Published in: Proceedings of The Australian Conference on Artificial Intelligence (AI'2004)
Queensland, Australia, 2004, pp. 681-693

Embedding Memoization to the Semantic Tree Search for Deciding QBFs

Mohammad GhasemZadeh, Volker Klotz, and Christoph Meinel

FB-IV Informatik, University of Trier,
D-54286 Trier, Germany
{GhasemZadeh, klotz, meinel}@TI.Uni-Trier.DE

Abstract. Quantified Boolean formulas (QBFs) play an important role in artificial intelligence subjects, specially in planning, knowledge representation and reasoning [20]. In this paper we present ZQSAT (sibling of our FZQSAT [15]), which is an algorithm for evaluating quantified Boolean formulas. QBF is a language that extends propositional logic in such a way that many advanced forms of reasoning can be easily formulated and evaluated. ZQSAT is based on ZDD, which is a variant of BDD, and an adopted version of the DPLL algorithm. The program has been implemented in C using the CUDD package. The capability of ZDDs in storing sets of subsets efficiently enabled us to store the clauses of a QBF very compactly and led us to implement the search algorithm in such a way that we could store and reuse the results of all previously solved subformulas with few overheads. This idea along some other techniques, enabled ZQSAT to solve some standard QBF benchmark problems faster than the best existing QSAT solvers.

Keywords: DPLL, Zero-Suppressed Binary Decision Diagram (ZDD), Quantified Boolean Formula (QBF), Satisfiability, QSAT.

1 Introduction

Propositional satisfiability (SAT) is a central problem in computer science with numerous applications. SAT is the first and prototypical problem for the class of NP-complete problems. Many computational problems such as constraint satisfaction problems, many problems in graph theory and forms of planning can be formulated easily as instances of SAT.

Theoretical analysis has showed that some forms of reasoning such as: belief revision, non monotonic reasoning, reasoning about knowledge and STRIPS-like planning have computational complexity higher than the complexity of the SAT problem. These forms can be formulated by quantified Boolean formulas and be solved as instances of the QSAT problem . Quantified Boolean formula satisfiability (QSAT) is a generalization of the SAT problem. QSAT is the prototypical problem for the class of PSPACE-complete problems. With QBFs we can represent many classes of formulas more concisely than conventional Boolean formulas.

ZDDs are variants of BDDs. While BDDs are better suited for representing Boolean functions, ZDDs are better for representing sets of subsets. Considering all the variables appearing in a QBF propositional part as a set, the propositional part of the formula

G.I. Webb and Xinghuo Yu (Eds.): AI 2004, LNAI 3339, pp. 681–693, 2004.

can be viewed as a set of subsets, this is why using ZDDs for representing a formula could potentially be beneficial. This idea is used in a number of related works [8, 10, 2] where the SAT problem is considered. They use ZDDs to store the CNF formula and the original DP algorithm to search its satisfiability. We also found ZDDs very suitable for representing and solving QSAT problems.

We represent the clauses in the same way in a ZDD, but we employ an adopted version of the DPLL [9] algorithm to search the solution. In fact, our adopted version simulates the "Semantic tree method in evaluating QBFs". It benefits from an adopted unit-mono-resolution operation which is very fast thanks to the data structure holding the formula. In addition, it stores all already solved subformulas along their solutions to avoid resolving same subproblems. Sometimes the split operation generates two subproblems which are equal. With ZDDs it is very easy to compare and discover their equality, therefore our algorithm can easily prevent solving both cases when it is not necessary. There are some benchmark problems which are known to be hard for DPLL (semantic tree) algorithms. ZQSAT is also a DPLL based algorithm, but it manages to solve those instances very fast. ZQSAT is still slow in some QBF instances, this is why we can not claim ZQSAT is the best conceivable algorithm, but it is the first work that shows how ZDDs along memoization can be used successfully in QBF evaluation.

2 Preliminaries

2.1 Quantified Boolean Formulas

Quantified Boolean formulas are extensions of propositional formulas (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, and negation. 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, in our research, the *prenex normal form*and *conjunctive normal form* (*CNF*) are important. In many problems including SAT and QSAT, normal forms do not affect the generality of the problem, instead they bring the problem in a form that can be solved more easily.

