Impact of the LZW-based Common Subexpression Elimination Algorithm on SAT-solving Efficiency

by
Jeriah Jn. Charles, B.S., B.A.

A Thesis

In

COMPUTER SCIENCE
Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

Approved

Dr. Yuanlin Zhang
Chair of Committee

Dr. Michael Gelfond

Dr. Richard Watson

Peggy Gordon Miller
Dean of the Graduate School

May, 2012

Copyright 2012, Jeriah Jn. Charles

ACKNOWLEDGMENTS

I would like to express my deepest gratitude and thanks to my advisor Dr. Yuanlin Zhang for his continuous support and guidance in the preparation and completion of my research paper. His invaluable knowledge inspired and motivated me to work diligently towards the completion of this thesis. I am also grateful to Dr. Michael Gelfond and Dr. Richard Watson for their valuable assistance in providing feedback in the development of this research paper. I wish to express special thanks to the KRLab members for their support during my research.

Special thanks to my parents Mr. Craig and Mrs. Mary Anna Jn. Charles, my sister Miah Jn. Charles, and my brothers Rohn and Daryl Jn. Charles for their unending support, care, and love. Their encouragement and guidance have made a difference in my educational attainment, especially during my master's research effort. I would also like to express special thanks to Justin Blount for his encouragement and support during my research.

TABLE OF CONTENTS

ACKNOWLEDGMENTS .... **ii**

ABSTRACT .... **v**

LIST OF TABLES .... **vi**

LIST OF FIGURES .... **vii**

I. INTRODUCTION **1**

II. PRELIMINARIES **4**

2.1 SAT Problem 4

2.2 SAT Solver 4

2.2.1 Davis-Putnam-Logemann-Loveland (DPLL) Algorithm 5

2.2.2 DIMACS CNF SAT File 6

III. RELATED WORK **7**

3.1 Common Subexpression Elimination and Declarative Programming 7

3.1.1 CSE During Flattening 7

3.1.2 Numerical Constraint Satisfaction Problems (CSPs) 8

3.1.3 Quantified Boolean Formulas (QBFs) 9

3.2 The Smallest Grammar Problem 9

3.3 Direct Acyclic Graphs in Compiler Optimization 10

3.4 Data Compression 11

3.4.1 Data Compression via Textual Substitution 11

3.4.2 Lempel–Ziv (LZ) Family of Data Compression Algorithms 12

3.5 Summary 14

IV. COMMON SUBEXPRESSION ELIMINATION IN SAT **15**

4.1 Example of CSE in SAT 15

4.2 Proposed CSE Method 16

4.3 Summary 16

V. LZW-BASED CSE APPROACH **17**

5.1 LZW Algorithm 17

5.1.1 Dictionary Structure 19

5.1.2 Dictionary Search 20

5.2 LZW-based CSE Algorithm 20
5.2.1 Dictionary 21
5.2.2 Algorithm 22

5.3 Summary 24

VI. EMPIRICAL STUDY 25

6.1 Benchmark Suite 25

6.2 Environment 27

6.3 Evaluation Metric 27

6.4 Experimental Results 28
6.4.1 Number of Variables/Clauses 30
6.4.2 Number of Decisions/Propagations 31
6.4.3 Satisfiabilty 33
6.4.4 CPU Time 34

6.5 Comprehensive Analysis 35

6.6 Limitations 36

6.7 Summary 36

VII. CONCLUSION 37
7.1 Summary 37

7.2 Future Work 38

BIBLIOGRAPHY 39

APPENDICES
A. RESULTS 41

ABSTRACT

The Satisfiability (SAT) problem is the problem of finding an assignment that satisfies a given propositional formula. SAT is effective in solving many important problems in areas such as automated reasoning, computer-aided design, and planning in Artificial Intelligence. The need to solve these problems in a reduced amount of time has geared considerable research in improving the performance of SAT solvers resulting in many solver algorithms being created or modified.

This research investigates how the removal of common subexpressions in a formula via the Lempel–Ziv-Welch (LZW)-based approach can affect the efficiency of SAT solving. By substituting common subexpressions for new variables in the original formula, we compare the results of passing the original formula and the new equivalent formula through a SAT solver. In this LZW-based approach, we modify the Lempel–Ziv-Welch data compression algorithm to find and substitute the common subexpressions in the formula.

LIST OF TABLES

3.1 Flattening and CSE……………………………………………………………...8

3.2 LZ77/LZ78 Data Compression Example………………………………………13

3.3 LZW Data Compression Example……………………………………………..14

5.1 LZW Example………………………………………………………………….19

6.1 Benchmarks…………………………………………………………………….26

6.2 Instances for Analysis…………………………………………………………28

6.3 Satisfiability……………………………………………………………………33

6.4 Fast New Instances……………………………………………………………..35

A.1 Application Instances…………………………………………………………41

A.2 Crafted Instances………………………………………………………………43

A.3 Random Instances……………………………………………………………..45

LIST OF FIGURES

2.1 Example DIMACS CNF SAT file……………………………………………....5

3.1 DAG for example block……………………………………………..................11

4.1 Resulting equivalent formula………………………………………….........15

5.1 LZW algorithm…………………………………………………………………...18

5.2 LZW-based algorithm………………………………………………………….23

6.1 Number of Variables…………………………………………………………...30

6.2 Number of Clauses……………………………………………………………..30

6.4 Number of Variables in new/Number of Variables in orig……………………….31

6.5 Number of Decisions…………………………………………………………..32

6.6 Number of Propagations……………………………………………………….32

6.7 CPU Time……………………………………………………………………...34

CHAPTER I

INTRODUCTION

The Satisfiability (SAT) problem is a well-known and intensively studied NP-complete problem in Computer Science and Artificial Intelligence. In a SAT problem, a propositional formula is given and the task is to find an assignment of its Boolean variables that satisfies the formula [20]. If such an assignment to a formula does not exist, it is unsatisfiable; otherwise, it is satisfiable.

Many important problems in various areas, such as automated reasoning, planning in Artificial Intelligence, and software and hardware verification, can be encoded into SAT instances and solved using a SAT solver [5]. Decreasing SAT solving time is important in these application problems. The need to solve these problems in a reduced amount of time has geared considerable research in improving the efficiency of SAT solving resulting in many solver algorithms being created or modified [20]. In addition, there is an annual SAT competition where persons compete to develop the most efficient SAT solver [18].

In this research, we examine the concept of Common Subexpression Elimination (CSE) in SAT, present a proposed method for this notion, and analyze this method on SAT solving efficiency. Common subexpressions (CS's) are subexpressions that are either syntactically or semantically equivalent [10]. Syntactically equivalent expressions are written the same way and semantically equivalent clauses have the same meaning. For example, in the expression $((a + b) - (a + b + c) * (b + a))$, the two occurrences of $a + b$ are syntactically equivalent and the subexpressions $a + b$ and $b + a$ are semantically equivalent. Another set of semantically equivalent subexpressions are $a*(b+c)$ and $a*b + a*c$.

Common Subexpression Elimination is the process of finding common subexpressions and replacing them with new variables. In the first example, after replacing the common subexpression $a+b$ by variable d, the expression becomes $(d - (d+c) * d)$. CSE has been used in many domains such as declarative programming. It is a

commonly used technique in compiler optimization to improve the efficiency of generated code [1].

In the case of the SAT problem, a subclause within a SAT formula can be thought of as a subexpression. We observe that many SAT instances contain common subclauses. The result of replacing these subclauses with new variables may have a positive impact on SAT solving. It may result in more propagation and thus, a reduced search space. Consider the formula: $\{\{A, B, C, D\}, \{A, C, E\}\}$. Replacing the common subexpression $\{A, C\}$ with a new variable X, we get new clauses $\{\{X, B, D\}, \{X, E\}\}$ and those resulting from $X \leftrightarrow \{A, C\}$. If variable E is first evaluated and assigned false, there is no unit propagation in the original formula; however, in the new formula, X evaluates to true after unit propagation.

In this paper, we study one novel approach to CSE based on the Lempel–Ziv-Welch (LZW) data compression algorithm which is the LZW-based approach, and investigate its impact on the efficiency of SAT solving. The LZW data compression has been efficient in reducing file size through many applications such as the UNIX compress program and the Graphics Interchange Format (GIF) [11]. Due to its effectiveness, we use this algorithm and apply it to CSE in SAT.

For the LZW-based approach, we look at common subclauses as those that are syntactically equivalent. We process each clause as a string; thus, the order of the variables within each clause of the formula matters. For example, the formula $\{\{A, B, C, D\}, \{A, B, D, E\}\}$ has only one common subclause: $\{A, B\}$. We expect to find that this algorithm improves SAT solving time for some particular formulas.

SAT is very important in the field of Artificial Intelligence. By using CSE to improve solving time, we may be able to solve more application problems encoded into SAT instances. So we would rely not only on the SAT solver for generating answers quickly but also on Common Subexpression Elimination in the SAT formula.

This chapter provided a brief introduction to the SAT Problem, the Common Subexpression Elimination notion, and the proposed CSE method, the LZW-based

approach. The rest of this paper is organized as follows. In Chapter 2, we provide the preliminaries needed to understand the SAT Problem. Chapter 3 discusses the work related to both CSE and the proposed LZW-based approach. Chapter 4 gives an insight into the concept of Common Subexpression Elimination in the context of SAT and the proposed method. In Chapter 5, we provide a detailed explanation of the LZW data compression algorithm and the LZW-based approach. Chapter 6 provides an empirical study on the impact of the LZW-based approach on SAT solving. It describes the environment and benchmarks that were used for the experiments conducted for this research and gives an analysis of the results. Chapter 7 provides the conclusion of this research effort.

CHAPTER II

PRELIMINARIES

This chapter provides an introduction to propositional calculus and the SAT problem needed for understanding the Common Subexpression Elimination notion in SAT and the Lempel-Ziv-Welch (LZW)-based approach.

Boolean variables range over the domain of truth values {true, false} [5, 8]. A *literal* is a Boolean variable or its negation: x, $\neg x$. x is a positive literal and is satisfied when x is assigned true. $\neg x$ is a negative literal and is satisfied when x is assigned false. A *clause* is a disjunction of literals. For example, clause $(\neg p \vee q \vee r)$ is denoted by $\{\neg p, q, r\}$. A clause is satisfied when one or more of its literals evaluate to true. A *conjunctive normal form* (*CNF*) *formula* is a conjunction of clauses. For example, formula $(p \wedge (\neg q \vee r))$ is denoted by $\{\{p\}, \{\neg q, r\}\}$. A CNF formula is satisfied when each of its clauses evaluate to true. An *assignment* is a consistent set of literals, i.e., one that does not contain both p and $\neg p$ for any variable p. It is a function that assigns a unique truth value to each variable in a formula. The formula $\{\{\neg p\}, \{q, r\}\}$ has an assignment $\{p, q, \neg r\}$, i.e., $\{p$=T, q=T, r=F$\}$ where T is true and F is false.

