University of New South Wales

A strategy game approach on solving QBF

Thesis A report

Author
Yifan He

Supervisor
Dr. Abdallah Saffidine

August 10, 2021

Abstract

Quantified Boolean Formulas or QBF were proposed in the 1970s as a natural generalization of the propositional logic. This more generalized framework is very important in many domains such as planning, games, and model checking. There are two types of QBF solvers expansion based and search based. For all of the existing search based QBF solvers, the term search refers to a depth first search based QDLL procedure. However, QBF is similar to two-player strategy game in a theoretical level, and depth first search is rarely used in two-player strategy game solving. This thesis project aims to investigate whether a best first search algorithm that has achieved massive success in two-player strategy game called proof number search can be combined with existing search based QBF solving algorithms and if this new combination can outperform the depth first search based solver on some popular benchmarks.

In this thesis A report, I will review some important search based QBF solving algorithms. I will also present how a 2001 QBF solving algorithm called backjumping can be implemented in the context of proof number search. Experiment result shows that replacing depth first search with best first search can potentially improve the search based QBF solvers on several families of benchmarks and this new combination is worthy to be further studied in thesis B and thesis C.

Contents

1 Introduction 3

2 Background 3

3 Literature Survey 6
- 3.1 QDLL Procedure . . . 7
- 3.2 Backjumping Trick . . . 8
  - 3.2.1 Reason computation at terminal state . . . 8
  - 3.2.2 Reason computation during backtracking . . . 9
  - 3.2.3 QDLL Backjumping algorithm . . . 10
- 3.3 QCDCL . . . 11
- 3.4 Branching Heuristic . . . 12
  - 3.4.1 Famous branching heuristics . . . 12
  - 3.4.2 Performance of different branching heuristics . . . 13
- 3.5 Strategy Game Interpretation . . . 14
- 3.6 Proof Number Search . . . 14
  - 3.6.1 Initialization . . . 16
  - 3.6.2 Selection . . . 17
  - 3.6.3 Expansion . . . 18
  - 3.6.4 Backpropagation . . . 18
  - 3.6.5 Backtracking . . . 19
  - 3.6.6 Seesaw effect and DeepPNS . . . 19

4 Current Work 20
- 4.1 Problem formulation . . . 20
- 4.2 Expected result . . . 20
- 4.3 Problem breakdown . . . 20
- 4.4 DeepPNS and Backjumping . . . 21
  - 4.4.1 Initialization with reason computation . . . 22
  - 4.4.2 Selection with Backjumping . . . 23
  - 4.4.3 Expansion with reason computation . . . 24
  - 4.4.4 Backpropagation with Backjumping . . . 26
  - 4.4.5 Backtracking with reason . . . 27
- 4.5 Implementation details and Experimental results . . . 27
  - 4.5.1 Enable SAT solver . . . 27
  - 4.5.2 R value in DeepPNS and Backjumping . . . 28

5 Future Study 31
- 5.1 Further investigate the seesaw effect . . . 31
- 5.2 Combine QCDCL and DeepPNS . . . 31
- 5.3 Minimal reason computation with SAT solver enabled . . . 31
- 5.4 Plan for thesis B and thesis C . . . 32

6 Appendix **34**
6.1 QDLLBJ algorithm pseudocode 34
6.2 Original QDLL algorithm 35

1 Introduction

Quantified Boolean Formulas or QBF were proposed in the 1970s as a natural generalization of the propositional logic. Quantified Boolean logic is more generalized framework compared to propositional logic, however, the performance of QBF solvers lag far behind that of SAT solvers. Modern SAT solvers can solve propositional formulas with thousands of variables and millions of clauses efficiently, but the formula capable to be solved by state of art QBF solver is far less than that scale. Another research field that achieved massive success over the past two decades was the strategy game [10]. But most of the research was related to "play well" not "play perfectly". QBF and 2-player strategy games are similar at a theoretical level, some researchers have already started to use QBF to encode a positional game [16]. However, the connections between the two areas remain unexplored in practice and the methods used for solving the two problems are vastly different. This thesis project focused on a heuristic search algorithm called proof number search that is frequently used in 2-player strategy game and how it can be applied to design and implement a QBF solver. Since the duration of the thesis project is only 1 year, it is unrealistic to design and implement a proof number search based QBF solver that can outperform state of art search based QBF solvers such as DepQBF. Instead, we will investigate how heuristic search algorithm that is frequently used in strategy game can be combined with some QBF solving algorithms such as dependency backtracking [4] and QCDCL [6], and if this new combination can potentially improve the performance of depth first search based QBF solver.

This report is structured as follows. In section 2, I will outline some key definitions and notations that are essential for the remaining report. In section 3, I will do the literature review by outlining some of the QBF and game solving techniques studied in the past 20 years. In section 4, I will introduce the motivation of applying proof number search in search based QBF solving, and how to design a QBF solver based on a variation of the proof number search algorithm called DeepPNS. In that section, I will also explain how an optimization trick in the QBF world called backjumping can be applied in the scenario of DeepPNS. Experimental results of this new algorithm combination can be found at the end of this section. I will end the report with future works and my plan for thesis B and thesis C in section 5.

2 Background

In this section, we will outline some of the important preliminaries on QBF and define the notations that will be used for the rest of the report.

A literal is a variable l or its negation $\neg l$. A clause C is a disjunction of literals. A propositional formula is in conjunctive normal form if it is a conjunction of clauses. We define a variable l occurs in a propositional formula, if at either l or $\neg l$ appears in the formula. A Quantified Boolean Formula (QBF) is an expression consists of blocks of existential ($\exists$) or universal ($\forall$) quantifiers followed by a propositional formula [16].

$$Q_1X_1Q_2X_2...Q_nX_n\Phi \tag{1}$$

(1) is a standard form of a QBF expression. Which $Q_i \in \{\exists, \forall\}$, $Q_i \neq Q_{i+1}$ for all i, X_i are the set of variables bounded by Q_i, and Φ is a propositional formula. We further restrict that for a valid QBF, no variable $x \in X_i$ and $x \in X_j$ such that $i \neq j$.

Definition 2.1 (Free variable)**.** A variable v is called free (or unbounded) in a QBF expression, if v occurs in Φ but $v \notin X_i$ for all $1 <= i <= n$.

Definition 2.2 (Closed and prenex form)**.** A QBF is *closed* if all variables are bounded. A QBF is in *prenex form*, if Φ is in conjunctive normal form.

If a QBF contains free variables, we assume these variables are bounded existentially in the leftmost quantifier block. If a QBF is not in prenex form, we can transform Φ to conjunctive normal form in linear time [6]. Hence, for the rest of the report, we assume that we only deal with QBF expression that is both closed and in prenex form.

Definition 2.3 (Quantifier level)**.** Suppose a variable $v \in X_i$ in the QBF expression 1, then the quantifier level of v is $\mathit{level}(v) = i$. We further define $v \in X_1$ as the variables in the outermost quantifier block (or has the highest decision level).

The aim of the QBF solving task is to tell whether a QBF expression of form 1 is satisfiable or not. This leads us to the following recursive definition on QBF semantic.

Definition 2.4 (QBF semantic)**.** If the QBF f contains an empty clause, then f is unsatisfiable. If f has all clauses satisfied (Φ is empty), then f is satisfiable. Otherwise, suppose f is $\exists x\psi\Phi$, f is satisfiable iff either f_x or $f_{\neg x}$ is satisfiable. Otherwise, if f is $\forall x\psi\ \Phi$, f is satisfiable iff both f_x and $f_{\neg x}$ are satisfiable [6].

In the above definition, ψ is the quantifier block obtained by removing a variable x from X_1 of the original quantifier prefix. And we define a clause to be empty (contradictory) if it contains no literal l such that the variable $|l|$ is bounded by an existential quantifier. We further define f_x as the QBF which propositional part Φ' derived from Φ by removing all the satisfied clauses C such that $x \in C$, and removing $\neg x$ from all clauses C such that $\neg x \in C$.

For the rest of the report, there are many statements "l is existential (or universal)", it is equivalent to expressing "the variable $|l|$ is bounded by an existential (or universal) quantifier in f". We would insist on this simplified version of the expression when no confusion can be raised.

A direct optimization to the above recursive procedure of solving QBF is to detect unit and pure literals, and simplify the expression.

Definition 2.5 (Unit literal)**.** A literal l is called *unit* in a QBF f, iff the variable is existential and there exists a clause $C_f \in \Phi$ such that l is the only existential variable in C_f and all other variables has a quantifier level greater than $|l|$. For the convenience of the rest of the report, we also call l is unit with respect to the clause C_f.

From the semantic of QBF, it is not hard to derive whenever l is unit, f is equivalent to f_l [3]. The process of repeatedly update f to f_l until there exists no more literal l such that l is unit is called **unit propagation**.

Definition 2.6 (Pure literal)**.** A literal l is called *pure* in a QBF f, iff l is existential and $\neg l$ does not occur in any clauses in Φ or l is universal and l does not occur in any clauses in Φ.

From the semantic of QBF, it is not hard to derive that whenever l is pure, f is equivalent to f_l [3]. The process of repeatedly update f to f_l until there exists no more literal l such that l is pure is called **pure literal elimination**.

