Published in: *Proceeding of 6th International Workshop on Boolean Problems (IWSBP), Freiberg, Germany, Sept. 2004, pp. 257-264*

Strengthening Semantic-tree Method in evaluating QBFs

Mohammad GhasemZadeh, Volker Klotz and Christoph Meinel
FB-IV Informatik, University of Trier, Germany
{GhasemZadeh, klotz, meinel}@TI.Uni-Trier.DE

Abstract

The semantic-tree approach is the most robust method for deciding QBFs. This method is very similar to the DPLL algorithm for evaluating SAT instances. In this paper we show how memorization can be embedded to this method to get better results. In other words we present an algorithm for evaluating QBFs, which is based on an adopted version of semantic-tree method and ZDDs (which are variants of BDDs). The capability of ZDDs in storing sets of subsets efficiently, enabled us to store the formula very compact and led us to implement the search algorithm in such a way that we could store and reuse the results of all already solved subformulas. We call this idea: 'strengthening semantic-tree method by memorization'. This idea along some other techniques, enabled our algorithm to solve a bunch of the standard QBF benchmark problems faster than the best existing QBF solvers.

Keywords: DPLL, Zero-Suppressed Binary Decision Diagram (ZDD), Quantified Boolean Formulae (QBF), Satisfiability, QSAT, Memorization, Dynamic Programming.

1 Introduction

Many computational problems such as constraint satisfaction problems, many problems in graph theory and forms of planning can be formulated easily in propositional logic and be solved as instances of SAT problem. Theoretical analysis has showed that some forms of reasoning such as nonmonotonic reasoning, reasoning about knowledge, and STRIPS-like planning, have computational complexity higher than the complexity of SAT problems. This forms can be formulated by quantified Boolean formulas and be solved as instances of QSAT problems (e.g., [9]).

Quantified Boolean formula satisfiability (QSAT) is a generalization of the SAT problem. QBFs give the possibility to represent many classes of formulas more concise than conventional propositional formulas. This additional conciseness lifts the complexity of evaluating QBFs to PSPACE-complete. However SAT and QSAT have a close connection, and this is why some recent QBF solvers[16, 12, 11, 10] are extensions of the Davis-Logemann-Loveland procedure [8].

ZDDs are variants of BDDs. While BDDs are better suited for representing Boolean functions, ZDDs are better for representing sets of subsets. A CNF formula can be viewed as a set of subsets. In our QSAT solver, we represent the body of the QBF formula in a ZDD, then we employ an adopted version of the "semantic tree method in evaluating a QBF formula" algorithm to search its satisfiability. It benefits from an adopted unit resolution which is very fast thanks to the data structure holding the formula. In addition it stores all already solved subformulas along their solutions and reuses them to avoid resolving same subproblems. We refer to this idea as 'strengthening semantic-tree method by memorization'. Sometimes the splitting operation generates two subproblems which are equal. With ZDDs it is very easy to compare and discover the equality, therefore our algorithm can easily prevent to solve both cases when it is not necessary. In fact since ZDDs are canonical, such a test can be performed with only one pointer comparison.

There are some benchmark problems which are known to be hard for DPLL (sematic-tree) algorithms. Our algorithm which we reference it by 'ZQSAT', is also a sematic-tree based algorithm, but it manages to solve those instances very fast. We believe, this superiority has obtained partly due to embedding memorization (tabulation/dynamic programming) to the search procedure and partly, thanks to ZDDs, due to fast simplification and unit/mono-resolution operations. We evaluated our algorithm over different known benchmarks presented in QBFLIB (Quantified Boolean Formula satisfiability LIBrary) [15]. We run FZQSAT along best existing QBF-Solvers such as QuBE [11], Decide [16], Semprop [12] and QSolve [10]

2 Preliminaries

2.1 Quantified Boolean Formulas

Quantified Boolean formula is an extension of propositional formula (also known as Boolean formula). A Boolean formula like $(x \vee (\neg y \rightarrow z))$ is a formula built up from Boolean variables and Boolean operators like conjunction, disjunction, negation and so on. In quantified Boolean formulas, quantifiers may also occur in the formula, like in $\exists x(x \wedge \forall y(y \vee \neg z))$. The $\exists$ symbol is called existential quantifier and the $\forall$ symbol is called universal quantifier. A number of normal forms are known for each of the above families. Among them, the *prenex normal form* and the *conjunctive normal form* (*CNF*) are important in QSAT and SAT problems.