2.1 SAT Problem

Given a set of n Boolean variables $x_1, \ldots, x_n$, a set of literals which are either x_i or $\neg x_i$ for some i, and a formula, a set of e distinct clauses $C_1, \ldots, C_e$, the SAT problem is to find an assignment of the variables such that every clause C_k $(1 \leq k \leq e)$ is evaluated to be true. If such an assignment is found, the formula is satisfiable; otherwise, it is unsatisfiable. For example, the formula $\{\{\neg p\}, \{q, r\}\}$ is satisfiable and has an assignment $\{\neg p, q, \neg r\}$. However, the assignment $\{p, q, \neg r\}$ does not satisfy this formula. In addition, the formula $\{\{p, \neg p\}\}$ is unsatisfiable.

2.2 SAT Solver

A SAT solver is a program that can determine whether a given SAT formula is satisfiable. Modern SAT solvers use the Davis-Putnam-Logemann-Loveland (DPLL) algorithm which is mainly characterized by branching, backtracking, and unit propagation [15]. This algorithm was introduced in 1962 by Martin Davis, Hilary

Putnam, George Logemann, and Donald W. Loveland. The format of the input for this algorithm is the DIMACS CNF file format.

2.2.1 Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

The DPLL algorithm is a depth-first search algorithm where each variable of a given formula is assigned a truth value recursively [5]. Branching in the algorithm generates a binary tree. In branching, an unassigned literal of the formula is selected and assigned a true value or false value. Under this assignment, another unassigned literal is assigned to be true or false. This branching continues until either one of two cases is reached: (i) the formula is unsatisfiable under this assignment; in this case, the solver recursively backtracks to the previous literal, evaluates it with the opposite truth value if this case has not been evaluated, and continues the branching mechanism, or (ii) the formula is satisfiable under this assignment; in this case, the solver outputs SAT and the assignment that was produced. If all possible assignments have been evaluated and there is no assignment that makes the formula true, then the solver outputs UNSAT.

Another important component of the DPLL algorithm that improves its efficiency is unit propagation. In unit propagation, if all but one literal of a clause has false values, then this one literal is assigned true in order to maintain the satisfiability of the clause [15]. A literal is chosen and assigned a truth value each time as the algorithm progresses and is propagated through the rest of the clauses. If the literal s is chosen and is assigned true, then we delete clauses containing s and delete $\neg s$ from other clauses. For example, if we have the set of clauses $\{\{p,q,s\}, \{q,\neg s\}, \{p\}\}$, then choosing literal s with a true value we get $\{\{q\}, \{p\}\}$ after unit propagation.

An example SAT solver is MiniSat. It is a free, open-source conflict-driven SAT Solver which was developed by Niklas Een and Niklas Sorensson in 2005 [15]. When a SAT instance is ran with MiniSat, MiniSat outputs not only SAT or UNSAT, but also the number of variables, clauses, propagations, decisions, conflicts, conflict literals, and restarts, the CPU time, and the amount of memory used. Selecting and setting the truth value of an unassigned literal is a *decision*. A *conflict* occurs when a variable occurs both positively and negatively in the search tree, such as x and $\neg x$. A *conflict literal* is a literal

which causes a conflict in the search tree. In a *restart*, the algorithm stops the search and restarts the branching mechanism from the beginning, keeping a record of the searched tree. This occurs when the algorithm reaches a difficult search region [20].

2.2.2 DIMACS CNF SAT File

The input to a SAT solver is a SAT formula usually in the DIMACS CNF file format [13]. The first set of lines may contain comments that begin with the character `c` followed by the line `p cnf #variables #clauses` where `#variables` is the number of variables and `#clauses` is the number of clauses in the formula. Each other line represents a clause with a trailing 0. Each positive literal is represented by its corresponding positive integer and each negative literal is represented by its corresponding negative integer. In Figure 2.1, the formula $\{\{A,\neg B,C\}, \{\neg A,C\}, \{B,C,D\}\}$ is represented. Positive literal *A* is denoted by *1*, negative literal $\neg B$ is denoted by *-2*, and so on. The file contains the line `p cnf 4 3` since the formula contains 4 variables *A*, *B*, *C*, and *D* and contains 3 clauses.

```
c comments here
c Example Formula:
c {{A,¬B,C},{¬A,C},{B,C,D}}
p cnf 4 3
1 -2 3 0
-1 3 0
2 3 4 0
```

Figure 2.1: Example DIMACS CNF SAT file

CHAPTER III

RELATED WORK

The Common Subexpression Elimination concept has been exploited in a variety of domains. Moreover, there exist problems in the Computer Science field that utilize a similar concept to CSE. In this chapter, we provide an overview of some of these problems.

3.1 Common Subexpression Elimination and Declarative Programming

3.1.1 CSE during Flattening

In the constraint programming community, we usually need to decompose more complex constraint expressions to simpler ones so that they are supported by existing solvers. This decomposition is called *flattening* [9]. It can introduce new variables. For example, the constraint expression $a+b+c \neq e*f$ is flattened for the Gecode solver as follows:

$$aux_1 = e*f$$
$$a+b+c \neq aux_1$$

The flattened subexpression is $e*f$ and the new variable introduced is aux_1.

Rendl et al. [10] propose using CSE during the flattening process. They note that common subexpressions exist in many constraint models. They show that CSE during flattening can reduce the size of the model and lead to a reduced search space. They incorporate CSE in the flattening process by recording each flattened subexpression and corresponding new variable in a hashmap. For each new subexpression that is to be flattened, they search in the hashmap for an equivalent one. If one is found, then the subexpression is replaced by the corresponding variable found in the hashmap. They provide the Table 3.1 as an illustration.

Table 3.1 Flattening and CSE

Unflattened	Flattened with CSE	Standard Flattening
$a + x * y = b$ $b + x * y = t$	$aux_1 = x * y$ $a + aux_1 = b$ $b + aux_1 = t$	$aux_1 = x * y$ $a + aux_1 = b$ $aux_2 = x * y$ $b + aux_2 = t$

Their results show that this approach can reduce solving time and flattening time in constraint solving and that additional propagation can be obtained by reusing the new variables.

3.1.2 Numerical Constraint Satisfaction Problems (CSPs)

Araya et al. [2] observed that in numerical Constraint Satisfaction Problems, replacing the common subexpressions in constraints can help reduce the number of operations in constraint propagation and result in more constraint propagation. They define a common subexpression as a numerical expression that occurs several times in one or several constraints. They present an algorithm that exploits CSE in interval analysis, in systems of equations ranging over real values.

The tree encoding a given system of equations is restructured so that common subexpressions are represented by nodes with at least two parents. Two nodes with common children and a common operator are detected as common subexpressions. These common nodes are merged in the reorganization of the tree. From this new tree, a new system of equations is constructed where each common subexpression f is replaced by its corresponding variable v and the new constraints $v = f$ corresponding to these subexpressions are included. For example, the following system of equations

$$x^2 + y + (y + x^2 + y^3 - 1)^3 + x^3 = 2$$

$$\frac{(x^2 + y^3)(x^2 + \cos(y)) + 14}{x^2 + \cos(y)} = 8$$

is transformed into the following new system after applying their CSE algorithm where the new variables are v_1, v_2, v_3, v_4,and v_5:

$v_2 + (v_3)^3 + x^3 - 2 = 0$	$v_1 = x^2$	$v_3 = -1 + y + v_4$
$\frac{v_4 \text{ x } v_5 + 14}{v_5} - 8 = 0$	$v_2 = y + v_1$	$v_4 = v_1 + y^3$
	$v_3 = v_2 + y^3 - 1$	$v_5 = v_1 + \cos(y)$

3.1.3 Quantified Boolean Formulas (QBFs)

Shlyakhter et al. [14] describe a technique for detecting and sharing common subformulas in Quantified Boolean formulas which are statements such as $\forall x P(x)$. They claim that this technique makes grounding out faster and creates smaller CNF formulas that are faster to solve.

First, they represent the QBF in an abstract constraint syntax tree. They then classify nodes of the tree whose ground forms may contain common subformulas. During grounding out, if the ground form of the current node matches an existing ground form in its class, then this common ground form will be shared among the current node and the node(s) specified in its class; thus, the nodes will have a common subtree. If there is no match, the ground form of the node is computed and recorded. The resulting tree is a directed acyclic graph (DAG). A CNF formula is then generated from the DAG.

Using a variety of benchmarks, the authors show that this generated formula is solved easier and faster. For the intractable benchmarks, the new generated CNF formula was easily solved.

3.2 The Smallest Grammar Problem

In their paper, Charikar et al. [3] propose an approximation algorithm for the smallest grammar problem. They define the size of a grammar to be the total number of symbols on the right sides of its production rules. The smallest grammar problem is to find the smallest context-free grammar that generates a given string. For example, the smallest context-free grammar that generates the string 'a rose is a rose is a rose' is given as follows.

S → BBA

A → a rose

B → A_is_

The size of this grammar is 14. Their approximation algorithm finds a grammar that generates the given string and whose length is not much larger than the length of the smallest grammar for that string. This $O(\log(n/m*))$ approximation, where n is the length of the given string and m is the length of the smallest grammar generating that string, uses a variant of the LZ77 data compression algorithm which transforms the string into a sequence of terminals (characters in the string) and pairs that represent substrings. This sequence is then translated into a grammar using one or more of the operations that the authors describe: AddPair, AddSequence, and AddSubstring.

As it applies to CSE, we can view nonterminal symbols as new variables and their corresponding sequence of symbols as a potential common subexpression. In the previous example, nonterminal symbol B represents a new variable with common subexpression 'A_is_.' The approximation algorithm, however, was not able to efficiently reduce the SAT formula.

3.3 Direct Acyclic Graphs in Compiler Optimization

In their 'Compilers' book, Aho et al. [1] explain the technique of constructing a Direct Acyclic Graph (DAG) for a block of code in which common subexpressions are eliminated. This technique is used in the intermediate code generation of the compiler front end. The leaves of the DAG represent atomic operands ands its inner nodes represent operators. Unlike the syntax tree for an expression, a DAG has shared subtrees representing common subexpressions. A node in the DAG has more than one parent if it represents a common subexpression. Aho et al. provide following steps on constructing the DAG for a basic block of expressions:

1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.

3. Node *N* is labeled by the operator applied at *s,* and also attached to *N* is the list of variables for which it is the last definition within the block.
4. Certain nodes are designated *output nodes.* These are the nodes whose variables are *live on exit* from the block; that is, their values may be used later, in another block of the flow graph. The flow graph displays the flow of control among blocks.