The process of repeatedly applying the recursive procedure defined in 2.4, unit propagation, and pure literal elimination is called **QDLL algorithm**, which we will discuss in section 3.1.

For the convenience of the rest of the report, it is necessary to introduce the following definitions before we move on to the next section.

Definition 2.7 (Assignment)**.** (adapt from [6]) An assignment for a QBF f is a sequence of literals $\mu = l_1; l_2; ...l_n$ such that $n >= 0$ and for each literal $l_i \in \mu$, l_i either unit or pure or is bounded in the outermost quantifier block in $f_{l_1;l_2...l_{i-1}}$. We call f_μ as the QBF subformula with the assignment μ. A variable that is neither unit nor pure in an assignment a *branching variable*.

Definition 2.8 (Forced Assignment Sequence)**.** Suppose f is a QBF expression that has neither an empty clause nor all clauses are satisfied, the *forced assignment sequence* $S[f]$ is the sequence of literals assigned while executing unit propagation and pure literal elimination on f.

The above definition can be written as the following pseudo code, and the forced assignment sequence $S[f]$ the return statement of the simplification procedure $simplify(f)$.

Algorithm 1: Simplification Procedure

```
1  SIMPLIFY($f$)
2      S := [ ]
3      g := $f$
4      while $g$ has a unit literal or pure literal do
5          if $g$ has a contradictory clause then
6              return $S$
7          if all clauses in $g$ are satisfied then
8              return $S$
9          if $l$ is unit in $g$ then
10             append $l$ to the end of S
11             $g := g_l$
12         if $l$ is pure in $g$ then
13             append $l$ to the end of S
14             $g := g_l$
15     f := g
16     return $S$
```

Need to be aware that the forced assignment sequence is a sequence not a set, this means that the order of elements in S must obey the chronological order the elements are assigned during the unit propagation and pure elimination procedure. This property is essential in algorithms that will be introduced in later sections.

3 Literature Survey

There are two main types of QBF solver, expansion based solver and search based solver. These two types of solvers are incomparable because these solvers use completely different proof system and both types of solvers are specialized on some families of benchmarks. Successful expansion based QBF solvers include QUANTOR [1], RAReQS [9], and caqe [18]. Successful search based QBF solvers include QUBE [4, 6], Quaffle [20], semprop [13], GhostQ [11], and DepQBF [14]. Most expansion based solver solves the formula by expanding either existential or universal variables and pass the expanded formula to the SAT solver to solve the QBF. Some expansion based solvers such as RAReQS and caqe also combines counter example guided abstraction refinement (CEGAR) which is frequently used in model checking in the expansion process. For the search based solver, the solver combines either dependency directed backtracking or clause learning with the QDLL prcedure to solve the QBF. In the thesis project, we will focus on the search based QBF solving approaches and for the remaining of the literature review section, we will review some search based QBF solving approaches from a technical perspective.

3.1 QDLL Procedure

In 1998, Cadoli, Ciovanardi, and Schaerf presented the baseline procedure of search based QBF solving which is called QDLL [3]. This approach is a modification of the famous Davis–Putnam–Logemann–Loveland (DPLL) SAT solving algorithm which variations are used in almost all of the search based solver. QDLL is a sound and complete procedure of QBF solving [6]. The baseline QDLL algorithm can be described as follows. In this pseudocode, f is the original input QBF expression, while μ is the partial assignment sequence. The initial call to the function is $QDLL(f, \emptyset)$

Algorithm 2: QDLL algorithm

```
Boolean QDLL($f$, $\mu$)
    if $f_\mu$ has a contradictory clause then
        return False
    if all clauses in $f_\mu$ are satisfied then
        return True
    if $l$ is unit in $f_\mu$ then
        return QDLL($f$, $\mu; l$)
    if $l$ is pure in $f_\mu$ then
        return QDLL($f$, $\mu; l$)
    $l$ = get_literal($f$, $\mu$)
    if $l$ is existential in $f$ then
        return QDLL($f$, $\mu; l$) or QDLL($f$, $\mu; \neg l$)
    return QDLL($f$, $\mu; l$) and QDLL($f$, $\mu; \neg l$)
```

As we can see from Algorithm 2, the conventional QDLL procedure would return false if the formula f_μ contains a contradictory clause; and would return true if the formula f_μ has all clauses satisfied. If there exists a unit or pure literal l in f_μ then the QDLL function would return the truth value of $f_{\mu;l}$. Otherwise, it would get a literal l in the outermost quantified block in f_μ. Details of how *get_literal*() works is going to be discussed in later sections. If l is existential, QDLL would return true if and only if either $f_{\mu;l}$ or $f_{\mu;\neg l}$ is true. Otherwise, QDLL would return true if and only if both $f_{\mu;l}$ and $f_{\mu;\neg l}$ are true.

Although Algorithm 2 is referred as the QDLL procedure in later papers such as [4, 13, 6], in the original 1998 paper [3], two additional special cases were considered (line 6-10 in algorithm 18). One is when f_μ contains only existential variables, f_μ is equivalent to a propositional formula. It might be helpful to call a SAT solver to solve f_μ. In addition, let G be the propositional formula obtained from f_μ by removing all the universal literals from f_μ. If the propositional formula G is satisfiable, then f_μ is also satisfiable. The later case is also called **trivial truth**. Since the difference between the original QDLL algorithm and the conventional one is insignificant, the pseudocode of the original QDLL algorithm is put in the appendix (see section 6.2 for details).

To avoid ambiguity, unless explicitly stated we refer Algorithm 2 as the

QDLL procedure in later sections.

3.2 Backjumping Trick

One way to optimize the baseline QDLL algorithm is the backjumping trick or dependency backtracking. This trick was proposed by Giunchiglia, Narizzano, and Tacchella in 2001. The aim of the algorithm is to reduce some unnecessary branching of the baseline QDLL procedure [4]. The algorithm frequently computes a set of literals called *reason* that are responsible for the satisfiability or unsatisfiability of the QBF formula f_μ. The following definition of reason is from the paper Backjumping for quantified Boolean logic satisfiability [4].

Definition 3.1 (Reason). A set of literals $\nu \in \mu$ is called a reason for f_μ satisfiability iff for all possible assignment ν' that satisfies the following 3 conditions:
• $\{|l| : l \in \nu'\} \subseteq \{|l| : l \in \mu\}$
• $\{l : l\ is\ universal, l \in \nu\} \subseteq \nu'$
• $\{l : l\ is\ existential, l \in \nu'\} \subseteq \{l : l \in \mu\}$, f_μ is satisfiable iff f'_ν is satisfiable. Similarly, a set of literals is called a reason for f_μ unsatisfiability iff for all possible assignment ν' that satisfies the following 3 conditions:
• $\{|l| : l \in \nu'\} \subseteq \{|l| : l \in \mu\}$
• $\{l : l\ is\ existential, l \in \nu\} \subseteq \nu'$
• $\{l : l\ is\ universal, l \in \nu'\} \subseteq \{l : l \in \mu\}$, f_μ is unsatisfiable iff f'_ν is unsatisfiable.

One intuitive interpretation of the reason is as follows. Define $Assign(f, \mu)$ as the QBF expression obtained from f by assigning all literals in μ. It is extremely important to note that $Assign(f, \mu) \neq f_\mu$, because for f_μ, μ must be an assignment (see Definition 2.7) of f. Then, let E_μ be all the existential literals in μ and U_μ be all the universal literals in μ. The reason ν for f_μ unsatisfiability is a set of existential literals in μ such that $Assign(f, E_\mu; \nu)$ is unsatisfiable. The reason ν for f_μ satisfiability is a set of universal literals in μ such that $Assign(f, U_\mu; \nu)$ is satisfiable.

The benefit of defining the reason is when the branching literal l is not in the reason for f_μ we can skip the exploration of $\neg l$ and prune the searching space. For the remaining of this subsection, we will review how the reason can be computed at a terminal state and how the reason can be maintained during backtracking.

3.2.1 Reason computation at terminal state

We call a searching state terminal if the formula f with assignment μ has all clauses satisfied or contains an empty clause. The method of computing the reason for unsatisfiability and reason for satisfiability is given by the following two theorems.

Theorem 3.1. *Suppose the QBF formula f_μ contains a contradictory clause C, then the set $\{\neg l : l \in C\ and\ l\ is\ existential\}$ is a reason for f_μ unsatisfiability.*

Proof. This theorem trivially holds by the definition of reason for unsatisfiability and the consequence of Theorem 1 in [4]. □

Theorem 3.2. *Suppose the QBF formula f_μ has all clauses satisfied, then the reason for f_μ satisfiability can be obtained by repeatedly removing universal literals from all universal literals in μ such that for each clause C in the original formula f, if l occurs in C, there is another literal $l' \in \mu$ such that l' occurs in C [5].*

Proof. This theorem trivially holds by the definition of reason for satisfiability and the consequence of Theorem 3 in [4]. □

Based on the above 2 theorems, we can write the pseudo code for *getReason* as follows.