A QBF Φ is in prenex normal form, if it is in the form: $\Phi = Q_1x_1 \ldots Q_nx_n\phi$, where $Q_i \in \{\forall, \exists\}$ and ϕ is a propositional formula over variables $x_1, \ldots, x_n$. The expression $Q_1x_1 \ldots Q_nx_n$ is called the prefix and ϕ the matrix of Φ. Sometimes we simply write $\Phi = Q.\phi$. A literal x or $\neg x$ is called a universal literal, if the variable x is bounded by a universal quantifier. Universal quantified variables are also denoted as $\forall - variables$. A clause containing only $\forall - literals$ is called universal clause or $\forall - disjunction$. Similar definitions exist for existential literals. QCNF denotes the class of QBF formulas with matrix in CNF. We remind that a propositional formula is in CNF form if it is a conjuction of disjunctions.(also known as product of sums - POS).

2.2 ZDDs, BDDs and the CUDD Package

Several years ago, Binary Decision Diagrams (BDDs) [4, 18, 13, 5] and their variants [3] entered the scene of computer science. Since that time, they have been used successfully in industrial CAD tools. In many applications, specially in problems involving sparce sets of subsets, the size of the BDD grows very fast, and causes inefficient processing. This problem can be solved by a variant of BDD, called ZDD (Zero suppressed Binary Decision Diagrams) [14, 1]. These diagrams are similar to BDDs with one of the underlying principles modified. While BDDs are better for representation of Boolean functions, ZDDs are better for representation of covers (set of subsets). We denote an internal node by $P(x, \Gamma_1, \Gamma_2)$ where x is the label of the node and Γ_1, Γ_2 are SubZDDs stand for its 'Then-child' and 'Else-child'. As an example, in Figure 1, the left diagram displays the ZDD representation for $S = \{\{a, b\}, \{a, c\}, \{c\}\}$, and the right diagram displays $F = ab + ac + c$, which is the characteristic function of S. We denote an internal node by $P(x, \Gamma_1, \Gamma_2)$ where x is the label of the node and Γ_1, Γ_2 are SubZDDs stand for its 'Then-child' and 'Else-child'. The size of a ZDD Γ is the number of its internal nodes denoted by $|\Gamma|$.

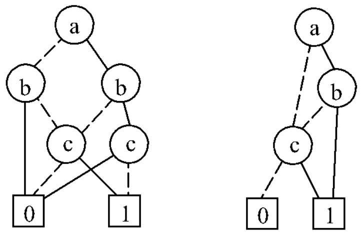


Figure 1: BDD versus ZDD.

CUDD-Colorado University Decision Diagram [17], is a package written in C for manipulating decision diagrams. The package provides a large set of operations on BDDs, ADDs, and ZDDs, and a large assortment of variable reordering methods.

2.3 The Semantic Tree Approach

This method is very similar to the well known DPLL algorithm. It iteratively splits the problem of deciding a formula of the form $Qx\Phi$ into two subproblems $\Phi[x = 1]$ and $\Phi[x = 0]$, then it decides according to the following rules:

- $\exists x\Phi$ is valid iff $\Phi[x = 1]$ or $\Phi[x = 0]$ is valid.
- $\forall x\Phi$ is valid iff $\Phi[x = 1]$ and $\Phi[x = 0]$ is valid.

In fact, this method searches the solution in a tree of variable assignments. Figure 2 displays the semantic tree for: $\Phi = \exists y_1 \forall x \exists y_2 \exists y_3 (C_1 \wedge C_2 \wedge C_3 \wedge C_4)$, where: $C_1 = (\neg y_1 \vee x \vee \neg y_2)$, $C_2 = (y_2 \vee \neg y_3)$, $C_3 = (y_2 \vee y_3)$ and $C_4 = (y_1 \vee \neg x \vee \neg y_2)$.

We can follow the tree and realize that Φ is invalid. Another interesting point can easily be seen in the tree. It is the duplication problem in semantic tree method. In other words, the same subproblem can appear two or more times. For a big QBF this situation can happen frequently. The superiority of our algorithm is recognizing these duplications and avoiding to examine them repeatedly.

