University of New South Wales

Solving QBF with proof number search

Thesis C report

Author
Yifan He

Supervisor
Dr. Abdallah Saffidine

April 26, 2022

Abstract

Quantified Boolean Formula or QBF was 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 on a theoretical level, and depth-first search is rarely used in two-player strategy game solving. For this thesis project, we 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 report, I will present how some well-known QBF algorithms such as backjumping and QCDCL can be implemented in the context of proof number search. The experiment result shows that in our current version of the solver, replacing depth-first search with proof number search is not beneficial on most benchmarks. However, proof number search based QBF solvers can solve a few instances that depth-first search is unable to solve, which implies that this new algorithm combination might be worthy to be studied further.

Contents

1 Introduction

Quantified Boolean Formulas or QBF were proposed in the 1970s as a natural generalization of the propositional logic. Quantified Boolean logic is a more generalized framework compared to propositional logic, however, the performance of QBF solvers lags 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 [12]. But most of the research was related to "game-playing" not "game-solving". QBF can be interpreted as an and-or strategy game in 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 two-player strategy games and QBF in the theoretical level. Based on this interpretation, the QDLL-based QBF solving procedure can be considered as a depth-first search approach to solve the and-or strategy game. However, depth-first search or iterative deepening search are rarely used in strategy game solvers. The reason is that both algorithms assume a solution to the game can be found at a relatively low searching depth. Hence, for strategy games that only exists a very deep solution, depth-first search is not ideal. This is mainly because depth-first search 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 [26]. 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 the searching depth exceeds 100. This means that QBF is also a game with a small branching factor but the solution only exists at a very deep level. Hence, although most of the search-based QBF solving procedure applies a depth-first search procedure, depth-first search might not be ideal for QBF.

This thesis project focused on a heuristic search algorithm called proof number search that has achieved massive success in 2-player strategy games. We would investigate 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, the target of this thesis is to investigate how a heuristic search algorithm that is frequently used in strategy games can be combined with some QBF solving algorithms such as backjumping [5] and QCDCL [6, 14, 29], and if these new combinations can potentially improve the performance of depth-first search based QBF solver. Although combining proof number search with backjumping is relatively straightforward, as we can see in later sections of the report that there are significant obstacles when QCDCL is implemented in the context of proof number search. One piece of work that is highly related to this thesis was the MCTS

based CDCL SAT solver [25]. How the techniques proposed in that paper can be generalized to the context of QBF is one of the most important aspect of this thesis.

This thesis report is structured as follows. We would firstly review QBF solving techniques and the proof number search algorithm. At the same time, we define the notations that will be used for the rest of the report. In section 3, we will describe how to design a proof number search based QBF solver that combines popular reasoning techniques such as backjumping and QCDCL. Implementation and experimental results of these new algorithm combinations can be found in section 4. Related work is going be discussed in section 5. In section 6, we would list some valuable future work.

2 Preliminaries

2.1 Quantified Boolean Logic

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 cube (**also known as term**) T is a conjunction of literals. A propositional formula is in conjunctive normal form if it is a conjunction of clauses, a propositional formula is in disjunctive normal form if it is a disjunction of cubes. 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 consisting of blocks of existential ($\exists$) or universal ($\forall$) quantifiers followed by a propositional formula [21].

$$Q_1 X_1 Q_2 X_2 ... Q_n X_n \Phi \tag{1}$$

(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 (a.k.a. outermost) quantifier block. If a QBF is not in prenex form, we can transform Φ to conjunctive normal form in linear time [6]. For the rest of the report, we assume that we only deal with QBF that is **initially** in closed and 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 $level(v) = n - i + 1$. 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 true or false. This leads us to the following recursive definition of QBF semantic.

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

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, if and only if 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 updating f to f_l is called **unit propagation**.

Definition 2.6 (Pure literal)**.** A literal l is called *pure* in a QBF f, if and only if 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 updating f to f_l is called **pure literal elimination**.

Need to be aware that, almost all search-based QBF solvers have implemented the unit propagation simplification procedure, but not all solvers have included pure literal elimination [29]. Unlike the SAT world where the importance of pure literal is debatable, it is widely believed that for QBF solving, pure literal elimination is beneficial [16].