Algorithm 3: get the reason for f_μ

```
1  Reason GETREASON($f$, $\mu$)
2      if ($f_\mu$ has a contradictory clause $C$) then
3          reason := all existential literals in $C$
4          return reason
5      if (all clauses in $f_\mu$ are satisfied) then
6          reason := all universal literals in $\mu$
7          while reason is not empty do
8              if < $l$ is in reason such that all clauses in the formula after
                 removing $l$ from $f_\mu$ are still satisfied > then
9                  reason := reason$\backslash\{l\}$
10                 $\mu := \mu\backslash\{l\}$
11         return reason
12     return NULL
```

Although it was not explicitly mentioned in the original paper [4], it is extremely important to note that in Theorem 3.2 and Algorithm 3, when we compute the reason for satisfiability, the order of removing universal literals is not arbitrary. It is valid to consider the universal literals from the last assigned to the first assigned but not from the first assigned to the last assigned. This is because a universal pure literal should never enter a reason for satisfiability, otherwise the all the reason computation rule described in section 3.2.2 no longer holds [6].

3.2.2 Reason computation during backtracking

Based on the definition of reason, the reason for $f_{\mu;l}$ is not equivalent to the reason for f_μ. Thus it is necessary to come up with an efficient method to maintaining a valid set of reasons during backtracking. One possible method of calculating the reason for f_μ given the reason for $f_{\mu;l}$ and $f_{\mu;\neg l}$ is clearly described in Theorem 2 and Theorem 4 in [4] and Section 5 in [13]. There are 12 possible cases during backtracking in the QDLL procedure. The reason computation rule of these 12 cases can be summarized to Table 1. In this table, we represent the reason for $f_{\mu;l}$ as ν and the reason for $f_{\mu;\neg l}$ as ν'. In this table,

#	**Property of** l	**Result of** $f_{\mu;l}$	**Result of** $f_{\mu;\neg l}$	**Reason for** f_μ
1	Pure	SAT/UNSAT	-	$\nu \backslash \{l\}$
2	Unit	SAT	-	ν
3	Unit $l \notin \nu$	UNSAT	-	ν
4	Unit $l \in \nu$	UNSAT	UNSAT	$\nu \cup \nu' \backslash \{l \cup \neg l\}$
5	$\exists$ Br.	SAT	-	ν
6	$\exists$ Br. $l \notin \nu$	UNSAT	-	ν
7	$\exists$ Br. $l \in \nu$	UNSAT	SAT	ν'
8	$\exists$ Br. $l \in \nu$	UNSAT	UNSAT	$\nu \cup \nu' \backslash \{l \cup \neg l\}$
9	$\forall$ Br.	UNSAT	-	ν
10	$\forall$ Br. $l \notin \nu$	SAT	-	ν
11	$\forall$ Br. $l \in \nu$	SAT	UNSAT	ν'
12	$\forall$ Br. $l \in \nu$	SAT	SAT	$\nu \cup \nu' \backslash \{l \cup \neg l\}$

Table 1: Reason computation rule during backtrack
- means skipped during QDLL procedure
Br. is short for branching

case 6 and 10 corresponds to the case which the right branch of the search tree would be explored the baseline QDLL procedure but skipped in the QDLLBJ algorithm.

3.2.3 QDLL Backjumping algorithm

QDLL Backjumping algorithm (QDLLBJ) can be implemented either recursively or iteratively. In order to see the body of the QDLL algorithm is not changed, in this subsection I will provide a recursive version of the algorithm. The pseudo code for QDLLBJ is quite long and is presented in the appendix.
As we can see from Algorithm 17, backjumping would not change the main structure of the QDLL procedure, it would just add the dynamic reason computation during the backtrack. The return statement of the algorithm is modified to be a pair that contains the truth value of the QBF expression f_μ and the reason for f_μ. The initial call to the algorithm is $QDLLBJ(f, \emptyset)$. Similar to the baseline QDLL procedure, if the formula f_μ contains a contradictory clause, the QDLLBJ would return false and the reason for unsatisfiability can be computed based on Theorem 3.1. If the formula f_μ has all clauses satisfied, QDLLBJ would return true and the reason for satisfiability can be computed based on Theorem 3.2.

If f_μ contains a unit literal (line 6-13), similar to QDLL algorithm, QDLLBJ would perform unit propagation and calculate the reason for f_μ based on case 2 to 4 in Table 1. If f_μ contains a pure literal (line 14-17), QDLLBJ would perform pure literal elimination and calculate the reason for f_μ based on case 1 in Table 1. If f_μ contains neither unit nor pure literal, QDLLBJ would get a literal in the outermost quantifier block and solve the two subproblems $QDLLBJ(f, \mu; l)$ and $QDLLBJ(f, \mu; \neg l)$ separately. Depending on whether l is existential or universal and the return statement of the two subproblems, the reason for f_μ will be calculated during backtracking based on case 5 to 12 in Table 1 (line 18-35).

There are two main advantages of QDLLBJ algorithm. Firstly, the implementation difficulty of QDLLBJ is relatively low and the computation overhead for the reason calculation is very little [4]. Secondly, as we can see in Table 1, case 6 and 10 corresponds to cases that QDLLBJ can skip exploring $f_{\mu;\neg l}$ while all the other cases QDLLBJ explores exactly the same searching space as baseline QDLL. Thus, in terms of the size of the searching space, QDLLBJ is guaranteed to perform not worse than baseline QDLL.

3.3 QCDCL

Although backjumping trick can reduce the search space by skipping the exploration of the right branch of the search space when some conditions are reached, it is rarely applied in state of art search based QBF solvers such as DepQBF and GhostQ. The main shortcoming of Backjumping algorithm is the reason computed at one state can only affect the search on the current path of the search tree. In other words, since conflicts and solutions can never affect the search on a different path of the search tree, the pruning effect of the backjumping trick is limited. One of the simplest formula that cannot be solved by QDLLBJ algorithm is the benchmark tree-exa2-50. Although there are only 100 variables and 52 clauses in this formula, the pruning condition in backjumping can never be triggered. The QDLLBJ algorithm is degenerated to baseline QDLL procedure and for this specific instance no implementation of the baseline QDLL procedure can solve the formula within one day [13]. However, for this specific instance, Q-resolution based solvers such as QUANTOR can easily solve it within 1 second. Thus it might be tempting to combine some resolution based solving technique with the search based algorithm, and such combination is called the QCDCL algorithm.

The QCDCL algorithm was proposed by Zhang [20], Giunchiglia [6], and Letz [13] in the early 2000s. This algorithm is a modification of the conflict-driven clause learning algorithm (CDCL) in SAT. Similar to backjumping, QCDCL also applies the QDLL procedure. The main idea of the algorithm is to learn additional clauses (resp. terms) caused by conflicts (resp. solutions) that can be added to the original formula conjunctively (resp. disjunctively). With these additional clauses and terms, more unit propagation might be triggered during the QDLL solving procedure, and hence the search space can be pruned. Unlike backjumping the reason for conflicts or solutions are immediately discarded after backtracking, in QCDCL the clauses and terms are calculated based on Q-resolution and added to the original formula [6]. This means that the conflicts or solutions in the searching procedure can not only affect the search on the current path of the search tree but also affect the search on different parts of the search tree. The shortcoming of the algorithm is that it does not guaranteed to outperform baseline QDLL in terms of the size of the searching space. For example, if clauses and terms are added to the formula without any consideration, pure literal elimination can be blocked. However, in many popular benchmarks QCDCL outperforms backjumping and it is applied as the cored QBF solving technique in many efficient QBF solvers such as QUBE, Quaffle and DepQBF.

Due to the lack of time, details of QCDCL will be learned and investigated in thesis B and C.

3.4 Branching Heuristic

In the QDLL procedure, if there is no unit or pure literal, a variable in the outermost quantifier block is selected as the branching variable. In this section, I will review how $get\_literal()$ is usually done in search based QBF solving algorithm. In the SAT world, because there is no quantifier block restriction, any literal can be selected. However, in the QBF world this is no longer hold. To determine the truth value of f_μ, we can only select literal l such that no variable p has a higher decision level than variable $|l|$. The heuristic that is used to select the next branching literal can also be called **Branching Heuristic** [15]. In general, for QBF solving, SAT solving heuristics are applied to the set of variables in the highest decision level.

In the remaining of the section, we will briefly review three of the most popular branching heuristics that are applicable to QBF solving.

3.4.1 Famous branching heuristics

• Dynamic largest combined sum (DLCS). When $get\_literal()$ is called, a variable v that is bounded in the outermost quantifier block and occurs the most frequent in f_μ is selected whether to explore v or $\neg v$ first is decided arbitrarily. In other words, suppose there are C_1 clauses in f_μ contains v and C_2 clauses in f_μ contains $\neg v$. Then, the variable v that is bounded in the outermost quantifier block with the largest $C_1 + C_2$ will be selected [15].
• BOMH's heuristic (BOMH). This heuristic was proposed by Speckenmeyer in the 1992 SAT competition [2]. For each variable v that occurs in the outermost quantifier block, we maintain a vector

$$BOMH_v =< H_1(v), H_2(v), ..., H_n(v) >.$$

which the kth term $H_k(v)$ is calculated by

$$H_k(v) = max(h_i(v), h_i(\neg v)) + 2 * min(h_i(v), h_i(\neg v))$$

which the $h_i(v)$ represents the number of clauses in f_μ contain variable v positively while $h_i(\neg v)$ represents the number of clauses in f_μ contain variable v negatively [15]. When $get\_literal()$ is called, the variable v that which has the maximal $H_k(v)$ will be selected as the next branching variable. The intuition of this heuristic is to quickly shorten the shortest undetermined clause, such that more unit propagation might be triggered. The 2004 solver QUBE-BJ used BOMH as its branching heuristic [4].

