Algorithm Analysis for Satisfiability problem

Adriano Pereira Jacques Fux
{adrianoc, jacfux}@dcc.ufmg.br

Department of Computer Science
Federal University of Minas Gerais
Belo Horizonte – Minas Gerais – Brazil

ABSTRACT

Satisfiability (SAT) is a pivotal problem in Computer Science with numerous applications that range from microprocessor verification to FPGA layout. A one-million dollar prize is offered by the Clay Institute for a complete polynomial time SAT solver or a proof that such an algorithm does not exist (the P-vs-NP problem). Additionally, industrial applications motivate intensive research in SAT algorithms that quickly solve real-life instances. The fundamental framework for state-of-the-art SAT algorithms was laid out in the 1960s, but a number of recent improvements in algorithms and implementation techniques have led to performance breakthroughs. In this work we study and analyze Davis-Putnam, Grasp, Sato, and Zchaff, algorithms used to solve the satisfiability problem. In order to validate the work we present several important case studies and analyze the empirical experiments.

1. INTRODUCTION

The Satisfiability (SAT) problem, the problem of deciding whether a given Boolean formula is satisfiable, is one of the well-known NP-complete problems which have been identified as a central to a number of areas in computing theory. Its generic form can be described as a boolean equation where we should find a group of values (0 or 1 - false or true) for the variables that turn the result 1, true. Besides, in case that group of values doesn't exist, it should be pointed. An instance of a sat formula it is a formula that has three components:

- A group of n variable : $x_1, x_2, x_3, \ldots, x_n$.
- A group of literal which is a variable ($Q = x$) or the negation of a variable ($Q = \bar{x}$).
- A group of different clauses: $C_1, C_2, C_3, \ldots, C_m$ that consist of just literal combined by the $\vee$ (or) logic connective.

The objective of a SAT problem is to determine a value assign that turns the following Conjunctive Normal Form (CNF) true: $C_1 \wedge C_2 \wedge \ldots \wedge C_m$ where $\wedge$ is a logical and connective . It is also possible to find a problem SAT that doesn't have a solution, therefore an algorithm to be complete needs to find the solution in case it exists or prove that the expression in subject doesn't have a solution. One way of solving the problems of formal verification is evaluating the boolean equations generated through an SAT solver algorithm. In this paper we present some algorithms to deal with SAT problems.

This paper is organized as follows. The Section 2 presents some known heuristics for the satisfiability problem and introduce important concepts about it. Section 3 describes some case studies and its analysis. Section 4 show related works, and Section 5 presents some conclusions.

2. HEURISTICS

Over the years, a large number of algorithms have been proposal for SAT problem. Many of the discrete constrained algorithms eliminate one variable at a time by making repeated use of resolution. Algorithms for solving SAT can be divided in 2 classes: complete (exact) algorithms and incomplete (inexact) algorithms. Complete algorithms can be thought of algorithms that scan all the search space for a solution. The classical algorithm for exact solving SAT problems is the Davis-Putnam (DP) algorithm [3]. It solves the problem by recursively simplifying CNF by detecting conditions for variables, splitting up the problem into independent sub-problems and assigning variables. SAT is a NP-complete problem which means that solving large problems is computational infeasible when using exact algorithms. Therefore procedures called Stochastic Local Search Algorithms can be used. These inexact algorithms can not guarantee that a solution is found (at least in finite time), but the use of random search and heuristics can often solve problems that could not be solved using exact algorithms.

The classic technique for solving SAT problems in CNF was proposed by Davis and Putnam in 1960 [3]. Unlike trying to enumerate each possible combination of values for the variables, that algorithm, showed in Figure 1, uses search backtrack to reduce the search space. An unbound variable is set one way, and the function is analyzed to determine the implication on all other inputs. Variables are assigned in a specific order, and backtracking is used to explore the entire such space.

