Spring 5-2011

Tri-State Boolean Satisfiability with Commit: An Efficient Partial Solution Using Hyperlogic

Kevin Michael Byrd
University of Southern Mississippi

The University of Southern Mississippi

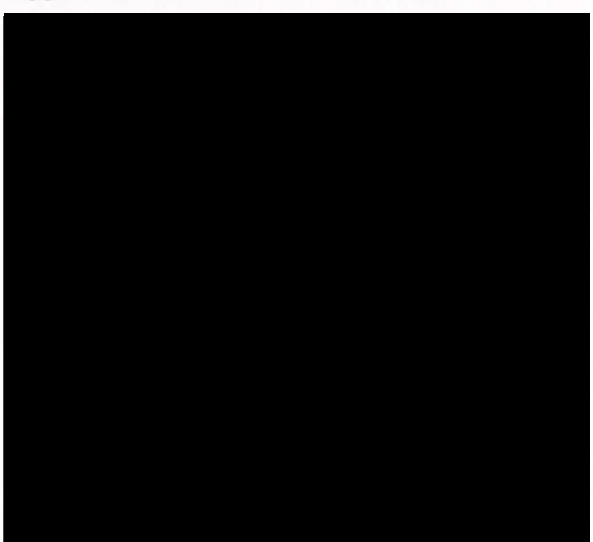
TRI-STATE BOOLEAN SATISFIABILITY WITH COMMIT: AN EFFICIENT PARTIAL SOLUTION USING HYPERLOGIC

by

Kevin Michael Byrd

A Thesis
Submitted to the Graduate School
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Master of Science

Approved:



Dean of the Graduate School

May 2011

ABSTRACT

TRI-STATE BOOLEAN SATISFIABILITY WITH COMMIT:

AN EFFICIENT PARTIAL SOLUTION WITH HYPERLOGIC

by Kevin Michael Byrd

May 2011

We present two implementation enhancements for the Boolean satisfiability problem and one visualization technique. The first is an expansion to a tri-nary logic system with a commit phase. The three states are (1) true, (2) false, and (3) don't care. We abstracted the operations of AND and OR to this hyperlogic system in a novel way. The commit phase works on one variable at a time and transitions values from temporary to permanent whenever possible. We viewed tri-state logic as a hyperspace above the binary (Boolean) logic. The second improvement is algorithmic. We modified the semantics of the classic 3 Conjunctive Normal Form Problem in order to develop a polynomial time algorithm for a simplified normal form – avoiding the need to examine all combinatoric limitations. In particular, we abandoned 3 CNF and used an unstructured left to right associativity. We do not claim that this new semantic is comprehensive. We do claim that it is simpler. Lastly, we introduced a node analogy to help us understand the algorithm itself.

TABLE OF CONTENTS

LIST OF TABLES

Table

CHAPTER I

INTRODUCTION

Problem Statement

Boolean logic is a binary logic system consisting of two states for every variable – true and false [Boolean Logic, 2010]. Its operations are well known [Hans Reichenbach, 1947]. One of the fundamental open questions in Computer Science is whether P is equal to NP [Deolalikar, 2010]. The answer to this question can be coded in terms of the Boolean Satisfiability Problem. In this thesis we expand the problem to a Tri-nary Satisfiability Problem. For the tri-state logic we use herein, we have three states – true, false, and don't care (T/F/D). We cover these in Chapter III.

Following [Linz, 2006, pg 346], "A … Boolean constant or variable is one that can take on exactly two values, true or false, … Boolean operators are used to combine Boolean constants and variables into Boolean expressions." Typically a Boolean Algebra includes at least the binary operation of AND and the unary operator NOT. It is also customary to work with the binary operator OR. For our purposes, then, a Tri-nary Algebra has three states and many more possible operations. In this work we consider only operations that form a hyperspace over Boolean Algebra. This topic is addressed further in Chapter II.

In Chapter III, we discuss the partial satisfiability algorithm which may lead to a solution. We lay out a way to follow the program's logic algorithmically as well. In Chapter IV, we analyze the results and their significance. We conclude in Chapter V. The code is given in the Appendixes.

CHAPTER II

REVIEW OF RELATED MATERIAL

Satisfiability

A Boolean formula is said to be satisfiable if the variables in the formula can be set in a way that makes the formula evaluate to True. A Boolean formula is said to be unsatisfiable if no such assignment is possible; for every combination of variables the formula evaluates to False. The term "satisfiability" is abbreviated to "SAT" and is used as a descriptor for formulas such as 3-SAT or 2-SAT; wherein, each clause in those particular formulas contain exactly 3 or 2 variables, respectively. [Boolean Satisfiability Problem, 2010].

To consider the SAT problem, we also need to introduce conjunctive normal form (CNF) and disjunctive normal form (DNF). The AND operator comprises conjunctives while the OR operator comprises disjunctives. Hence an expression of the form

$$a_1 \vee a_2 \vee a_3 \vee \ldots \vee a_n$$

is in CNF, where each a_i term is a disjunctive clause such as

$$a_i = d_1 \vee d_2 \vee \ldots \vee d_m.$$

Negations can be restricted to the disjunctive expression without loss of generality.

It is straightforward to write a program to test whether a combination of variables exists that make the expression evaluate to True. However, the brute force, "easy," program will take exponential time to execute – on average. This is easy to see. For n Boolean variables, there are 2n possible combinations that may need to be tested.

