generator and provides support for heuristic-guided solution guessing with backtracking. The graphical interface displays the status of all variables and clauses in the expression with color highlights that indicate their current status as the expression is processed. Counts of the positive and negative occurrences of unassigned variables are displayed to aid the user in selecting and setting the truth value of the next variable. Expression simplification and backtracking are automatic. The program keeps track of the number of computational steps required to find a solution or determine that none exists.

The graphical 3-SAT program is equally valuable for research and instruction. Student users can quickly gain an understanding of the satisfiability problem and come to appreciate both its simplicity and its inherent complexity. The program also illustrates the techniques implemented by the fastest known deterministic algorithms for satisfiability, and it is sufficiently robust to provide a testbed for satisfiability research.

Introduction

3-SAT is the problem of determining the satisfiability of a Boolean expression where the expression is in conjunctive normal form with three literals per clause. A Boolean expression is satisfiable if and only if its value is true for some assignment of values to its variables. The problem of determining whether an arbitrary Boolean expression is satisfiable is a well-known NP-complete problem. The satisfiability problem is commonly presented in undergraduate-level courses on algorithms and complexity as an example of a problem that is computationally intractable. The problem seems simple enough for small expressions, but as the number of variables becomes large, the running time explodes exponentially.

The fastest deterministic algorithms for 3-SAT can be applied to expressions with a few hundred variables. The algorithms with the best running times are variations of heuristic-guided solution guessing with backtracking. Since the heuristics play a critical role in the efficiency of the search, it is useful for researchers to have an interactive graphical tool for empirical testing of sample expressions where various heuristics can be applied by hand. This paper describes such a tool. The program presents a graphical display of a randomly generated expression and allows the user to search for a satisfying truth assignment. Counts of the positive and negative occurrences of the remaining variables are displayed to aid the user in selecting and setting the truth value of the next variable. Expression simplification and backtracking are automatic. The program keeps track of the number of computational steps required to find a solution or determine that none exists.

The graphical 3-SAT program is equally valuable for research as for instruction. It can be presented as a game where the goal is to determine the satisfiability of an expression in as few steps as possible. In running the program, students can quickly gain an understanding of the satisfiability problem and come to appreciate both its simplicity and its inherent complexity. The program provides a vehicle for stimulating interest in theoretical computer science and for bringing research on a classical problem in computing to the classroom.

The Satisfiability Problem

A Boolean variable is an object whose value is true or false. A Boolean expression specifies the application of logical operations conjunction, disjunction, and negation to Boolean variables or subexpressions. A Boolean expression is satisfiable if and only if its value is true for some assignment of values to its variables. The problem of determining whether an arbitrary Boolean expression is satisfiable is an NP-complete problem [1]. All known algorithms for problems in this class take exponential time in the worst case, and it remains unknown whether polynomial-time algorithms exist.

An arbitrary Boolean expression can be transformed to an equivalent expression in conjunctive normal form (CNF). A CNF expression is a conjunction of clauses. Each clause is a disjunction of literals, and each literal is a variable or a negated variable.

Procedure DP

Given a set of clauses *E* defined over a set of variables *V*:

- If *E* is empty, return "satisfiable"
- If *E* contains an empty clause, return "unsatisfiable"
- *Unit Clause Rule*: If *E* contains a unit clause *C*, assign to the variable mentioned the truth value which satisfies *C*, and return the result of calling DP on the simplified expression.
- *Branching Rule*: Select from *V* a variable *v* that has not been assigned a truth value. Assign it a value and call DP on the simplified expression. If this call returns "satisfiable", then return "satisfiable". Otherwise, set *v* to the opposite truth value and return the result of calling DP on the re-simplified expression.

Figure 1: The basic Davis-Putnam procedure.

When each clause is restricted to contain no more than 3 literals, the expression is said to be 3-CNF. The satisfiability problem for 3-CNF expressions is known as the 3-SAT problem.