There have been some successful extensions to the basic DP algorithm:

- A lot of branching rules were proposed each aiming at accelerating the search under differ-

```
Davis-Putnam( CNF f ) {
  if f is empty return SATISFIABLE
  if □ ∈ f return UNSATISFIABLE
  /* unit propagation */
  if ∃c ∈ f with c = {l})
    then satisfy l and simplify f to f'
         return Davis-Putnam( f' )
    else
         /* branching rule */
         select unassigned variable x_i and an
         assignment φ_i(x_i) = b;
         simplify f to f';
         if Davis-Putnam( f' ) == SATISFIABLE
           return SATISFIABLE
         else
           change assignment of x_i to φ_i(x) = ¬b;
           simplify f to f'';
           return Davis-Putnam( f'' )
}
```

Figure 1: Davis-Putnam Algorithm

ent preferences (e.g., eliminating small clauses, and weighting variables that occur often as positive as well as negative literal).

- Conflict analysis is used to analyze contradictions occurring during the search. As a result of the conflict analysis one or more additional clauses are learned, so this issue is called Conflict Learning [10]. Learning clauses from a conflict prevents the algorithm in the future to search parts of the search space that will lead to the same conflict.
- Non-chronological backtracking is an issue closely related to conflict analysis. In the original DP, when testing variable assignment $x = c$ fails, this assignment is automatically flipped $x = \neg c$. But when a conflict occurred and a clause is learned, it often still be unsolved when flipping the last decision assignment. Instead, we can identify the last assignment contained in the learned clause and backtrack to this assignment. Thereby, several decision levels can be skipped, leading to non-chronological backtrack behavior of the SAT algorithm.
- Restarts abort the search after exceeding some parameter threshold, and starts it from scratch. There are several possibilities for the parameter threshold, as the number of backtracks, and the number of decisions. Restarts prevent the SAT algorithm to lose in non-relevant parts of the search space. Conflict learning affects the concept of restarts once new clauses are added during two restarts, modifying the clause database. In general, branching rules will then behave different, leading to a varying exploration of the search space.

In the next subsections we present Sato (Satisfiability Testing Optimized) [16], Grasp (Generic seaRch Algorithm for the Satisfiability Problem) [13], that uses some basic extensions, and Zchaff [10] which incorporates all the extensions presented.

2.1 The Sato Algorithm

One of the major motivations to develop Sato was to attack open Latin square problems. Given a set S, a Latin square indexed by S is an $|S|$x$|S|$ array such that each row and each column of the array is a permutation of the elements in S. $|S|$ is called the order of the Latin square. The goal of this heuristic is to integrate two known techniques: splitting rules and conflict analysis. [16] presents the source code in C and explanation of Sato. A parallel version of Sato is described in [17].

The kernel of this heuristic is a new unit-propagation that needs amortized linear time for unit resolution but amortized constant time for unit subsumption [18]. It uses a simple data structure and achieve better results than DP, adopting the same idea of unit propagation, illustrated by Figure 2.

```
proc unit_propagation ()
  /* initiation */
  OK := true
  UNITS := empty
  for c: clause in CLAUSES
    if head_index(c) = tail_index(c) do
      L: literal := ith_literal(c,head_index(c))
      stack_push(L, UNITS)
    end for
  /* main loop */
  while (OK and not stack_empty?(UNITS)) do
    /* handle unit clauses */
    L: literal := stack_pop(UNITS)
    if truth_value(L) = true
      then continue
      else if truth_value(L) = false
      then OK := false
      else if positive(L)
      then { truth_value(L) := true
                propagate_true_value(L) }
      else { truth_value(-L) := false
                propagate_false_value(-L) }
    end while
  if (OK) then write("satisfiable if Horn")
    else write("unsatisfiable")
end proc
```

Figure 2: Sato Unit Propagation

2.2 The Grasp Algorithm

The conflict analysis procedure previously explained has been incorporated in GRASP. This procedure is tightly coupled with Boolean Constraint Propagation (BCP) and the causes of conflicts do not need necessarily to correspond to decision assignments. Grasp employs techniques to prune the search by analyzing the implication structure generated by BCP.