If we arrive at a combination that evaluates to True before we have tested all combinations, we may short circuit the execution and report that the expression is satisfiable. However, for unsatisfiable expressions, the entire suite of combinations must

be checked. If we had a computer that could implement a fully non-deterministic solution to this problem, then each path through that very broad graph would require only order n time. These two options form the fundamental open question in the discipline of Computer Science and a full solution to problem SAT is considered the holy grail of that discipline.

Tri-State Logic

In this thesis we examine a solution that lies in between these two extremes. Our approach does not always produce a conclusive answer. Indeed, occasionally, we get a False result. However, on average, it covers more than one path through the very broad non-deterministic graph by allowing us to combine some of the pathways using the third state in our tri-state logic system. For example, consider the very simple expression

$$e = a_1 \vee a_2 \vee a_3 \vee a_4.$$

All possible combinations would require us to check 16 possible combinations (16 paths, all of which can be checked in parallel in a NDFSA).

Table 1

The 16 Possible Combinations of 'e = a_1 v a_2 v a_3 v a_4'

a_1	a_2	a_3	a_4
T	T	T	T
T	T	T	F
T	T	F	T
T	T	F	F
T	F	T	T
T	F	T	F
T	F	F	T
T	F	F	F
F	T	T	T
F	T	T	F
F	T	F	T
F	T	F	F
F	F	T	T
F	F	T	F
F	F	F	T
F	F	F	F

However, this problem can be compressed by the addition of a third "I don't care" logic state. In tri-nary logic we may consider Table 2.

Table 2

The Possible Combinations of 'e = a_1 v a_2 v a_3 v a_4' Using Tri-nary Logic

a_1	a_2	a_3	a_4
T	D	D	D
D	T	D	D
D	D	T	D
D	D	D	T

This is our first simplification.

Our second simplification is a short-circuit operation. Variables are "born" in the D state. Only when we conclude that a variable *must* be T or F to enforce satisfiability do we commit a D to a T or F. So, for example, if we have

a_1 v a_2 v a_3 v a_4,

then all variables must evaluate to True.

Suppose we have a singleton. Then that singleton must be True. Here we commit that singleton to a True value. Should it appear later in the expression in another singleton that is negated, we are forced to exit the computation and conclude the expression is unsatisfiable. This short circuit allows us to save execution time. There are additional patterns that can be short-circuited as well.

We also introduce the idea of betting which value is more likely to occur. Towards this end we introduce a variety of tri-state operations that can be incorporated in the algorithm that "predict" a final state for a variable. Consider, for example, the pessimistic OR operation defined below. Here we are essentially predicting that, when a

variable is committed to its underlying T or F state, it will be False. We have no direct justifications for these assumptions. However, the mathematics in our hyperspace requires that we extend the OR, AND, and NOT operations fully and doing this requires and/or allows these types of operations. We utilize them, herein, because they add the possibility of further shortening our computational run time.

CHAPTER III

TRI-STATE LOGIC WITH COMMIT

Building Our Tri-State Logic with Commit

In this chapter we describe the tri-state logic with commit that we used to compute satisfiability to our simplified problem. Our tri-state logic approach is similar to the approach used in [Kullman, 2007] and [Frisch, Peugnieze].

In the tri-state logic used here, state D is semantically like being both True and False in the sense that we don't care what its truth value is.

In Table 3, we defined an Optimistic OR operator. Here, we assumed that from any OR operation that included a D State, the D would eventually become True. Table 4 presents the Pessimistic OR operation; where, from any D state, we refused to assume that one of those states would eventually become True. Note that the projection of both operations onto binary logic is idempotent – we are working in a hyperspace.

Table 3

Optimistic OR (OOR) with Assumptions Shown in Bold

Optimistic OR	F	T	D
F	F	T	**T**
T	T	T	T
D	**T**	T	**T**

Table 4

Pessimistic OR (POR) with Assumptions Shown in Bold

Pessimistic OR	F	T	D
F	F	T	**F**
T	T	T	T
D	**F**	T	**F**

Other hyper-OR definitions are possible. The above tables, for example, do not carry the D state forward. Our Irresolute OR (IOR) is an operation that does carry the D state forward. This is defined in Table 5.

Table 5

Irresolute OR (IOR) with Assumptions Shown in Bold

Irresolute OR	F	T	D
F	F	T	**F**
T	T	T	T
D	**F**	T	**D**

For the work here, we utilized an optimistic OR, where the table can be reduced to the two combinations that we would encounter during our simplified Boolean Satisfiability problem:

Not F (ie. T or D) V Not F (ie. T or D) -> T

F V F -> F.

We can also extend the binary AND to tri-state logic in many ways. Tables 6 and 7 define an Optimistic AND (OAND) and a Pessimistic AND (PAND).

Table 6

Optimistic AND (OAND) with Assumptions Shown in Bold

Optimistic AND	F	T	D
F	F	F	**F**
T	F	T	**T**
D	**F**	**T**	**T**

Table 7

Pessimistic AND (PAND) with Assumptions Shown in Bold

Pessimistic AND	F	T	D
F	F	F	**F**
T	F	T	**F**
D	**F**	**F**	**F**

We can also extend the NOT operation to tri-state logic. Here we have an idempotent third state, although other selections are possible.

NOT T = F,

NOT F = T,

NOT D = D.

A NOT operation becomes a no-op for the D state. Another possibility would be to switch from pessimistic to optimistic (or vice-versa) when encountering a NOT D. For

example, let DT and DF represent an optimistic and a pessimistic view of state D, respectively. Then !DT = DF, and !DF = DT.