3 Representation and Algorithm

3.1 Representation of a CNF Formula in a ZDD

A ZDD can be used to represent a set of subsets. Since each propositional CNF formula $\phi = c_1 \cdot c_1 \cdot \ldots \cdot c_n$ can be represented as a set of clauses $[\phi] = \{[c_1], \ldots, [c_n]\}$ where $[c_i] = \{l_1, \ldots, l_m\}$ and l_j is a literal in c_i, we can represent a CNF formula in a ZDD. In ZDDs, each path from the root to the 1-terminal corresponds to one clause of the set. In a path, if we pass through $x_i = 1$ (toward its 'Then-child'), then x_i exists in the clause, but if we pass through $x_i = 0$ (toward its 'Else-child') or we don't pass through x_i, then x_i does not exist in the clause.

In representing boolean functions, which often include positive and negative literals, we need to assign two successive ZDD indices to each variable, one index for positive and the next for its complemented form [7]. This idea has several benefits, among them we mention the possibility of detecting and removing the subsumed clauses [2, 7]. Figure 3 shows how this idea works for a small CNF formula.

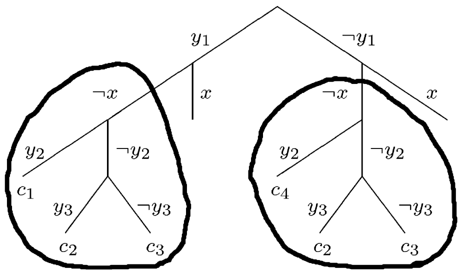
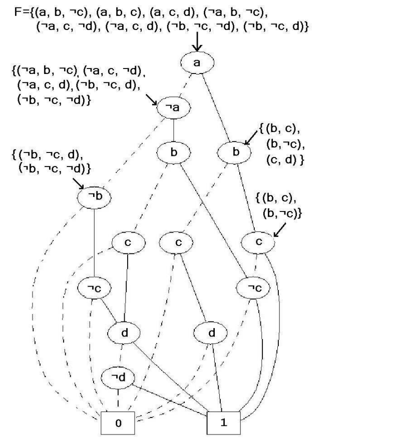


Figure 2: A semantic tree proof.



F={(a, b, ¬c), (a, b, c), (a, c, d), (¬a, b, ¬c), (¬a, c, ¬d), (¬a, c, d), (¬b, ¬c, ¬d), (¬b, ¬c, d)}
{(¬a, b, ¬c), (¬a, c, ¬d), (¬a, c, d), (¬b, ¬c, d), (¬b, ¬c, ¬d)}
{(b, c), (b,¬c), (c, d)}
{(¬b, ¬c, d), (¬b, ¬c, ¬d)}
{(b, c), (b,¬c)}
a
¬a
b
b
¬b
c
c
c
¬c
¬c
d
d
¬d
0
1

Figure 3: ZDD encoding of a CNF formula.

In evaluating QBFs, the freedom of variable selection is strongly restricted (in general, must respect the prefix order). The semantic-tree method processes the variables according to their oder in prefix of the formula. We also consider the same order for the literals in the ZDD representing the formula.

3.2 Benefits of using ZDDs

In Figure 3, we can also see another interesting characteristic of the ZDDs, that is, their possibility of sharing nodes and subgraphs. In fact each node in a ZDD stands for a (sub)function. In many situations, this property lets ZDDs to hold new (sub)functions with producing a few or no additional nodes. In our search procedure, after unit-mono-resolution and after the splitting step, new (sub)functions emerge. We noticed that many of this (sub)functions are the same, therefore we let ZQSAT to retain all already produced (sub)functions along their solutions, to prevent resolving same (sub)functions. This idea also helped ZQSAT to generate fewer function calls. In fact, after inserting this possibility, ZQSAT managed to solve the instances known to be hard for DPLL-based methods very fast (see Table 1).

Considering ZDDs as the data structure holding the formula, affects the search algorithm and its complexity considerably. In other words, operations like: detecting the unit clauses, detecting mono variables, performing the unit/mono resolution and detecting the SAT/UNSAT conditions depend strongly on the data structure holding the formula. Here we give some rules concerning above operations. These rules can be concluded from the basic properties known for QBFs, some lemmas presented in [6] and the properties of representing CNF clauses in a ZDD. Performing these operations with other data structures is often slower.