Definition 1. *A Boolean formula is in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals, that is,* $\phi = c_1 \wedge c_2 \wedge \ldots \wedge c_n$*, where* $c_i = (l_{i1} \vee \ldots \vee l_{im_i})$ *and* l_{ij} *is a negative or positive literal. The disjunctions are referred as clauses.*

Each Boolean formula can be transformed into a logically equivalent Boolean formula which is in conjunctive normal form (CNF). Generally this transformation can not be done efficiently.

Definition 2. *A QBF* Φ *is in prenex normal form, if it is in the form:*

$$\Phi = Q_1 V_1 Q_2 V_2 \ldots Q_n V_n \phi,$$

where $Q_i \in \{\forall, \exists\}$, V_i $(1 \leq i \leq n)$ are disjoint sets of propositional variables and ϕ is a propositional formula over the variables $x_1, \ldots, x_n$. The expression $Q_1 V_1 Q_2 V_2 \ldots Q_n V_n$ is called the prefix and ϕ the matrix of Φ.

2.2 The DPLL Algorithm for the SAT Problem

Most former and recent SAT solvers are in some way extensions of the DPLL [9] algorithm. DPLL tries to find a satisfying assignment for a CNF formula by making an exhaustive search. Each variable is assigned with a truth-value (`true` or `false`) which leads to some simplifications of the function. Since the function is in CNF, the assignment can be done efficiently. If an assignment forces the formula to be reduced to `false` then a backtrack will take place to make another possible assignment. If none of the possible assignments satisfy the function then the function is unsatisfiable. In order to prune the search space we have to consider unit clauses. A unit clause is a clause with exactly one literal. For example in $f = (a \vee \neg b \vee \neg c) \wedge (a \vee \neg c \vee d) \wedge (b) \wedge (a \vee b \vee c)$, the third clause is a unit clause. Unit resolution is the assignment of proper truth values to the literals appearing in unit clauses and removing them from the formula. For instance, in the above example, b receives the value `true`, which lets f be simplified to: $f_1 = (a \vee \neg c) \wedge (a \vee \neg c \vee d)$. If all literals in a clause simplify without satisfying the clause then DPLL immediately returns "UNSATISFIABLE", but if all the clauses satisfy and be removed then it returns "SATISFIABLE". When no more simplification is possible, DPLL splits the simplified function over one of the remaining variables (which can receive either the value `true` or `false`). This step removes one variable, and consequently a number of clauses. Two smaller CNF formulas will be generated of which at least one must be satisfiable to make the original formula satisfiable.

2.3 The Semantic Tree Approach for the QSAT Problem

This method is very similar to the DPLL algorithm. It iteratively splits the problem of deciding a QBF of the form $Qx\,\Phi$ into two subproblems $\Phi[x = 1]$ and $\Phi[x = 0]$ (the unique assignment of each x respectively with `true` or `false`), and 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.

Figure 1 displays the pseudocode of this algorithm, which we have called QDPLL.

The differences between QDPLL (for QBFs) and the DPLL algorithm (for Boolean satisfiability) can be enumerated as follows:

1. In the Unit-Resolution step (line 1), if any universally quantified variable is found to be a unit clause then the procedure can immediately conclude the UNSAT result and terminate.
2. In the Mono-Reduction step (line 1), if any universally quantified variable is found to be a mono-literal, then it can be removed from all the clauses where it occurs (rather than removing the clauses, as it applies to existentially quantified mono literals). We call a literal *monotone* if its complementary literal does not appear in the matrix of the QBF. The Mono-Reduction step can result in new unit clauses. Therefore the procedure must continue line 1 as long as new simplifications are possible.

```
Boolean QDPLL( Prenex-CNF  F )
 {
1  F=Simplify F by repeated Unit-Resolution removal of subsumed
     clauses and possible Mono-Reductions;

2  if (F is primitive ) return Solution;
   // the same as in DPLL, if F include Empty-Clause return
   // UNSAT, but if F is Empty return SATISFIABLE

3  F0,F1=choose x as the splitting variable; Split F;
         // Very little freedom in choosing variables.

4  Solution=DPLL(F0);
5  if (Solution==TRUE  and  Existential_Literal(x) ) return TRUE;
6  if (Solution==FALSE and  Univarsal_Literal(x)   ) return FALSE;

  // for the other two cases
7  return DPLL(F1);
 }
```