Simplified Semantics

In this section of Chapter III we define the semantics of our simplified satisfiability problem examined in this paper. Computation proceeds from left to right. The input for the software program we developed is consumed left to right. Also, each prefix is linked to its immediate right adjacent neighbor. For example, a series of input OR's, A V B V C, processed from left to right, is equivalent to (A V B) ^ (B V C). A series of input AND's such as A ^ B ^ C is equivalent to (A ^ B) ^ (B ^ C).

Consider the following expression:

(A V B) ^ (B ^ C) ^ (C V D) ^ (D ^ E).

Now consider each parenthesized 2-tuple in our conjunctive and normalized formula as a unique node. This allows us to view the above expression as

(Node1) ^ (Node2) ^ (Node3) ^ (Node4).

(FVI. Our program would encode this expression in the input form A V B ^ C V D ^ E.) Our program would then formulate the following pairs:

Node 1:=	Var1 = A	Ops = Optimistic OR (OOR)	Var2 = B
Node 2:=	Var1 = B	Ops = Optimistic AND (OAND)	Var2 = C
Node 3:=	Var1 = C	Ops = Optimistic OR (OOR)	Var2 = D
Node 4:=	Var1 = D	Ops = Optimistic AND (OAND)	Var2 = E.

For our initial condition, all variables would begin in state D, the "don't care" state. Our algorithm would then proceed from left to right, stopping at each node to check if we could modify any of the variable's states. The 2-tuples are checked to

determine if one or both variables require a commitment to a particular state (once a variable is committed, it can never be changed) or if nothing requires attention. If we run into a node where the 2-tuple can not be made True, the algorithm halts its progress, explains where the expression possibly becomes unsatisfiable and exits. If the algorithm is able to step from node 1 to node n, then the expression is said to be possibly satisfiable.

Visualization

In this section we describe our binary "running satisfaction" visualization tool which we refer to as a ladder representation. Our illustration views our passage through the expression as successively larger prefix satisfiability expressions.

Consider the input string

A V B V C ^ D ^ B V D

which is equivalent to the Boolean expression

(A V B) ^ (B V C) ^ (C ^ D) ^ (D ^ B) ^ (B V D).

The semantics of this expression, broken down into our 2-tuple nodes, is

Node1:= A V B,

Node2:= B V C,

Node3:= C ^ D,

Node4:= D ^ B,

Node5:= B V D.

Where, again, the Optimistic AND (OAND) and Optimistic OR (OOR) are used.

Begin by assigning all variables the D state value. Our algorithm would produce the following values as it traversed from node to node, working left to right.

Table 8

Variable Values as the Algorithm Traverses the Nodes From Left to Right of the Input 'AvBvC^D^BvD'

Linear Looping	A	B	C	D
After Node = 1	D(T)	D(T)	D	D
After Node = 2	D(T)	D(T)	D(T)	D
After Node = 3	D(T)	D(T)	T*	T*
After Node = 4	D(T)	T*	T*	T*
After Node = 5	D(T)	T*	T*	T*

* indicates that the value for that variable has been committed

D(T) indicates that the value for that variable is state D and is Optimistic

The final answer obtained is:

A = Don't Care state (Optimistic = True),

B = True,

C = True,

D = True.

where B, C, and D have been committed to a specific value and A is still in our tri-nary state of "don't care / doesn't matter."

Now consider a second expression that contains a NOT operator:

A V B V C ^ D ^ NOT A

which is equivalent to the Boolean expression

(A V B) ^ (B V C) ^ (C ^ D) ^ (D ^ NOT A).

This has a ladder representation of:

Node1:= A V B

Node2:= B V C

Node3:= C ^ D

Node4:= D ^ NOT A

Table 9 shows how the above expression will be processed.

Table 9

Variable Values as the Algorithm Traverses the Nodes Left to Right of the Input 'AvBvC^D^NOT A'

Linear Looping	A	B	C	D
After Node = 1	D(T)	D(T)	D	D
After Node = 2	D(T)	D(T)	D(T)	D
After Node = 3	D(T)	D(T)	T*	T*
After Node = 4	F*	D(T)	T*	T*

* means the value for that variable has been committed

D(T) means the value for that variable is state D and is Optimistic

The final answer obtained is

A = False,

B = Don't Care state (Optimistic = True),

C = True,

D = True,

where A, C, and D have been committed to a specific value and B is still in our tri-nary state of "don't care / doesn't matter."

Input Processing

If the given expression is not initially linked (it does not have the linked form for all adjacent tuples), we must modify the expression before we can encode it as input for our algorithm.

For example, a nicely linked expression that would not need to be converted might be (A V B) ^ (B ^ C). A not nicely linked expression might be (A V B) ^ (C ^ D).

Expressions like (A V B) ^ (C ^ D) must be re-written using an artificial variable which we call E. E begins with a committed True value. When pre-processing is

complete, our encoded input file represents the 2CNF/SAT problem with the following linked structure:

(A V B) ^ (B V E) ^ (E V C) ^ (C ^ D).

We can map any 2CNF/SAT formula into our input encoding.

We can take any arbitrary 2CNF / 2CF expression and rewrite the expression into our algorithm's needed encoded input by adding these 'linking' variables that are initially committed the value True.

