Classics Illustrated:
A Visualization Tool for Theorem-Proving Procedures

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

Abstract

In a classic paper in theoretical computer science from the year 1962, Davis, Logemann, and Loveland [2] presented two methods for testing the consistency of logical formulas. The problem they described remains central to computer science, and their methods for solving it (subsequently dubbed DLL procedures in the research literature) are subject to recurring study and analysis to this day. This paper describes a Java program with a graphical user interface that enables users to randomly generate logical expressions and to observe, step-by-step, the results of applying either of the original DLL methods to determine whether the expression contains a contradiction. The DLL visualization program has value as a tool for both research and instruction. Researchers can use it to gain a clear understanding of the standard methods for automated processing of Boolean expressions. For computer science students, the program can be used in courses on algorithms, complexity, or artificial intelligence. It can even be used in introductory courses on computing as a hands-on demonstration of one of the classic problems in computing – a seemingly efficient program for a simple problem can quickly become overwhelmed by a combinatorial explosion as the problem size increases.

1 Introduction

The satisfiability of Boolean expressions is one of the most thoroughly studied problems in the discipline of computer science. Computing is, after all, just automated Boolean logic. A thorough understanding of Boolean logic is necessary to grasp the foundations of computing, and the ability to manipulate Boolean expressions efficiently is essential for practical computing. Researchers have used computers for automated theorem-proving since the early days of the discipline, and the methods developed in those days remain at the core of the best-known algorithms for satisfiability testing. This paper describes **VisiDLL**, a Java program with a graphical interface that illustrates the classical methods for satisfiability testing. Researchers can use the program as a tool to become quickly grounded in theorem-proving methods that are both classical and current. Educators can use the program to help students quickly grasp the complexity of one of the core problems in our discipline.

A Boolean expression is a formula containing Boolean variables and the logical operators *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). A Boolean expression is satisfiable if and only if its value is **true** for some assignment of values to its variables. An expression that is unsatisfiable contains a contradiction, and it is possible to relate satisfiability-testing to theorem-proving as follows: a theorem, stated as a Boolean expression, is valid if and only if its negation is unsatisfiable. In one of the earliest papers on theorem-proving procedures, Davis and Putnam [3] describe a method for demonstrating the unsatisfiability of an expression. Their method later came to be known as resolution-refutation. A few years later, Davis, Logemann, and Loveland [2] published a follow-up article that proposed a space-saving alternative to the resolution process called the splitting rule. While these two methods are based on the same underlying logical operations, their appearance at run-time is quite distinct. They were originally developed and run on an IBM 704, and the scope of the testing was understandably very limited. It was apparent that the running time could grow exponentially with the size of the expression being tested, and a decade or so later, Cook [1] established satisfiability as the defining problem for a class called NP-complete.

The VisiDLL program provides a graphical interface for generating random Boolean expressions and testing them for satisfiability using either of the methods described by Davis, Logemann, and Loveland (DLL methods). For each method, some heuristics are employed to speed up the expression processing and to give the user a feel for the challenges and experiments that drive continuing satisfiability research. The splitting method is commonly called backtracking, and **VisiDLL** employs an enhanced version of the previously published **3-SAT Backtracker** [4] to illustrate this method. The interface and implementation for the resolution method are new. Both methods are described in more detail in the sections that follow.

2 The Resolution Method

Davis, Logemann, and Loveland described their "Rule for Eliminating Atomic Formulas" very concisely, as shown in Figure 1. A lengthier description of the method is given in the paragraphs that follow. The term "resolution" was apparently introduced by J. A. Robinson [5] in a description of this method that was published a few years later.

> Let the given formula F be put into the form
>
> $$(A \vee p) \,\&\, (B \vee \neg p) \,\&\, R$$
>
> where A, B, and R are free of p. Then F is inconsistent if and only if $(A \vee B) \,\&\, R$ is inconsistent.

Figure 1: Rule III from Davis, Logemann, and Loveland [2].