This algorithm uses the notation $x = y@d$ to show that "x becomes equal to y at decision level d" [13]. We can classify this approach as a dynamic learning search mechanism based on diagnosing the causes of conflict. It considers the occurrence of a conflict, which is unavoidable for an unsatisfiable instance unless the formula is complete.

As an opportunity to learn from the mistake that led to the conflict and introduces additional implicates to the clause database only when it stumbles.

For example, consider a formula Θ containing the clause $\omega = x + \neg y$ and assume $y = 1$. Note that for any satisfying assignment Θ, ω must have $x = 1$, so $y = 1$ implies that $x = 1$ due ω. Figure 4 shows the Grasp algorithm.

```
// Global variables:      Clause database φ
//                        Variable assignment A
// Return value:          FAILURE or SUCCESS
// Auxiliary variables:   Backtracking level β
//
GRASP()
{
    return (Search (0, β) != SUCCESS) ?
        FAILURE : SUCCESS;
}

// Input argument:        Current decision level d
// Output argument:       Backtracking level β
// Return value:          CONFLICT or SUCCESS
//
Search (d, &β)
{
    if (Decide (d) == SUCCESS)
        return SUCCESS;
    while (TRUE) {
        if (Deduce (d) != CONFLICT) {
            if (Search (d + 1, β) == SUCCESS)
                return SUCCESS;
            else if (β != d) {
                Erase(); return CONFLICT;
            }
        }
        if (Diagnose (d, β) == CONFLICT) {
            Ease(); return CONFLICT;
        }
        Erase();
    }
}

Diagnose (d, &β)
{
    ω_C(κ) = Conflict_Induced_Clause(); // From (4)
    Update_Clause_Database (ω_C(κ));
    β = Compute_Max_Level();            // From (7)
    if (β != d) {
        add new conflict vertex κ to I;
        record A(κ);
        return CONFLICT;
    }
    return SUCCESS;
}
```

Figure 3: Grasp Algorithm

The function Decide chooses a decision assignment at each stage of the search process. Decision procedures are commonly based on heuristic knowledge. Then we can see the function Deduce which implements BCP (implicitly) and maintains the assignment. To identify the causes of conflicts we use the function Diagnose. If some assignment is not satisfied, the function Erase deletes the actual decision level.

2.3 The Zchaff Algorithm

Now, we present another heuristic that uses all the extensions described in Section 2. Zchaff is a new version of the first one implemented called Chaff [10]. We can see below the procedures used in this heuristic.

Decide next branch: this function implements the branching rule. Each variable x maintains *literal count* counters for counting the occurrences of x and $\neg x$ in clauses contained in the clause database. Additionally,*scores* are maintained for x and $\neg x$.

Deduce: this function incorporates the *unit propagation* rule in an iterated manner, and this process is commonly called Boolean Constraint Propagation (BCP). The main function of *deduce* is to iteratively pop an implication from the implication queue and check all unsatisfied clauses if they are now satisfied or if they are becoming a unit clause.

```
zChaff() {
  while ( 1 ) {
    if( decide_next_branch() ) {
      while ( deduce() == CONFLICT ) {
        blevel = analyze_conflicts()
        if( blevel == 0 ) {
          return UNSATISFIABLE
        }
        else {
          back_track( blevel )
        }
      }
    }
    else {
      return SATISFIABLE
    }
  }
}
```

Figure 4: Zchaff Algorithm

Analyze conflicts: this function analyzes the conflict occurred in the deduce function. A conflict occurs when a contradictory variable assignment is deduced. For example, suppose that x is already assigned with 1 and within deduce a unit clause $c = \neg x$ is identified, so we can conclude that $x = 0$ which is a contradiction.

2.4 Data Structures

Most backtrack SAT solvers are conceptually composed of three main stages: the decision stage; the deduction stage; and the diagnosis state, that we had already presented. Recent SAT solvers are also characterized by using very efficient data structures, intended to reduce CPU time. Data structures can be organized in two main categories: data structures based on adjacency lists, and lazy data structures [13]. We present data structures for each algorithm, pointing out the differences between them.