Definition 2.7 (Minimal Form)**.** A clause in a QBF with Φ in conjunctive normal form is in **minimal form** if and only if the literal with minimum quantifier level is existential. A cube in a QBF with Φ in disjunctive normal form

is in **minimal form** if and only if the literal with minimum quantifier level is universal.

Definition 2.8 (Universal Reduction [6])**.** The process of repeatedly removing literals with minimum quantifier level from a clause in a QBF in conjunctive normal form until the clause is in minimal form is called the **Universal Reduction** of the clause, we denote the reduced clause as $min(C)$. The universal reduction procedure is sound.

Definition 2.9 (Existential Reduction [6])**.** The process of repeatedly removing literals with minimum quantifier level from a cube in a QBF in disjunctive normal form until the cube is in minimal form is called the **Existential Reduction** of the cube, we denote the reduced cube as $min(T)$. The existential reduction procedure is sound.

With the soundness of universal and existential reduction, we assume we only deal with QBF in minimal form.

Definition 2.10 (Model)**.** A cube T is called a **model** of a QBF if the literals in T is a solution to the propositional formula Φ. If T is a model of a QBF, $min(T)$ is also a model.

Definition 2.11 (Q-resolution)**.** Suppose C_1 and C_2 are two clauses in minimal form, and there exists exactly one literal l such that l is in C_1 and $\neg l$ is in C_2, then the Q-resolvent of C_1 and C_2 is

$$\frac{C_1 \qquad\qquad C_2}{min(C_1 \cup C_2 \setminus \{l, \neg l\})}$$

The procedure of resolving C_1 and C_2 to their Q-resolvent is called **clause resolution**. Suppose T_1 and T_2 are two cubes in minimal form, and there exists exactly one literal l such that l is in T_1 and $\neg l$ is in T_2, then the Q-resolvent of T_1 and T_2 is

$$\frac{T_1 \qquad\qquad T_2}{min(T_1 \cup T_2 \setminus \{l, \neg l\})}$$

The procedure of resolving T_1 and T_2 to their Q-resolvent is called **term resolution**. The collection of clause resolution and term resolution is called **Q-resolution**.

Note that Q-resolution can be interpreted as the combination of universal (resp. existential) reduction and the resolution procedure in propositional logic. Similar to resolution in propositional logic, Q-resolution is a sound and complete procedure of solving QBF in conjunctive (resp. disjunctive) normal form. In

other words, if we repeatedly perform Q-resolution and add the resolvent to the formula conjunctively (resp. disjunctively), the original QBF is true (resp. false) if and only if the empty clause (resp. term) is not derivable.

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 2.12 (Assignment [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 2.13 (Forced Assignment Sequence)**.** Suppose f is a QBF expression in conjunctive normal form 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 until the simplified expression contains either an empty clause or contains neither unit nor pure literals.

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

Algorithm 1: Simplification Procedure

```
   Input  : Any QBF formula f in conjunctive normal form
   Output: The forced assignment sequence of f, and f is simplified
 1 Function 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             continue
13         if Pure_literal_enabled and l is pure in g then
14             append l to the end of S
15             g := g_l
16     f := g
17     return S
```

There are two remarks for the above pseudocode. Firstly, there is a Boolean flag called Pure_literal_enabled. Pure literal elimination is going to be executed

only if this flag is set to be true. For the rest of the report, we will explicitly state the status of this flag when required. Secondly, we should 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 critical in algorithms that will be introduced in later sections. To avoid confusion, we would use $\mu;\nu$ to represent the "concatenation" of two sequences μ and ν, and $set(\mu)$ to represent the a **set** that contains the all elements that appears in the sequence μ. For example, $[1,2,3];[4,5,6] = [1,2,3,4,5,6]$ and $set([1,3,1]) = \{1,3\}$. We further define the operator $\in$ between an element l and a sequence μ: $l \in \mu$ if and only if $l \in set(\mu)$.

2.2 Search-based QBF solving techniques

There are two main types of QBF solvers, expansion-based solver, and search-based solver. These two types of solvers are incomparable because they use different proof systems and both types of solvers are specialized on some families of benchmarks. Successful expansion-based QBF solvers include QUANTOR [2], RAReQS [10], and caqe [24]. Successful search-based QBF solvers include QUBE [5, 6], Quaffle [29], semprop [14], GhostQ [13], and DepQBF [17]. In this section, we will review some of the cored search-based QBF solving techniques.