They present the following block as an example with its corresponding DAG in Figure 3.1.

a = b + c
b = a - d
c = b + c
d = a – d

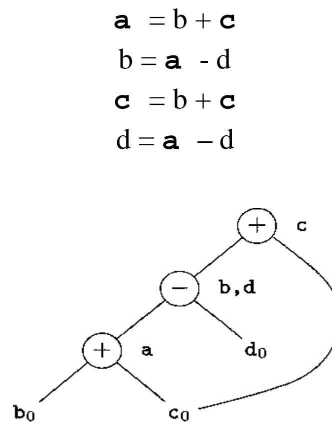

Figure 3.1 DAG for example block

3.4 Data Compression

Data compression exploits the existence of redundant characters and substrings in data. It involves the use of common subexpressions. Many algorithms have been developed to efficiently compress data, reducing it to hopefully fewer bits [11]. These algorithms are useful in many applications such as file sharing, video streaming, and image compression. In this area, we have studied the following algorithms.

3.4.1 Data Compression via Textual Substitution

In their paper, Storer et al. [16] introduce four models of compressing given strings. These models identify common substrings and replace them by a pointer which is a pair of integers *(a,b)* where *a* is the position of the first character of the substring in the original string that it represents and *b* is the length of that substring. For example, in the

internal model, the string *aaBccDaacEaccFacac* is compressed to *aaBccD(1,2)cEa(4,2)Fac(13,2)*, obtaining a compression of $^{15}/_{18}$, where the pointer *(1,2)* represents the substring *aa*, *(4,2)* represents the substring *cc*, and *(13,2)* represents the substring *ac*.

3.4.2. Lempel–Ziv (LZ) Family of Data Compression Algorithms

In the Lempel–Ziv family of data compression algorithms, we examined the LZ77, LZ78, and LZW algorithms. The LZ77 and LZ78 algorithms are two lossless compression algorithms that were developed by Abraham Lempel and Jacob Ziv and the Lempel–Ziv-Welch (LZW) algorithm is a variant of the LZ78 algorithm published by Terry Welch [19]. These algorithms have been employed in many applications such as the compression of GIF images, zip, and gzip files. In these three algorithms, a data structure, the dictionary, is used to reference substrings of the given string to be compressed.

3.4.2.1 LZ77 Algorithm

The LZ77 algorithm compresses a string into a sequence of 3-tuples [6]. It searches backwards in the earlier processed part of the string for the longest substring that matches the prefix of the current input. If there is a match, it outputs the 3-tuple *(i,j,c)* where *i* is the number of characters from the beginning of the substring that precedes the current input, *j* is the length of the substring, and *c* is the next non-matching character in the current input. Otherwise, it outputs *(0,0,c)* where *c* is the current input character. In both cases, the current input now becomes the unprocessed part of the string and the algorithm repeats. The earlier processed part of the string in which the algorithm searches is used as the dictionary.

For example, part of the compression of the string *abracadabra* is as follows. The symbol | separates the earlier processed part of the string and the current input.

|*abracadabra*: search for *a* is unsuccessful, output (0,0,a)

a|*bracadabra*: search for *b* is unsuccessful, output (0,0,b)

ab|*racadabra*: search for *r* is unsuccessful, output (0,0,r)

abr|acadabra: search for *a* is successful, search for *ac* is unsuccessful, output (3,1,c)

abrac|adabra: search for *a* is successful, search for *ad* is unsuccessful, output (2,1,d) …

3.4.2.2 LZ78 Algorithm

The LZ78 algorithm compresses a string into a sequence of pairs [3]. It searches for the shortest, nonempty prefix of the current input string that is not in the dictionary. If the prefix is a single character *c*, it outputs *(0,c)* and adds *c* to the dictionary. Otherwise, the prefix represents *αc* where *α* is a substring already in the dictionary and *c* is a single character. The algorithm then outputs *(i,c)* where *i* is the dictionary index of *α*. The current input now becomes the unprocessed part of the string and the algorithm repeats.

As an example for the LZ77 and LZ78 algorithms, the string *abracadabra* is compressed in Table 3.2.

Table 3.2 LZ77/LZ78 Data Compression Example

LZ77		LZ78		
Dictionary	Output	Dictionary		Output
	(0,0,a)	0	null	
a	(0,0,b)	1	a	(0,a)
ab	(0,0,r)	2	b	(0,b)
abr	(3,1,c)	3	r	(0,r)
abrac	(2,1,d)	4	ac	(1,c)
abracad	(7,4,")	5	ad	(1,d)
		6	ab	(1,b)
		7	ra	(3,a)

3.4.2.3 LZW Algorithm

One variant of the LZ78 algorithm is the Lempel–Ziv-Welch (LZW) compression algorithm. The dictionary is first initialized to all possible ASCII and Extended ASCII (256) characters and their corresponding codes in the range 0-225 [7]. The next available codes begin from 256 and represent substrings containing at least two characters. The algorithm finds the longest prefix of the current input that matches a string in the dictionary, then outputs the code *c* for this prefix and creates a record in the dictionary

containing *c*, the code for the next input character *p*, and the next available code. The current input then becomes the rest of the string beginning from *p*. The resulting compressed string is a string of codes. For example, the string *aabbababaabb* is compressed in Table 3.3.

Table 3.3 LZW Data Compression Example

Dictionary			Output
Code	Substring		
97	97	a	
98	97	b	
…	…		
256	97 97	aa	97
257	97 98	ab	97
258	98 98	bb	98
259	98 97	ba	98
260	257 97	aba	257
261	260 97	abaa	260
262	257 98	abb	257
			98

3.5 Summary

This chapter discussed the work related to the Common Subexpression Elimination concept. This work included problems in the declarative programming paradigm, the Smallest Grammar Problem, Direct Acyclic Graphs in compiler optimization, and data compression. The next chapter provides a detailed discussion of CSE in context to the SAT problem.

CHAPTER IV

COMMON SUBEXPRESSION ELIMINATION IN SAT

In this chapter, we discuss the Common Subexpression Elimination notion, provide a brief context to the SAT problem, and underline the proposed Lempel-Ziv-Welch (LZW)-based method.

A subclause F, $\{x_1, \ldots, x_n\}$ $(n > 1)$, is a common subexpression of a set of clauses $C_1,\ldots,C_k$ $(k > 1)$ if there exists $i, j \in [1,k]$, $i \neq j$, such that $F \subseteq C_i$ and $F \subseteq C_j$. The frequency of a subclause F appearing in a set S of clauses is the number of clauses of S that are a superset of F. We would like to replace every common subexpression F by a new variable, with the addition of clauses on the equivalence of the new variable and F.

Given a SAT problem P with clauses $C_1,\ldots,C_k$ $(k > 1)$ and a common subexpression F of these clauses, the resulting SAT problem of eliminating F is: $C \cup C' \cup E$ where C is the set of clauses of P that are not a superset of F, $C' = \{C \mid \exists i \in [1,k], F \subseteq C_i, C = (C_i - F) \cup \{X\}$ where X is a new Boolean variable$\}$, and $E = \{X \leftrightarrow F \mid X$ is a new variable in $C'\}$.

4.1 Example of CSE in SAT

The formula $\{\{A, B\}, \{A, B, C\}, \{A, C\}\}$ contains common subexpression, $\{A, B\}$. Substituting $\{A, B\}$ with a new variable D, we get $\{\{D\}, \{D,C\}, \{A,C\}\}$ and add $D \leftrightarrow \{A,B\}$ to form a new equivalent formula. Using this formula to illustrate, we give a brief overview of how a new equivalent SAT formula is created from a given SAT formula. Figure 4.1 depicts the CSE method for this original formula.

Original		New	
1 2		4	
1 2 3	→	4 3	
1 3		1 3	
		-4 1 2	} 4 ↔ 1 2
		-1 4	
		-2 4	

Figure 4.1: Resulting equivalent formula

As mentioned earlier, variable D (*4* in this case) replaces common subexpression $\{A,B\}$ (*1 2* in this case) and the equivalence clauses denoting $D \leftrightarrow \{A,B\}$are added to the new formula. Since a CNF formula is a conjunction of clauses, we cannot explicitly add $D \leftrightarrow \{A,B\}$ to it; however, we can add its equivalent CNF representation as denoted by the last three clauses of the new formula in Figure 4.1: $\{\{\neg D,A,B\},\{\neg A,D\},\{\neg B,D\}\}$.

4.2 Proposed CSE Method

There are many possible algorithms for CSE in SAT. In this thesis, we choose an approach based on the Lempel-Ziv-Welch (LZW) data compression algorithm. In this approach, we process each clause as a string rather than as a set; thus, the order of the variables within each clause of the SAT formula matters. The common subexpressions are the syntactically equivalent subclauses containing at least two variables and occurring in at least two clauses in the formula. For example, the formula $\{\{A, B\}, \{A, B, C\}, \{A, C\}\}$ has only one common subclause, $\{A, B\}$.

Though this approach does not detect all common subclauses in the SAT formula, we expect that for some particular SAT instances solving time would improve and that it can be beneficial in some application problems.

4.3 Summary

This chapter discussed the Common Subexpression Elimination concept in relation to SAT. It also provided a brief introduction to the proposed CSE method. The following chapter provides a detailed discussion of the LZW-based approach to Common Subexpression Elimination in SAT.

CHAPTER V

LZW-BASED CSE APPROACH

In this section, we provide a detailed explanation of the Lempel–Ziv-Welch (LWZ)-based approach to Common Subexpression Elimination in SAT. We first explain the LZW data compression algorithm and then present the LWZ-based algorithm. The limitations of this approach are also outlined in this section.

A given string *aba* takes 3 bytes to encode: 97, 98, 97. If we use the code 256 to encode the string *aba*, we use 9 bits and get a considerable amount of compression. The LZW algorithm uses this approach. It is a well-known lossless data compression algorithm that has been used in many applications such as the Graphics Interchange Format (GIF) [11]. The LZW-based algorithm uses some of the techniques used in the LZW compression algorithm to efficiently replace syntactically equivalent subclauses in a given SAT formula.

5.1 LZW Algorithm

In Section 3.4.2.3, we provided a brief overview of the LZW algorithm. In this section, we describe it in detail. This algorithm compresses a given string by replacing its substrings with codes [7]. First, we define the following terms:

code for character in given string: ASCII code of character in the range 0-255

code for substring (length > 1): > 255

dictionary/string table: data structure to hold substrings and their corresponding codes

The code for character *a* is 97, character *b* is 98, and so on. Substring *ac* may have a code of 257. These substrings, each with length > 1, are assigned codes as the algorithm progresses. Figure 5.1 depicts the LZW algorithm as provided by Mark Nelson.

```
1.  STRING = get input character
2.  WHILE there are still input characters DO
3.      CHARACTER = get input character
4.      IF STRING+CHARACTER is in the string table then
5.          STRING = STRING + CHARACTER
6.      ELSE
7.          output the code for STRING
8.          add STRING+CHARACTER to the string table
9.          STRING = CHARACTER
10.     END of IF
11. END of WHILE
12. output the code for STRING
```

Figure 5.1: LZW algorithm

In each iteration, the algorithm gets the next character from the given string and checks whether the string formed from the concatenation of the previous character/substring (*STRING)* and this new character (*CHARACTER*) is in the dictionary. For example, if the given string is *abaca*, then *STRING* = *a*, *CHARACTER*= *b*, and the algorithm searches for the substring *ab* (*STRING* + *CHARACTER*) in the dictionary. If it is found, then *STRING* = *ab*. In the next iteration, *CHARACTER* = *a,* which is the next input character. The algorithm will then search for substring *aba* (*STRING* + *CHARACTER*) in the dictionary. If the substring *ab* is not found, then the code for *a* (*STRING*) is sent to output, the substring *ab* is assigned the next available code and stored in the dictionary, and STRING becomes the current character. The algorithm continues until there are no more input characters.