• Variable State Independent Decaying Sum (VSIDS). This branching heuristic is a modification of the DLCS heuristic was firstly used by the famous SAT solver Chaff, detailed description of this heuristic can be found in [17]. However, we need to note that this branching heuristic is only useful when new clauses can be added to the original formula during the QBF solving procedure which would never happen for algorithms implemented in thesis A. Hence, this heuristic might only be useful to the project in thesis B and thesis C after QCDCL is implemented.

3.4.2 Performance of different branching heuristics

During thesis A, both BOMH and DLCS heuristics were implemented and tested. For all the experiments, I used the self implemented QDLL backjumping solver, and set a time limit of 15 minutes for each benchmark. The number of branching nodes required to solve a benchmark is recorded and used as a way to measure the performance of the branching heuristic. As a comparison, I will also list the number of branching nodes QUBE-BJ produced for these benchmarks, and all the data related to the performance of QUBE-BJ are retrieved from [4, 7]. Need to be aware that in the QBFeval 2004, although QUBE-BJ showed strong performance in terms of the number of solved instances, it produced wrong output in many instances [12]. From my implementation experience, if there are bugs in the backjumping implementation, invalid pruning might occur and the searching space can be reduced. Thus, the result of QUBE-BJ might not accurately reflect the actual number of node expansions required to solved the instance.

Despite the potential inaccurate results from QUBE-BJ, from Table 2 we can see that even if all three solvers use the same QDLL and Backjumping algorithm as the QBF solving method, the performance on different benchmarks are mixed, no solver expand fewer nodes than others in all benchmarks, and no branching heuristic outperforms the other. Hence unless explicitly stated, we will use DLCS heuristics to select the branching variable for all later experiments in this report. In addition, even for the same branching heuristic, after picking the branching variable v, whether to explore v or $\neg v$ first can produce a significant difference to the solver's performance, this can be seen from the performance of different solvers on the Impl family. However, instead of trying to come up with a heuristic to deal with this issue, we should note that this is one of the drawback of the depth first search based solving procedure. For example suppose v is existential and both f_v and $f_{\neg v}$ are satisfiable. If solving f_v is significantly simpler than $f_{\neg v}$ but the solver explored $f_{\neg v}$ first, depth first search based solving procedure might be trapped on the wrong side of the search tree and fail to solve the instance efficiently. How this issue might be solved is one of the main target of the thesis project.

Instance	**Result**	**DLCS**	**BOMH**	**QUBE-BJ**
chain12v.13	T	8215	8215	4119
adder-2-sat	T	3383	2131	1643
adder-2-unsat	F	17946	41129	20256
BLOCK3ii4.3	F	-	-	59390
BLOCK3iii4	F	110714	98574	18952
BLOCK3ii5.2	F	-	-	525490
BLOCK3iii.5	T	-	58559	-
Impl18	T	37	37	22991578
Impl20	T	41	41	-
TOILET6.1.iv.11	F	1896123	1414371	139505
TOILET6.1.iv.12	T	218581	207319	56587
TOILET7.1.iv.14	T	3036638	3007306	688621
L*A1	F	-	-	265610
k_path_n-2	T	13837	30007	N/A
k_path_n-3	T	1010031	-	N/A

Table 2: Number of node expansions for DLCS heuristic, BOMH heuristic, and QUBE-BJ solver
- means the solver timeout after 15 minutes
N/A means the solver solved the instance but the number of branching node is unknown

3.5 Strategy Game Interpretation

QBF can be interpreted as an and-or strategy game which the existential player is the maximizer that controls the *or* node and tries to satisfy the formula while the universal player is the minimizer that controls the *and* node and tries to falsify the formula. This interpretation indicates the similarity between strategy games and QBF in theoretical level. Based on this interpretation, QDLL procedure can be considered as a depth first search approach to solve the and-or strategy game. In the last subsection of the literature review part of the report, we will review a very famous two-player strategy game solving algorithm called proof number search.

3.6 Proof Number Search

Proof number search algorithm was proposed by Victor Allis in 1994. It is a best first search algorithm that can be used for solving two player strategy games. The algorithm prioritize the search on the direction of the most proving node [10]. In recent years variations of this algorithm such as DeepPNS, PN2, PN* and df-pn has achieved massive success in many two player and-or strategy game such as Hex, Othello, tsume-shogi, and connect6 [8, 19]. For the remaining of the section, I will review how the baseline version of the PNS algorithm works. Since proof number search is a best first search algorithm, the entire search tree is stored in memory. Each node stores the proof number (*node.pn*), the disproof number (*node.dn*), pointers to the two children ($node.left$ and *node.right*), pointer to the parent (*node.parent*), and the depth of the node (*node.depth*). The proof number indicates the minimum number of leaves in the subtree that must be proved such that the entire subtree can be proved while the disproof number indicates the minimum number of leaves in the subtree that must be disproved such that the entire subtree can be disproved. In proof number search a node can have 3 states, satisfied (proved), falsified (disproved), and undetermined, which is described by the following definition.

Definition 3.2. A proof number search tree node is **falsified (disproved)** if and only if its disproof number is 0 and proof number is ∞. A node is **satisfied (proved)** if and only if its disproof number is ∞ and proof number is 0. A node that is neither satisfied nor falsified is called **undetermined**. A node that is not undetermined is called **solved**.

Need to be aware that for general proof number search algorithm, a node can have **more than 2 children**. However, for the purpose of the project, we are only interested in the proof number search algorithm with branching factor of 2.

The main procedure of PNS algorithm can be described as follows. It can be seen from Algorithm 9, before the start of the search, a root node is created based on the information of the initial game state (line 2). After that we repeatedly execute the following procedure until the number of iterations exceed the limit or the root has been proved or disproved.

- Increment the number of iterations by 1.
- Select the most proving node (line 6,7).
- Expand the most proving node (line 8).
- Backtrack from the current node all the way back to the root until the a node's proof number (*current.pn*) and a node's disproof number (*current.dn*) are not changed (line 11-15). During backtracking, we back-propagate the information of the two children of *current* to the current node (line 10).

Algorithm 4: PNS Algorithm

```
Boolean PNS($game\_state$)
    $root.depth = 1$, $current := root :=$ initialize($game\_state$)
    $iter := 0$
    while ($root$ is not solved and $iter <= maximum\_iteration$) do
        $iter := iter + 1$
        while $*next$ = selection($current$, $game\_state$) is not null ) do
            $current := next$
        expand($current$, $game\_state$)
        while ($current$ is not null ) do
            backpropagate($current$)
            if ($current$ is root) then
                break
            if ($current.pn$ and $current.dn$ are not changed ) then
                break
            $current := backtrack(current, game\_state)$
    if ($root$ is not solved ) then
        return UNSOLVED
    if ($root$ is falsified ) then
        return FALSE
    return TRUE
```

In this algorithm, there are 5 important procedures which are initialization, selection, expansion, back-propagation, and backtrack. For the remaining of the section, I will review how these procedures work based on the paper [10].

3.6.1 Initialization

As we can see from Algorithm 5, during initialization we will create the root for the search tree. If the current game state is a win for the minimizer then we initialize $root.pn = \infty$ and $root.dn = 0$. If the game state is a win for the maximizer then we initialize $root.pn = 0$ and $root.dn = \infty$. Otherwise, we would assign $root.pn = 1$ and $root.dn = 1$.

Algorithm 5: Initialize

```
PNSNODE INITIALIZE($game\_state$)
    $node.left := node.right := node.parent := NULL$
    if $game\_state$ is not terminal then
        $node.pn = 1$, $node.dn = 1$
    else
        if $game\_state$ is a win for the maximizer then
            $node.pn = 0$, $node.dn = \infty$
        else
            $node.pn = \infty$, $node.dn = 0$
    return $node$
```

Note that if the game state is not terminal, the initialization of the proof number and disproof number is not unique. Another alternatives is to initialize $pn = 1$ and $dn = 2$ if the node is controlled by the maximizer while $pn = 2$ and $dn = 1$ if the node is controlled by the minimizer. This alternative initialization method is called **mobility initialization** [10].

3.6.2 Selection

Select the most proving node is the most important procedure of any variations of proof number search algorithm.

Algorithm 6: Find most proving node

```
PNSNODE SELECTION($node, game\_state$)
    if ($node$ has no children ) then
        return $NULL$
    $ret := \arg\min_{c \in node.child} dpn(c)$
    if ($ret$ is the left child ) then
        $game\_state$ := game_state after taking the action on the left
    else
        $game\_state$ := game_state after taking the action on the right
    return $ret$
```

As described in Algorithm 6, the selection procedure would return null if the current node does not have children (line 2, 3) and the most proving node (MPN) is found. Otherwise, the most proving node is in the child with the minimum dpn value (line 4) [8]. The dpn value helps to determine the location of the MPN which will be further discussed in later section. For baseline proof number search, the dpn value of a child is defined as

$$dpn(c) = \begin{cases} c.pn & \text{if } node \text{ is controlled by maximizer} \\ c.dn & \text{if } node \text{ is controlled by minimizer} \end{cases} \tag{2}$$

[10]. Need to be aware is in this expression, the dpn value of a child of the

current node is related to the player that controls the current node. If the MPN is in the subtree of the left child of the current node, we update the game state by taking the option on the left; otherwise, the MPN is in the subtree of the right child of the current node and we update the game state by taking the option on the right.