Suppose we have read the clauses and represented them in a ZDD, Γ then the following rules are applicable when we are examining the satisfiability of the formula:

Rule 1 (Finding all unit clauses): A unit clause is a clause with exactly one literal. If the literal is universally quantified, then the clause and subsequently the QBF is unsatisfiable. If the literal is existentially quantified, then the truth value of the literal can be determined uniquely. In our ZDD $\Gamma = P(y, \Gamma_1, \Gamma_2)$, where y is the topmost literal in the variable order, then a literal x can is a unit clause in Γ if:

$x = y$ and Γ_1 contains the empty set. In other words, the literal appearing in the root is a unit clause if moving to its Then-child followed by moving always toward the Else-child leads us to the 0-Terminal.

$x \in var(\Gamma_2)$ and x is a unit clause in Γ_2.(Note: if $x \in var(\Gamma_1)$ then it can not be a unit clause.)

Finding all unit clauses can be accomplished at most with $(2 \cdot n - 1)/2$ comparisons, where n is the number of variables in the set of clauses represented by Γ_1.

Rule 2 (Trivial UNSAT): If x is a unit-clause and it is universally quantified, then the QBF formula is unsatisfiable. This operation needs only one comparison instruction and can be done during the step of finding the unit clauses.

Rule 3 (Trivial UNSAT): If x is an existentially quantified unit-clause and its complementary literal is also a unit clause, then the QBF formula is unsatisfiable. This operation can be performed during the identification of unit clauses.

Rule 4 (Variable assignment/ Splitting operation): Let $\Gamma = (x, \Gamma_1, \Gamma_2)$ be our ZDD. Considering x to be 'True', simplifies Γ to $Union(Then(\Gamma_2), Else(\Gamma_2))$. Similarly considering x to be 'False', simplifies Γ to $Union(\Gamma_1, Else(\Gamma_2))$. This operation is quadratic in the size of the ZDD.

Rule 5 (Propagation of a unit clause): If x is a unit clause and located in the root node then Γ can be simplified to Γ_2. If Γ_2 has complement of x at its root then the result will be: $Union(Then(\Gamma_2), Else(\Gamma_2))$. On the other hand, if x is a unit clause but not located in the root node then, first we must remove all the clauses including x as a literal from Γ by $\Gamma' = Subset0(\Gamma, x)$, then remove the complementary literal of x, denoted by $\overline{x}$ from Γ' by $\Gamma'' = Union(Subset1(\Gamma', \overline{x}), Subset0(\Gamma', \overline{x}))$.

Rule 6 (Mono Variables): A literal l is monoton if its complementary literal does not appear in the QBF. If l is existentially quantified we can replace it by 'True', which simplifies Γ to Γ_2, but if l is universally quantified we must replace it by 'False', which simplifies Γ to $Union(\Gamma_1, \Gamma_2)$.

Rule 7 (Detecting SAT/UNSAT): If the ZDD reduces to the 1-terminal then the QBF is SAT. Similarly, if the ZDD reduces to 0-terminal then the QBF is UNSAT. This operation needs only one comparison instruction.

3.3 Algorithm

ZQSAT consists of two main parts, the first part gets the QBF and represents it as a ZDD. The second part which we called it ZQDPLL, is an adopted version of the well known DPLL algorithm. ZQDPLL can also be seen as an adopted version of 'the semantic tree approach in evaluating QBFs' which is the most robust method in deciding QBFs. Q-DIMACS is a suggested format for QBF input files. Almost all QBF benchmarks are presented in this format. Naturally we also considered this format in our implementation. First the algorithm reads and stores the prefix of the QBF in an array. This prefix holds the quantifiers of the QBF variables. Next it reads the QBF clauses and makes a ZDD represeting the formula. The variable order of this ZDD is the same as the variable order in the prefix. We assign two successive indices to each variable, the first for its positive and the second for its complimented form. This is exactly the same as what we saw in Figure 3. This form of assignment is very important and useful when we process the ZDD to find the solution. Then the recursive function ZQDPLL, examines the satisfiability of the QBF (represented in a ZDD). This function, like its original version, is based on unit resolution and splitting over variables, but ZQDPLL is different in some aspects. For example, the unit and monoliteral resolution is sometimes different because of the universally quantified variables allowed in the formula. We implemented the algorithm in C using CUDD package. Figure 4 displays the pseudocode of our algorithm. What are the special points of this algorithm along ZDDs? First, as we mentioned in Section 2, it is possible to store the original and next ZDDs, generated as a result of simplification or splitting, with a few overhead. This allowed ZQSAT to retain any already solved subformula along its solution in a table and avoid to resolve them in future occurrences. This idea could decrease the process time exponentially. Second, the splitting step could generate equal subformulas. In ZDDs two functions are equal if and only if both point to the same ZDD node. Therefore equality check could be done by only one comparison instruction. With other data structures, this could not be so easy. Perhaps most of them always solve both subformulas. Third, the unit and monoliteral resolution step is very cheap with ZDDs. It needs to consider the 0- or 1-subgraphs of the root, or union of sub-ZDDs (as mentioned in above rules).