Fig. 1. The semantic tree approach for QBFs

3. In the splitting step (line 3), there is a little freedom in choosing the splitting variable. In fact, in a block of consecutive variables under the same kind of quantifier we are allowed to consider any order, but before processing all the variables in the leftmost block we are not allowed to assign values to any variable from other blocks. In other words, Iterations of quantified blocks must be considered exactly in the same order as they appear in the QBF prefix. As an example in the QBF $\forall x_1 \forall x_2 \forall x_3 \exists y_1 \exists y_2 \exists y_3 \forall z_1 \forall z_2 \forall z_3 \phi$, all x_i must be assigned before any assignment for any y_j take place, in the similar way, all y_j must be assigned before any assignment for any z_k take place, but we are allowed to consider any order when we are processing the variables of for example x block.
4. After solving one of the branches, even if the result is `true` (line 7), it could be necessary to solve the other branch as well. Due to universal variables allowed to appear in QBFs, the `false` result (line 6) for one of the branches can signify the UNSAT result and terminate the procedure without checking the other branch.

From another point of view, this method searches the solution in a tree of variable assignments. Figure 2 [13] 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:

$$C_1 = (\neg y_1 \vee x \vee \neg y_2), C_2 = (y_2 \vee \neg y_3), C_3 = (y_2 \vee y_3), \text{ and } C_4 = (y_1 \vee \neg x \vee \neg y_2).$$

We can follow the tree and realize that Φ is invalid. A very interesting point can be easily seen in the tree. It is the duplication problem in semantic tree method, namely, the same subproblem can appear two or more times during the search procedure. In a

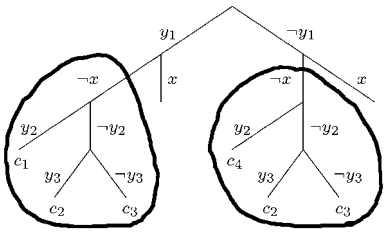

Fig. 2. A semantic tree proof

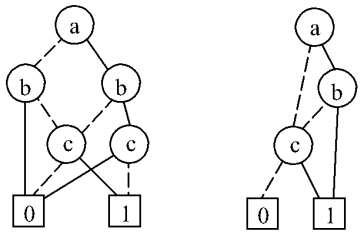

Fig. 3. BDD versus ZDD

big QBF this situation can frequently happen in different levels. The superiority of our algorithm which we will present later, is its possibility to detect and avoid to examine such duplications repeatedly.

2.4 BDDs Versus ZDDs

Here we give a very short background for BDDs and ZDDs. Several years ago, Binary Decision Diagrams (BDDs) [5, 6, 21, 3, 16] and their variants [4] entered the scene of the computer science. Since that time, they have been used successfully in industrial CAD tools. In many applications, specially in problems involving sparse 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) [17, 1]. These diagrams are similar to BDDs with one of the underlying principles modified. While BDDs are better suited for the representation of functions, ZDDs are better suited for the representation of covers (set of subsets). Considering all the variables appearing in a QBF (propositional part) as a set, the propositional part of the formula can be viewed as a set of subsets, this is why using ZDDs for representing a formula could potentially be beneficial. As an example [18], in Figure 3, the left diagram displays the ZDD representing $S = \{\{a, b\}, \{a, c\}, \{c\}\}$, and the right diagram displays $F = (a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge c) \vee (\neg a \wedge \neg b \wedge c)$, which is the characteristic function of S. In a ZDD (or BDD) we represent an internal node by $P(x, \Gamma_0, \Gamma_1)$ where x is the label of the node, and Γ_1, Γ_0 are SubZDDs rooted in it 'Then-child' and 'Else-child' respectively. The size of a ZDD Γ, denoted by $|\Gamma|$, is the number of its internal nodes.