3.6.3 Expansion

After the selection process terminates, we have reached the MPN of the current search tree (let us denote it as *current*). From the property of the dpn value, as we will described in later section, it is guaranteed that *current* has neither been proved nor disproved. Since *current* is the most promising node for the search, based on the best first search strategy, we should continue our search along this direction. Therefore, PNS will expand the 2 children of *current* in the following way. As we can see from Algorithm 7, during the expansion, we initialize the left child of *current* based on the information of the game state after taking the left move of the current node (line 2-3). Similarly, we initialize the right child of *current* based on the information of the game state after taking the right move of the current node (line 4-5). After we have obtained the two children, we set their parent to *current* (line 6).

Algorithm 7: Expansion

```
1 EXPAND($current$, $game\_state$)
2     $fl$ := game state after taking the left move at $game\_state$
3     $L := initialize(fl)$
4     $fr$ := game state after taking the right move at $game\_state$
5     $R := initialize(fr)$
6     $L.parent := R.parent := current$
7     $L.depth := R.depth := current.depth + 1$
```

3.6.4 Backpropagation

To make sure the information stored in a tree node is consistent during the entire search, these information must be updated during the backtracking procedure. This update procedure is called **backpropagation**. During the backpropagation procedure, we update the current node's information based on the information stored in its left and right child.

Two information need to be updated during the backpropagation, the proof number *current.pn*, the disproof number *current.dn*. It can be seen from Algorithm 8, the update of the proof and disproof number is separated into 2 dual cases (line 2-7).

- If *current* is controlled by the maximizer, then we update the proof number of the current node as the minimum proof number of the children and the disproof number of the current node as the sum of disproof number of the children [10].

- Otherwise, we update the proof number of the current node as the sum of proof number of the children and the disproof number of the current node as the minimum of disproof number of the children [10].

In order to save memory, if a node is solved, we remove its left and right child (line 8, 9).

Algorithm 8: Backpropagation

```
1 BACKPROPAGATE($node$)
2     if $node$ is controlled by maximizer then
3         $node.pn := min(node.left.pn, node.right.pn)$
4         $node.dn := node.left.dn + node.right.dn$
5     else
6         $node.dn := min(node.left.dn, node.right.dn)$
7         $node.pn := node.left.pn + node.right.pn$
8     if ($node$ is solve) then
9         $node.left := node.right := NULL$
```

3.6.5 Backtracking

During the backtracking procedure, we cannot simply let the current node point to its parent, because this would make our game state inconsistent. We must reverse the game state to the state before we take the move at the parent node.

3.6.6 Seesaw effect and DeepPNS

Seesaw effect is always one of the challenging problem for proof number search. It is the effect that the location of the most proving node changes rapidly during the proof number search procedure. This effect can potentially weakened the performance of proof number search [8]. A variation of the proof number search called DeepPNS can potentially reduce the seesaw effect. DeepPNS deals with the seesaw effect by its most proving node selection formula. During the initialization, *node.deep* is assigned as $\frac{1}{node.depth}$. We have mentioned in Algorithm 6, PNS would prioritize the search in a child which has the minimum dpn value. In DeepPNS the dpn value of a node is defined as

$$dpn(c) = \begin{cases} \delta' * R + d * (1 - R) & \text{if } node \text{ is undetermined} \\ \infty & \text{otherwise} \end{cases} \tag{3}$$

In this expression $d = node.deep$, and R is a parameter between 0 and 1 used to control seesaw effect, and

$$\delta' = \begin{cases} (1 - \frac{1}{node.dn}) & \text{if } node \text{ is controlled by the minimizer} \\ (1 - \frac{1}{node.pn}) & \text{if } node \text{ is controlled by the maximizer} \end{cases} \tag{4}$$

[8]. Furthermore, during backpropagation the *deep* value of the current node is updated to the deep value of the child with the minimum dpn value. An

interpretation of the above expression is, as R increase from 0 to 1, the algorithm steadily transferred from pure depth first search to pure best first heuristic search. In the field of Hex, a good parameter for R is 0.65 and in Othello a good parameter is 0.5 [8] .

4 Current Work

Depth first search or iterative deepening search are rarely used in strategy game solvers. The reason is because both algorithms assume a solution of the game can be found at a relatively low searching depth. However, for strategy games that only exists a very deep solution, depth first search is not ideal. This is mainly because depth first based game solvers might be trapped on the wrong side of the searching space. For example, in shogi a depth first search based solver can rarely solve a game with more than 17 moves. But proof number search based solvers can solve a game with more than 100 moves [19]. In section 3.5 we reviewed how QBF can be interpreted as a strategy game with branching factor 2. For some non-trivial QBF instances a terminal state cannot be reached before a searching depth of 50. This means that QBF is also a game with small branching factor but the solution only exists at a very deep level. Hence, although most of the search based QBF solving procedure applies depth first search procedure, depth first search might not be ideal for QBF.

4.1 Problem formulation

For this thesis project, we will investigate how best first search algorithms that are used for and-or strategy game can be applied to QBF solving and analyse its strength and weaknesses. The ultimate goal is to test if best first search algorithms can improve the performance of the depth first search based QBF solvers. In other words, I will attempt to replace the depth first search procedure in the Backjumping or QCDCL algorithm by some best first search algorithms such as proof number search and see if this new combination can potentially boost the performance of the QBF solver.

4.2 Expected result

Since combining best first search with search based QBF solving algorithms is a novel idea, it is difficult to predict the exact outcome of the thesis project. In the ideal case, at the end of thesis C, I can present how to correctly combine best search search with some search based QBF solving algorithms and I can identify on some specific families of QBF benchmarks, combining best first search with search based QBF solving algorithms is beneficial.

4.3 Problem breakdown

There are two main tasks for the thesis project, one is to combine proof number search with backjumping algorithm while the other is to combine proof number search with QCDCL algorithm. For the backjumping and proof number search the theory and implementation difficulty is relatively low hence the target for this part of research is to implement a QBF solver based on proof number search and backjumping and compare its performance against the standard QDLL and backjumping solver and identify its strength and weakness. In comparison, for the QCDCL and proof number search combination, both the theory and implementation difficulty are high and whether it is feasible to implement QCDCL in the context of best first search is unknown. For this part of the research, our target is to investigate the algorithm combination from a theoretical level. I will start the implementation of QCDCL and proof number search only if the combination can be proved to be feasible from a theoretical level.

During thesis A, I combined DeepPNS and the Backjumping algorithm. In the next section, I will discuss how Backjumping can be implemented in the context of DeepPNS and some experimental results.

4.4 DeepPNS and Backjumping

If we implement Backjumping trick in the context of DeepPNS, the game state is the QBF formula f_μ, and for each node in the search tree we store the following information. The proof number (*node.pn*), the disproof number (*node.dn*), pointers to the two children (*node.left* and *node.right*), pointer to the parent (*node.parent*), the depth of the node (*node.depth*), the deep value of a node (*node.deep*), the branching variable corresponds to the node (*node.variable*), and the reason for satisfiability/unsatisfiability of the node (*node.reason*). Note that in my implementation, each proof number search tree node corresponds to a branching variable (i.e. the variable stored in the node is neither unit nor pure), for implementation convenience, the following invariant must be kept during the entire searching procedure.

Invariant 4.1. The reason for each undetermined node is NULL.

Invariant 4.2. For any none root node, if *current* is the left (resp. right) child of the parent node *parent*, assume f_μ is the formula corresponds to the parent node. The reason stored in the current node is the reason for $f_{\mu;parent.variable}$ (resp. $f_{\mu;\neg parent.variable}$) satisfiability/unsatisfiability.

The main procedure of DeepPNS and Backjumping algorithm is almost the same as the baseline PNS procedure. As we can see from Algorithm 9 the only modifications to the baseline PNS procedure is we replace the game state with the formula corresponds to each search tree node. And the truth value of the original formula f is given by $DeepPNSsolve(f)$.

Algorithm 9: DeepPNS based QBF solver

```
BOOLEAN DEEPPNSSOLVE($f_\phi$)
    $root.depth = 1$, $current := root :=$ initialize($f_\phi$)
    $iter := 0$
    while ($root$ is not solved and $iter <= maximum\_iteration$) do
        $iter := iter + 1$
        while ($next$ = MPN($current$, $f_\mu$) is not null) do
            $current := next$
        expand($current$, $f_\mu$)
        while ($current$ is not null) do
            backpropagate($current$)
            if ($current$ is root) then
                break
            if ($current.pn$ and $current.dn$ are not changed ) then
                break
            $current := backtrack(current, f_\mu)$
    if ($root$ is not solved) then
        return UNSOLVED
    if ($root$ is falsified) then
        return FALSE
    return TRUE
```

Similar to the proof number search algorithm, for the remaining of the section I will discuss how initialization, selection, expansion, backpropagation, and backtracking can be implemented in the context of DeepPNS and Backjumping.