CHAPTER IV

ANALYSIS OF DATA

Analysis

We have taken the Boolean Logic of True and False and have added a third state D. We call it the "I don't care" or "It doesn't matter" state. We now have tri-state logic with {T, F, D}. We are not the only researchers utilizing tri-state logic for this problem [Fey et al., 2006], but there is not extensive literature on the subject. For example, we looked for previously defined operators to fit this application but found only those currently used in electronics. The definitions given here are our own.

We point out that after testing, a subset of input expressions was found that has the potential to yield a False Positive result. These expressions were of the form (A V B) ^...^ (~A ^ X) ^ ...^(~B ^ X) where variable X can be any variable that is not A or B. (This expression is not satisfiable; however, our algorithm claimed it is satisfiable).

Clearly, because the algorithm did not have a backtracking utility, expressions of the above form were not able to be evaluated correctly. However, we believe that future work may alleviate these subsets of problems by pre-modifying the expression into a valid form before running our SAT program.

Also, another subset of input expressions was found that could yield a False Positive result. These expressions were of the form (A V B) ^...^(~A V ~B) where the expression is in 2 Conjunctive Normal Form. (This expression is not satisfiable; however, our algorithm claimed it is satisfiable). Again, future work may be able to modify these expressions a-priori to alleviate this issue.

CHAPTER V

SUMMARY

Future Work

For future work we would like to examine other subsets of the 3CNF/SAT equations. For example, if we have the expression (3v1) ^ (1v~2) ^ (~2v4) ^ (4v1), we may be able to convert this to (1) ^ (~2v3v4) ^ (~2v4).

We noticed that when we encoded expressions in reverse order many of the False Positives were caught by the algorithm; however, it introduced a new subset of False Positives that were a mirror of the original expression's False Positives. For future work we would like to study the subsets of expressions satisfied by left to right and right to left processing to discover the structure of the expressions not addressable with only one pass, but addressable with left to right and right to left processing.

Third we would like to examine the orderings for our 2CNF to study when it is possible to compensate for backtracking with a better ordering. For example, (A V B) ^ (NOT B ^ C) ^ (NOT A ^ D) would give a False Positive result and thus would be beneficial to have backtracking; however, reordering the two-tuples as (NOT A ^ D) ^ (NOT B ^ C) ^ (A V B) would give the correct result of unsatisfiable and would not require backtracking.

For the present, however, we have presented a polynomial time algorithm that solves some SAT problems, and leaves you with additional information (the D state) whenever possible.

APPENDIX A

MYPROGRAM.CPP (CONTAINS MAIN)

```
/*************************************************************
        Author:     Kevin Byrd
        Purpose:    To attempt to build a data structure that runs in
                    Polynomial time and can determine
                    whether a Boolean formula is satisfiable.

        Usage:      argv[0] = myProgram.exe
                    argv[1] = file you wish to open
                            = defaults to equation.txt
                    argv[2] = user choice if PrintSpindle()
                              and PrintTable() functions are
                              called
                            = default is 0 (or True)
*************************************************************/

#include <iostream>
#include <iomanip>
#include <fstream>
#include <set>
#include <algorithm>
#include "spindle.h"

using namespace std;
```

```
int main(int argc, char* argv[])
{
	char *filename;
	int choice = 0;

	if(argc >= 2)
		filename = argv[1];
	else
		filename = "equation.txt";

	if(argc == 3)
		choice = atoi(argv[2]);

	ifstream inFile(filename);

	if(!inFile)
	{
		cout << "ERROR - cannot open myEquation.txt for read." << endl;
		cout << "Program will now terminate." << endl;
		return(0);
	}

	//Begin reading in data from file
	//We assume the file is in the CORRECT format
```

```
int point1, point2, total;
char ops;

inFile >> total;
inFile >> point1;
inFile >> ops;
inFile >> point2;

spindle *mySpin = new spindle(point1, ops, point2, total);

while(!inFile.eof())
{
        inFile >> point1;
        inFile >> ops;
        inFile >> point2;

        mySpin->Insert(point1, ops, point2);
}

inFile.close();

//Once the spindles are loaded into the spoke, we spin it
//I used spindles, spoke, and spinning to remind myself
//of how cloth threads were made back in the day and in a
//sense my algorithm takes the data and attempts to spin it
```

```
        //into a hopefully satisfiable outcome
        mySpin->Begin();

        mySpin->PrintSpindle(choice);
        mySpin->PrintTrTable(choice);

        return(true);
}
```

APPENDIX B

SPINDLE.CPP

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include "spindle.h"

using namespace std;

spindle::spindle(int p1, char op, int p2, int total)
{
        struct pNode *newNode = new pNode;

        if(p1 < 0)
        {
                newNode->point1 = p1 * -1;
                newNode->p1_isNot = true;
        }
        else
        {
                newNode->point1 = p1;
                newNode->p1_isNot = false;
        }

        if(p2 < 0)
```

```
{
        newNode->point2 = p2 * -1;
        newNode->p2_isNot = true;
}
else
{
        newNode->point2 = p2;
        newNode->p2_isNot = false;
}

newNode->andOr = op;
newNode->next = NULL;
newNode->previous = NULL;

head = newNode;

truth_Table = new point[total+1];
totalNumOfIndexes = total;

truth_Table[p1].value = false;
truth_Table[p1].is_both = false;
truth_Table[p1].is_set = false;

truth_Table[p2].value = false;
truth_Table[p2].is_both = false;
```

```
        truth_Table[p2].is_set = false;


        isSAT = true;
}

int spindle::Insert(int p1, char op, int p2)
{
        struct pNode *newNode = new pNode;


        if(p1 < 0)
        {
                newNode->point1 = p1 * -1;
                newNode->p1_isNot = true;
        }
        else
        {
                newNode->point1 = p1;
                newNode->p1_isNot = false;
        }


        if(p2 < 0)
        {
                newNode->point2 = p2 * -1;
                newNode->p2_isNot = true;
        }
```