2.2.1 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. 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

```
   Input  : QBF formula f and the empty partial assignment
   Output: The truth value of the QBF expression f
1  Function QDLL(f, μ)
2      if f_μ has a contradictory clause then
3          return False
4      if all clauses in f_μ are satisfied then
5          return True
6      if l is unit in f_μ then
7          return QDLL(f, μ; l)
8      if Pure_literal_enabled and l is pure in f_μ then
9          return QDLL(f, μ; l)
10     l = get_variable(f, μ)
11     if l is existential in f then
12         return QDLL(f, μ; l) or QDLL(f, μ; ¬l)
13     return QDLL(f, μ; l) and QDLL(f, μ; ¬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 variable l in the outermost quantified block in f_μ. Details of how *get_variable*() works is normally related to branching heuristics [20] which is not the focus of this thesis. 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. Note that in the default setting, the *Pure_literal_enabled* flag is set to true.

Although Algorithm 2 is referred as the QDLL procedure in later papers such as [5, 14, 6], in the original 1998 paper [3], two additional special cases were considered (line 6-10 in algorithm 15). 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 B for details).

To avoid ambiguity, we refer Algorithm 2 as the QDLL procedure in later sections.

2.2.2 Backjumping

It is widely believed that QDLL alone is incompetitive. One way to optimize the baseline QDLL algorithm is the Backjumping algorithm or dependency back-

tracking. This algorithm 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 [5]. The algorithm attempts to associate each false search-state with a μ-contradicted clause, and each true state with a μ-truth cube. The QBF expression is false (resp. true) if and only if the empty clause (resp. cube) can be derived.

Definition 2.14 (μ-contradicted clause [6])**.** A clause C is said to be μ-contradicted if and only if each literal $l \in C$, we have $l \notin \mu$, and for each existential literal $l \in C$, we have $\neg l \in \mu$.

Definition 2.15 (μ-satisfied cube [6])**.** A cube T is said to be μ-satisfied if and only if each literal $l \in T$, we have $\neg l \notin \mu$, and for each universal literal $l \in T$, we have $l \in \mu$.

Definition 2.16 (μ-false clause)**.** A clause C is said to be μ-false if and only if each literal $l \in C$, we have $\neg l \in \mu$.

Definition 2.17 (μ-truth cube)**.** A cube T is said to be μ-satisfied if and only if each literal $l \in T$, we have $l \in \mu$.

There are two remarks about these definitions. Firstly, in the context of backjumping we associate each false (resp. true) state with a μ-contradicted clause (μ-truth cube), not all μ-contradicted clauses (resp. μ-truth cube) are valid. These clauses must be either a clause (resp. a model) of the original formula or clauses (resp. cubes) that are derivable by Q-resolutions of the clauses (resp. models) of original formula. Secondly, in the context of SAT, we are able to associate each false state with a μ-false clause instead of μ-contradicted. However, because of the existence of existential unit literals not in the outermost quantifier block in QBF, we can only ensure it is possible to associate each false state with a μ-contradicted clause [6]. Note that this is also the reason why in backjumping algorithm, we can associate each true state with a μ-truth cube. Because in the context of backjumping, the concept of universal unit propagation does not exist.

In this thesis we also refer the μ-contradicted clause associated with a state as the **reason for unsatisfiability** of the state, and the μ-satisfied cube associated with a state as the **reason for satisfiability** of that state. Be aware that our definition of reason for unsatisfiability (resp. satisfiability) is different from its original definition in [5]. In that paper, the reason for unsatisfiability refers to a set of existential literals such that its negation appears in the μ-contradicted clause, while the reason for satisfiability refers to the set of universal literals in the μ-satisfied cube. However, to avoid confusion, in our thesis we would unify the definition of reason and μ-contradicted clauses (resp. μ-satisfied cube) . Note that such unification is valid because under both definitions, the depth-first search tree generated by the backjumping algorithm is going to be exactly the same as long as the reason calculation rule is modified accordingly [6].

The benefit of defining the reason is when the branching literal l (resp. $\neg l$) does not appear the reason for $f_{\mu;l}$ satisfiability (resp. unsatisfiability), we can

skip the exploration of $\neg l$ and prune the searching space. We would review one possible way of computing the reason at terminal state and propagating the reason during backtracking.

We call a searching state terminal if the formula f in conjunctive normal form 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 2.1. *Suppose the QBF formula f_μ contains a contradictory clause C, then C is a reason for f_μ unsatisfiability.*

Proof. This theorem trivially holds by the definition of reason for unsatisfiability (μ-contradicted clause). □

Theorem 2.2 (Model Generation [5])**.** *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 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. Let μ' be the resulting partial assignment, and T be the cube that contains all literals in μ'. Then, T is the reason for f_μ satisfiability.*

Proof. This theorem trivially holds by the definition of reason for satisfiability (μ-truth cube). □

Note that, because of the soundness of existential reduction and universal reduction, the minimal form of a reason for unsatisfiability (resp. satisfiability) is also a reason for unsatisfiability (resp. satisfiability). Based on the above 2 theorems, we can write the pseudocode for *getReason* as follows.

Algorithm 3: get the reason for f_μ