3 Our Algorithm

ZQSAT is the name we used for our QSAT solver. The major points which are specific to our algorithm are:

1. Using ZDDs to represent the QBF matrix (the formula clauses). (We adopted this idea from [8, 10, 2] then established the specific rules suitable for QBF evaluation).
2. Embedding memoization to overcome mentioned duplication problem (to avoid solving the same subproblem repeatedly).

Figure 4 displays the pseudocode for MQDPLL, which stands for our 'DPLL with memoization' procedure. This procedure forms the search strategy used by ZQSAT.

```
Boolean MQDPLL( Prenex-CNF  F )
 {
1  if ( F is Primitive or AlreadySolved ) return Solution;
2  S=Simplify F by repeated Unit-Resolution, removal of
     subsumed clauses and possible MonoLiteral-Reductions;
3  if ( S is Primitive or AlreadySolved )
     {Add F along with the Solution to SolvedTable;
      return Solution; }

4  F0,F1=choose x as the splitting variable then Split S;
5  Solution=DPLL(F0); // Why this branch? Read in text !!
6  if (F0==F1) or
     (Solution==TRUE  and  Existential-Literal(x) ) or
     (Solution==FALSE and  Univarsal-Literal(x)   )
      {Add F along with the Solution to SolvedTable;
       return Solution; }
7  Solution=DPLL(F1);
8  Add F along with the Solution to SolvedTable;
   return Solution;
 }
```

Fig. 4. MQDPLL: Our 'DPLL with memoization' procedure

MQDPLL is different from QDPLL in some aspects. Firstly, it benefits from a memoization strategy (dynamic programming tabulation method) to store and reuse the results of already solved subproblems (lines 1, 3, 6, 8 in the above pseudocode). Secondly, the situation where the two subfunctions f_0 and f_1 are equal can be detected and the subproblem would be solved only once (line 6).

In line 4, the algorithm needs to choose a variable for the splitting operation. At this point we must respect the order of iterations of quantification blocks, but when we are working with a quantification block we are allowed to choose any variable order. In our implementation we used the order which appears in the initial QBF formula. In fact we tried to investigate other possibilities, but since we obtained no benefits in our first effort we did not continue to investigate the issue in detail. We believe in this regard we can potentially find useful heuristics in our future research.

In line 5, the algorithm needs to choose the next branch (F0 or F1) to continue the search process. There are a number of possibilities like: always F0 first, always F1 first, random choice, according to the number of nodes in the ZDDs representing F0 and F1, according to the indices appearing in the root nodes of the ZDDs representing F0 and F1, Considering the number of positive/negative appearance of the variables in F0 and F1 clauses and so on. We tried most of these possibilities and realized that in our implementation they behave more or less the same, but we still believe that at this point we can potentially improve the performance of our algorithm.

Storing all already solved subproblems and detecting the equality of two subproblems (functions) is usually very expensive. We managed to overcome these difficulties thanks to ZDDs. This data structure lets us to store the QBF matrix very efficiently and allowed

us to store every subfunction created in the splitting step or obtained after the simplification operations, with no or very little overheads (see Figure 5).

3.1 Using ZDDs to Represent a CNF Formula

A ZDD can be used to represent a set of subsets. We use this property to represent the body of the QBF, which is supposed to be a propositional function in CNF. Since each propositional CNF formula ϕ can be represented as a set of sets of literals $[\phi]$ we can represent a CNF formula by means of 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.

To represent the sets of clauses, i.e., a set of sets of literals, we assign two successive ZDD indices to each variable, one index for positive and the next for its complemented form [8]. Figure 5 shows how this idea works for a small CNF formula [10, 2]. In ZDDs (like BDDs), the variable order can considerably affect the shape and size of the resulting graph. As we pointed out earlier, in evaluating QBFs, the variable selection is strongly restricted. In general the order of the prefix must be respected. In representing and evaluating a QBF like $\Phi = \exists x_1 \ldots \forall x_n \phi$ using ZDDs, we consider the extended literal order $x_1 \leq \neg x_1 \leq \ldots \leq x_n \leq \neg x_n$. The following theorem [8] gives a good estimate for the size of the ZDD representing a CNF formula in the mentioned method.