If the discipline of computer science is old enough to have classical problems, satisfiability has to be placed in that category. Boolean expressions are simple logical formulas, but they are sufficiently powerful to provide a model for all of computation, since every computer program and computation can be represented as a Boolean expression. As computer programmers, we write code for various applications and test it by assigning specific values to input variables, running the code, and analyzing the output. In testing we repeatedly perform experiments that answer the question "what output will I get from this specific set of inputs?" Satisfiability raises the opposite question – "what inputs are required to get this specific output?" It has many immediate practical applications. In the design of life-critical systems, for example, one approach to establishing the reliability of a system is to identify unacceptable outputs and then to prove that no set of inputs will produce the unwanted outputs. Thus satisfiability is a fundamental problem in both theoretical and applied computing, and it is very appropriate to present it to undergraduates in courses on algorithms or theoretical computer science. The program described here provides an excellent tool for introducing the topic to students.

The Davis-Putnam Procedure

A number of deterministic algorithms for the satisfiability problem have been studied over the past four decades. Solution-guessing with backtracking is perhaps the oldest algorithm, and with appropriate heuristics and optimization, it remains the fastest. Algorithms based on a backtracking search are referred to in research literature as simple Davis-Putnam (DP) procedures [3]. Figure 1 contains a common description of the DP

Number of Variables	Number of Clauses	Average Branching Steps
20	85	1
40	171	4
60	255	9
80	340	18
100	425	35
120	510	71
140	595	145
160	680	292
180	765	604
200	850	1238
220	935	2543
240	1020	5217
260	1105	10549
280	1190	21783
300	1275	44240

Figure 2: Benchmark data for DP algorithms.

algorithm (from [5]). The expression is a conjunction of clauses, and each clause initially has three literals. With each assignment of a variable, the expression is simplified by removing the clauses that are satisfied by the assignment and by removing from the remaining clauses the literals that contradict the assignment. If a clause becomes empty after repeated simplifications, it is unsatisfied. In that case backtracking will be applied to try a different assignment. If a clause in the simplified expression contains only one literal, the Unit Clause Rule is applied to satisfy that clause. Repeated application of the Unit Clause Rule is called unit propagation.

Various heuristics can be used when the Branching Rule is applied to select the next variable and truth value to be assigned. As with any backtracking search, it is better to exploit constraints as early as possible to reduce the size of the search. In the context of a satisfiability search, the goal is to satisfy as many clauses as possible and to produce as many unit clauses as possible with each step. Thus it makes sense to choose the variable that occurs most frequently among the remaining clauses. Also, if there are more positive occurrences of a variable than negative occurrences, it makes sense to assign the variable *true* before *false*. Frequency of occurrence in 2-literal clauses is another factor to consider. When a variable in a 2-literal clause is assigned, the clause will either be satisfied or it will become a 1-literal clause, and unit propagation automatically processes 1-literal clauses. So, to take full advantage of unit propagation, it is beneficial to create as many 1-literal clauses as possible.

The Pure Literal Rule is another common heuristic. A literal is pure if all occurrences of it in the expression have the same polarity (all positive or all negative). In that case, there is no need to try both truth values of the associated variable – just choose the value that will satisfy all the literals. It may also be beneficial to look for literals that occur only once (singletons) or twice (doubletons) in the expression. If a literal x occurs in 6 clauses and the literal $\neg x$ occurs in only one, then the literal x is "almost pure," and it makes

sense to try $x = true$ first. These heuristics are applied, for example, in Jon Freeman's POSIT algorithm [4].

Figure 2 contains some data from an empirical study undertaken by Crawford and Auton [2]. They ran a highly optimized, heuristic-guided DP algorithm called TABLEAU on batches of 10,000 randomly generated expressions. The number of variables n ranged from 20 to 300. The table shows data for expressions where the number of clauses m is $4.25n$. These expressions are in the critical region $4n < m < 5n$ where the probability that the expression is satisfiable drops rapidly from nearly 100% to nearly 0%, where the expressions are most difficult to solve, and where the running time for deterministic algorithms is the highest. Since running times are highly machine-dependent, the table gives only the average number of branching steps (applications of the Branching Rule) required for each computation. Applications of the Unit Clause Rule are not counted, since unit propagation can be accomplished in polynomial time. It is the Branching Rule that induces exponential time complexity, and we see that the number of branches approximately doubles with each increment of $20n$. This data provides a benchmark for empirical satisfiability testing.