```
   Input  : The QBF expression f and a partial assignment μ
   Output: The reason for satisfiability or unsatisfiability of f_μ
 1 Function getReason(f, μ)
 2     if (f_μ has a contradictory clause C) then
 3         return min(C)
 4     if (all clauses in f_μ are satisfied) then
 5         reason := μ
 6         while reason is not empty do
 7             if There exists l ∈ reason and l is universal such that all
                clauses in the formula after removing l from f_μ are still
                satisfied then
 8                 reason := reason\{l}
 9                 μ := μ\{l}
10             else
11                 break
12         return min(reason)
13     return NULL
```

Although it was not explicitly mentioned in the original paper [5], it is extremely important to note that in Theorem 2.2 and Algorithm 3, when we compute the reason for satisfiability, the order of removing universal literals is not arbitrary, if we enabled pure literal elimination. 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.

Once the reason at the terminal state is calculated, we need to propagate the reason during backtracking. Based on the definition of reason, the reason for $f_{\mu;l}$ is often not equivalent to the reason for f_μ. Thus it is necessary to come up with an efficient method to maintain a valid set of reasons during backtracking. The most popular way of computing the reason during backtracking is summarized in Table 1. In this table, we represent the reason for $f_{\mu;l}$ as R and the reason for $f_{\mu;\neg l}$ as R' (R and R' can be either a clause or a cube). In this table, case

#	**Property of l**	**Truth of $f_{\mu;l}$**	**Truth of $f_{\mu;\neg l}$**	**Reason for f_μ**
1	Pure	SAT/UNSAT	-	R
2	Unit	SAT	-	R
3	Unit $\neg l \notin R$	UNSAT	-	R
4	Unit $\neg l \in R$	UNSAT	UNSAT	$resolve(f_\mu, R, R', l)$
5	$\exists$ Br.	SAT	-	R
6	$\exists$ Br. $\neg l \notin R$	UNSAT	-	R
7	$\exists$ Br. $\neg l \in R$	UNSAT	SAT	R'
8	$\exists$ Br. $\neg l \in R$ and $l \notin R'$	UNSAT	UNSAT	R'
9	$\exists$ Br. $\neg l \in R$ and $l \in R'$	UNSAT	UNSAT	$resolve(f_\mu, R, R', l)$
10	$\forall$ Br.	UNSAT	-	R
11	$\forall$ Br. $l \notin R$	SAT	-	R
12	$\forall$ Br. $l \in R$	SAT	UNSAT	R'
13	$\forall$ Br. $l \in R$ and $\neg l \notin R'$	SAT	SAT	R'
14	$\forall$ Br. $l \in R$ and $\neg l \in R'$	SAT	SAT	$resolve(f_\mu, R, R', l)$

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

6 and 11 corresponds to the case in which the right branch of the search tree would be explored the baseline QDLL procedure but skipped in the backjumping algorithm. If we disable the pure literal elimination rule, case 1 can never be triggered. There are many ways of implementing $resolve(f_\mu, R, R', l)$ [6, 14, 29, 18] but one must be aware that under all settings $resolve(f_\mu, R, R', l)$ is not just a simple Q-resolution between R and R' (an explanation to this can be found in [6, 14, 28, 29]). In this report, we refer $resolve(f_\mu, R, R', l)$ as the $Rec - Resolve$ procedure described in Figure 9 of [6], and we would put the

pseudocode in Appendix C.

There are two main advantages of the backjumping algorithm. Firstly, the implementation difficulty of backjumping is relatively low and the computation overhead for the reason calculation is usually very little with the optimized reason computation method proposed in [5]. Secondly, case 6 and 11 in Table 1 correspond to cases that backjumping can skip exploring $f_{\mu;\neg l}$ while all the other cases backjumping explores exactly the same searching space as baseline QDLL. Thus, in terms of the size of the searching space, backjumping is guaranteed to perform not worse than baseline QDLL.

2.2.3 QCDCL

Although Backjumping algorithm 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 as the main solving technique in state of art search-based QBF solvers such as DepQBF. The main shortcoming of the 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 algorithm is limited. One of the simplest formula that cannot be solved by backjumping 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 backjumping algorithm degenerates to the baseline QDLL procedure and for this specific instance, no implementation of the baseline QDLL procedure can solve the formula within a day [14]. However, any QCDCL solver can solve the instance within several seconds.

The QCDCL algorithm is a sound and complete QBF solving algorithm that was proposed by Zhang [29], Giunchiglia [6], and Letz [14] in the early 2000s. This algorithm is a modification of the conflict-driven clause learning algorithm (CDCL) in SAT. The algorithm is very similar to backjumping, despite that it adds clauses and cubes to the formula during the search procedure. The main idea of the algorithm is to learn additional clauses (resp. cubes) caused by conflicts (resp. solutions) that can be added to the original formula conjunctively (resp. disjunctively). Concretely, the QBF f is extended to

$$Q_1X_1Q_2X_2...Q_nX_n[\Phi, \theta, \psi] \tag{2}$$

. Here, Φ represents the clauses of the original formula, θ represents the clauses learned because of **clause resolution** of clauses in Φ and θ, while ψ represents the cubes learned because of model generation of Φ and **term resolution** of the cubes in ψ. Note that initially both θ and ψ are empty, and the original QBF is always logically equivalent to

$$Q_1X_1Q_2X_2...Q_nX_n(\Phi \ \wedge \ \theta) \tag{3}$$

, and

$$Q_1X_1Q_2X_2...Q_nX_n(\Phi \ \vee \ \psi) \tag{4}$$

[6] Furthermore, we have to modify the definition of terminal state, unit literal, and pure literal to the following.