Grasp is a SAT solver that utilizes adjacency lists. A clause is unsatisfied if the unsatisfied literal counter equals the number of literals; it is satisfied if the counter of satisfied literals is greater than one; it is unit if the unsatisfied literal counter equals the number of literals minus one, and there is still one unassigned literal. When a clause is declared unit, the list of literals is traversed to identify which literal needs to be assigned.

The first lazy data structure proposed for SAT was the Head/Tail (H/T) data structure used in the Sato. It associates two references with each clause. Initially the head reference points to the first literal, and the tail reference points to the last literal. Each time a literal pointed to by either the head or tail reference is assigned, a new unassigned literal is searched for. In case an unassigned literal is identified, it becomes the new head (or tail) reference, and a *new* reference is created and associated with the literal's variable.

Zchaff proposes a new data structure that solves some of the problems posed by H/T. In the Watched Literals, as H/T lists, two references are associated with each clause. However, in contrast with H/T lists, there is no order relation between the two references. The lack of order has the key advantage that no literal needs to be updated when backtracking takes place.

2.5 Algorithm Analysis

We presented three different heuristics based on Davis-Putnam algorithm. We can see clearly that algorithms are similar, but their behaviour differs, as described in Section 3. These differences show us that small improvements could increase the capacity to solve SAT problems.

3. EXPERIMENTAL RESULTS

In order to evaluate the algorithms we execute many different tests, comparing the results obtained for each one. This section describes the case studies we adopt in our experiments, the empirical results and their analysis.

3.1 Graph Coloring Problem - GCP

The first case study is the GCP, a well-known combinatorial problem from graph theory. Given a graph $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ is the set of vertices and E is the set of edges connecting the vertices, find a coloring $C : V \to N$, such that connected vertices always have different colors. There are two variants of this problem: in the optimization variant, the goal is to find a coloring with a minimal number of colors, whereas in the decision variant, the question is to decide whether for a particular number of colors, a coloring of the given graph exists. The two variants are tightly related, as optimal colorings can be found by solving a series of decision problems, using, for instance, binary search to determine the minimal number of colors. In the context of SAT-encoded Graph Coloring problems, we focus on the decision variant.

Following earlier approaches in literature, we use a straightforward strategy for encoding GCP instances into SAT: each assignment of a color to a single vertex is represented by a propositional variable; each coloring constraint (edge of the graph) is represented by a set of clauses ensuring that the corresponding vertices have different colors, and two additional sets of clauses ensure that valid SAT assignments assign exactly one color to each vertex.

We generate this test using a flat graph generator. It consists of 3-colorable flat graph and the connectivity of which is adjusted in such a way that it is very hard to solve (in average) for graph coloring algorithms like the Brelaz heuristic [14]. The tests showed that the generated instance is satisfiable.

The Table 1 shows the results of the GCP case study for the three algorithms studied.

SATISFIED	Algorithm		
	Grasp	Sato	Zchaff
Time (sec)	19.53	0.16	0.21
Number of Decisions	1631	2304	2237
#Clauses / #Literals	2237 / 4674		

Table 1: Graph Coloring Problem

It shows that the Sato algorithm solves the graph coloring problem faster than Grasp and Zchaff, but Zchaff has a good result either. This result has already been expected, once Sato is very good to solve satisfiability problem that can be modeled using Latin squares technique [7].

3.2 All-Interval Series Problem - AIS

The AIS problem is an arithmetic problem [11] that was inspired by a well-known problem occurring in serial musical composition, which, in its original form, can be stated in the following way: given the twelve standard pitch-classes $(c, c\#, d, ...)$, represented by numbers $0, 1, 2, ..., 11$, find a series in which each pitch-class occurs exactly once and in which the musical intervals between neighboring notes cover the full set of intervals from the minor second (1 semitone) to the major seventh (11 semitones).Therefore for each of these intervals, there is a pair of neighboring pitch-classes in the series, between which this interval appears. The problem of finding such series can be easily formulated as an instance of a more general arithmetic problem in Z_n, the set of integer residues modulo n: for given positive integer n, find a vector $s = (s_1, ..., s_n)$, such that s is a permutation of $Z_n = \{0, 1, ..., n-1\}$; and the interval vector $v = (|s_2 - s_1|, |s_3 - s_2|, ...|s_n - s_{n-1}|)$ is a permutation of $Z_n - 0 = \{1, 2, ..., n-1\}$. The problem of finding such a series is called the All-Interval Series Problem of size n.