The 3-SAT Backtracker

The 3-SAT Backtracker is a program that provides a graphical user interface for experimenting with the DP algorithm on randomly generated 3-CNF Boolean expressions. The program was developed using Java (JDK 1.3) and the Swing graphical component library. The expression generator is a separate unit written in C. The program has the following components, laid out as shown in Figure 3:

Main Menu -- contains items for creating expressions, setting the control mode, and controlling heuristics.
Variable Panel -- contains lists of *free*, *true*, and *false* variables.
Expression Panel -- contains lists of 3-literal clauses, 2-literal clauses, 1-literal clauses, and satisfied clauses.
Statistics Panel -- contains a table that shows the number of occurrences of each literal in 3-literal and 2-literal clauses.
Control Panel -- contains control buttons and step counts.

The main menu contains options for creating expressions and for controlling the processing mode and heuristics. The user is allowed to generate new expressions, to read existing expressions from files, and to save expressions to files. To generate an expression, an external C program is invoked, and the output from the C program is read from a temporary file. The expression format is simple – the literals are just positive or negative integers, and it is assumed that three consecutive literals represent a clause. The user specifies the number of variables n and the number of clauses m, and the generator produces a list of $3m$ literals, randomly selected from the range $1..n$, and randomly assigned a positive or negative polarity. The generator will not form two literals from the same variable in any single clause, but it there is nothing to prevent it from generating duplicate clauses.

The variable panel contains three scrollable lists that communicate the current truth values of all the variables. The variable names are just positive integers displayed in black if they are unassigned (*free*), blue if they are *true*, and red if they are *false*. There are separate lists for *free* variables, for *true* variables, and for *false* variables, each with a counter box below it to display the size of the list. The user makes an assignment by selecting a variable from one of the lists and pressing a button on the control panel. The variables are moved from one list to another as dictated by the assignment.

The expression panel contains four scrollable lists of clauses that communicate the current status of the expression. There are separate lists for satisfied clauses and for unsatisfied clauses with 3 literals, 2 literals, and 1 literal. As with the variable lists, each clause list has a box below it displaying the size of the list. Clauses move from one box to another as the user makes assignments. Literals within the clauses are highlighted to indicate their status. In the list of satisfied clauses, literals that match the current assignment are highlighted in yellow. In the other lists, literals that are inconsistent with the current assignment are highlighted (lowlighted?) with gray.

The statistics panel contains a scrollable table that displays the number of occurrences of each literal in 3-literal clauses and 2-literal clauses. It contains all the information needed to implement the common heuristics used by DP algorithms. The user can activate or deactivate the table. When the table is active, the occurrence counts are automatically updated with each assignment.

The control panel contains the buttons required for the user to test an expression and a display of the number of assignments made and the number of branching steps. There are