Definition 2.18 (Terminal state in QCDCL)**.** Let μ be a partial assignment, $[\Phi, \theta, \psi]_\mu$ is said to be in a terminal false state if and only if $(\Phi \wedge \theta)_\mu$ contains an empty clause. Similarly, $[\Phi, \theta, \psi]_\mu$ is said to be in a terminal true state if and only if Φ_μ has all clauses satisfied or ψ has a satisfied cube.

Definition 2.19 (Unit literal in QCDCL [6])**.** An existential literal l is called *unit* in a QBF f during the QCDCL procedure, if and only if there exists a clause $C_f \in \Phi$ or $C_f \in \theta$ such that l is the only existential variable in C_f and all other variables has a quantifier level greater than $|l|$. Similarly, a universal literal l is called *unit* in a QBF f during the QCDCL procedure, if and only if there exists a cube $T_f \in \psi$ such that $\neg l$ is the only universal variable in C_f and all other variables have a quantifier level greater than $|l|$.

Definition 2.20 (Pure literal in QCDCL [6])**.** A literal l is called *pure* in a QBF f during the QCDCL procedure, if and only if l is existential and $\neg l$ does not occur in any clauses in $\Phi \cup \theta$ or l is universal and l does not occur in any clauses or cubes in $\Phi \cup \psi$.

One motivation of the QCDCL algorithm is, with the additional clauses and cubes learned during backtracking, more unit propagation might be triggered during the QDLL solving procedure, and hence the search space can be pruned. QCDCL algorithm might be interpreted as a *modified* backjumping algorithm. Same as backjumping, the QCDCL algorithm is going to return true (resp. false) if and only if the empty cube (resp. clause) is derivable. The algorithm attempts to associate each false state with a μ-contradicted clause, and each true state with a μ-satisfied cube (note that because of the existence of universal unit propagation, we are no longer able to associate each true state with a μ-truth cube). The major difference between backjumping and QCDCL is, in QCDCL, we add the reason of satisfiability and unsatisfiability computed by the backjumping algorithm to θ and ψ during the middle of the search instead of discarding it. 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 the search on different parts of the search tree. Unfortunately, there are two main shortcomings of the QCDCL algorithm. One is that it does not guarantee to outperform baseline QDLL in terms of the size of the searching space. For example, when new clauses and cubes are added to the formula, the number of pure literal eliminations can be reduced [16]. Another shortcoming of the QCDCL algorithm is the size of the formula might be exponential if clauses and terms are added to the formula without any consideration. The very popular 1-UIP learning schema is always used to tackle this issue [6]. 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.

2.3 Proof number search

Both backjumping and QCDCL are improvements to the baseline depth-first search QDLL procedure from the reasoning perspective. However, it is possible that partial-assigned QBF formulas on one side of the searching space is significantly harder to be solved than partial-assigned QBF formulas on a different part of the searching space. For depth-first search procedure, once it reaches the wrong side of the searching space, unless we restart the search, the search is trapped. Heuristic search algorithms, however, are less likely to be trapped on one side of the searching space. One of the main goals of the thesis is to figure out if it is beneficial to replace the depth-first search procedure in backjumping and QCDCL with heuristic search. The heuristic search algorithm that we are particularly interested in for this thesis is called proof number search.

The 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 prioritizes the search in the direction of the most proving node [12]. In recent years variations of this algorithm such as DeepPN, PN2, PN*, and df-pn have achieved massive success in many two-player and-or strategy games such as Hex, Othello, tsume-shogi, and connect6 [8, 26]. Since proof number search is a best-first search algorithm, the entire search tree is stored in memory. In the baseline setting, 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 estimates the minimum number of leaves in the subtree that must be proved such that the entire subtree can be proved while the disproof number estimates the minimum number of leaves in the subtree that must be disproved such that the entire subtree can be disproved. 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 thesis, we are only interested in the proof number search algorithm with a branching factor of 2. In proof number search, a node can have 3 states, satisfied (proved), falsified (disproved), and undetermined, which is described by the following definition.

Definition 2.21. 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**.

The main procedure of PNS algorithm can be described as follows. It can be seen from Algorithm 4 that, before the start of the search, a root node is created based on the information of the initial game state. After that, we repeatedly execute the following procedure until the number of iterations exceeds the limit or the root has been proved or disproved.

- Increment the number of iterations by 1.

- Select the most proving node based on information of the proof and disproof number of the 2 children.

- Expand the most proving node, and initialize the 2 newly created nodes. During initialization, if the node is undetermined, the proof and disproof number can be determined by heuristics. In the default setting, for an undetermined node, we initialize both the proof and disproof number to be 1.

- Backtrack from the current node all the way back to the root. During backtracking, we back-propagate the information of the two children of *current* to the current node.

Algorithm 4: PNS Algorithm