4.4.1 Initialization with reason computation

As we can see from Algorithm 10, the logic for initialization is similar to the baseline PNS. One modification is we apply mobility initialization when the node is not a terminal node, and we also store the next branching variable in *node.variable*. Another modification to the procedure is when the formula is evaluated to be true or false, the reason for satisfiability or unsatisfiability is stored in *node.reason*. The way to calculate the initial reason is exactly the same as the QDLL case which is given by Theorem 3.1 and 3.2. Need to be aware that if the formula f_μ corresponds to the newly created node is true or false, the initial reason stored in the current node breaks invariant 4.2. Assume the current node is the left child of the parent, the reason calculated by *getReason*() is the reason for $f_{\mu;parent.variable;S[f_\mu;parent.variable]}$ satisfiability/unsatisfiability not for $f_{\mu;parent.variable}$ satisfiability/unsatifsiability. Here $S[f_{\mu;parent.variable}]$ is the forced assignment sequence of $f_{\mu;parent.variable}$ as defined in Definition 2.8. However, since except the creation of the root, the only place node initialization can be called is within the *expand*() method, we can just temporarily stored the reason in the newly created node, and let the *expand*() method to maintain invariant 4.2.

Algorithm 10: Initialize with reason

```
PNSNODE INITIALIZE($f$)
    $node.left := node.right := node.parent := NULL$
    $node.deep := \frac{1}{node.depth}$
    if $evaluate(f)$ is not Unknown then
        $node.variable := get\_variable(f)$
        if isexist(node) then
            $node.pn = 1$, $node.dn = 2$
        else
            $node.pn = 2$, $node.dn = 1$
        $node.reason :=$ NULL
    else
        if $evaluate(f)$ is True then
            $node.pn = 0$, $node.dn = \infty$
            $node.reason := getReason(f, \mu)$
        else
            $node.pn = \infty$, $node.dn = 0$
            $node.reason := getReason(f, \mu)$
    return node
```

Need to mention is the *evaluate*(f) evaluates whether f has been proved or disproved which is shown in Algorithm 11.

Algorithm 11: Evaluation

```
EVALUATE($f_\mu$)
    if $f_\mu$ has a contradictory clause then
        return False
    if all clauses in $f_\mu$ are satisfied then
        return True
    return Unknown
```

4.4.2 Selection with Backjumping

Similar to initialization, selection with backjumping is very similar to the selection procedure described in the baseline proof number search algorithm.

Algorithm 12: Find most proving node

```
PNSNode SELECTION($node$, $f_\mu$)
    if ($node$ $has$ $no$ $children$ ) then
        return $NULL$
    $v$ := node.variable
    $ret := \arg\min_{c \in node.child} dpn(c)$
    if ($ret$ $is$ $the$ $left$ $child$ ) then
        $f := f_{\mu;node.variable}$
    else
        $f := f_{\mu;\neg node.variable}$
    simplify($f$)
    return $ret$
```

As described in Algorithm 12, the selection procedure would return null if the current node does not have children (line 2, 3) and we know the current node is the most proving node. Otherwise, the most proving node is located in the subtree of a child with the minimum dpn value (line 5) [8]. If the most proving node is in the left subtree of the current node, we assign *node.variable* to the formula; otherwise, we assign $\neg$*node.variable* to the formula (line 6-9). In order to guarantee each node in the search tree is a branching node, after we assign a literal, we would simplified the newly updated formula by calling the simplify procedure described in Algorithm 1 and return child such that its subtree contains the most proving node (line 10,11).

4.4.3 Expansion with reason computation

The expansion procedure in DeepPNS and Backjumping is very different from the expansion introduced in the baseline PNS algorithm (Algorithm 7). One of the main difference between DeepPNS and QDLL is for DeepPNS the entire search tree is stored in memory. Hence, in order to save memory, we only store the branching nodes. In other words, for all nodes in the deep proof number search tree *node.variable* is a branching variable (i.e. neither unit nor pure). As mentioned in the previous subsection, invariant 4.2 might be broken after new node initialization. Hence, the expansion method also needs to update the reason stored in the children calculated by *getReason*() to maintain invariant 4.2. In order to achieve this, we define the method undo, which would compute the reason for $f_{\mu;parent.variable}$ based on the reason for $f_{\mu;parent.variable;S[f_\mu;parent.variable]}$ satisfiability/unsatisfiability.

Algorithm 13: Compute the reason for $f_{\mu;parent.variable}$

```
Reason undo($f_{\mu;parent.variable;S[f_\mu;parent.variable]}$, $current$)
    $reason := current.reason$
    $g := f_{\mu;parent.variable;S[f_\mu;parent.variable]}$
    while ($g_{\nu;l} \neq f_{\mu;parent.variable}$) do
        if ($reason \neq NULL$) then
            if ($l$ is unit in $g_\nu$) then
                if ($g_{\nu;l}$ is unsatisfied and $l \notin reason$) then
                    C := a clause in $g_\nu$ such that $l$ is a unit w.r.t. $C_{g_\nu}$
                    other := $\{l : \neg l \in C\} \cap \{l : l \in \nu \wedge isexist(l)\}$
                    reason := $reason \cup other \backslash \{l\}$
                else
                    reason := reason$\backslash\{l\}$
            if ($l$ is pure in $g_\nu$) then
                reason := reason$\backslash\{l\}$
        $current.reason := reason$
        $g := g_\nu$
    return $reason$
```

As we can see from Algorithm 13, we repeatedly unassign the last assigned literal from the current formula $f_{\nu;l}$, update the reason of the current node to the reason for f_ν based on the first 4 reason update rules listed in table 1 until the formula equals to $f_{\mu;parent.variable}$. Concretely, if the last assigned literal l is unit, we calculate the reason for f_ν from the reason for $f_{\nu;l}$ based on case 2-4 from Table 1 and when the last assigned literal l is pure, we calculate the reason for f_ν by dropping l from the reason for $f_{\nu;l}$.

The remaining of the expansion procedure would be exactly the same as the expansion procedure introduced in section 3.6.3 despite the game state is the QBF expression f_μ.

Algorithm 14: Expansion with backjumping

```
EXPAND($current$, $f_\mu$)
    $v := current.variable$
    $fl := f_{\mu;v}$
    simplify($fl$)
    $L := initialize(fl)$
    $L.reason := undo(fl, L)$
    $fr := f_{\mu;\neg v}$
    simplify($fr$)
    $R := initialize(fr)$
    $R.reason := undo(fr, R)$
    $L.parent := R.parent := current$
    $L.depth := R.depth := current.depth + 1$
```

4.4.4 Backpropagation with Backjumping

The backpropagation procedure in DeepPNS and Backjumping is slightly more complicated than the backpropagation procedure in standard proof number search. During backpropagation, we not only need to update the proof number, the disproof number, and the deep value but also need to compute the reason of the current node, if the reason of the current node can be determined. In order to save memory, if a node is solved, we can remove its left and right child (line 12, 13).

Algorithm 15: Backpropagation with Backjumping

```
1  BACKPROPAGATE($node$)
2      $v := node.variable$
3      $ret := \arg\min_{c \in node.child} dpn(c)$
4      $node.deep := ret.deep$
5      if $isexist(v)$ then
6          $node.pn := min(node.left.pn, node.right.pn)$
7          $node.dn := node.left.dn + node.right.dn$
8      else
9          $node.dn := min(node.left.dn, node.right.dn)$
10         $node.pn := node.left.pn + node.right.pn$
11     process_reason()
12     if ($node\ is\ solve$) then
13         $node.left := node.right := NULL$
```

As we can see from Algorithm 15, except *process_reason*(), the backpropagation procedure in DeepPNS and Backjumping is exactly the same as the backpropagation procedure described in section 3.6. We updated the proof and disproof number based on Algorithm 8, and we update the deep value of a node based on the description in section 3.6.6.

We will focus on how *process_reason*() works in backppropagation with backjumping. Despite the order of exploring the children can be different in DeepPNS, the way to compute the reason for a branching node is exactly the same as the rules in Table 1 in the QDLLBJ scenario. There are 7 cases to be considered for the backpropagation in the backjumping scenario.

• If both the left and right child are satisfied. This can only happen when *current.variable* is a universal branching variable. The reason of the current node can be computed the same as case 12 in Table 1.

• If both the left and right child are falsified. This can only happen when *current.variable* is an existential branching variable. The reason of the current node can be computed the same as case 8 in Table 1.

• Only the left (resp. right) child is satisfied, and *current.variable* is existential. This is equivalent to case 5 (resp. case 7) in Table 1 and the reason of the current node is the same as the reason stored in the left (resp. right) child.

• Only the left (resp. right) child is falsified, and *current.variable* is universal. This is equivalent to case 9 (resp. case 11) in Table 1 and the reason of the current node is the same as the reason stored in the left (resp. right) child.