The resulting compressed string is a string of codes. In the example mentioned earlier, the string *aabbababaabb* is compressed to *97 97 98 98 257 260 257 98* with the dictionary in Table 5.1.

Table 5.1 LZW Example

Dictionary		
Code	Substring	
97	97	a
98	97	b
…	…	
256	97 97	aa
257	97 98	ab
258	98 98	bb
259	98 97	ba
260	257 97	aba
261	260 97	abaa
262	257 98	abb

Nelson provides a C implementation of this LZW algorithm. In this implementation, the dictionary or string table is first empty. When the algorithm *gets an input character*, it gets the ASCII code associated with that character. *STRING* represents the code for a substring and *CHARACTER* represents the code for the next character in the given string.

5.1.1 Dictionary Structure

The dictionary does not store the single characters since their codes can be automatically generated using the *getc()* function. A dictionary entry is represented as a tuple: <*code, prefix, character*> where *prefix* is *STRING*, *character* is *CHARACTER*, and *code* is the code for the substring represented by *prefix* + *character*. Note that *code* > 255 since the codes 0-255 are reserved for the ASCII characters. An *index* of the dictionary references substrings with their corresponding codes. For example, if the algorithm is searching for substring *aba* in the dictionary, it may find it at *index 7* with dictionary entry <260, 257, 97>. Code 260 represents substring *aba*, code 257 represents substring *ab*, and code 97 represents character *a*. In addition, the dictionary entry for substring *ab* is <257, 97, 98>; in order words, <257, *a*, *b*>. We can observe that for every substring *s* in the dictionary, $s = s_1 + c$ where s_1 is a character or a substring in the dictionary and *c* is a character.

5.1.2 Dictionary Search

When searching for a substring in the dictionary, we need a way to do so efficiently without much overhead. Nelson used a hash function where *index* = hash(*prefix*, *character*). In other words, index(s) = hash(s) = hash($s_1 + c$). For example, if the algorithm needs to search for the substring *ab*, it computes the following:

$s = ab$, $s_1 = a$, $c = b$

index(*ab*) = hash(*ab*) = hash(*a*,*b*) = hash(*97, 98*) = 1601

It then checks whether the *code* of the dictionary entry at the computed index *1601* represents *ab*. If it does not, then substring *ab* is not in the dictionary. If it does and the next character is *a*, it computes the following:

$s = aba$, $s_1 = ab$, $c = a$

index(*aba*) = hash(*aba*) = hash(*ab*,*a*) = hash(257, 97) = 1809

Here *prefix* + *character* = *ab* + *a*. The algorithm then checks whether the *code* of the dictionary entry at the computed index *1809* represents *aba*.

The hash function uses two bitwise operators, namely left shift and XOR. It returns only when the entry at the computed index *i* either is empty or matches the substring being searched for.

5.2 LZW-based CSE Algorithm

For the new algorithm based on the LZW compression algorithm, the SAT formula is a set of clauses containing integers (similar to characters in the LZW algorithm). Recall the DIMACS CNF format for the SAT formula. Here, a clause is a string of integers. We use the following terms to describe the new algorithm and the modifications made to the original algorithm.

- *integer*: literal in formula
- *nvars*: number of variables in the original formula
- *existing variable*: variable in original formula
- *new variable*: variable representing potential common subclause in formula
- *code for an existing variable*: existing variable itself
- *code for a new variable*: > *nvars*

- *newCode*: code for a new variable
- *cp:* position of current literal being processed in formula
- *equivClauses*: set of clauses that represent the equivalence between new variables and their corresponding common subclauses
- *dictionary*: data structure to store potential common subclauses and their corresponding new variables

For example, the original SAT formula $\{\{A,B\},\{A,B,C\}\}$ is represented as follows:

Original		New
1 2		4
1 2 3	→	4 3
		-4 1 2
		-1 4
		-2 4

The original formula contains two clauses: 1 2 and 1 2 3. The number of variables, n*vars*, it contains is 3. The existing variables are 1, 2, and 3. The next available code/*new variable* is 4 and its corresponding subclause is 1 2. The *equivClauses* data structure contains 4 ↔ 1 2.

5.2.1 Dictionary

The dictionary used for the new algorithm is similar to that of the LZW algorithm; *<new_var, prefix_var, integ>* where *new_var* is a new variable, *prefix_var* is an integer or a subclause, *integ* is an integer. However, the tuple contains three additional elements which are described below.

clause_no: clause in which subclause first occurs

loc: position of subclause in formula

substituted: indicator of whether subclause in dictionary has been substituted
= 0 if it has not been
= 1 if it has been

We also create another dictionary which records the index of the new variables in the first dictionary. It contains the element *index2* that holds these indices. This element

is needed at the end of the algorithm when it appends the *equivClauses* to the new formula.

5.2.2 Algorithm

The main operations of the new algorithm which is presented in Figure 5.2 are:

1. Modify original SAT formula
2. Replace common subclauses with new variables
3. Add equivalence clauses

The original SAT formula is first modified to contain only clauses with no trailing zeros and no comments so that it is easier to be manipulated. The dictionary is first empty as well as the new formula. For each clause in the SAT file, the algorithm searches from left to right for the longest subclause from the current position *cp* that matches a subclause in the dictionary. Like the LZW compression algorithm, it uses a hash function to index into the dictionary in order to reduce the amount of time taken to match the subclauses.

If a match is found, the algorithm checks whether the subclause has been substituted already and/or still exists in its original clause. If it has been substituted, the new variable corresponding to the subclause is appended to the new formula. If it has not been substituted and still exists in its original clause, then new variable corresponding to the subclause is appended to the new formula, substituted in *clause_no*, and added to *equivClauses*. Otherwise, it does not exist in its original clause; its dictionary entry is updated and *prefix_var* is appended to the new formula. If a match is not found, the algorithm adds the subclause, represented by the pair of consecutive literals from *cp,* to the dictionary. After all clauses have been scanned, the clauses in *equivClauses* are appended to the new formula.

```
Algorithm LZW-based CSE

Input: A sequence F of clauses (each clause is a sequence of literals).
Output: A sequence cF of clauses, equivalent to F,
            resulting from the elimination of some common subexpressions of F.

if F is empty, set cF = <> and terminate;
let nvars be the number of distinct literals in F;
newCode := nvars where newCode denotes the next available new code;
set the dictionary to be empty;
curClause := 1 where curClause denotes the current clause under processing;
cF = <>;
equivClauses := <> where eqivClauses holds all new clauses introduced due to elimination;
for every pair of consecutive literals < i_code1, i_ code2> in the first clause C in F {
        insert entry <code, i_code1, i_ code2, clause_no, loc, substituted> into the dictionary
            where code = newCode, clause_no = curClause, loc is the position of i_code1 in C, and substituted = 0;
        newCode := newCode +1;
}
curClause := curClause + 1;
while F is not empty {
        cp := 0 where cp denotes the position of the current literal of C;
        while end of C is not reached {
            if cp is at the end of C
                append literal at cp to cF;
                 cp := cp + 1;
            else
                search for the longest subclause c of C starting from cp that is in the dictionary;
                if not found
                    insert entry <code, i_code1, i_ code2, clause_no, loc, substituted > to the dictionary
                        where code = newCode, <i_code1, i_ code2> are consecutive literals from cp, clause_no =
                         curClause, loc is the position of i_code1, and substituted =0;
                    append i_code1 to cF;
                    newCode := newCode +1;
                    cp := cp + 1;
                else
                    let the dictionary entry for c be <code, i_code1, i_ code2, clause_no, loc, substituted >;
                    if substituted
                        append code to cF;
                        cp := cp + length(c);
                    else
                        if clause_no of c still has the string <i_code1, i_ code2> at position loc
                            replace the common subclause c in clause_no by code;
                            append code to cF;
                            update substituted of dictionary entry of c to be 1;
                            append clauses resulting from c ↔ code to equivClauses;
                            cp := cp + length(c);
                        else // c does not appear in clause_no any more
                            update clause_ no and loc of the dictionary entry of c to be curClause and cp respectively;
                            append i_code1 to cF;
                            cp := cp + length(c) - 1;
        }
        curClause := curClause + 1;
}
append equivClauses to cF;
```

Figure 5.2: LZW-based Algorithm

5.3 Summary

This chapter discussed the Lempel–Ziv-Welch (LWZ)-based approach to Common Subexpression Elimination in SAT. A description of the LZW algorithm was first provided. This algorithm compresses a given string of characters into a string of codes using a dictionary to store potential common substrings and their corresponding codes [7]. Each code in the compressed string represents a single character or a substring stored in the dictionary. From this algorithm, the LZW-based algorithm was developed to exploit CSE in a given SAT formula. The LZW-based algorithm finds and replaces only syntactically equivalent subclauses in the formula. By considering each clause in the CNF formula as a string, the algorithm searches each clause for common subclauses and records potential ones in a dictionary. The next chapter provides an empirical study on this approach.

CHAPTER VI

EMPIRICAL STUDY

In this chapter, we present the benchmarks, system, and tools used to evaluate the proposed Lempel–Ziv-Welch (LWZ)-based approach to Common Subexpression Elimination in SAT, describe the results obtained from the experiments, and provide an analysis of these results. We carried out experiments on both the original instance and the new instance created by the LZW-based algorithm. Various parameters were used to evaluate this CSE method on SAT-solving efficiency.

6.1 Benchmark Suite

To test the impact of the LZW-based approach on SAT solving, we used SAT benchmarks from the SAT 2011 Competition [12]. The SAT Competition is an annual event of the International Conference on Theory and Applications of Satisfiability Testing where researchers from various regions as well as areas of research present and compare their SAT solvers on various parameters, mainly CPU solving time, by running them on the various benchmark problems [17]. The solvers presented in the competition are developed in the hopes of potentially performing more efficiently than existing or newly created solvers. The competition aims to present challenging SAT instances and fast solvers needed for SAT-related research purposes.

All the benchmarks of the competition are represented in DIMACS CNF format. They are grouped in three categories: Application, Crafted, and Random. The Application instances are usually large instances encoding application problems from domains such as planning. The Crafted instances are designed to be hard and challenging for the SAT solver. The Random instances are randomly generated k-SAT instances where k represents the number of literals per clause [13]. As an example, 3-SAT instances contain 3-literal clauses.

For the experiments, we select 68 benchmarks from the SAT 2011 Competition that are small in size (<20 MB): 27 instances from Application, 32 instances from Crafted, and 9 instances from Random. They are listed in Table 6.1.