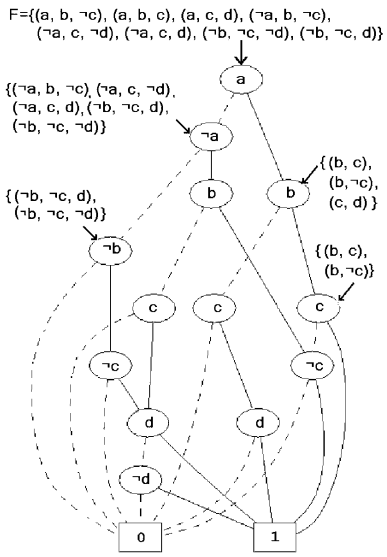

Fig. 5. ZDD encoding of a CNF formula

Theorem 1. *Let f be a formula in conjunctive normal form. The number of nodes of the ZDD Γ_f encoding the set of clauses of f is always at most equal to the total number of literals of f.*

Due to the page limit we removed the proof (please contact the authors for the proof).

3.2 Benefits of Using ZDDs Along Our MQDPLL-Algorithm

In Figure 5, 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 unique function. In our search procedure, after the simplification operations and after the splitting step, new functions arise. We noticed that many of these functions are the same, therefore we let ZQSAT to retain all already produced functions along their solutions, to prevent resolving the same functions (memoization). We mentioned earlier, this idea is embedding dynamic programming/memoization to the DPLL Algorithm. 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. 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 these operations. The rules can be derived from the basic properties known for QBFs, some lemmas presented in [7] and the properties of representing CNF clauses in a ZDD. Performing these operations with other data structures is often much slower. Reminding that Minato [17] has presented efficient algorithms for set operations on ZDDs. His algorithms are mostly based on dynamic programming and efficient caching techniques. We used them (through the CUDD package) in our research work.

In the following rules we suppose we have read the clauses and represented them in a ZDD Γ. The rules are applicable when we are examining the satisfiability of Γ:

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. Let $\Gamma = P(l_1, \Gamma_1, \Gamma_2)$ be a ZDD where l_1 is the topmost literal in the variable order, then the literal l_2 is a unit clause in Γ iff:

- $l_2 = l_1$ and Γ_1 contains the empty set. In other words, the literal appearing in the root of the ZDD is a unit clause if moving to its Then-child followed by moving always toward the Else-child leads us to the 1-terminal.
- $l_2 \in var(\Gamma_2)$ and l_2 is a unit clause in Γ_2. Note: if $l_2 \in var(\Gamma_1)$ then it can not be a unit clause.

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

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 is located in the root node then Γ can be simplified to Γ_2. If Γ_2 has complement of x in 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 $\Gamma' = Subset0(\Gamma, x)$. After this we must 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 monotone 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.

These rules are the basic stones in implementing the operations needed in ZQSAT, specially the unit resolution and mono literal reduction in MQDPLL procedure.

4 Experimental Results

We evaluated our algorithm by different known benchmarks presented in QBFLIB (QBF satisfiability LIBrary) [19]. We run ZQSAT along the best existing QBF-Solvers such as QuBE [12], Decide [20], Semprop [14] and QSolve [11]. The platform was a Linux system on a 3000-Mhz, 2G-RAM desktop computer. We also considered 1G-RAM limit which were never used totally by any of the above programs, and a 900 second timeout which was enough for most solvers to solve many of benchmark problems.

The results we obtained show that ZQSAT is very efficient and in many cases better than state-of-the-art QSAT solvers. It solves many instances which are known hard for DPLL (semantic-tree) method, in a small fraction of a second (see Table 1 and Table 2). Like almost all other QSAT solvers it 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 [16]. In the following we give more detailed information.