The AIS can be represented as a Constraint Satisfaction Problem (CSP) in a rather straightforward way. Each element of s and v is represented as a variable; the domains of these variables are Z_n and $Z_n - 0$, respectively; and the constraint relations encode the two conditions from the definition previously given. From this CSP formulation, instances of the all-interval series problem can be easily transformed into a SAT instance by using essentially the same encoding as for the Graph Coloring Problem, where each assignment of a value to a single CSP variable is represented by a propositional variable; each of the three sets of constraints from the definition above is represented by a set of clauses; and two additional sets of clauses ensure that valid SAT assignments assign exactly one value to each CSP variable.

Our benchmark set provided to this case study contains the two (satisfiable) SAT instances for $n = 8$ and $n = 10$. The larger problem instance has found to be extremely hard to solve for state-of-the-art SLS-based SAT algorithms [11].

The Table 2 shows the results of the AIS case study of size $n = 8$, and the Table 3 for the same problem of size $n = 10$.

SATISFIED	Algorithm		
	Grasp	Sato	Zchaff
Time (sec)	0.04	0.01	0.02
Number of Decisions	90	32	520
#Clauses / #Literals	1520 / 3515		

Table 2: All-Interval Series - size 8

SATISFIED	Algorithm		
	Grasp	Sato	Zchaff
Time (sec)	6.16	0.02	1.1
Number of Decisions	1940	346	7778
#Clauses / #Literals	3151 / 7255		

Table 3: All-Interval Series - size 10

Similar to the graph coloring problem, here the Sato algorithm gets the best result in both instance, more significative in the problem of size 10, where it was 55 times faster than Zchaff and 308 times than Grasp.

3.3 Blocks World Planning Problem - BWP

The BWP is a very well-known problem domain in Artificial Intelligence (AI) research. The general scenario in BWP comprises a number of blocks and a table. The blocks can be piled onto each other, where the down most block of a pile is always on the table. For the benchmark problems, taken from [12], there is only one operator which moves the top block of a pile to the top of another pile or onto the table. Given an initial and a goal configuration of blocks, the problem is to find a sequence of operators which, when applied to the initial configuration, leads to the goal situation. Such a sequence is called a (linear) plan. Blocks can only be moved when they are clear, i.e., no other block is piled on top of them, and they can be only moved on top of blocks which are clear or onto the table. If these conditions are satisfied, the move operator always succeeds. There is an optimization and a decision variant of the BWP: in the optimization variant, the goal is to find a shortest plan, whereas in the decision variant, the question is to decide whether a plan of a given length exists. The two variants are tightly related, as shortest plans can be found by solving a series of decision problems. SAT-based approaches to Blocks World Planning typically focus on the decision variant.

A linear encoding strategy was used for translating BWP instances into CNF. Details on the SAT encoding used to generate the benchmark problems can be found in [12]. The Table 4 shows the results of the BWP for 9 blocks, and the Table 5 for the same problem for 11 blocks.

SATISFIED	Algorithm		
	Grasp	Sato	Zchaff
Time (sec)	0.12	0.03	0.01
Number of Decisions	22	28	42
#Clauses / #Literals	4675 / 10809		

Table 4: Blocks World Planning Problem

SATISFIED	Algorithm		
	Grasp	Sato	Zchaff
Time (sec)	0.88	0.14	0.03
Number of Decisions	77	198	254
#Clauses / #Literals	13772 / 31767		

Table 5: Blocks World Planning Problem

Zchaff gets the best result for each instance, getting 4.66 times faster than Sato and 29.3 times than Grasp.