```
else
{
        newNode->point2 = p2;
        newNode->p2_isNot = false;
}

newNode->andOr = op;
newNode->next = NULL;

struct pNode *current = head;

if(current != NULL)
{
        while(current->next != NULL)
                current = current->next;
}

current->next = newNode;
newNode->previous = current;

truth_Table[newNode->point1].value = false;
truth_Table[newNode->point1].is_both = false;
truth_Table[newNode->point1].is_set = false;

truth_Table[newNode->point2].value = false;
```

```
        truth_Table[newNode->point2].is_both = false;
        truth_Table[newNode->point2].is_set = false;

        return(true);
}

int spindle::Begin()
{
        struct pNode *current = head;

        if(current != NULL)
        {
                char currentOps = current->andOr;
                int p1 = current->point1;
                int p2 = current->point2;

                //If point1 is a NOT
                if(current->p1_isNot)
                {
                        truth_Table[p1].value = false;
                        truth_Table[p1].is_both = true;
                        truth_Table[p1].is_set = false;
                }
                //If point1
                else
```

```
{
        truth_Table[p1].value = true;
        truth_Table[p1].is_both = true;
        truth_Table[p1].is_set = false;
}

mySolution = new set<int>;

mySolution->insert(current->point1);

//We now check the initial condition of our pNode chain

//Our initial condition is an OR statement
if(currentOps == '*')
{
        //If point2 is a NOT
        if(current->p2_isNot)
        {
                if(!truth_Table[p2].is_set)
                {
                        truth_Table[p2].value = false;
                        truth_Table[p2].is_both = true;
                        truth_Table[p2].is_set = false;
                }
        }
```

```
//If point2
else
{
        if(!truth_Table[p2].is_set)
        {
                truth_Table[p2].value = true;
                truth_Table[p2].is_both = true;
                truth_Table[p2].is_set = false;
        }
}

        if(truth_Table[p1].value == true && current->p1_isNot &&
truth_Table[p2].value == true && current->p2_isNot)
        {
                cout << endl << "******************" << endl;
                cout << "No solution can be found - An OR statement
fails." << endl;
                cout << "(";
                if(current->p1_isNot)
                        cout << "~";
                cout << p1 << " " << currentOps << " ";
                if(current->p2_isNot)
                        cout << "~";
                cout << p2;
                cout << ")" << endl;
                cout << "******************" << endl << endl;
```

```
                    isSAT = false;
                    return(0);
                }
                else if(truth_Table[p1].value == false && !current->p1_isNot &&
truth_Table[p2].value == false && !current->p2_isNot)
                {
                    cout << endl << "*****************" << endl;
                    cout << "No solution can be found - An OR statement
fails." << endl;
                    cout << "(";
                    if(current->p1_isNot)
                        cout << "~";
                    cout << p1 << " " << currentOps << " ";
                    if(current->p2_isNot)
                        cout << "~";
                    cout << p2;
                    cout << ")" << endl;
                    cout << "*****************" << endl << endl;

                    isSAT = false;
                    return(0);
                }
                else
                    mySolution->insert(p2);
            }
```

```
//Our initial condition is an AND statement
else if(currentOps == '&')
{
    //If point2 is a NOT
    if(current->p2_isNot)
    {
        //If point2 has been used before and is_set and value is true
        if(truth_Table[p2].is_set == true && truth_Table[p2].value
== true)
        {
            cout << endl << "******************" << endl;
            cout << "No solution can be found - A point became
both true and false." << endl;
            cout << "(";
            if(current->p1_isNot)
                cout << "~";
            cout << p1 << " " << currentOps << " ";
            if(current->p2_isNot)
                cout << "~";
            cout << p2;
            cout << ")" << endl;
            cout << "******************" << endl << endl;

            isSAT = false;
            return(0);
        }
```

```
                    //If point2 can adapt to the AND statement
                    else
                    {
                        truth_Table[p2].value = false;
                        truth_Table[p2].is_both = false;
                        truth_Table[p2].is_set = true;
                    }
                }
                //If point2
                else
                {
                    //If point2 is set in stone as false but it is not negative
                    if(truth_Table[p2].value == false && truth_Table[p2].is_set
== true)
                    {
                        cout << endl << "******************" << endl;
                        cout << "No solution can be found - A point became
both true and false." << endl;
                        cout << "(";
                        if(current->p1_isNot)
                            cout << "~";
                        cout << p1 << " " << currentOps << " ";
                        if(current->p2_isNot)
                            cout << "~";
                        cout << p2;
                        cout << ")" << endl;
```

```
                cout << "******************" << endl << endl;


                isSAT = false;
                return(0);
            }
            else
            {
                truth_Table[p2].value = true;
                truth_Table[p2].is_both = false;
                truth_Table[p2].is_set = true;
            }
        }

        //If point1 is false and positive
        if(truth_Table[p1].value == false && !current->p1_isNot)
        {
            cout << endl << "******************" << endl;
            cout << "No solution can be found - An AND fails." <<
endl;
            cout << "(";
            if(current->p1_isNot)
                cout << "~";
            cout << p1 << " " << currentOps << " ";
            if(current->p2_isNot)
                cout << "~";
```