```
   Input  : The initial game state
   Output: If the player plays first can win
 1 Function PNS(game_state)
 2     root.depth = 1, current := root := initialize(game_state)
 3     iter := 0
 4     while (root is not solved and iter <= maximum_iteration) do
 5         iter := iter + 1
 6         while (next = selection(current, game_state) is not null )
            do
 7             current := next
 8         expand(current, game_state)
 9         while (current is not null ) do
10             backpropagate(current)
11             if (current is root) then
12                 break
13             current := backtrack(current, game_state)
14     if (root is not solved ) then
15         return UNSOLVED
16     if (root is falsified ) then
17         return FALSE
18     return TRUE
```

In this algorithm, there are 5 important procedures which are initialization, selection, expansion, backpropagation, and backtracking. Details of how these procedures work depend on use cases, while their behavior in the most standard setting can be found in [12]. We will provide a detailed discussion on how these 5 procedures work in the context of QBF in later sections.

One shortcoming of the standard proof number search algorithm is the seesaw effect. It is the effect that the location of the most proving node changes rapidly during the proof number search procedure. This effect can potentially weaken the performance of the proof number search [8]. Variations of the proof number search such as DeepPN [8] can potentially reduce the seesaw effect. Besides, the worst-case memory usage for proof number search is exponential. Large memory usage is also a tricky problem for standard proof number search. Techniques like PN2, and Df-pn with $1+\epsilon$ trick [23] can potentially ease this issue.

Before we end this section, We would briefly review the most proving node selection formula of DeepPN. DeepPN deals with the seesaw effect by its most proving node selection formula. In DeepPN, each node contains an additional information *node.deep*. During the initialization, *node.deep* is assigned as $\frac{1}{node.depth}$. During selection, DeepPN would prioritize the search in a child which has the minimum dpn value. In DeepPN the dpn value of a node is defined as

$$dpn(node) = \begin{cases} \delta' * R + d * (1 - R) & \text{if } node \text{ is undetermined} \\ \infty & \text{otherwise} \end{cases} \tag{5}$$

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

$$\delta' = \begin{cases} (1 - \frac{1}{node.dn}) & \text{if } node.parent \text{ is controlled by the minimizer} \\ (1 - \frac{1}{node.pn}) & \text{if } node.parent \text{ is controlled by the maximizer} \end{cases} \tag{6}$$

[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, while the *pn* and *dn* are updated the same way as they are in the standard proof number search [12] as shown in equation (7) and (8).

$$node.pn = \begin{cases} node.left.pn \ + \ node.right.pn & \text{if } node \text{ is a min node} \\ \min(node.left.pn, node.right.pn) & \text{if } node \text{ is a max node} \end{cases} \tag{7}$$

$$node.dn = \begin{cases} node.left.dn \ + \ node.right.dn & \text{if } node \text{ is a max node} \\ \min(node.left.dn, node.right.dn) & \text{if } node \text{ is a min node} \end{cases} \tag{8}$$

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

3 Proof Number Search and QBF solving

In this section, we will discuss how to design a proof number search based QBF solver, and how optimization techniques such as backjumping and QCDCL can be implemented in the context of proof number search. We will apply DeepPN as our searching method, and for simplicity, we will **disable** the *Pure_literal_enabled* flag. This means that in the simplification procedure, only unit propagation is going to be executed. Although disabling pure literal elimination might weaken the solver's performance, a detailed reason on why *Pure_literal_enabled* is set to false is going to be discussed at the end of this section.

3.1 DeepPN and Backjumping

If we combine Backjumping algorithm with DeepPN, 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 our solver, each proof number search tree node corresponds to a branching variable (i.e. the variable stored in the node is not unit). For implementation convenience, the following invariant must be kept during the entire searching procedure.

Invariant 3.1. The reason for each undetermined node is NULL.

Invariant 3.2. For any solved node, assume f_μ is the formula corresponding to the current node. The reason stored in the current node is a μ-contradicted clause (resp. μ-truth cube) if the node is disproved (resp. proved).

The main procedure of the DeepPN and Backjumping algorithm is almost the same as the baseline PNS procedure. As we can see from Algorithm 5 the only modification to the baseline PNS procedure is we replace the game state with the formula corresponding to each search tree node. And the truth value of the original formula f is given by $DeepPNsolve(f)$.

Algorithm 5: DeepPN based QBF solving procedure