DLL algorithms operate on Boolean expressions in conjunctive normal form (CNF). A CNF expression is a conjunction of clauses where each clause is a disjunction of literals, and each literal is a variable or a negated variable. When each clause is restricted to contain no more than k literals, the expression is said to be k-CNF. The following expression, for example, is a 3-CNF expression over the variable set $V = (w, x, y, z)$:

$$(w \vee x \vee \neg y) \wedge (\neg w \vee x \vee y) \wedge (w \vee x \vee z) \wedge (\neg x \vee \neg y \vee z).$$

The set of clauses in an expression is a set of constraints. In a resolution-based satisfiability test, this set is expanded by performing the resolution operation on pairs of clauses to add additional constraints with the goal of deriving a contradiction. The resolution operation is defined as follows: given two clauses $c_1 = x \vee c_1$' and $c_2 = \neg x \vee c_2$' where x is a variable, the resolvent clause $c_1' \vee c_2'$ is added to the set of clauses. The addition of the resolvent will not affect the satisfiability of the clause set. If the variable x is assigned **true**, then clause c_1 is satisfied and clause c_2 is not. The clause c_2 will have to be satisfied by an assignment to one of the variables in c_2'. If x is assigned **false**, on the other hand, then clause c_2 is satisfied and c_1 will have to be satisfied by an assignment to one of the variables in c_1'. So it is clear that the resolvent clause $c_1' \vee c_2'$ does not add any new constrains, since it will be satisfied if both c_1 and c_2 are satisfied.

At first glance, it appears that resolution process will continually add more and longer clauses to the clause set. But there are cases in which the resolvent is shorter than the two clauses being resolved. Suppose, for example, that $c_1 = x \vee c$' and $c_2 = \neg x \vee c$'. In this case, the two clauses are the same except that one contains x and the other contains $\neg x$, and the resolvent $c' \vee c' = c'$ is shorter than the clauses it resolves. Now consider what happens if we resolve $c_1 = x$ and $c_2 = \neg x$. The resolvent is the empty clause, which signals that a contradiction has been discovered. So if an expression is unsatisfiable, the resolution process will eventually terminate when it generates an empty clause. We can also apply other operations on clauses to minimize the size of the clause set. Whenever one clause contains all the literals of another clause, the longer clause can be deleted.

That is, if the clause set contains clauses c_1 and $c_2 = c_1 \vee c_2$', then c_2 can be deleted because any assignment that satisfies c_1 and c_2 also satisfies just c_1. As a special case when c_2' is empty, this rule also suppresses multiple copies of the same clause.

If an expression is satisfiable, the resolution process will continue until no new clauses can be created by resolving pairs already in the set. To make the process more systematic, the order of resolution operations can be controlled in a way that allows variables (and the all clauses that mention them) to be eliminated one by one. We pursue this strategy by selecting a variable x and gathering together all the clauses that mention it. After resolving all pairs where one clause contains literal x and the other contains $\neg x$, the clauses containing x can be removed. The resulting clause set with one less variable is satisfiable if and only if the original clause set is satisfiable. This process must obviously come to a stop. With only one variable left, either an empty clause is generated (indicating a contradiction) or no new clause is generated (indicating all clauses have been satisfied).

3 The Splitting Method

The resolution algorithm of Davis, Logemann, and Loveland was running out of memory on an IBM 704, so they proposed the splitting rule as a space-saving alternative. Instead of adding resolvents to a single growing set of clauses, the expression is split into two smaller versions and the two versions are both tested for satisfiability (if necessary). To test an expression F, the algorithm selects a variable x and assigns it to **true**. The expression F is simplified to get F' by removing all clauses that contain the literal x and by removing the literal $\neg x$ from any remaining clauses that contain it. The expression F' is then tested by recursive invocation of the splitting rule. If F' is found to be satisfiable, the algorithm terminates. Otherwise, x is assigned to **false** and F is simplified to get another F', this time by removing the clauses containing $\neg x$ and shortening the clauses that contain x. The satisfiability of F is determined by the satisfiability of the second F'.

Let the given formula F be put in the form

$$(A \vee p) \,\&\, (B \vee \neg p) \,\&\, R$$

where A, B, and R are free of p. Then F is inconsistent if and only if $A \,\&\, R$ and $B \,\&\, R$ are both inconsistent.