Table 6.1 Benchmarks

Application	Crafted	Random
aes_32_1_keyfind_1	battleship-5-8-unsat	unif-k3-r4.26-v250-c1065-S427778075-026.UNKNOWN
aes_32_2_keyfind_1	battleship-6-9-unsat	unif-k3-r4.26-v250-c1065-S534787376-095.UNKNOWN
aes_32_3_keyfind_1	battleship-7-12-unsat	unif-k3-r4.26-v250-c1065-S931339469-052.UNKNOWN
aes_64_1_keyfind_1	battleship-7-13-sat	unif-k3-r4.26-v250-c1065-S984689729-080.UNKNOWN
AProVE11-12	battleship-8-15-sat	unif-k3-r4.26-v250-c1065-S1397126856-076.UNKNOWN
AProVE11-15	crn_11_99_u	unif-k3-r4.26-v250-c1065-S1822479556-014.UNKNOWN
AProVE11-09	crn_11_100_s	unif-k3-r4.26-v300-c1278-S768688898-014.UNKNOWN
AProVE11-16	rnd_100_27_s	unif-k3-r4.26-v300-c1278-S862932513-073.UNKNOWN
AProVE11-11	rnd_100_28_u	unif-k3-r4.26-v300-c1278-S1194590195-083.UNKNOWN
AProVE11-02	rnd_100_28_s	
traffic_3b_unknown	rnd_100_32_s	
traffic_kkb_unknown	GreenTao_2-3-5_527	
traffic_b_unsat	GreenTao_2-3-5_528	
traffic_f_unknown	sgen3-n120-s12930489-sat	
aaai10-planning-ipc5-pathways-13-step17	sgen3-n120-s55656844-unsat	
aaai10-planning-ipc5-pathways-17-step20	sgen3-n130-s30940966-unsat	
aaai10-planning-ipc5-pathways-17-step21	sgen3-n140-s18527668-sat	
E05X15	srhd-sgi-m27-q225-n25-p15-s58217873	
E07N15	srhd-sgi-m27-q255-n25-p15-s2076598	
E15N15	srhd-sgi-m27-q255-n25-p30-s39712998	
E02F17	srhd-sgi-m27-q225-n25-p30-s70617701	
korf-15	VanDerWaerden_2-3-12_135	
korf-17	VanDerWaerden_2-3-13_160	
korf-18	VanDerWaerden_2-3-14_186	
slp-synthesis-aes-bottom12	VanDerWaerden_pd_2-3-19_348	
slp-synthesis-aes-bottom13	VanDerWaerden_pd_2-3-20_381	
slp-synthesis-aes-bottom14	VanDerWaerden_pd_2-3-20_390	
	289-sat-4x8	
	289-sat-5x8.cnf	
	289-sat-7x6.cnf	
	289-sat-11x4.cnf	
	289-sat-6x8.cnf	

This particular set of instances showed the potential of containing syntactically equivalent subclauses and the majority could be solved in less than a day. Excluding the GreenTao instances, at least 3 instances from each problem were used in the experiments. For example, for the Battleship problem, 5 instances were used.

6.2 Environment

We performed experiments on an Ubuntu 10.04 LTS workstation with 3.6 GB RAM and a CPU clock rate of 2.66 GHz. We used one version of the MiniSat solver, minisat2-070721 [4], to solve both the original formula and the new equivalent formula after CSE using the LZW-based approach. The LZW-based algorithm was developed under the GNU C++ compiler version 4.4.1.

The Bash shell script that was used in the experiments performed the following operations in sequence on each instance in the set of benchmarks selected:

1. Solve the original instance with MiniSat.
2. Run the compiled LZW-based code on the original instance to generate the new equivalent instance.
3. Solve the new instance with MiniSat.

For each instance, the script stored the output of each operation in a separate text file-the results from MiniSat and the new instance. The results were then tabulated and evaluated according to various parameters for analysis.

6.3 Evaluation Metric

We compare the results of the experiments based on the following parameters for each problem instance and each new instance generated after running it with the LZW-based algorithm:

- size (S) in KB
- number of variables (V)
- number of clauses (C)
- ratio of the number of variables in the original instance to the number of variables in the new instance

- number of decisions (D)
- number of propagations (P)
- satisfiability (Sa)
- CPU time (T) in seconds

The size of the SAT file, the number of clauses, and the number of variables were obtained directly from each instance. The number of decisions, number of propagations, satisfiability, and CPU time for each instance were obtained from the results of the MiniSat solver. For each instance, the ratio of the number of variables in the original instance to the number of variables in the new instance was computed automatically.

6.4 Experimental Results

The overall results from the experiments are shown in Tables A.1, A.2, and A.3 in Appendix A. Out of the 68 instances, 27 instances showed little or no difference in results:

- In the Application instances, the E07N15 and E15N15 instances were solved instantly by unit propagation.
- In the Crafted instances, the battleship, two of the sgen-n3, and srhd-sgi instances did not contain syntactically equivalent subclauses; the 289-sat instances showed no improvement.
- The Random instances showed little improvement or none.

Thus, we evaluated the remaining 41 instances: 25 from Application and 16 from Crafted. These particular instances and their corresponding reference numbers are listed in Table 6.2. We first evaluate them on each criterion: number of variables/clauses, ratio of the number of variables in the original instance to the number of variables in the new instance, number of propagations/decisions, satisfiability, and CPU time. In the graphs that follow, o*rig* represents the original instances and *new* represents the new instances generated after running them with the LZW-based algorithm.

Table 6.2 Instances for Analysis

Application	Ref #	Crafted	Ref #
aes_32_1_keyfind_1	1	crn_11_99_u	26
aes_32_2_keyfind_1	2	crn_11_100_s	27
aes_32_3_keyfind_1	3	rnd_100_27_s	28
aes_64_1_keyfind_1	4	rnd_100_28_u	29
AProVE11-12	5	rnd_100_28_s	30
AProVE11-15	6	rnd_100_32_s	31
AProVE11-09	7	GreenTao_2-3-5_527	32
AProVE11-16	8	GreenTao_2-3-5_528	33
AProVE11-11	9	sgen3-n120-s55656844-unsat	34
AProVE11-02	10	sgen3-n130-s30940966-unsat	35
traffic_3b_unknown	11	VanDerWaerden_2-3-12_135	36
traffic_kkb_unknown	12	VanDerWaerden_2-3-13_160	37
traffic_b_unsat	13	VanDerWaerden_2-3-14_186	38
traffic_f_unknown	14	VanDerWaerden_pd_2-3-19_348	39
aaai10-planning-ipc5-pathways-13-step17	15	VanDerWaerden_pd_2-3-20_381	40
aaai10-planning-ipc5-pathways-17-step20	16	VanDerWaerden_pd_2-3-20_390	41
aaai10-planning-ipc5-pathways-17-step21	17		
E05X15	18		
E02F17	19		
korf-15	20		
korf-17	21		
korf-18	22		
slp-synthesis-aes-bottom12	23		
slp-synthesis-aes-bottom13	24		
slp-synthesis-aes-bottom14	25		

6.4.1 Number of Variables/Clauses

The change in the number of variables and clauses for the instances is illustrated in Figure 6.1 and Figure 6.2 The number of variables and number of clauses always increases if syntactically equivalent subclauses are found since for each new variable, there are three additional clauses created for the equivalence between that variable and its corresponding common subclause. For example, the aes_32_1_keyfind_1 new instance contains 110 new variables and therefore would have 330 new clauses.

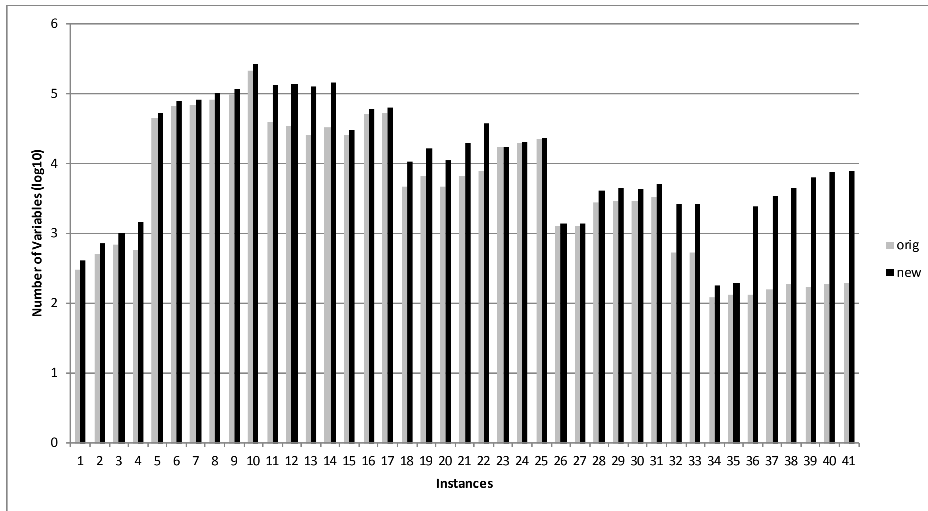

Figure 6.1: Number of Variables

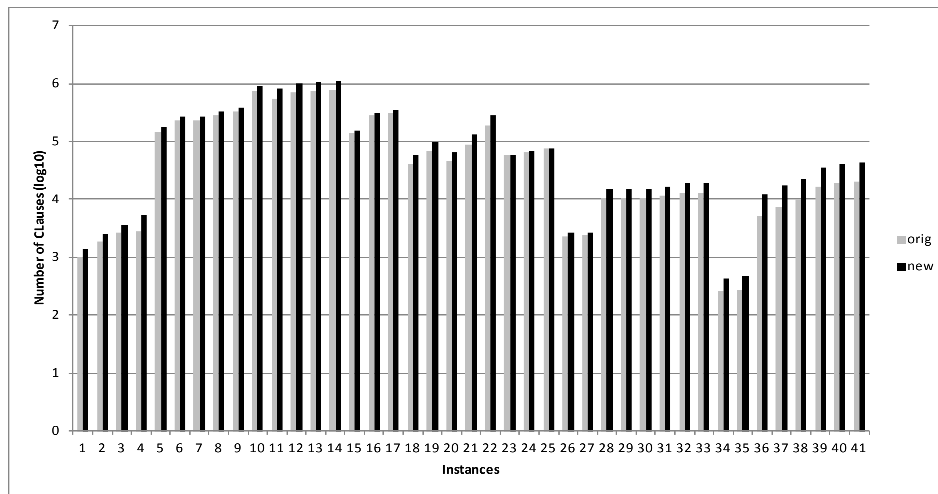

Figure 6.2: Number of Clauses

From Figure 6.3, the VanDerWaerden family of instances from the Crafted category showed the highest increases in the number of variables; the number of variables in the new instances was between 18 and 42 times more than the number of variables in the original instances, indicating that there were many common subclauses in these instances. Other instances which showed a considerable increase in the number of variables included the traffic, E0, and korf instances from the Application group and the GreenTao instances from the Crafted group. The slp-synthesis-aes-bottom family of instances from the Application group showed the lowest percent increase in the number of variables, indicating that they contained relatively few common subclauses.