```
                cout << p2;
                cout << ")" << endl;
                cout << "******************" << endl << endl;

                isSAT = false;
                return(0);
        }
        //If point1 is true and negative
        else if(truth_Table[p1].value == true && current->p1_isNot)
        {
                cout << endl << "******************" << endl;
                cout << "No solution can be found - An AND fails." <<
endl;
                cout << "(";
                if(current->p1_isNot)
                        cout << "~";
                cout << p1 << " " << currentOps << " ";
                if(current->p2_isNot)
                        cout << "~";
                cout << p2;
                cout << ")" << endl;
                cout << "******************" << endl << endl;

                isSAT = false;
                return(0);
```

```
                }
                //Else we may have a solid AND statement!!
                else
                {
                        if(truth_Table[p1].is_both == true)
                        {
                                truth_Table[p1].is_both = false;
                                truth_Table[p1].is_set = true;
                        }

                        mySolution->insert(p2);
                }
        }
    }

    //If we have more spindles to cascade - we continue
    if(current->next != NULL)
        Continue(current->next);

    return(true);
}

int spindle::Continue(struct pNode *current)
{
    char currentOps = current->andOr;
```

```
int p1 = current->point1;
int p2 = current->point2;

//If our operation is an OR statement
if(currentOps == '*')
{
        //If point2 is a NOT
        if(current->p2_isNot)
        {
                //If our point2 value is not set in stone
                if(!truth_Table[p2].is_set)
                {
                        truth_Table[p2].value = false;
                        truth_Table[p2].is_both = true;
                        truth_Table[p2].is_set = false;
                }
        }
        //If point2
        else
        {
                //If our point2 value is not set in stone
                if(!truth_Table[p2].is_set)
                {
                        truth_Table[p2].value = true;
                        truth_Table[p2].is_both = true;
```

```
                truth_Table[p2].is_set = false;
            }
        }

        //Check for valid OR statement
        //If both points are true but both are negative
        if(truth_Table[p1].value == true && current->p1_isNot &&
truth_Table[p2].value == true && current->p2_isNot)
        {
            cout << endl << "******************" << endl;
            cout << "No solution can be found - An OR statement fails." <<
endl;
            cout << "(";
            if(current->p1_isNot)
                cout << "~";
            cout << p1 << " " << currentOps << " ";
            if(current->p2_isNot)
                cout << "~";
            cout << p2;
            cout << ")" << endl;
            cout << "******************" << endl << endl;

            isSAT = false;
            return(0);
        }
        //If both points are false and positive
```

```
            else if(truth_Table[p1].value == false && !current->p1_isNot &&
truth_Table[p2].value == false && !current->p2_isNot)
            {
                cout << endl << "******************" << endl;
                cout << "No solution can be found - An OR statement fails." <<
endl;
                cout << "(";
                if(current->p1_isNot)
                    cout << "~";
                cout << p1 << " " << currentOps << " ";
                if(current->p2_isNot)
                    cout << "~";
                cout << p2;
                cout << ")" << endl;
                cout << "******************" << endl << endl;

                isSAT = false;
                return(0);
            }
            //Else we may have a true OR statement
            else
                mySolution->insert(p2);

        }
        //Else our operation is an AND statement
        else if(currentOps == '&')
```

```
        {
                if(current->p2_isNot)
                {
                        //If point2 is set in stone and is true
                        if(truth_Table[p2].is_set == true && truth_Table[p2].value ==
true)
                        {
                                cout << endl << "*****************" << endl;
                                cout << "No solution can be found - An AND statement
fails." << endl;
                                cout << "(";
                                if(current->p1_isNot)
                                        cout << "~";
                                cout << p1 << " " << currentOps << " ";
                                if(current->p2_isNot)
                                        cout << "~";
                                cout << p2;
                                cout << ")" << endl;
                                cout << "*****************" << endl << endl;

                                isSAT = false;
                                return(0);
                        }
                        //Else point2 can adapt to the AND statement
                        else
                        {
```

```
                    if(truth_Table[p2].is_set == false)
                    {
                        truth_Table[p2].value = false;
                        truth_Table[p2].is_both = false;
                        truth_Table[p2].is_set = true;
                    }
                }
            }
            else
            {
                //If point2 is set in stone and is false
                if(truth_Table[p2].is_set == true && truth_Table[p2].value ==
false)
                {
                    cout << endl << "******************" << endl;
                    cout << "No solution can be found - An AND statement
fails." << endl;
                    cout << "(";
                    if(current->p1_isNot)
                        cout << "~";
                    cout << p1 << " " << currentOps << " ";
                    if(current->p2_isNot)
                        cout << "~";
                    cout << p2;
                    cout << ")" << endl;
                    cout << "******************" << endl << endl;
```

```
                    isSAT = false;
                    return(0);
                }
                //Else point2 can adapt to the AND statement
                else
                {
                    if(!truth_Table[p2].is_set)
                    {
                        truth_Table[p2].value = true;
                        truth_Table[p2].is_both = false;
                        truth_Table[p2].is_set = true;
                    }
                }
            }

            //Check for valid AND statement
            //If point1 is false and positive
                if(truth_Table[p1].value == false && !current->p1_isNot)
                {
                    cout << endl << "******************" << endl;
                    cout << "No solution can be found - An AND statement
fails." << endl;
                    cout << "(";
                    if(current->p1_isNot)
```