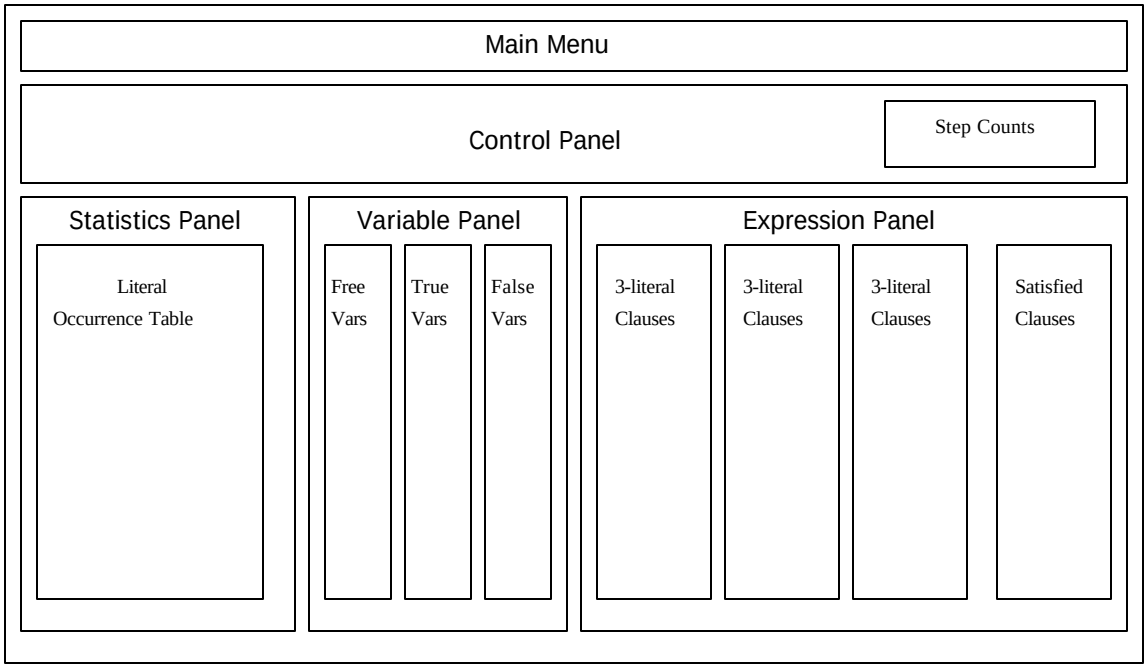

Figure 3: Panel layout for 3-SAT Backtracker.

three processing modes: manual, interactive backtracking, and automatic backtracking. In manual mode, the user can select variables from the variable lists and assign them to *true*, *false*, or *free*. The program updates the variable lists, clause lists, and statistics table accordingly, but there is no automated support for backtracking. Unit propagation will not happen automatically, but it can be initiated by pressing a button.

In the backtracking modes, the program employs a stack of variables to control the sequence of assignments. With interactive backtracking, the user makes an assignment for each branching step, but unit propagation and backtracking occur automatically. The program terminates and declares the expression satisfiable when all clauses reach the satisfied clause list. It terminates and declares the expression unsatisfiable when the variable stack becomes empty. The statistics table can be activated to help the user make decisions regarding which variable and truth value to try next at each branching step.

With automatic backtracking, the user simply presses the *run* button and waits until the expression is declared satisfiable or unsatisfiable. Variables are automatically selected for assignment throughout the process. If the statistics panel is inactive, the first variable on the *free* list is selected for each branching step, and it is assigned *true* first, followed by *false* if necessary. If the statistics panel is active, heuristics are used in the selection of a variable and initial truth value for each branching step. Two standard heuristics are employed: 1) if the expression contains a pure literal, choose an assignment to satisfy it, and 2) choose a variable that occurs most frequently among the remaining clauses.

Experimenting with the Backtracker

There are plenty of experiments to try with the 3-SAT Backtracker. For instructional purposes, running the Backtracker in manual mode with small expressions provides an excellent introduction to the satisfiability problem, and running it in manual mode with large expressions instills in the user an understanding of the word “intractable”. For research projects, the user can compare the step counts from running the program with and without heuristics. Interactive backtracking can be used to try to match or improve upon the benchmark performance data in Figure 2. After developing heuristics in interactive or manual mode, the user can modify the Java code to automate the new heuristics. Using this methodology, various sets of heuristics can be compared.

In summary, the 3-SAT Backtracker is a flexible tool for instruction and research. It provides an extensible framework for experimentation with and gives immediate access to one of the classical problems in computing.

References

1. Cook, S. (1971). The complexity of theorem-proving procedures. *Proceedings of the third ACM symposium on theory of computing,* 151-158. ACM, New York.

2. Crawford, J., and L. Auton. (1996). Experimental results on the crossover point in random 3-SAT. *Artificial intelligence* 81:31-57.

3. Davis, M., and H. Putnam. (1960). A computing procedure for quantification theory. *Journal of the Association for Computing Machinery* 7:201-215.

4. Freeman, J. (1996). Hard random 3-SAT problems and the Davis-Putnam procedure. *Artificial intelligence* 81:183-198.

5. Selman, B., D. Mitchell, and H. Levesque. (1996). Generating hard satisfiability problems. *Artificial intelligence* 81:17-29.