4 Experimental results

We evaluated our algorithm over different known benchmarks presented in QBFLIB (Quantified Boolean Formula satisfiability LIBrary) [15]. We run FZQSAT along best existing QBF-Solvers such as QuBE [11], Decide [16], Semprop [12] and QSolve [10]. Our platform was a Linux system on a 3000-Mhz, 2G-RAM desktop computer. We also considered 1G-RAM limit which never used totally by any of above programs, and a 900 second timeout which was enough for must solvers to solve many of benchmark problems. The results we obtained can be summarized as follows:

1. FZQSAT is very efficient and in many cases better than state-of-the-art QSAT solvers. It solves many instances which are known to be hard for DPLL (semantic-tree) method, in a small fraction of a second. This was only possible when we strengthen ZQDPLL with memorization.

2. ZQSAT like almost all other QSAT solvers is inefficient in solving random QBFs. According to the well known counting theorem, the representation and evaluation of random instances could not be done efficiently.

Here we give detailed information on above findings.

Structured formulas: Most structured Formulas come form real word problems represented as a QBF. We used the benchmarks of Letz [15] and Rintanen [16]. The benchmarks of Letz include instances known to be hard for DPLL (tree-based) QBF solvers. Table 1 shows how FZQSAT is faster than other recent QBF solvers in evaluating these benchmark problems. FZQSAT is also a DPLL based algorithm, but it manages to solve those instances because of data structure holding the formula and the memorization techinqe embedded to the search algorithm. Next, we considered the benchmarks of Rintanen, where some problems from AI planning and other structured formulas are included. The experimental results for these benchmarks are presented in Table 2. This table show that ZQSAT works well on most instances. We are comparable and in many times cases better than other solvers. It is needed to mention that 'Decide' is specially designed to work efficiently for planning instances.

Random formulas: For random formulas we used the benchmarks of Massimo Narizzano [15]. ZQSAT is inefficient in big unstructured instances. ZDDs are very good in representing sets of subsets, but they are less useful, if the information is unstructured. In other words, ZDDs explore and use the relation between the set of subsets, therefore if there is no relation between the subsets (clauses) then it could not play its role. Fortunately in real word problems there is always some connection between the problem components. In our effort to investigate why ZQSAT is slow on the given instances, we found that in these cases the already solved subformulas never or too few times used again, also the mono and unit resolution functions could not reduce the size of the (sub)formula noticeably.

```
main() /*ZQSAT*/
{
 Get QBF and represent its clauses in a ZDD;
 Result=ZQDPLL(ZDD); Output the Result;
}

/*----------------------------------------*/

Boolean ZQDPLL( FormulaZDD  F )
{
 if ( F is Primitive or AlreadySolved ) return Solution;
 S=Simplify F by UnitMonoReduction;
 if ( S is Primitive or AlreadySolved )
   {Add F along the Solution to SolvedTable;return Solution;}
 Split F according to RootNodeLiteral to get F0 and F1;
 Solution=ZQDPLL( F0 );
 Add F along the Solution to SolvedTable;
 if( F0==F1 ) return Solution;
 if( SplittedLiteral is Universal and Solution==FALSE ) return FALSE;
 if( SplittedLiteral is Exsitentil and Solution==TRUE  ) return TRUE;
 Solution=ZQDPLL( F1 );
 Add F along the Solution to  SolvedTable;
 return Solution;
}

/*----------------------------------------*/

UnitMonoReduction( FormulaZDD  F )
 {
   do{
     do{
         1.Find all Unit-Literals in F, but if any Universally Quantified
           Literal found to be Unit return UNSAT, also if a literal is Unit
           and its complement is also Unit return UNSAT;
         2.Reduce F for all found unit Literals;
       } while(more iterations needed);

     while( Index of rootnode is mono Lteral)
       if (the Index is universally quantified)
          F=Else(F);
       else
          F=Union(Else(F),Then(F));

  }while(more reductions is possible);
 }

/*----------------------------------------*/

UnitResolution(FormulaZDD F, ListOfNodes  L)
 {
  for all Indices in L
    if (Index is in root of F)
        F=Else(F);
    else
       {
          F=RemoveClauses(F,Index);
          F=RemoveLiteral(F,NotIndex);
       }
 }
```