Structured Formulas: Most structured Formulas come form real word problems represented as a QBF. We used the benchmarks of Letz [19] and Rintanen [20]. The benchmarks of Letz include instances known to be hard for DPLL (tree-based) QBF solvers. ZQSAT is also a DPLL based algorithm, but it manages to solve those instances very fast. In a real problem there are always some connections between its components, which remain in some form in its corresponding QBF representation. This feature causes similar subproblems to be generated during the search step, also assignment of values to

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 [19])

Problem		QuBE						Nr. Lookup		No. Mem.
tree-exa-	ZQSAT	BJ	Rel	Decide	Semprop	QSolve	Z	Total	Succ	Rec.Calls
10-10	$< .01$	$< .01$	$< .01$	$< .01$	$< .01$	$< .01$	1	49	14	1277
10-15	$< .01$	$< .01$	$< .01$	0.06	0.01	$< .01$	1	79	24	40957
10-20	$< .01$	$< .01$	0.01	1.89	0.27	$< .01$	1	109	34	1310717
10-25	$< .01$	0.01	0.07	63.95	8.51	$< .01$	1	139	44	not solved
10-30	$< .01$	0.11	0.75	(?)	273.28	0.03	1	169	54	not solved
2-10	$< .01$	$< .01$	$< .01$	$< .01$	$< .01$	$< .01$	1	31	6	not solved
2-15	$< .01$	$< .01$	$< .01$	0.01	$< .01$	$< .01$	1	51	11	not solved
2-20	$< .01$	0.01	$< .01$	0.1	0.01	$< .01$	1	71	16	not solved
2-25	$< .01$	0.12	$< .01$	1.16	0.1	0.04	1	91	21	not solved
2-30	$< .01$	1.29	$< .01$	12.9	1.06	0.53	1	111	26	not solved
2-35	$< .01$	14.42	$< .01$	144.16	11.98	5.85	1	131	31	not solved
2-40	$< .01$	158.41	$< .01$	(?)	130.19	65.73	1	151	36	not solved
2-45	$< .01$	(?)	$< .01$	(?)	(?)	729.7	1	171	41	not solved
2-50	$< .01$	(?)	$< .01$	(?)	(?)	(?)	1	191	46	not solved
(?): Not solved in 900 seconds										

variables causes sharp simplification on generated subformulas. Therefore, our memoization idea helps very much in these circumstances. Table 1 shows how ZQSAT is faster than other recent QBF solvers in evaluating these benchmark problems.

The four rightmost columns in the table are provided to show the role and effect of our memoization idea. The two columns which stand for the number of lookups (total, successful) give us an estimate of the hit ratio, i.e. 'successful lookups' versus 'all lookups, which in our implementation is the same as the total recursive DPLL calls'. We must be careful analyzing these numbers, because the number of total calls depends strongly (sometimes exponentially) on the number of successful lookups. In order to avoid such a misinterpretation we provided the rightmost column which displays the number of DPLL recursive calls when no memoization is considered. In this condition our implementation only managed to solve three smallest instances of Letz benchmarks (in above mentioned platform and our 900 second time out). The column labeled with 'Z' is in connection with construction of the initial ZDD for the formula. In fact we realized that even in failing benchmarks, ZQSAT managed to make the initial ZDD soon.

Next, we considered the benchmarks of Rintanen, where some problems from AI planning and other structured formulas are included. They include some instances form blocks world problem, Towers of Hanoi, long chains of implications, as well as the bw-large.a and bw-large.b blocks world problems. The experimental results for these benchmarks are presented in Table 2. This table shows that ZQSAT works well on most instances. We are comparable and in many cases better than other solvers. Let us mention that 'Decide' is specially designed to work efficiently for planning instances.

In Table 2 we see that our implementation could not solve any instance of blocks-world. We observed that MQDPLL benefited very few times from the already solved subformulas. In other words, our pruning method was not successful for this problem. In fact this is a matter of pruning strategy, different pruning strategies behave differently facing various sets of QBF benchmarks.