```
   Input  : The QBF expression f with empty assignment
   Output: The truth value of the QBF
 1 Function DeepPNsolve(f_φ)
 2     root.depth = 1, current := root := initialize(f_φ, 1)
 3     iter := 0
 4     while (root is not solved and iter <= maximum_iteration) do
 5         iter := iter + 1
 6         while (next = selection(current, f_μ) is not null) do
 7             current := next
 8         expand(current, f_μ)
 9         while (current is not null) do
10             backpropagate(current, f_μ)
11             if (current is root) then
12                 break
13             current := backtrack(current, f_μ)
14     if (root is not solved) then
15         return UNSOLVED
16     if (root is falsified) then
17         return FALSE
18     return TRUE
```

For the remaining of this section, I will discuss how initialization, selection, expansion, backpropagation, and backtracking can be implemented in the context of DeepPN and Backjumping.

3.1.1 Initialization with reason computation

The algorithm for initialization is shown in Algorithm 6. In this algorithm, the *evaluate*(f) method would return *True* if the partial assigned formula f has all clauses satisfied, and it would return *False* if the formula under the current partial assignment has a contradictory clause. Otherwise, the function would return *Unknown*. As we can from Algorithm 6, we initialize the proof and disproof number with the standard mobility initialization [12] when the node is not a solved node. This means that during initialization, if the newly created node is undetermined, we initialize its proof (resp. disproof) number to be 1 and disproof (resp. proof) number to be the branching factor which is 2 when the newly created node is controlled by the existential (resp. universal) variable. We also select and store the next branching variable in *node.variable*. Otherwise, when the formula contains a contradictory clause or has all clauses satisfied, we mark the node as proved or disproved, and the reason for satisfiability or unsatisfiability is calculated and stored in *node.reason*. The way to calculate the initial reason is exactly the same as the QDLL case which is given by Theorem 2.1 and 2.2.

Algorithm 6: Initialize with reason

Input : The partial assigned QBF expression f
Output: A newly created proof number search tree node corresponds to f