• Only the left (resp. right) child is satisfied, and *current.variable* is universal, *current.variable* (resp. $\neg$*current.variable*) does not occur in the reason stored in the left (resp. right) child. Similar to case 10 in the QDLLBJ scenario, this is when the universal pruning occurs in DeepPNS. The reason of the current node is the same as the reason stored in the left (resp. right) child, and we directly set the current node's proof number to be 0 and disproof number to be ∞.

• Only the left (resp. right) child is falsified, and *current.variable* is existential, *current.variable* (resp. $\neg$*current.variable*) does not occur in the reason stored in the left (resp. right) child. Similar to case 6 in the QDLLBJ scenario, this is when the existential pruning occurs in DeepPNS. The reason of the current node is the same as the reason stored in the left (resp. right) child, and we directly set the current node's proof number to be ∞ and disproof number to be 0.

• If none of the above cases happen, the current node is undetermined and the reason can not be determined as well.

4.4.5 Backtracking with reason

Similar to baseline proof number search, during the backtracking procedure, we cannot simply let the current node point to its parent, because this would make our formula inconsistent. We must undo all the variable assignments happened at this node during the selection procedure. These variables include *current.variable* and all variables that are assigned during the simplification procedure (line 10 of Algorithm 12). Since we are applying Backjumping trick, we have to calculate the reason of the parent node based on the reason of the current node by calling undo. One thing to note is after we call backtrack, *node.reason* is no longer the reason for f_μ satisfiability/unsatisfiability. It is the reason for $f_{\mu;node.parent.variable}$ (resp. $f_{\mu;\neg node.parent.variable}$) if *node* is the left child (resp. right) of *node.parent*. The aim of this operation is to maintain invariant 4.2 and make the reason computation in the backpropagation procedure more convenient.

Algorithm 16: DeepPNS backtracking with reason

```
1 PNSNODE BACKTRACK($node$, $f_\mu$)
2     $node.reason := undo(f_\mu, node)$
3     return $node.parent$
```

4.5 Implementation details and Experimental results

4.5.1 Enable SAT solver

In section 3.1, we have discussed that in the original QDLL algorithm two additional cases are considered. One is when the partial assigned QBF f_μ contains no universal literals, the truth value of the QBF is simply $SAT(f_\mu)$, and the other is the trivial truth condition. Interestingly, in the original backjumping algorithm none of these cases were considered. In my current implementation of both the QDLLBJ solver and the DeepPNS solver, I ignore the trivial truth condition but when the partial assigned QBF f_μ contains no universal literals a SAT solver is called to solve f_μ and the reason for conflict (resp. solution) is all existential (resp. universal) literals in μ.

4.5.2 R value in DeepPNS and Backjumping

In section 3.6.6, we discussed that DeepPNS can control the seesaw effect of proof number search by introducing the parameter R. In general, the R value can mix proof number search and depth first search together. And in that section, we introduced that a good R value for the game Othello is 0.5 [8]. However, in the QBF world experiments shows that when R equals 0.5, the seesaw effect is still very serious. This can be seen from the following experiment. We run the DeepPNS and Backjumping solver with $R = 0.5$ (DeepPNSBJ) and QDLLBJ on the same set of benchmarks. We further restrict both solvers use DLCS as branching heuristic and enable the SAT solver when the partial-assigned formula contains no universal variables. Instead of comparing the run time, we use the number of node expansions as the performance metric and we set a limitation on the number of branching nodes as 4 million. The reason for setting the number of node expansions as the performance metric is because it directly measures the size of the searching space. Besides, under my current implementation, the formula data structure is not as efficient as the data structures used in state of art solvers, the run time is not the main concern at this stage.

If we compare DeepPNSBJ solver with $R = 0.5$ with the solver based on QDLL and backjumping, it can be seen from Table 3, in terms of the number of node expansions, DeepPNSBJ performs worse in almost all benchmarks. This does not mean that proof number search is completely unsuitable to solve QBF but because of the QBF game is very different from Othello. For most non-trivial benchmarks, a solved node cannot be reached without completely expanding the first quantifier block. And normally, the first quantifier block contains more than 30 quantifiers. Since the proof and disproof number cannot directly tell whether the formula is satisfiable or not, we need a sufficient number of solved nodes to proof or disproof the formula. Execute heuristic search at the very early stage of the searching process might not be suitable. For example, in the 3qbf-5cnf-50var-500cl.2 instance, if we use DeepPNS with $R = 0.5$, no terminal node was reached in the first 4000 iterations. Because proof and disproof number is irrelevant to the difficulty of QBF itself, without any solved nodes, DeepPNS has no benefits compared to depth first search. One thing to note is we cannot put the R value to be too small, because only with a R value of reasonable size, proof number search can alternate the search branch when the current search branch is too difficult to be proved or disproved.

I would provide a potential way of combining proof number search and depth first search so that the solver might take the benefit of both. Instead of using a uniform R during the entire searching process, we variate R between 0 and 0.5. For the current solver, I set let R to be 0 if the number of iterations divide by 5000 is even, and let R to be 0.5 otherwise. The intuition of this approach is to combine heuristic search and depth first search together. We start from depth first search, for every 5000 iterations if the current searching branch is too difficult for depth first search to solve, we use heuristic search to choose the searching branch for the next round of depth first. This dynamic R approach significantly improve the performance of the DeepPNSBJ solver.

In the following experiment, we compare the performance of the QDLLBJ solver and the DeepPNSBJ solver that alters the R parameter during the search. Let us denote the later solver by DeepPNSBJ-alt. Other restrictions are exactly the same as the preivous experiment.

Instance	**DeepPNS-BJ**	**QDLL-BJ**
chain12v.13	38307	8215
chain14v.15	185613	32795
chain18v.19	3270761	524323
TOILET6.1.iv.11	1896123	1441537
TOILET6.1.iv.12	264577	218581
TOILET10.1.iv.20	-	-
adder-2-sat	5895	3543
adder-2-unsat	39467	46335
k_d4_n-1	138585	582
k_lin_p-2	1872643	1188617
k_path_n-3	-	505581
k_t4p_n-1	-	1816773
k_lin_n-2	-	139775
impl20	553942	41
k_dum_p-3	2317805	200362
3qbf-5cnf-50var-500cl.2	-	-

Table 3: Number of node expansions for the QDLL-backjumping solver and DeepPNS-backjumping solver
- means the solver fails to solve the formula within 4 million node expansions

The previous result of DeepPNS with $R = 0.5$ shows that proof number search is worse than QDLL in almost all instances and none of the instances unsolvable by QDLLBJ can be solved by DeepPNS. However, with the help of the dynamic R trick it can be observed from Table 4 and Table 5 the performance of solver improved significantly. When the R value is alternating between 0 and 0.5, the proof number search can solve 23 instances that cannot be solved by the QDLL-BJ solver. However, it also fails to solve 5 instances that can be solved by QDLL-BJ solver.

This result indicates that although proof number and disproof number are irrelevant to the actual difficulty of the QBF expression, proof number search based algorithm can switch to a different branch of the search tree when the

Instance	**PNS-BJ-alt**	**QDLL-BJ**
chain12v.13	8239	8215
chain13v.14	16435	16409
chain14v.15	32823	32795
chain15v.16	65595	65565
chain16v.17	131135	131103
chain17v.18	262211	262177
chain18v.19	524359	524323
TOILET6.1.iv.11	1896123	1441537
TOILET6.1.iv.12	10073	218581
TOILET7.1.iv.14	3970381	3036638
TOILET10.1.iv.20	10373	-
TOILET16.1.iv.32	11271	-
adder-2-sat	4519	3543
adder-2-unsat	45923	46335
k_d4_n-1	809	582
k_d4_p-3	1482245	-
k_lin_p-2	3522933	1188617
k_path_n-2	28313	10927
k_path_n-3	-	505581
k_path_p-4	346099	151620
k_t4p_n-1	2446447	1816773
k_lin_n-2	466359	139775
impl20	81	41
k_dum_p-3	441255	200362
3qbf-5cnf-50var-500cl.2	29053	-

Table 4: Number of node expansions for the QDLL-backjumping solver and DeepPNS-backjumping solver with dynamic R
- means the solver fails to solve the formula within 4 million node expansions

Total	**Either**	**Both**	**DeepPNSBJ-alt**	**QDLL-BJ**
95	51	28	47	32

Table 5: Number of solved encoded gttt4x4 instances by dfs and PNS

current search branch is too difficult to be proved. Conversely, since for most non-trivial QBF instances a solved node cannot be reached by doing very little variable assignment, the seesaw effect might be severed because of the generation of too many undetermined nodes during expansion. Therefore, combine pure depth first search in the searching process can be beneficial. Overall, the experimental results shows that heuristic search has the potential of improving the performance of the search based QBF solvers that are based on pure depth first search at least on some family of benchmarks such as the generalized tic-tac-toe (gttt4x4).

5 Future Study

5.1 Further investigate the seesaw effect

Currently, the R value for the DeepPNS solver follows the same pattern for all instances. However, this might not be the best way for the alteration of R. For example, on some family of instances such as BLOCK, because of the huge outermost quantifier block, PNSBJ-alt can still suffer from the problem of seesaw effect by creating no solved nodes during the heuristic search process. For this family of instances, between the 5001 and 10000 iterations, the disproof number would increase by 10000. According to the of way we initialize proof number and disproof number in Algorithm 10, the only way to make this happen is between these 5000 iterations, no solved node was expanded. Since different family of instances corresponds to different real world problems, it might be beneficial to find a suitable way to alter this R value for different family of instances.