Table 2. Comparison of different QBF solvers on a number of QBFs from the set of benchmarks of Rintanen [20, 19]

Problem		QuBE						Nr. Lookup		No. Mem.
tree-exa-	ZQSAT	BJ	Rel	Decide	Semprop	QSolve	Z	Total	Succ	Rec.Calls
B*3i.5.4	(?)	(?)	(?)	1.84	(?)	(?)	1	(?)	(?)	not solved
B*3ii.4.3	(?)	0.81	0.01	0.01	2.25	(?)	1	(?)	(?)	not solved
B*3ii.5.2	(?)	23.25	0.41	0.02	65.76	(?)	1	(?)	(?)	not solved
B*3ii.5.3	(?)	(?)	33.12	0.36	160.93	(?)	1	(?)	(?)	not solved
B*3iii.4	(?)	0.25	0.01	$< .01$	12	(?)	1	(?)	(?)	not solved
B*3iii.5	(?)	(?)	0.48	0.1	0.53	(?)	1	(?)	(?)	not solved
B*4i.6.4	(?)	(?)	264.76	1.28	(?)	(?)	1	(?)	(?)	not solved
B*4ii.6.3	(?)	(?)	27.64	1.1	(?)	(?)	1	(?)	(?)	not solved
B*4ii.7.2	(?)	(?)	(?)	2.28	(?)	(?)	1	(?)	(?)	not solved
B*4iii.6	(?)	(?)	13.62	0.59	(?)	(?)	1	(?)	(?)	not solved
B*4iii.7	(?)	(?)	(?)	67.28	(?)	(?)	1	(?)	(?)	not solved
C*12v.13	2.66	0.12	1.41	0.19	0.06	1.96	1	72	11	12310
C*13v.14	3.76	0.26	3.44	0.38	0.13	6.52	1	78	12	24600
C*4v.15	5.27	0.55	9.17	0.77	0.27	21.98	1	84	13	49178
C*15v.16	7.08	1.22	24.21	1.62	0.54	62.53	1	90	14	98332
C*16v.17	9.43	3.09	60.68	3.31	1.14	205.72	1	96	15	196638
C*17v.18	12.49	5.86	148.58	6.9	2.43	633.44	1	102	16	393248
C*18v.19	16.2	12.87	352.21	14.4	5.12	(?)	1	108	17	786466
C*19v.20	21.01	31.93	840.26	30.29	10.59	(?)	1	114	18	1572900
C*20v.21	26.69	91.23	(?)	61.93	22.24	(?)	1	120	19	3145766
C*21v.22	33.17	195.12	(?)	129.24	46.61	(?)	1	126	20	not solved
C*22v.23	40.8	494.26	(?)	272.24	98.53	(?)	1	132	21	not solved
C*23v.24	50.24	(?)	(?)	571.12	202.3	(?)	1	138	22	not solved
i*02	$< .01$	$< .01$	$< .01$	$< .01$	$< .01$	$< .01$	1	8	0	8
i*04	$< .01$	$< .01$	$< .01$	$< .01$	$< .01$	$< .01$	1	14	0	14
i*06	$< .01$	$< .01$	$< .01$	0.01	$< .01$	$< .01$	1	20	0	20
i*08	$< .01$	$< .01$	$< .01$	0.14	0.02	0.01	1	26	0	26
i*10	$< .01$	0.01	$< .01$	1.12	0.14	0.07	1	32	0	32
i*12	$< .01$	0.04	$< .01$	8.69	1.04	0.5	1	38	0	38
i*14	$< .01$	0.18	$< .01$	65.27	7.74	3.69	1	44	0	44
i*16	$< .01$	0.74	$< .01$	482.97	56.88	27.04	1	50	0	50
i*18	$< .01$	3.12	$< .01$	(?)	423.41	200.82	1	56	0	56
i*20	$< .01$	13.06	$< .01$	(?)	(?)	(?)	1	62	0	62
T*10.1.iv.20	2.65	(?)	(?)	0.58	(?)	(?)	1	43	0	43
T*16.1.iv.32	26.08	(?)	(?)	7.38	(?)	(?)	1	66	0	66
T*2.1.iv.3	$< .01$	$< .01$	$< .01$	$< .01$	$< .01$	$< .01$	1	10	0	10
T*2.1.iv.4	$< .01$	$< .01$	$< .01$	$< .01$	$< .01$	$< .01$	1	10	0	10
T*6.1.iv.11	(?)	2.1	205.75	4.78	2.25	3.66	1	(?)	(?)	not solved
T*6.1.iv.12	0.24	0.79	29.44	0.04	0.4	2.65	1	27	0	27
T*7.1.iv.13	(?)	37.45	(?)	63.87	39.7	134.02	1	(?)	(?)	not solved
T*7.1.iv.14	0.5	12.25	521.59	0.09	5.22	64.17	1	30	0	30