Figure 4: The ZQSAT algorithm.

problem		QuBE				
tree-exa-	ZQSAT	BJ	Rel	Decide	Semprop	QSolve
10-10	< .01	< .01	< .01	< .01	< .01	< .01
10-15	< .01	< .01	< .01	0.06	0.01	< .01
10-20	< .01	< .01	0.01	1.89	0.27	< .01
10-25	< .01	0.01	0.07	63.95	8.51	< .01
10-30	< .01	0.11	0.75	(?)	273.28	0.03
2-10	< .01	< .01	< .01	< .01	< .01	< .01
2-15	< .01	< .01	< .01	0.01	< .01	< .01
2-20	< .01	0.01	< .01	0.1	0.01	< .01
2-25	< .01	0.12	< .01	1.16	0.1	0.04
2-30	< .01	1.29	< .01	12.9	1.06	0.53
2-35	< .01	14.42	< .01	144.16	11.98	5.85
2-40	< .01	158.41	< .01	(?)	130.19	65.73
2-45	< .01	(?)	< .01	(?)	(?)	729.7
2-50	< .01	(?)	< .01	(?)	(?)	(?)
(?): Not solved in 900 seconds						

Table 1: Comparison of the runtimes of different QBF solvers over a number of QBFs. The instances are hard for tree-based QBF solvers (see Letz [15]).

		QuBE				
problem	ZQSAT	BJ	Rel	Decide	Semprop	QSolve
B*3i.4.4	(?)	(?)	0.02	0.02	(?)	(?)
B*3i.5.3	(?)	(?)	516.67	10.51	(?)	(?)
B*3i.5.4	(?)	(?)	(?)	1.84	(?)	(?)
B*3ii.4.3	(?)	0.81	0.01	0.01	2.25	(?)
B*3ii.5.2	(?)	23.25	0.41	0.02	65.76	(?)
B*3ii.5.3	(?)	(?)	33.12	0.36	160.93	(?)
B*3iii.4	(?)	0.25	0.01	< .01	12	(?)
B*3iii.5	(?)	(?)	0.48	0.1	0.53	(?)
B*4i.6.4	(?)	(?)	264.76	1.28	(?)	(?)
B*4ii.6.3	(?)	(?)	27.64	1.1	(?)	(?)
B*4ii.7.2	(?)	(?)	(?)	2.28	(?)	(?)
B*4iii.6	(?)	(?)	13.62	0.59	(?)	(?)
B*4iii.7	(?)	(?)	(?)	67.28	(?)	(?)
C*12v.13	2.66	0.12	1.41	0.19	0.06	1.96
C*13v.14	3.76	0.26	3.44	0.38	0.13	6.52
C*4v.15	5.27	0.55	9.17	0.77	0.27	21.98
C*15v.16	7.08	1.22	24.21	1.62	0.54	62.53
C*16v.17	9.43	3.09	60.68	3.31	1.14	205.72
C*17v.18	12.49	5.86	148.58	6.9	2.43	633.44
C*18v.19	16.2	12.87	352.21	14.4	5.12	(?)
C*19v.20	21.01	31.93	840.26	30.29	10.59	(?)
C*20v.21	26.69	91.23	(?)	61.93	22.24	(?)
C*21v.22	33.17	195.12	(?)	129.24	46.61	(?)
C*22v.23	40.8	494.26	(?)	272.24	98.53	(?)
C*23v.24	50.24	(?)	(?)	571.12	202.3	(?)
i*02	< .01	< .01	< .01	< .01	< .01	< .01
i*04	< .01	< .01	< .01	< .01	< .01	< .01
i*06	< .01	< .01	< .01	0.01	< .01	< .01
i*08	< .01	< .01	< .01	0.14	0.02	0.01
i*10	< .01	0.01	< .01	1.12	0.14	0.07
i*12	< .01	0.04	< .01	8.69	1.04	0.5
i*14	< .01	0.18	< .01	65.27	7.74	3.69
i*16	< .01	0.74	< .01	482.97	56.88	27.04
i*18	< .01	3.12	< .01	(?)	423.41	200.82
i*20	< .01	13.06	< .01	(?)	(?)	(?)
l*A0	0.06	< .01	< .01	< .01	?	0.01
l*A1	(?)	50.28	5.79	0.67	3.68	(?)
l*B0	0.2	< .01	< .01	0.01	?	0.01
l*B1	(?)	407.65	14.62	3.25	13.91	(?)
T*10.1.iv.20	2.65	(?)	(?)	0.58	(?)	(?)
T*16.1.iv.32	26.08	(?)	(?)	7.38	(?)	(?)
T*2.1.iv.3	< .01	< .01	< .01	< .01	< .01	< .01
T*2.1.iv.4	< .01	< .01	< .01	< .01	< .01	< .01
T*6.1.iv.11	(?)	2.1	205.75	4.78	2.25	3.66
T*6.1.iv.12	0.24	0.79	29.44	0.04	0.4	2.65
T*7.1.iv.13	(?)	37.45	(?)	63.87	39.7	134.02
T*7.1.iv.14	0.5	12.25	521.59	0.09	5.22	64.17
(?): Not solved in 900 seconds						