3.4 Logistics Planning Problem - LP

In the logistics planning domain, packages have to be moved between different locations in different cities. Within cities, packages are carried by trucks while between cities they are flown in planes. Both, trucks and airplanes are of limited capacity. The problem involves 3 operators (load, unload, move) and two state predicates (in,at). The initial and goal state specify locations for all packages, trucks, and planes; the plans allow multiple actions to be executed simultaneously, as long as no conflicts arise from their preconditions and effects. Like in the Blocks World Planing problem, there is an optimization and a decision variant of the Logistics Planning problem: In the optimization variant, the goal is to find a shortest plan, whereas in the decision variant, the question is to decide whether a plan of a given length exists. Again, the two variants are tightly related, as shortest plans can be found by solving a series of decision problems. SAT-based approaches to Logistics Planning typically focus on the decision variant.

A state-based encoding strategy was used for translating Logistics instances into CNF. In this encoding, only the state predicates are explicitly represented by propositional variables, whereas the operators are only implicitly represented by constraints on transitions of the state predicates.

The SAT encoding used for generating the benchmark problems relies critically on two important techniques for reducing the size of the CNF: operator splitting and simple propositional reductions [12]. Operator splitting replaces a predicate which take three or more arguments by a number of binary predicates. This reduces the number of propositional variables for the given problem from $O(k*n^3)$ to $O(k*n^2)$ where n is the number of blocks and k the number of plan steps.

The Table 6 shows the results of the LP for 8 packages and 5 number of plan steps, and the Table 7 for 5 packages and 13 number of plan steps .

SATISFIED	Algorithm		
	Grasp	Sato	Zchaff
Time (sec)	7.65	0.26	0.03
Number of Decisions	2415	3388	4086
#Clauses / #Literals	6718 / 17915		

Table 6: Logistics Planning Problem

SATISFIED	Algorithm		
	Grasp	Sato	Zchaff
Time (sec)	8.89	62.06	0.05
Number of Decisions	2743	899536	6160
#Clauses / #Literals	7301 / 19024		

Table 7: Logistics Planning Problem

The analysis of Tables 6 and 7 shows that Zchaff gets the best result for each instance, reaching 1241 times faster than Sato and 178 times than Grasp in the second instance (Table 7). Note that in the first instance of this problem Sato gets better result than Grasp, but the opposite occurs in the second instance. This last observation can be explained by the number of decision demanded by Sato in the second instance (more than 899 thousand).

3.5 A Pipelined DLX Processor

Here we validate the algorithms through the Boolean condition for the correctness of a single-issue pipelined DLX processor [9] with multi cycle functional units, exceptions, and branch prediction, as formally verified in [15]. The evaluation of the same formula with BDDs [2, 1] takes 6 seconds of CPU time. This benchmark suite is distributed by Carnegie Mellon University.

The Table 8 shows the results of this experiment for each studied algorithm. Its analysis allows to conclude that Zchaff algorithm is very efficient testing satisfiability of processor's architecture. It gets almost 100 times faster than Grasp. It is important to point out that Sato does not solve this case study in 12000 seconds, so we decided to abort its execution.

UNSATISFIED	Algorithm		
	Grasp	Sato	Zchaff
Time (sec)	9.93	> 12000	0.10
Number of Decisions	1795	-	3624
#Clauses / #Literals	3725 / 10045		

Table 8: A Pipelined DLX Processor

3.6 Pigeon Hole Problem - PH

The PH problem asks whether it is possible to place $n+1$ pigeons in n holes without two pigeons being in the same hole.

The SAT encoding of this problem is very straightforward. For each pigeon i we have a variable x_{ij} which means that pigeon i is placed in hole j. Then we have $n+1$ clauses which say that a pigeon has to be placed in some hole. Then for each hole we have a set of clauses ensuring that only one single pigeon is placed into that hole. This encoding leads to in total $n*(n+1)$ propositional variables and to $(n+1)+n*(n*(n+1)/2)$ clauses.

The Table 9 shows the results for this problem. It shows that Zchaff is the best solver for the satisfiability of this PH. Zchaff solves the problem in only 0.44 seconds,3 times faster than Sato, and more than 7 times faster than Grasp.