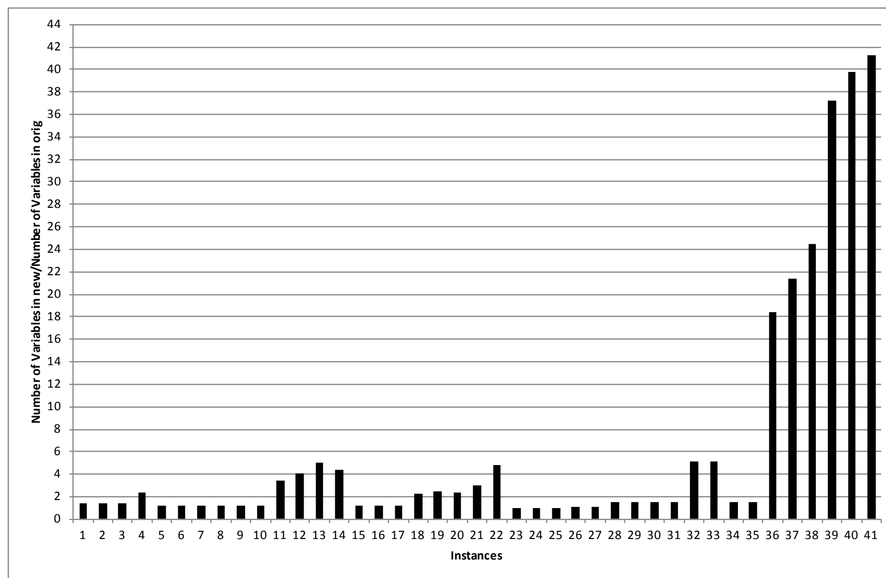

Figure 6.3: Number of Variables in new/Number of Variables in orig

6.4.2 Number of Decisions/Propagations

As shown in Figure 6.4 and Figure 6.5, the number of propagations and decisions for most new instances was more than that of the original instances. The sgen3-n120-s55656844-unsat instance from the Crafted group exhibited the highest increase in the number of decisions while the korf-18 instance from the Application group exhibited the highest increase in the number of propagations. The sgen3-n130-s30940966-unsat instance exhibited the highest decrease in both the number of decisions and the number of propagations.

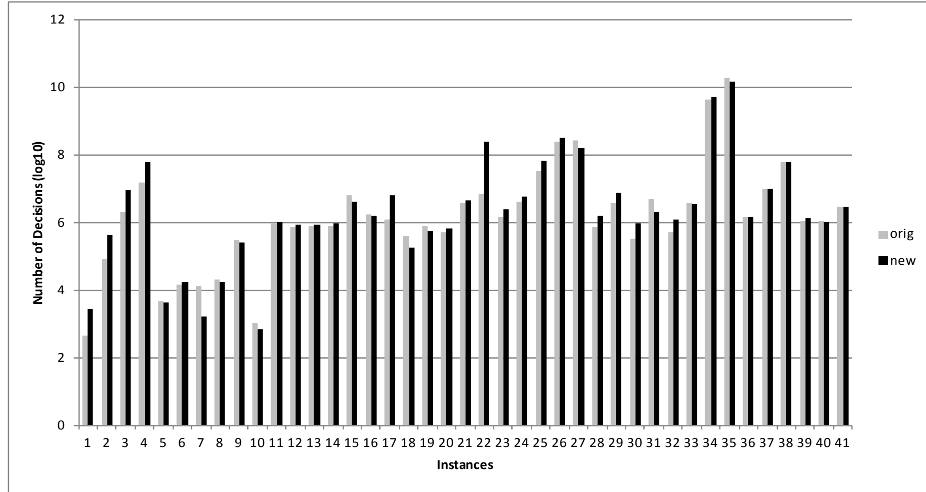

Figure 6.4: Number of Decisions

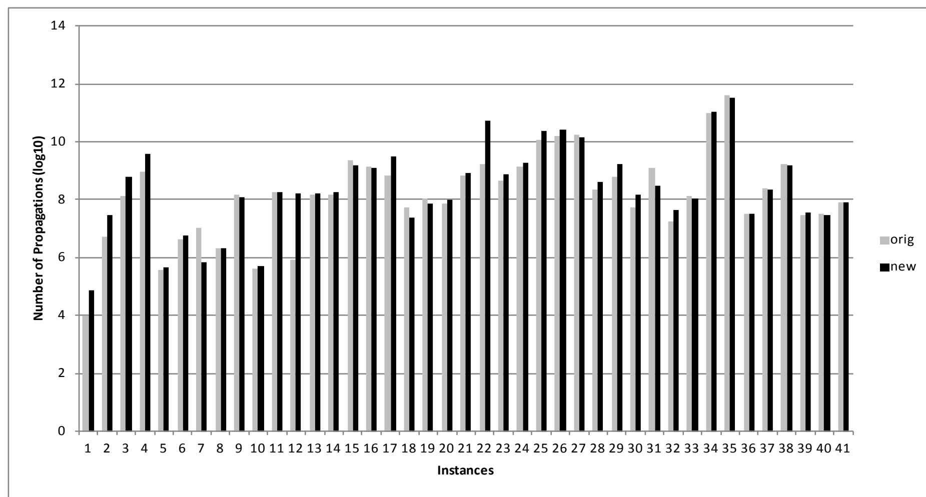

Figure 6.5: Number of Propagations

The traffic_kkb_unknown instance from the Application group showed the highest percent increase in the number of propagations – the number of propagations for the new instance was at least 185 times greater than that of the original instance. The korf-18 instance showed the highest percent increase in the number of decisions – the number of decisions for the new instance was approximately 34 times higher than that of the original. The AProVE11-09 instance from the Application group showed the highest percent decrease in both the number of propagations and decisions - approximately 93% and 88% respectively.

Among the families of instances, the aes_keyfind instances of the Application group combined showed the highest percent increase in both the number of decisions and the number of propagations whereas the AProVE instances showed the highest percent decrease and the VanDerWaerden instances showed the lowest percent decrease. The lowest percent increase in the number of decisions was exhibited by the traffic instances and that in the number of propagations was exhibited by the VanDerWaerden instances.

6.4.3 Satisfiability

As shown in Table 6.3, 26 instances were unsatisfiable (UNSAT). Out of these instances, 15 were Application instances and 11 were Crafted instances. The other 15 instances were satisfiable (SAT); 5 of these instances are from the Application group and 10 of them are from the Crafted group.

Table 6.3 Satisfiability

Application	Ref #	Sa	Crafted	Ref #	Sa
aes_32_1_keyfind_1	1	SAT	crn_11_99_u	26	UNSAT
aes_32_2_keyfind_1	2	SAT	crn_11_100_s	27	SAT
aes_32_3_keyfind_1	3	SAT	rnd_100_27_s	28	SAT
aes_64_1_keyfind_1	4	SAT	rnd_100_28_u	29	UNSAT
AProVE11-12	5	SAT	rnd_100_28_s	30	SAT
AProVE11-15	6	SAT	rnd_100_32_s	31	SAT
AProVE11-09	7	SAT	GreenTao_2-3-5_527	32	SAT
AProVE11-16	8	SAT	GreenTao_2-3-5_528	33	UNSAT
AProVE11-11	9	UNSAT	sgen3-n120-s55656844-unsat	34	UNSAT
AProVE11-02	10	SAT	sgen3-n130-s30940966-unsat	35	UNSAT
traffic_3b_unknown	11	UNSAT	VanDerWaerden_2-3-12_135	36	UNSAT
traffic_kkb_unknown	12	UNSAT	VanDerWaerden_2-3-13_160	37	UNSAT
traffic_b_unsat	13	UNSAT	VanDerWaerden_2-3-14_186	38	UNSAT
traffic_f_unknown	14	UNSAT	VanDerWaerden_pd_2-3-19_348	39	UNSAT
aaai10-planning-ipc5-pathways-13-step17	15	UNSAT	VanDerWaerden_pd_2-3-20_381	40	UNSAT
aaai10-planning-ipc5-pathways-17-step20	16	UNSAT	VanDerWaerden_pd_2-3-20_390	41	UNSAT
aaai10-planning-ipc5-pathways-17-step21	17	SAT			
E05X15	18	UNSAT			
E02F17	19	UNSAT			
korf-15	20	UNSAT			
korf-17	21	UNSAT			
korf-18	22	UNSAT			
slp-synthesis-aes-bottom12	23	UNSAT			
slp-synthesis-aes-bottom13	24	UNSAT			
slp-synthesis-aes-bottom14	25	UNSAT			

6.4.4 CPU Time

As can be seen in Figure 6.6, the korf-18 instance exhibited the worst performance; its new instance was at most 60 times slower than the original instance. The original instance took approximately 2.5 hours to solve whereas the new instance took approximately 7 days to solve. The aaai10-planning-ipc5-pathways-17-step21 instance showed the second highest increase in CPU time; its new instance was at most 12 times slower than the original instance. The original instance took approximately 28 minutes to solve whereas the new instance took approximately 5 hours to solve. Excluding the korf-18 and aaai10-planning-ipc5-pathways-17-step21 instances, solving time for the instances that showed worse performance increased in the range of approximately 10 to 466 percent. All the new instances of the aes_keyfind, traffic, and slp-synthesis-aes-bottom problems exhibited slower CPU times.

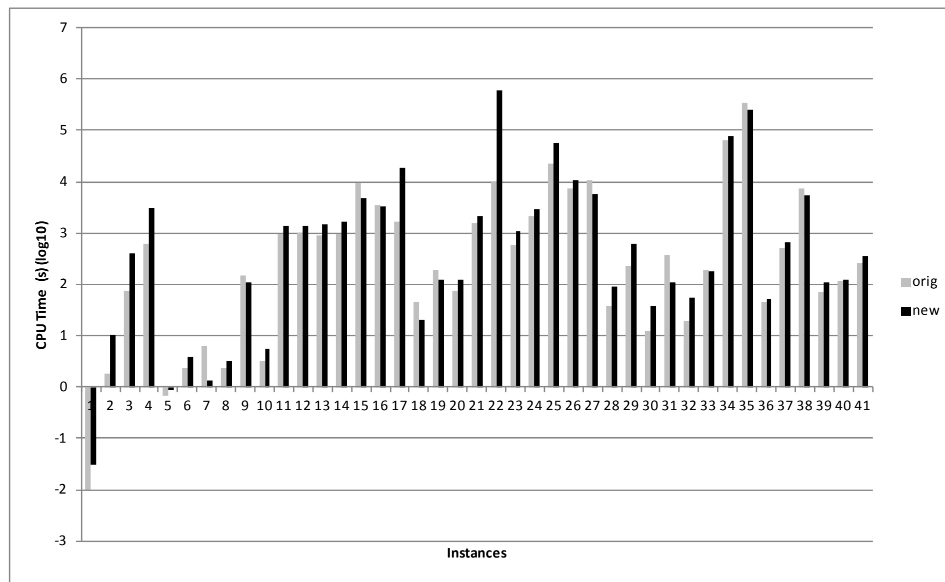

Figure 6.6: CPU Time

For 11 out of the 41 instances, solving time decreased in the range of approximately 4 to 78 percent. Table 6.3 lists these instances. The rest showed worse performance with increases in solving time ranging from 10 to 6000 percent.