Table 2: Comparison of different QBF solvers on a number of QBFs from the set of benchmarks of Rintanen [16, 15].

5 Conclusion

The experimental results show that our algorithm is comparable and in many cases much faster than the best existing QSAT solvers. We realized that this superiority is partly form the idea of strengthening the search method with memorization and partly from fast unit-mono-resolution, thanks to ZDDs used to represent the formula.

References

[1] O. Achröer and I. Wegener. The Theory of Zero-Suppressed BDDs and the Number of Knight's Tours. *Formal Methods in System Design*, 13(3), November 1998.

[2] F. A. Aloul, M. N. Mneimneh, and K. A. Sakallah. ZBDD-Based Backtrack Search SAT Solver. In *International Workshop on Logic Synthesis (IWLS)*, pages 131–136, New Orleans, Louisiana, 2002.

[3] J. Bern, C. Meinel, and A. Slobodová. OBDD-Based Boolean manipulation in CAD beyound current limits. In *Proceedings 32nd ACM/IEEE Design Automation Conference*, pages 408–413, San Francisco, CA, 1995.

[4] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[5] R. E. Bryant and C. Meinel. *Ordererd Binary Decision Diagrams in Electronic Design Automation*, chapter 11. Kluwer Academic Publishers, 2002.

[6] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An Algorithm to Evaluate Quantified Boolean Formulae and Its Experimental Evaluation. *Journal of Automated Reasoning*, 28(2):101–142, 2002.

[7] P. Chatalic and L. Simon. Multi-Resolution on Compressed Sets of Cluases. In *Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'00)*, 2000.

[8] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communication of the ACM*, 5:394–397, 1962.

[9] T. Eiter, V. Klotz, H. Tompits, and S. Woltran. Modal Nonmonotonic Logics Revisited: Efficient Encodings for the Basic Reasoning Tasks. In *Proceedings of the Eleventh Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-2002)*, 2002.

[10] R. Feldmann, B. Monien, and S. Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, 2000.

[11] E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability. In *Proceedings of the International Joint Conference on Automated Reasoning*, pages 364–369, 2001.

[12] R. Letz. Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In *Proceedings of TABLEAUX 2002*, pages 160–175. Springer-Verlag Berlin, 2002.

[13] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer, 1998.

[14] S. Minato. Zero-suppressed BDDs for set Manipulation in Combinatorial Problems. In *proceedings of the 30th ACM/IEEE Design Automation Conference*, 1993.

[15] QBFLIB - QBF Satisfibility Library. http://www.mrg.dist.unige.it/ qube/qbflib/.

[16] J. Rintanen. Improvements to the Evaluation of Quantified Boolean Formulae. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1192–1197, 1999.

[17] F. Somenzi. CUDD: CU Decision Diagram Package. ftp://vlsi.colorado.edu/pub/.

[18] I. Wegener. *Branching Programs and Binary Decision Diagrams – Theory and Applications*. SIAM Monographs on Discrete Mathematics and Applications, 2000.