```
                cout << "~";
            cout << p1 << " " << currentOps << " ";
            if(current->p2_isNot)
                cout << "~";
            cout << p2;
            cout << ")" << endl;
            cout << "******************" << endl << endl;

            isSAT = false;
            return(0);
        }
        //If point1 is true and negative
        else if(truth_Table[p1].value == true && current->p1_isNot &&
truth_Table[p1].is_both == false)
        {
            cout << endl << "******************" << endl;
            cout << "No solution can be found - An AND statement
fails." << endl;
            cout << "(";
            if(current->p1_isNot)
                cout << "~";
            cout << p1 << " " << currentOps << " ";
            if(current->p2_isNot)
                cout << "~";
            cout << p2;
            cout << ")" << endl;
```

```
            cout << "******************" << endl << endl;


            isSAT = false;
            return(0);
    }
    //Else we may have a solid AND statement!!
    else
    {
            if(truth_Table[p1].is_both == true)
            {
                    if(current->p1_isNot)
                    {
                            truth_Table[p1].value = false;
                            truth_Table[p1].is_both = false;
                            truth_Table[p1].is_set = true;
                    }
                    else
                    {
                            truth_Table[p1].value = true;
                            truth_Table[p1].is_both = false;
                            truth_Table[p1].is_set = true;
                    }
            }

            mySolution->insert(p2);
```

```
                }
        }

        //If we have more spindles to cascade - we continue
        if(current->next != NULL)
                Continue(current->next);
        /*****************************************************************
                                CURRENTLY NOT NEEDED
        //Else we have reached the end - we check the satisfyability
        //else
        //      CheckSolution();
        *****************************************************************/

        return(1);
}

int spindle::TraceBack(pNode *current)
{
        if(current->previous != NULL)
                current = current->previous;

        if(current->andOr == '*')
        {
                if(truth_Table[current->point1].value = true && !current->p1_isNot)
                {
```

```
            cout << "TraceBack Complete - Found valid Or Statement at: "
                    << current->point1 << " with " << current->point2 << endl;
            if(truth_Table[current->point1].is_set == false)
                    truth_Table[current->point1].is_set = true;
            if(truth_Table[current->point1].is_both == true)
                    truth_Table[current->point1].is_both = false;
            return(1);
    }
    else if(truth_Table[current->point1].value = false && current->p1_isNot)
    {
            cout << "TraceBack Complete - Found valid Or Statement at: "
                    << current->point1 << " with " << current->point2 << endl;
            if(truth_Table[current->point1].is_set == false)
                    truth_Table[current->point1].is_set = true;
            if(truth_Table[current->point1].is_both == true)
                    truth_Table[current->point1].is_both = false;
            return(1);
    }
    else if(truth_Table[current->point2].value = true && !current->p2_isNot)
    {
            cout << "TraceBack Complete - Found valid Or Statement at: "
                    << current->point1 << " with " << current->point2 << endl;
            if(truth_Table[current->point2].is_set == false)
                    truth_Table[current->point2].is_set = true;
            if(truth_Table[current->point2].is_both == true)
```

```
                        truth_Table[current->point2].is_both = false;
                    return(1);
                }
                else if(truth_Table[current->point2].value = false && current->p2_isNot)
                {
                    cout << "TraceBack Complete - Found valid Or Statement at: "
                        << current->point1 << " with " << current->point2 << endl;
                    if(truth_Table[current->point2].is_set == false)
                        truth_Table[current->point2].is_set = true;
                    if(truth_Table[current->point2].is_both == true)
                        truth_Table[current->point2].is_both = false;
                    return(1);
                }
                else
                    TraceBack(current);
            }
            else
                TraceBack(current);

            return(1);
}

int spindle::TraceForward(pNode *current)
{
            if(current->next != NULL)
```

```
        current = current->next;


if(current->andOr == '*')
{
        if(truth_Table[current->point1].value = true && !current->p1_isNot)
        {
                cout << "TraceForward Complete - Found valid Or Statement at: "
                        << current->point1 << " with " << current->point2 << endl;
                if(truth_Table[current->point1].is_set == false)
                        truth_Table[current->point1].is_set = true;
                if(truth_Table[current->point1].is_both == true)
                        truth_Table[current->point1].is_both = false;
                return(1);
        }
        else if(truth_Table[current->point1].value = false && current->p1_isNot)
        {
                cout << "TraceForward Complete - Found valid Or Statement at: "
                        << current->point1 << " with " << current->point2 << endl;
                if(truth_Table[current->point1].is_set == false)
                        truth_Table[current->point1].is_set = true;
                if(truth_Table[current->point1].is_both == true)
                        truth_Table[current->point1].is_both = false;
                return(1);
        }
        else if(truth_Table[current->point2].value = true && !current->p2_isNot)
```

```
        {
                cout << "TraceForward Complete - Found valid Or Statement at: "
                        << current->point1 << " with " << current->point2 << endl;
                if(truth_Table[current->point2].is_set == false)
                        truth_Table[current->point2].is_set = true;
                if(truth_Table[current->point2].is_both == true)
                        truth_Table[current->point2].is_both = false;
                return(1);
        }
        else if(truth_Table[current->point2].value = false && current->p2_isNot)
        {
                cout << "TraceForward Complete - Found valid Or Statement at: "
                        << current->point1 << " with " << current->point2 << endl;
                if(truth_Table[current->point2].is_set == false)
                        truth_Table[current->point2].is_set = true;
                if(truth_Table[current->point2].is_both == true)
                        truth_Table[current->point2].is_both = false;
                return(1);
        }
        else
                TraceForward(current);
}
else
        TraceForward(current);
```