UNSATISFIED	Algorithm		
	Grasp	Sato	Zchaff
Time (sec)	3.15	1.32	0.44
Number of Decisions	1163	41581	3450
#Clauses / #Literals	297 / 648		

Table 9: Pigeon Hole Problem

3.7 "Morphed" Graph Coloring Problem

An interesting class of Graph Coloring Problems is obtained by morphing regular ring lattices with random graphs [4]: A (type B) p-morph of two graphs $A=(V,E_1)$ and $B=(V,E_2)$ is a graph $C=(V,E)$ where E contains all the edges common to A and B, a fraction p of the edges from E_1-E_2 (the remaining edges of A), and a fraction $1-p$ of the edges from E_2-E_1. The test-set considered here are obtained by morphing regular ring lattices, where the vertices are ordered cyclically and each vertex is connected to its k closest in this ordering, and random graphs from the well-known class Gnm [5]. The morphing ratio p controls the amount of structure in the problem instances, and by varying p, the behaviour of various algorithms depending on the degree of structure and randomness can be studied [4].

Following earlier approaches in literature, we use a straightforward strategy for encoding GCP instances into SAT: each assignment of a color to a single vertex is represented by a propositional variable; each coloring constraint (edge of the graph) is represented by a set of clauses ensuring that the corresponding vertices have different colors, and two additional sets of clauses ensure that valid SAT assignments assign exactly one color to each vertex.

The test-set was generated using a generator program. All graphs have 100 vertices and 400 edges, the regular ring lattice used for morphing connects each vertex to its 8 nearest neighbors in the cyclic ordering. For $0 < p < 1$, the chromatic number of the graphs thus obtained varies. Using a graph coloring program, we filtered out all instances with $c \neq 5$ colors (the regular ring lattice, i.e., the morphed graph for $p = 0$, has chromatic number 5). The instance was satisfiable.

The Table 10 shows the results for this problem. It shows that Zchaff is the best solver for this case study, but Sato achieves a good result either. Note this problem is very similar to the first case study presented, but the result was the opposite. As expected, Grasp is not a good solver for this kind of problem (it spent 169 times the execution of Zchaff).

SATISFIED	Algorithm		
	Grasp	Sato	Zchaff
Time (sec)	1.69	0.02	0.01
Number of Decisions	1682	40	741
#Clauses / #Literals	3100 / 6500		

Table 10: "Morphed" Graph Coloring

4. RELATED WORK

The key to the efficiency of the model checking algorithms is the use of *binary decision diagrams* [1, 2] to represent the labeled state-transition graph and to verify if a timing property is true or not. Model checkers can exhaustively check the state space of systems with more than 10^{30} states in a few seconds. Binary decision diagrams (BDDs) consist of a similar approach that can be used to solve the satisfiability problem. BDDs are a canonical representation for boolean formulas [1]. A BDD is obtained from a binary decision tree by merging identical subtrees and eliminating nodes with identical left and right siblings. The resulting structure is a directed acyclic graph rather than a tree which allows nodes and substructures to be shared. The internal vertices are labeled with boolean variables. Leaves are labeled with 0 and 1. Canonicity is ensured placing a strict total order on the variables as one traverses a path from "root" to "leaf". The edges are labeled with 0 or 1. For every truth assignment there is a corresponding path in the BDD such that at vertex x, the edge labeled 1 is taken if the assignment sets x to 1; otherwise, the edge labeled 0 is taken. If the path end in the "leaf" labeled 0 then the formula will not be satisfied, conversely, if it end in the "leaf" labeled 1 then the formula will be satisfied - the assignment made to each variable satisfies the formula. Figure 5 illustrates the BDD for the boolean formula $(a \wedge b) \vee (c \wedge d)$.