Figure 2: Splitting Rule III* from Davis, Logemann, and Loveland [2].

The splitting algorithm is an example of backtracking search of a solution space. It can be very space-efficient if a single copy of the expression is modified for each forward step and restored with each backward step in the process. So backtracking can successfully suppress an exponential space requirement. Its worst-case time requirement,

however, is still exponential. If both possible truth values are tested for each of n variables, the algorithm requires 2^n steps.

4 The Interface for Resolution

The **VisiDLL** program has three major components: the expression generator, the resolution component, and the backtracking component. It contains a class to encapsulate CNF Boolean expressions. The expression generator requests from the user the number of literals per clause k, the number of variables n, and the number of clauses m. It then randomly generates a CNF expression object with n variables, m clauses, and exactly k literals per clause. Within a clause, there are no repeated variables. The expression, however, may contain repeated clauses. The names of the variables are just the natural numbers 1, 2, ..., n.

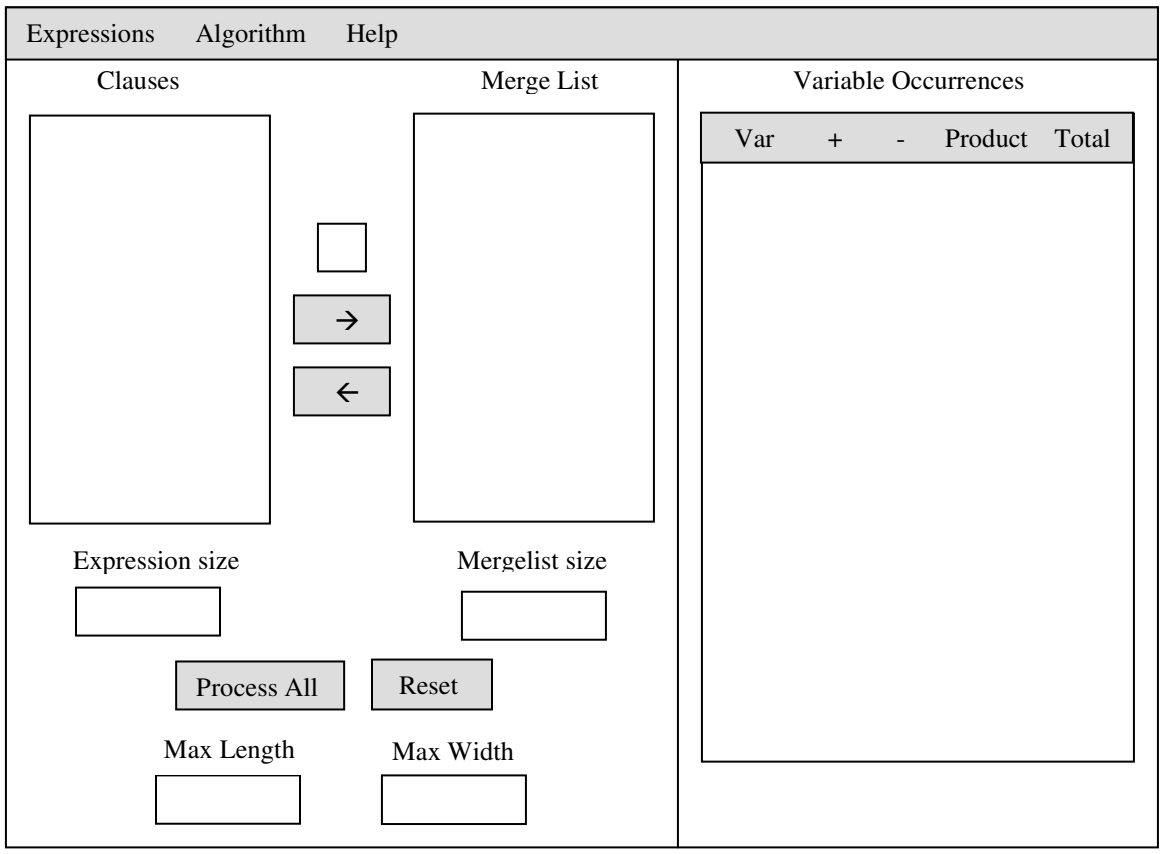

Figure 3: The resolution interface.