```
        return(1);
}

int spindle::CheckSolution()
{
        //Not implemented yet as I may have discovered that my algorithm
        //Doesn't even need this!!  Because if there are no kick outs from
        //The Begin() and Continue() functions, then the equation IS BOOLEAN
        //SATISFYABLE.

        return(1);
}

void spindle::PrintSpindle(int choice)
{
        if(choice == 1)
        {
                struct pNode *current = head;

                if(current != NULL)
                {
                        cout << endl << endl;

                        while(current->next != NULL)
                        {
```

```
        cout << "Point1 = ";
        if(current->p1_isNot)
                cout << " ~" << setw(2) << current->point1;
        else
                cout << setw(4) << current->point1;
        cout << "  Ops = " << current->andOr;
        cout << "  Point2 = ";
        if(current->p2_isNot)
                cout << " ~" << setw(2) << current->point2;
        else
                cout << setw(4) << current->point2;
        cout << endl;

        current = current->next;
}

cout << "Point1 = ";
if(current->p1_isNot)
        cout << " ~" << setw(2) << current->point1;
else
        cout << setw(4) << current->point1;
cout << "  Ops = " << current->andOr;
cout << "  Point2 = ";
if(current->p2_isNot)
        cout << " ~" << setw(2) << current->point2;
```

```
                else
                        cout << setw(4) << current->point2;
                cout << endl << endl;
        }
    }

}

void spindle::PrintTrTable(int choice)
{
    if(choice == 1)
    {
        cout << endl << endl;

        for(int x = 1; x < totalNumOfIndexes + 1; x++)
        {
            cout << setw(2) << x << " = ";
            if(truth_Table[x].is_both == true)
                cout << "True and False";
            else if(truth_Table[x].value == false)
                cout << "False";
            else if(truth_Table[x].value == true)
                cout << "True";
            cout << endl;
        }
```

```
            cout << endl;
        }

        cout << endl;
        if(isSAT)
            cout << "It was found to be Satisfiable";
        else
            cout << "It was found to be unSatisfiable";
        cout << endl;
}

spindle::~spindle()
{
}
```

APPENDIX C

SPINDLE.H

```
#ifndef SPINDLE_H
#define SPINDLE_H

#include <set>
#include <algorithm>

using namespace std;

//A point is simple a variable A,B,etc...
struct point
{
        bool value;
        bool is_both;
        bool is_set;
};

//A pNode is a spoke in the spindle
//It holds two points and an operation that
//Binds the two points together
struct pNode
{
        int point1;
        bool p1_isNot;
```

```
        char andOr;
        int point2;
        bool p2_isNot;

        struct pNode *next;
        struct pNode *previous;
};

//The main class that drives the programs goals
/*******************************
Insert() inserts a spoke/pNode into the spindle
Begin() starts the spinning
Continue() is used to continue spinning
TraceBack() and Forward() are not used
CheckSolution() is not needed nor used
PrintSpindle() and TrTable() (Truth Table) are for output

I use a head pointer for my spokes/pNodes
I have a dynamic table of points (variables) that
are index for Big O(1) speed!!
I keep track of my indexes with totalNumOfIndexes
I use myIter because I use a set of integers saved into a
solution set.
Of note, mySolution isn't necessarily needed at this time.
*******************************/
```

```
class spindle
{
public:
        spindle(int, char, int, int);
        int Insert(int, char, int);
        int Begin();
        int Continue(struct pNode *);
        int TraceBack(struct pNode *);
        int TraceForward(struct pNode *);
        int CheckSolution();
        void PrintSpindle(int);
        void PrintTrTable(int);
        ~spindle();

        struct pNode *head;
        struct point *truth_Table;
        int totalNumOfIndexes;
        bool isSAT;

        set<int> *mySolution;
        set<int>::iterator myIter;
};

#endif
```

BIBLIOGRAPHY

Alekhnovich, Michael; Ben-Sasson, Eli; Razborov, Alexander A.; Wigderson, Avi. Pseudorandom Generators in Propositional Proof Complexity. Supported by INTAS grant #96-753. Moscow State University. August 4, 2003.

Deolalikar, Vinay. P <> NP. HP Research Labs, Palo Alto. August 11, 2010.

Fey, Goerschwin, Junhao Shi, and Rolf Drechsler, Efficiency of Multi-Valued Encoding in SAT-based ATPG, 36th International Symposium on Multiple-Valued Logic (ISMVL), 2006.

Frisch, Alan M.; Peugnieze, Timothy J. Solving Non-Boolean Satisfiability Problems with Stochastic Local Search. University of York. <http://www.cse.wustl.edu/~zhang/teaching/cs518/frisch.ps>

Kullmann, Oliver. An algorithmic platform for efficient satisfiability-based problem solving. Final Report on EPSRC grant GR/S58393/01. Swansea University. September 12, 2007.

Linz, Peter. An Introduction to Formal Languages and Automata, 4th Ed., Jones and Bartlett, Sudbury, MA, 2006.

Reichenbach, Hans. Elements of Symbolic Logic. New York: The Free Press, 1947.

Wikipedia. "Boolean Logic." Last Modified: Nov 23, 2010. <http://en.wikipedia.org/wiki/Boolean_logic>

Wikipedia. "Boolean Satisfiability Problem." Last Modified: Nov 20, 2010. <http://en.wikipedia.org/wiki/Boolean_satisfiability_problem>