(?): Not solved in 900 seconds

Random Formulas: For random formulas we used the benchmarks of Massimo Narizzano [19]. 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. 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 very well. Fortunately, in real word problems there are always some connections 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 were never or too few times used again, also the mono and unit resolution functions could not reduce the size of the (sub)formula noticeably.

5 Conclusion

In this paper we have presented ZQSAT, an algorithm to evaluate quantified Boolean formulas. The experimental results show how it is comparable and in some cases faster than the best existing QBF solvers. However, we still do not claim ZQSAT is the best conceivable algorithm, but it shows how ZDDs along memoization can be used successfully in QBF evaluation.

References

1. Olaf Achröer and Ingo Wegener. The Theory of Zero-Suppressed BDDs and the Number of Knight's Tours. *Formal Methods in System Design*, 13(3), November 1998.
2. Fadi A. Aloul, Maher N. Mneimneh, and Karem A. Sakallah. ZBDD-Based Backtrack Search SAT Solver. In *International Workshop on Logic and Synthesis (IWLS)*, pages 131–136, New Orleans, Louisiana, 2002.
3. www.bdd-portal.org. http://www.bdd-portal.org/.
4. Jochen Bern, Christoph Meinel, and Anna 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.
5. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.
6. Randal E. Bryant and Christoph Meinel. *Ordererd Binary Decision Diagrams in Electronic Design Automation*, chapter 11. Kluwer Academic Publishers, 2002.
7. Marco Cadoli, Marco Schaerf, Andrea Giovanardi, and Massimo Giovanardi. An Algorithm to Evaluate Quantified Boolean Formulae and Its Experimental Evaluation. *Journal of Automated Reasoning*, 28(2):101–142, 2002.
8. Philippe Chatalic and Laurent Simon. Multi-Resolution on Compressed Sets of Cluases. In *Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'00)*, 2000.
9. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communication of the ACM*, 5:394–397, 1962.
10. Karem A. Sakallah Fadi A. Aloul, Maher N. Mneimneh. Backtrack Search Using ZBDDs. In *International Workshop on Logic and Synthesis (IWLS)*, page 5. University of Michigan, June 2001.
11. Rainer Feldmann, Burkhard Monien, and Stefan Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, 2000.
12. Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability. In *Proceedings of the International Joint Conference on Automated Reasoning*, pages 364–369, 2001.
13. Reinhold Letz. Efficient Decision Procedures for Quantified Boolean Formulas. Vorlesung WS 2002/2003 TU Muenchen: Logikbasierte Entscheidungsverfahren.
14. Reinhold Letz. Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In *Proceedings of TABLEAUX 2002*, pages 160–175. Springer Berlin, 2002.
15. C. Meinel M. GhasemZadeh, V. Klotz. FZQSAT: A QSAT Solver for QBFs in Prenex NNF (A Useful Tool for Circuit Verification) . In *International Workshop on Logic and Synthesis (IWLS)*, pages 135–142. California, USA, June 2004.
16. Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design*. Springer, 1998.

17. S. Minato. Zero-suppressed BDDs for set Manipulation in Combinatorial Problems. In *proceedings of the 30th ACM/IEEE Design Automation Conference*, 1993.
18. Alan Mishchenko. An Introduction to Zero-Suppressed Binary Decision Diagrams. http://www.ee.pdx.edu/ alanmi/research/dd/zddtut.pdf, June 2001.
19. QBFLIB - Quantified Boolean Formulas Satisfiability Library. http://www.mrg.dist.unige.it/ qube/qbflib/.
20. Jussi 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.
21. Ingo Wegener. *Branching Programs and Binary Decision Diagrams – Theory and Applications*. SIAM Monographs on Discrete Mathematics and Applications, 2000.