The resolution component contains a version of the resolution algorithm and a user interface for conducting resolution refutation under user control (see Figure 3). The resolution interface panel contains a list box that displays all the clauses currently in the clause set. Initially, this box contains just the clauses of the original expression. A second list box labeled the "Merge List" lies to the right of the clause box. This box is a list of resolvents created by a resolution operation. There is a text box between the two list boxes in which the user types the next variable to be used for resolution. The right

arrow button is used to initiate the operation. It causes all the resolvents of clauses that mention the resolution variable *x* to be added to the merge list, and then all clauses that mention *x* are removed from the clause list. The user completes the resolution operation by pressing the left arrow button, which causes the merge list to be merged into the clause list. The panel contains text boxes that display the number of clauses in the clause list and the merge list, as well as boxes that display the maximum number of clauses and maximum width of any clause during the processing of the entire expression. The maximum size of the clause list is a metric for the complexity of the algorithm. The panel also contains a "Process All" button that automatically runs the resolution process to completion and a "Reset" button that restores the expression to its initial state.

The rightmost region of the resolution panel contains a table of statistics to guide the user in selecting the next variable to be resolved. For each variable in the expression, the table displays the number of positive occurrences, the number of negative occurrences, the product of the positive and negative occurrences, and the total number of occurrences. The product column is provided as a measure of the balance between positive and negative occurrences. The default heuristic for selecting the next variable is to choose the one with the smallest value in the product field. The variable with the smallest product should produce the shortest list of resolvents. With resolution, the best strategy seems to be keeping the clause list as short as possible throughout the process.

5 The Interface for Splitting

The backtracking component is an enhanced version of the previously published **3-SAT Backtracker** [4]. It contains an implementation of the splitting algorithm and an interface that allows the user to control the order in which variables are selected and assigned.

The interface contains a main menu and five panels: the variable panel, the expression panel, the statistics panel, the control panel, and the tree panel. The main menu contains items for creating expressions and setting the control mode. The variable panel contains lists of free (unassigned), true, and false variables. The expression panel contains the clauses of the expression partitioned into lists of 3-literal clauses, 2-literal clauses, 1-literal clauses, and satisfied clauses. The lists are dynamically updated as the user makes assignments to the variables. The statistics panel contains a table that shows the number of occurrences of each literal in 3-literal and 2-literal clauses. This information helps the user decide what the next assignment should be. The control panel contains the control buttons appropriate to the control mode and two counts: the total number of assignments and the number of branching nodes in the backtracking tree. The tree panel, not included in the earlier **3-SAT Backtracker**, contains a graphical display of the assignments made so far in the form of a backtracking tree. Figure 4 shows the layout of the interface in the interactive control mode.

When using the backtracking algorithm, the user is challenged to determine satisfiability by making as few assignments as possible. The default heuristic for automated

assignment selection is to choose the variable that occurs most frequently and assign it the **true/false** value that matches its most frequent polarity.