The BDD is efficient to search for optimal solutions for arbitrarily complicated Boolean expressions. In

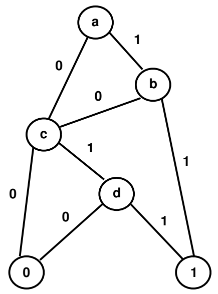

Figure 5: BDD for $(a \wedge b) \vee (c \wedge d)$

VLSI circuit design, many practical problems require the enumeration of all possible assignments for a given Boolean formula. The best assignment that yields the minimum cost (e.g., minimal circuit structure, minimum chip area, and maximum circuit speed) is then selected from these possible assignments. Since most algorithms for satisfiability testing are designed for finding one truth assignment, they are impractical for selecting an optimal assignment. BDDs are very useful in such situations, since a simple and incremental enumeration of all possible paths from the root node to the leaf node 1 yields all the truth assignments. Thus, once the BDD for a Boolean function has been constructed, it is straightforward to enumerate all assignments or find an optimal solution. The BDD method can effectively handle small and medium size formulas. For larger size formulas, a partitioning into a set of smaller sub-formulas before applying the BDD algorithms has been suggested. This approach works well for asynchronous computer circuit design problems [8].

Another very good SAT-solver is BerkMin [6]. It inherits such features of GRASP, SATO, and ZChaff as clause recording, fast BCP, restarts, and conflict clause "aging". At the same time BerkMin introduces a new decision making procedure and a new method of clause database management. In [6], the authors experimentally compare BerkMin with ZChaff, the leader among SAT-solvers used in the EDA domain. Experiments show that BerkMin is more robust than ZChaff. BerkMin solved all the instances we used in experiments including very large CNFs from a microprocessor verification benchmark suite. On the other hand, ZChaff was not able to complete some instances even with the timeout limit of 16 hours.

5. CONCLUSION

This paper describes SAT solvers and compare the number of decisions and time of each heuristics. It's clearly that Zchaff is more robust and solve the problems without abortion, which is not true for Sato. However we present some results showing that Sato solves some problems faster than others, which is verified by the knowledge of database structure and mostly because of the benchmark used.

Indeed GRASP presents the worst performance, however it is an important approach dealing with SAT problem because once it was the first one that used the backtrack search trying to solve SAT problem.

In general, the heuristics presented are very similar because they are based on Davis-Putnam algorithm. We verified that small modifications can vary the performance of the algoritms and each one has own merit for specific class of problems.

In this paper we see that a lot of problems can be formulated as a SAT problem, but sometimes is not easy. For example, if we have a difficult problem,try to write this problem in a conjunctive normal formula and maybe you will reach good results. Our final conclusion is that Zchaff is the best algorithm analyzed in this paper.

6. REFERENCES

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[3] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[4] I. P. Gent, H. H. Hoos, P. Prosser, and T. Walsh. Morphing: Combining structure and randomness. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)*, pages 654–660, Orlando, Florida, 1999.

[5] I. P. Gent, H. H. Hoos, P. Prosser, and T. Walsh. Morphing: Combining structure and randomness. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)*, pages 654–660, Orlando, Florida, 1999.

[6] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, Mar. 2002.

[7] C. Gomes and D. Shmoys. Completing quasigroups or latin squares: A structured graph coloring problem. In *Proceedings of the Computational Symposium on Graph Coloring and Extensions*, Lecture Notes in Computer Science, Ithaca, New York, 2002. Springer-Verlag.

[8] J. Gu and R. Puri. Asynchronous circuit synthesis with boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 14(8):961–973, 1995.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd edition.* Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[10] M. Herbstritt. zchaff: Modifications and extensions. Technical Report TR-188, Albert-Ludwigs University, July 2003.

[11] H. H. Hoos and K. O'Neill. Stochastic local search methods for dynamic SAT- an initial investigation. Technical Report TR-00-01, 1, 2000.

[12] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In H. Shrobe and T. Senator, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1194–1201, Menlo Park, California, 1996. AAAI Press.

[13] J. P. Marques-Silva and K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[14] Tad Hogg. Refining the phase transition in combinatorial search. *Artificial Intelligence*, (81):127–154, 1996.

[15] M. N. Velev and R. E. Bryant. Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction. In *Design Automation Conference*, pages 112–117, 2000.

[16] H. Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.

[17] H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4):543–560, 1996.

[18] H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale (Florida USA), 1996.