Table 6.4 Fast New Instances

Application	Ref #	Crafted	Ref #
AProVE11-09	7	crn_11_100_s	27
AProVE11-11	9	rnd_100_32_s	31
aaai10-planning-ipc5-pathways-13-step17	15	GreenTao_2-3-5_528	33
aaai10-planning-ipc5-pathways-17-step20	16	sgen3-n130-s30940966-unsat	35
E05X15	18	VanDerWaerden_2-3-14_186	38
E02F17	19		

The AProVE11-09 and rnd_100_32_s new instances showed the most significant decrease in solving time - approximately 78% and 70% decrease respectively. Out of these fast instances, the solving times for the crn_11_100_s, aaai10-planning-ipc5-pathways, sgen3-n130-s30940966-unsat, and VanDerWaerden_2-3-14_186 instances were reduced with the lowest CPU time of 58 minutes reducing to 56 minutes, and the highest CPU time of 93 hours reducing to 71 hours.

The execution of the LZW-based algorithm on each instance took less than a minute. Considering this preprocessing time for each of the fast instances, those with a difference in solving time of less than a minute would become slightly slower. These instances were the AProVE11-09, AProVE11-11, E05X15, and GreenTao_2-3-5_528 instances.

6.5 Comprehensive Analysis

Approximately 27% of the new instances, listed in Table 6.1, showed a decrease in solving time in the range of approximately 4 to 78 percent. Out of these instances which performed well, the VanDerWaerden_2-3-14_186 instance showed the most significant increase in the number of variables - approximately 24 times higher. For all instances which showed improvement, the number of decisions and propagations decreased. In addition, 8 out of these 11instances were unsatisfiable (UNSAT).

The instances which showed worse performance had increases in solving time in the range of approximately 10 to 6000 percent. The korf-18 new instance was the slowest; its new instance was at most 60 times slower than its original instance.

Additionally, 18 of these 30 instances were unsatisfiable (UNSAT) and the number of decisions and propagations increased for 80% of these slow instances.

6.6 Limitations

Some of the difficulties and limitations experienced with this LZW-based approach are as follows.

- We had to understand thoroughly the LZW data compression algorithm and implementation.
- Executing the SAT solver with some of the original instances (before executing them with the LZW-based algorithm) took a considerable amount of time; some instances would take days to solve.
- The LZW-based algorithm finds only syntactically equivalent subclauses.

One observation that we keep in mind is that not all common subclauses are eliminated. The new formula may contain common subclauses containing new variables.

6.7 Summary

This chapter described the tools used to carry out the experiments for the proposed LZW-based approach, provided a detailed evaluation of the results obtained from these experiments, and highlighted the difficulties and limitations with the approach. The instances were evaluated in terms of size, number of clauses, variables, decisions, and propagations, ratio of the number of variables in the original instance to the number of variables in the new instance, satisfiability, and CPU time. After the LZW-based method was applied, the majority (approximately 73%) of the instances were solved slower. These instances showed worse performance with increases in solving time ranging from 10 to 6000 percent. Approximately 27% of the new instances showed a reduced amount of solving time. The following chapter summarizes the goals and findings of the research effort and discusses future work for the proposed LZW-based CSE method.

CHAPTER VII

CONCLUSION

The Lempel–Ziv-Welch (LWZ)-based approach to Common Subexpression Elimination in SAT was introduced and evaluated in terms of its impact on SAT-solving efficiency. In this chapter, we summarize this research effort and provide a brief discussion on the future work related to the proposed CSE approach.

7.1 Summary

The Satisfiability problem is a well-known problem in many areas of Computer Science. Many important application problems can be encoded into SAT instances and solved using a SAT solver. Considerable research as well as events such as the SAT Competition have been undertaken in this area [15]. In this thesis, we examined the Common Subexpression Elimination concept in relation to the SAT problem and studied the LZW-based CSE approach.

In Common Subexpression Elimination, we replace syntactically or semantically equivalent subexpressions within an expression by new variables [10]. The related work has shown that CSE is applicable in many domains. In compiler optimization, a Direct Acyclic Graph is formed for a block of code where the common subexpressions are shared among nodes [1]. In constraint modeling, the goal of this elimination is to generate an expression that can be solved more efficiently [9]. Other problem areas related to CSE include the Smallest Grammar problem and Data Compression.

The proposed CSE method used in this thesis was based on the Lempel–Ziv-Welch (LZW) data compression algorithm. The LZW algorithm compresses a given string into a string of codes [7]. The main data structure used in this algorithm is a dictionary that stores potential common substrings. The algorithm also uses a hash function for efficiently searching the dictionary for a given substring. In the LZW-based algorithm, potential common subclauses are stored in the dictionary and each clause in the SAT formula is processed as a string. The LZW-based method only applies to syntactically equivalent subclauses in SAT formulas. As mentioned in the limitations, this method does not remove all common subclauses in the SAT formula.

For investigating the impact of the proposed LZW-based CSE approach on SAT-solving efficiency, we selected a set of benchmarks from the SAT 2011 Competition and performed three operations on each instance within this set. We first solved the original instance with the MiniSat solver, run it with the proposed algorithm, and solve the generated instance with MiniSat. The parameters used for the evaluation of the algorithm was the size, number of clauses, variables, decisions, and propagations, ratio of the number of variables in the original instance to the number of variables in the new instance, satisfiability, and CPU time.

The experimental results have showed that for the majority of the instances the LZW-based algorithm had a negative impact on SAT-solving efficiency. The slowest new instance was 60 times slower than its original instance and took approximately 7 days to solve. There was little or no impact on the random, battleship, E07N15 and E15N15 instances, two of the sgen-n3, and srhd-sgi instances. The algorithm had a positive impact on approximately 27% of the instances used in the analysis – they showed a decrease in solving time in the range of approximately 4 to 78 percent.

7.2 Future Work

It would be interesting to identify the classes of problems for which the LZW-based CSE method can solve SAT formulas more efficiently. Future work would be to enhance this algorithm to eliminate all common subclauses and expand it so that each clause is processed as a set rather than as a string. The algorithm could be a basis on which to extend this research. In addition, further work would be to find the most efficient algorithm that exploits CSE in SAT formulas. Other approaches could be investigated and compared to find an effective algorithm such that SAT solving time for most instances drastically improves. The experiments can be performed with a variety of SAT solvers and benchmarks. The benefits of Common Subexpression Elimination in SAT can be employed within the SAT community.

BIBLIOGRAPHY

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed.: Addison Wesley, 2007, pp. 358-359, 533-535.

[2] I. Araya, B. Neveu, and G. Trombettoni, "Exploiting Common Subexpressions in Numerical CSPs," in *Procdeedings of CP*, 2008, pp. 342-357.

[3] M. Charikar et al., "The Smallest Grammar Problem," *IEEE Transactions on Information Theory*, vol. 51, no. 7, pp. 2554-2576, 2005.

[4] N. Eén and N. Sörensson. The MiniSat Page. http://www.minisat.se/MiniSat.html

[5] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, "Satisfiability Solvers," in *Handbook of Knowledge Representation*, F. van Harmelen, V. Lifschitz, and B. Porter, Eds.: Elsevier, 2008.

[6] J. Müller, "Data Compression LZ77," 2008. http://jens.quicknote.de/comp/LZ77-JensMueller.pdf

[7] M. Nelson, "LZW Data Compression," *Dr. Dobb's Journal*, vol. 14, no. 10, pp. 29-37, October 1989. http://marknelson.us/1989/10/01/lzw-data-compression

[8] A. Nerod and R. Stone, *Logic for Applications*, 2nd ed.: Springer-Verlag NY, Inc., 1997.

[9] A. Rendl, I. Miguel, I. P. Gent, and C. Jefferson, "Automatically Enhancing Constraint Model Instances during Tailoring," in *SARA*, 2009.

[10] A. Rendl, I. Miguel, and I. P. Gent, "Eliminating Common Subexpressions during Flattening," 2008. http://www-circa.mcs.st-and.ac.uk/Preprints/ERCIMCSE08.pdf

[11] D. Salomon, *Data Compression: The Complete Reference*.: Springer, New York, 2004.

[12] SAT Competition 2011. http://www.satcompetition.org/2011

[13] SAT Competition 2011: Benchmark Submission Guidelines. http://www.satcompetition.org/2011/format-benchmarks2011.html

[14] I. Shlyakhter, M. Sridharan, R. Seater, and D. Jackson, "Exploiting Subformula Sharing in Automatic Analysis of Quantified Formulas," in *SAT*, Portofino, Italy, May, 2003.

[15] N. Sörensson, *Effective SAT Solving*, Chalmers University of Technology and University of Gothenburg, Sweden, Ph.D. Thesis, 2008.

[16] J. A. Storer and T. G. SzymanskI, "Data Compression via Textual Substitution," *Journal of the ACM*, vol. 19, no. 4, pp. 928-951, October 1982.

[17] The International Conferences on Theory and Applications of Satisfiability Testing (SAT). http://www.satisfiability.org

[18] The International SAT Competitions web page. http://www.satcompetition.org

[19] C. Zeeh, "The Lempel Ziv Algorithm," 2003. http://w3studi.informatik.uni-stuttgart.de/~zeehca/Seminar/LempelZivReport.pdf

[20] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in Boolean Satisfiability Solver," in *the International Conference on Computer-Aided Design*, 2001, pp. 279–28.