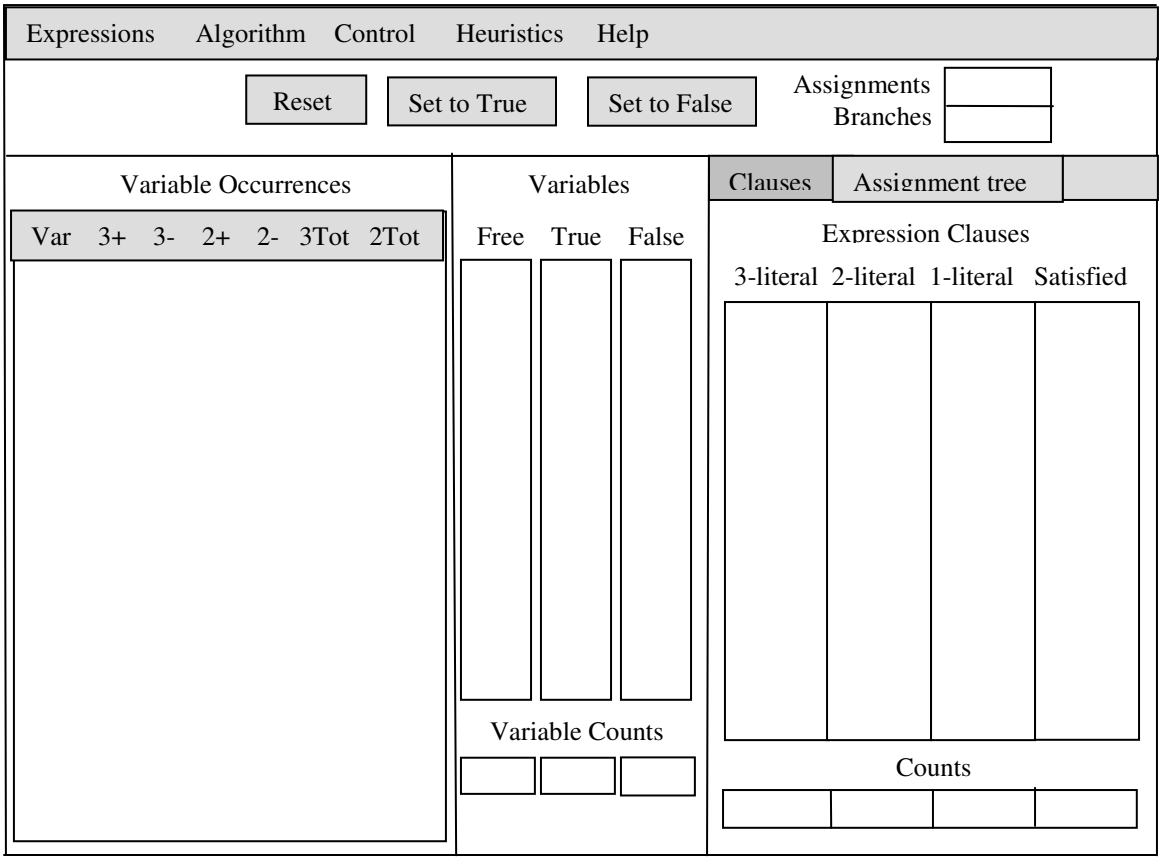

Figure 4: The splitting interface.

6 Conclusion

The **VisiDLL** program provides a conceptualization tool for two standard algorithms for a classical problem in computing. While the two algorithms are closely related, their behavior and appearance at run-time is strikingly different. The resolution method represents a breadth-first expansion of a set of constraints, while the backtracking method is a depth-first search of the set of possible solutions. In both cases, the order in which the variables are processed has a huge impact on the running-time, and the standard heuristics are opposite. With resolution, it is generally best to choose the variable that occurs least frequently. With backtracking, on the other hand, it is best to choose the variable that occurs most frequently.

VisiDLL was developed with two purposes in mind. First, a good conceptualization tool is essential for research. Any progress in satisfiability research must be founded on a thorough understanding of the standard methods that have been so thoroughly studied over the years. Second, a concrete graphical implementation is the most effective way to present abstract material to students. The satisfiability problem remains one of the core

problems in the discipline of computer science, and it is a standard topic in current undergraduate texts on algorithms or complexity. **VisiDLL** is an excellent enhancement to classroom instruction. It provides a way for students to follow the foot-steps of our discipline's first generation of scientists in a journey that began nearly fifty years ago.

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] Davis, M., G. Logemann, and D. Loveland (1962). A Machine Program for Theorem-Proving. *Communications of the Association for Computing Machinery* 5:394-397.

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

[4] O'Neil, T. E. (2001). A Graphical 3-SAT Program for Research and Instruction. *Proceedings of the 34th Annual Midwest Instruction and Computing Symposium (MICS '01)*, Cedar Falls, IA.

[5] Robinson, J. A. (1965). A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery* 12(1):23-41.