5.2 Combine QCDCL and DeepPNS

During thesis A, I have implemented a solver that combines backjumping and deep proof number search. From the experimental results, it shows that deep proof number search can potentially improve the performance of depth first search based QBF solver. However, backjumping is not used by any search based state of art QBF solver as discussed in section 3.3. Since DeepPNS combine with backjumping outperforms QDLL and backjumping on many benchmarks, it would be interesting to see how DeepPNS works with QCDCL.

Unfortunately, the implementation of combining proof number search and QCDCL is extremely technical. Not only because this algorithm involves adding new clauses and terms into the original formula and it will breaks the invariant of my current formula data structure, but also because with the addition of new clauses and terms, it can be viewed as the rule of the original and-or strategy game is changed. The proof number search algorithm need to be modified to deal with this change of game rule during the search as well.

5.3 Minimal reason computation with SAT solver enabled

In the thesis report, I have mentioned a SAT solver can be called when no universal quantifier exists in f_μ and how to compute reason for this case was not discussed in the original paper [4]. The way I deal with this case was discussed in section 4.5.1. However, this also means that no pruning might happen along the current searching path. In the BLOCK group for example, enable SAT solver this way significantly weakened the performance of the backjumping solver.

Instead of solving 1 to 4 instances depending on different branching heuristic, no instance in the BLOCK family can be solved if the SAT solver is enabled. Therefore, it is essential to develop a way to compute a more accurate set of reason for the case the SAT solver is enabled. If this can be achieved, the backjumping based solver's performance can be improved.

5.4 Plan for thesis B and thesis C

Out of the three mentioned tasks, combining QCDCL and DeepPNS is definitely the most difficult because whether QCDCL can be applied in the context of proof number search is still unknown at this stage. Thus, this task will be marked with the highest priority for thesis B while the remaining 2 tasks will be done concurrently. Tasks for thesis C really depends on how the combination of QCDCL and DeepPNS goes. If a relatively simple way to combine QCDCL and proof number search can be found in thesis B, during thesis C, I would implement the new combination. Otherwise, I would investigate how to combine proof number search with other QBF solving techniques from a theoretical level. A tentative schedule for thesis B and thesis C is as follows.

Period	**Week**	**Task**
B	1 - 2	reading on QCDCL
	3 - 5	implement the data structure part of QCDCL investigate reason computation with SAT enabled
	6 - 9	investigate seesaw effect in QBF world investigate how to combine QCDCL and PNS
	10	prepare for thesis B presentation
	11	thesis B presentation
C	1 - 6	QCDCL and PNS implementation or investigate other QBF solving techniques
	7 - 8	prepare for thesis C presentation
	9 - 11	thesis C report

Table 6: Schedule for thesis B and C

We can see from this table, in thesis B the focus would be on improving the current DeepPNS and Backjumping solver and investigate QCDCL and PNS combination from a theoretical level. In thesis C, depending on the feasibility of combining QCDCL and PNS I would either implement the QCDCL and PNS solver or investigate some other popular QBF solving techniques such as counterexample guided abstraction refinement (CEGAR) [9] in the context of proof number search.

6 Appendix

6.1 QDLLBJ algorithm pseudocode

Algorithm 17: QDLLBJ algorithm with Backjumping

```
Pair<Boolean, Reason> QDLLBJ($f$, $\mu$)
    if ($f_\mu$ has a contradictory clause) then
        return <False, getReason($f$, $\mu$)>
    if (all clauses in $f_\mu$ are satisfied) then
        return <True, getReason($f$, $\mu$)>
    if ($l$ is unit in $f_\mu$) then
        ret, reason := QDLLBJ($f$, $\mu; l$)
        if ret is False and $l \notin$ reason then
            C := a clause in $f$ such that $l$ is a unit with respect to $C_{f_\mu}$
            other := $\{l : \neg l \in C\} \cap \{l : l \in \mu \wedge isexist(l)\}$
            reason := $reason \cup other \setminus \{l \cup \neg l\}$
        reason := $reason \setminus \{l\}$
        return <ret, reason>
    if ($l$ is pure in $f_\mu$) then
        ret, reason := QDLLBJ($f$, $\mu; l$)
        reason := reason$\setminus\{l\}$
        return <ret, reason>
    $l$ = get_literal($f$, $\mu$)
    if $l$ is existential in $f$ then
        L, reasonL := QDLLBJ($f$, $\mu; l$)
        if (L is True or $l$ is not in reasonL) then
            return <L, reasonL>
        R, reasonR := QDLLBJ($f$, $\mu; \neg l$)
        if (R is True) then
            return <R, reasonR>
        reasonR := (reasonL $\cup$ reasonR) $\setminus\{l \cup \neg l\}$
        return <R, reasonR>
    L, reasonL := QDLLBJ($f$, $\mu; l$)
    if (L is False or $l$ is not in reasonL) then
        return <L, reasonL>
    R, reasonR := QDLLBJ($f$, $\mu; \neg l$)
    if (R is False) then
        return <R, reasonR>
    reasonR := (reasonL $\cup$ reasonR) $\setminus\{l \cup \neg l\}$
    return <R, reasonR>
```

6.2 Original QDLL algorithm

Algorithm 18: Original QDLL algorithm

```
Boolean QDLL($f$, $\mu$)
    if $f_\mu$ has a contradictory clause then
        return False
    if all clauses in $f_\mu$ are satisfied then
        return True
    if $f_\mu$ has no universal variables then
        return SAT($f_\mu$)
    G := the formula obtained by removing all universal literals from $f_\mu$
    if SAT(G) then
        return True
    if $l$ is unit in $f_\mu$ then
        return QDLL($f$, $\mu; l$)
    if $l$ is pure in $f_\mu$ then
        return QDLL($f$, $\mu; l$)
    $l$ = get_literal($f$, $\mu$)
    if $l$ is existential in $f$ then
        return QDLL($f$, $\mu; l$) or QDLL($f$, $\mu; \neg l$)
    return QDLL($f$, $\mu; l$) and QDLL($f$, $\mu; \neg l$)
```

References

[1] Armin Biere. Resolve and expand. In *International conference on theory and applications of satisfiability testing*, pages 59–70. Springer, 2004.

[2] Michael Buro and H Kleine Büning. *Report on a SAT competition.* Fachbereich Math.-Informatik, Univ. Gesamthochschule, 1992.

[3] Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An algorithm to evaluate quantified boolean formulae. *AAAI/IAAI*, 98:262–267, 1998.

[4] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Backjumping for quantified boolean logic satisfiability. *Artificial Intelligence*, 145(1-2):99–120, 2003.

[5] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Backjumping for quantified boolean logic satisfiability. 2003b.

[6] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Clause/term resolution and learning in the evaluation of quantified boolean formulas. *Journal of Artificial Intelligence Research*, 26:371–416, 2006.

[7] Enrico Giunchiglia, Massimo Narizzano, Armando Tacchella, et al. Learning for quantified boolean logic satisfiability. In *AAAI/IAAI*, pages 649–654, 2002.

[8] Taichi Ishitobi, Aske Plaat, Hiroyuki Iida, and Jaap van den Herik. Reducing the seesaw effect with deep proof-number search. In *Advances in Computer Games*, pages 185–197. Springer, 2015.

[9] Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. Solving qbf with counterexample guided refinement. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 114–128. Springer, 2012.

[10] Akihiro Kishimoto, Mark HM Winands, Martin Müller, and Jahn-Takeshi Saito. Game-tree search using proof numbers: The first twenty years. *Icga Journal*, 35(3):131–156, 2012.

[11] William Klieber, Samir Sapra, Sicun Gao, and Edmund Clarke. A non-prenex, non-clausal qbf solver with game-state learning. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 128–142. Springer, 2010.

[12] Daniel Le Berre, Massimo Narizzano, Laurent Simon, and Armando Tacchella. The second qbf solvers comparative evaluation. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 376–392. Springer, 2004.

[13] Reinhold Letz. Lemma and model caching in decision procedures for quantified boolean formulas. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 160–175. Springer, 2002.

[14] Florian Lonsing and Uwe Egly. Depqbf 6.0: A search-based qbf solver beyond traditional qcdcl. In *International Conference on Automated Deduction*, pages 371–384. Springer, 2017.

[15] Joao Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Portuguese Conference on Artificial Intelligence*, pages 62–74. Springer, 1999.

[16] Valentin Mayer-Eichberger and Abdallah Saffidine. Positional games and qbf: The corrective encoding. *arXiv preprint arXiv:2005.05098*, 2020.

[17] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

[18] Markus N Rabe and Leander Tentrup. Caqe: a certifying qbf solver. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pages 136–143. IEEE, 2015.

[19] Masahiro Seo, Hiroyuki Iida, and Jos WHM Uiterwijk. The pn2̆217-search algorithm: Application to tsume-shogi. *Artificial Intelligence*, 129(1-2):253–277, 2001.

[20] Lintao Zhang and Sharad Malik. Conflict driven learning in a quantified boolean satisfiability solver. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 442–449, 2002.