APPENDIX A

RESULTS

Table A.1 Application Instances

File	S	V	C	D	P	Sa	T
aes_32_1_keyfind_1	15	300	1016	471	9796	SAT	0.01
lzw-based	19	410	1346	2762	71784	SAT	0.03
aes_32_2_keyfind_1	29	504	1840	87742	5370413	SAT	1.85
lzw-based	35	724	2500	442562	28112598	SAT	10.47
aes_32_3_keyfind_1	42	708	2664	2190213	139948793	SAT	74.56
lzw-based	52	1038	3654	9150101	594950819	SAT	398.45
aes_64_1_keyfind_1	51	596	2780	15877384	951100039	SAT	626.5
lzw-based	74	1440	5312	61819873	3922038666	SAT	3163.48
AProVE11-12	2655	44805	149118	4837	365538	SAT	0.7
lzw-based	2992	53340	174723	4351	480188	SAT	0.87
AProVE11-15	4117	66715	228274	15244	4351188	SAT	2.29
lzw-based	4676	80545	269764	18261	6075026	SAT	3.92
*AproVE11-09	4208	68779	234693	13922	10102604	SAT	6.36
lzw-based	4716	81288	272220	1628	662187	SAT	1.37
AProVE11-16	5132	84025	282766	20713	2167908	SAT	2.34
lzw-based	5848	101710	335821	18084	2040026	SAT	3.13
*AProVE11-11	5917	96526	325263	317083	145432625	UNSAT	149.17
lzw-based	6811	118627	391566	268469	123916978	UNSAT	109.36
AProVE11-02	14661	214734	743081	1055	402383	SAT	3.23
lzw-based	16691	262344	885911	711	531422	SAT	5.47
traffic_3b_unknown	13015	39151	533919	993502	173801757	UNSAT	974.64
lzw-based	12989	135819	823923	1021997	183866700	UNSAT	1409.75
traffic_kkb_unknown	17211	34510	701694	725881	857566	UNSAT	995.72
lzw-based	15984	140634	1020066	868686	160181884	UNSAT	1385.58
traffic_b_unsat	18262	26061	742909	798783	149276076	UNSAT	907.29
lzw-based	16287	130048	1054870	870774	163415973	UNSAT	1484.17
traffic_f_unknown	18734	33320	763101	798054	155290243	UNSAT	946.48
lzw-based	17270	146620	1103001	948198	180597259	UNSAT	1672.01
*aaai10-planning-ipc5-pathways-17-step21	4961	53919	308235	1294938	693797803	UNSAT	1698.77
lzw-based	5378	64505	339993	6727019	2973781183	UNSAT	18884.5
*aaai10-planning-ipc5-pathways-17-step20	4561	50277	283903	1718192	1395390531	UNSAT	3523.89
lzw-based	4935	60012	313108	1578141	1217671905	UNSAT	3370.13
aaai10-planning-ipc5-pathways-13-step17	2229	25631	142227	6461697	2227915729	SAT	9413.5
lzw-based	2390	30036	155442	4309542	1596615538	SAT	4882.58
*E05X15	939	4740	41379	416740	52231412	UNSAT	46.5
lzw-based	891	10586	58917	187965	23087908	UNSAT	20.83
E07N15	1024	4740	41363			UNSAT	
lzw-based	921	11028	60227			UNSAT	
E15N15	1093	4740	44327			UNSAT	
lzw-based	996	11818	65561			UNSAT	
*E02F17	1675	6664	69700	810123	108878089	UNSAT	188.95
lzw-based	1503	16665	99703	580643	75077991	UNSAT	124.82
korf-15	1192	4740	45569	519495	72846030	UNSAT	76.96

File	S	V	C	D	P	Sa	T
lzw-based	1001	11121	64712	663041	97712626	UNSAT	124.19
korf-17	2194	6664	89966	3715811	693455016	UNSAT	1533.84
lzw-based	1962	19933	129773	4450237	800660395	UNSAT	2129.9
korf-18	6477	7794	186934	7119807	1685138076	UNSAT	10192.1
lzw-based	4603	37560	276232	246185227	52346063928	UNSAT	609163
slp-synthesis-aes-bottom12	926	17298	57292	1514061	438710058	UNSAT	592.84
lzw-based	937	17640	58318	2574976	737855325	UNSAT	1100.89
slp-synthesis-aes-bottom13	1092	19995	66333	4361069	1343271962	UNSAT	2092.27
lzw-based	1103	20369	67455	6157054	1944186107	UNSAT	2999.95
slp-synthesis-aes-bottom14	1271	22886	76038	34388817	11634001453	UNSAT	22125.9
lzw-based	1283	23292	77256	70773764	24492875648	UNSAT	57690.2

Table A.2 Crafted Instances

File	S	V	C	D	P	Sa	T
battleship-5-8-unsat	2	40	105	7213	70810	UNSAT	0.02
lzw-based	2	40	105	7213	70810	UNSAT	0.03
battleship-6-9-unsat	3	54	171	31122	347095	UNSAT	0.13
lzw-based	3	54	171	31122	347095	UNSAT	0.15
battleship-7-12-unsat	5	84	301	9381030	104301350	UNSAT	95.32
lzw-based	5	84	301	9381030	104301350	UNSAT	93.73
battleship-7-13-sat	5	91	322	498	3753	SAT	0.01
lzw-based	5	91	322	498	3753	SAT	0.01
battleship-8-15-sat	8	120	484	384	3453	SAT	0.01
lzw-based	8	120	484	384	3453	SAT	0.01
crn_11_99_u	38	1287	2332	241757872	16061142472	UNSAT	7627.7
lzw-based	41	1386	2629	325278629	25560621407	UNSAT	10507.3
crn_11_100_s	39	1300	2355	271099627	18126831180	SAT	10687.6
lzw-based	42	1400	2655	159164047	14805086396	SAT	5691.06
rnd_100_27_s	163	2754	10377	727921	214207819	SAT	39.07
lzw-based	207	4212	14751	1672385	409049292	SAT	90.45
rnd_100_28_u	167	2856	10578	4003936	606714925	UNSAT	237.06
lzw-based	216	4424	15282	7657978	1674209848	UNSAT	628.84
rnd_100_28_s	167	2856	10578	345418	52944481	SAT	12.56
lzw-based	209	4285	14865	924596	145433511	SAT	37.12
rnd_100_32_s	182	3264	11382	4987499	1194662264	SAT	375.52
lzw-based	239	5056	16758	2141687	298140872	SAT	111.49
GreenTao_2-3-5_527	171	526	12647	511693	17222071	SAT	19.3
lzw-based	246	2689	19136	1248973	42109248	SAT	54.98
GreenTao_2-3-5_528	171	527	12692	3811558	129478113	UNSAT	193.29
lzw-based	247	2694	19193	3421862	115013172	UNSAT	175.74
sgen3-n120-s12930489-sat	4	120	288	520951	13215541	SAT	5.26
lzw-based	4	120	288	520951	13215541	SAT	5.25
sgen3-n120-s55656844-unsat	4	121	252	4348109617	93975767418	UNSAT	63357.7
lzw-based	5	183	438	5110131886	113014137996	UNSAT	76886.2
sgen3-n130-s30940966-unsat	4	133	276	18751398084	416232837568	UNSAT	336563
lzw-based	6	199	474	15294494912	323204885411	UNSAT	255929
sgen3-n140-s18527668-sat	4	140	336	101605	2878853	SAT	1.01
lzw-based	4	140	336	101605	2878853	SAT	1.01
srhd-sgi-m27-q255-n25-p15-s2076598	339	545	29734	2659452	120036582	SAT	1476.88
lzw-based	339	545	29734	2659452	120036582	SAT	1496.71
srhd-sgi-m27-q225-n25-p15-s58217873	406	550	35586	317387	16713777	SAT	100.05
lzw-based	406	550	35586	317387	16713777	SAT	130.07
srhd-sgi-m27-q255-n25-p30-s39712998	518	666	45238	17362148	782735883	SAT	25708.2
lzw-based	518	666	45238	17362148	782735883	SAT	26032.2
srhd-sgi-m27-q225-n25-p30-s70617701	583	671	50773	1025207	53445065	SAT	1189.14
lzw-based	583	671	50773	1025207	53445065	SAT	1207.38
VanDerWaerden_2-3-12_135	90	135	5251	1515352	33328261	UNSAT	45.07
lzw-based	155	2481	12289	1464206	32335802	UNSAT	52.56
VanDerWaerden_2-3-13_160	130	160	7308	9708456	234597576	UNSAT	529.29
lzw-based	226	3422	17094	9675332	233629497	UNSAT	655.94

File	S	V	C	D	P	Sa	T
VanDerWaerden_2-3-14_186	181	186	9795	6279873	1653504858	UNSAT	7323.33
lzw-based	307	4545	22872	60172081	1586710289	UNSAT	5550.94
VanDerWaerden_pd_2-3-19_348	311	174	16458	1112952	28646376	UNSAT	69.07
lzw-based	481	6470	35346	1372316	34733157	UNSAT	108.23
VanDerWaerden_pd_2-3-20_381	365	191	19482	1117080	31480791	UNSAT	115.18
lzw-based	569	7602	41715	1090762	30719091	UNSAT	127.62
VanDerWaerden_pd_2-3-20_390	398	195	20607	2917047	80525764	UNSAT	269.07
lzw-based	610	8052	44178	2949440	81399049	UNSAT	347.11
289-sat-4x8	15	128	896	60	96	SAT	0.01
lzw-based	25	576	2240	60	96	SAT	0.01
289-sat-5x8.cnf	25	160	1400	277	1098	SAT	0.01
lzw-based	35	720	3080	298	1201	SAT	0.02
289-sat-7x6.cnf	28	168	1554	80	177	SAT	0.01
lzw-based	32	588	2814	81	193	SAT	0.01
289-sat-11x4.cnf	29	176	1628	78	132	SAT	0
lzw-based	27	440	2420	78	132	SAT	0.02
289-sat-6x8.cnf	37	192	2016	305	1181	SAT	0.02
lzw-based	47	864	4032	318	1258	SAT	0.01

Table A.3 Random Instances

File	S	V	C	D	P	Sa	T
unif-k3-r4.26-v250-c1065-S427778075-026.UNKNOWN	15	250	1065	447563	17205942	UNSAT	5.8
lzw-based	15	255	1080	447563	17205942	UNSAT	5.93
unif-k3-r4.26-v250-c1065-S534787376-095.UNKNOWN	15	250	1065	167041	6185293	UNSAT	1.99
lzw-based	16	260	1095	167041	6185292	UNSAT	2
unif-k3-r4.26-v250-c1065-S931339469-052.UNKNOWN	15	250	1065	147975	5399172	UNSAT	1.74
lzw-based	15	251	1068	147975	5399172	UNSAT	1.74
unif-k3-r4.26-v250-c1065-S984689729-080.UNKNOWN	15	250	1065	150443	5603139	UNSAT	1.77
lzw-based	16	263	1104	185485	6928751	UNSAT	2.23
unif-k3-r4.26-v250-c1065-S1397126856-076.UNKNOWN	15	250	1065	247421	9301302	UNSAT	3.05
lzw-based	16	262	1101	247421	9301302	UNSAT	3.04
unif-k3-r4.26-v250-c1065-S1822479556-014.UNKNOWN	15	250	1065	69830	2576522	UNSAT	0.82
lzw-based	15	257	1086	69830	2576522	UNSAT	0.8
unif-k3-r4.26-v300-c1278-S768688898-014.UNKNOWN	18	300	1278	802379	33159731	UNSAT	11.92
lzw-based	19	312	1314	802379	33159732	UNSAT	12.39
unif-k3-r4.26-v300-c1278-S862932513-073.UNKNOWN	18	300	1278	823819	34480611	UNSAT	12.72
lzw-based	19	316	1326	823819	34480617	UNSAT	12.56
unif-k3-r4.26-v300-c1278-S1194590195-083.UNKNOWN	18	300	1278	1111820	47128598	UNSAT	17.9
lzw-based	19	306	1296	1227657	52108629	UNSAT	19.93