```
1  Function Initialize(f_μ, depth)
2      node.left := node.right := node.parent := NULL
3      node.depth := depth
4      node.deep := 1/depth
5      if evaluate(f_μ) is Unknown then
6          node.variable := get_variable(f_μ)
7          if isexist(node) then
8              node.pn = 1, node.dn = 2
9          else
10             node.pn = 2, node.dn = 1
11         node.reason := NULL
12     else
13         if evaluate(f_μ) is True then
14             node.pn = 0, node.dn = ∞
15             node.reason := getReason(f, μ)
16         else
17             node.pn = ∞, node.dn = 0
18             node.reason := getReason(f, μ)
19     return node
```

3.1.2 Selection with Backjumping

During the selection procedure, we follow the standard DeepPN selection heuristic, which is we search in the direction with the minimum *dpn* value.

Algorithm 7: Find the child contains the most proving node

Input : The current node and the partial assigned formula
Output: The child of the current node which subtree contains the most proving node

```
1  Function selection(node, f_μ)
2      if (node has no children ) then
3          return NULL
4      v := node.variable
5      ret := arg min_{c∈node.child} dpn(c)
6      if (ret is the left child ) then
7          f := f_{μ;node.variable}
8      else
9          f := f_{μ;¬node.variable}
10     simplify(f)
11     return ret
```

As described in Algorithm 7, the selection procedure would return null if the current node does not have children 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 [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. In order to guarantee each node in the search tree is a branching node, after we assign a literal, we would simplify the formula by calling the simplification procedure described in Algorithm 1, and return the child such that its subtree contains the most proving node.

3.1.3 Expansion with reason computation

Expansion is the procedure of creating new nodes in the search tree. In the context of backjumping, during expansion, we have to both create the two children of the most proving node, initialize their proof and disproof number, and initialize the reason if the created node is a solved node. The pseudocode of the expansion procedure is given by Algorithm 8. In this algorithm, we assume that the partial assigned formula corresponds to the left (resp. right) child of a node is given by: 1) assign the branching variable positively (resp. negatively), 2) simplify the formula until there exists no unit literals.

Algorithm 8: Expansion with backjumping

Input: The node for expansion and the partial assigned formula f_μ

```
1 Function Expand(current, f_μ)
2     v := current.variable
3     fl := f_{μ;v}
4     simplify(fl)
5     node.left := initialize(fl, 1 + node.depth)
6     fr := f_{μ;¬v}
7     simplify(fr)
8     node.right := initialize(fr, 1 + node.depth)
9     L.parent := R.parent := current
```

3.1.4 Backpropagation with Backjumping

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. One of the main difference between proof number search and depth-first search is for proof number search 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 proof number search tree, the node is either a solved node or *node.variable* is a branching variable (i.e. not unit). Therefore, instead of calculating the reason for f_μ given the reason for $f_{\mu;l}$ and $f_{\mu;\neg l}$, in the context of proof number search, we calculate the reason for f_μ given the reason for $f_{\mu;l;S[f_{\mu;l}]}$ and the reason for $f_{\mu;\neg l;S[f_{\mu;\neg l}]}$. Here the notation $S[f_{\mu;l}]$ is the force assignment sequence of the formula $f_{\mu;l}$ that is defined in definition 2.13.

The strategy of calculating the reason for f_μ given $f_{\mu;l;S[f_{\mu;l}]}$ and $f_{\mu;\neg l;S[f_{\mu;\neg l}]}$ is as follows. We firstly calculate the reason for $f_{\mu;l}$ (resp. $f_{\mu;\neg l}$) given the reason for $f_{\mu;l;S[f_{\mu;l}]}$ (resp. $f_{\mu;\neg l;S[f_{\mu;\neg l}]}$). Then we use the reason for $f_{\mu;l}$ and $f_{\mu;\neg l}$ to derive the reason for f_μ. To achieve the first part, we define the method undo, which would compute the reason for $f_{\mu;l}$ based on the reason for $f_{\mu;l;S[f_{\mu;l}]}$.

Algorithm 9: Compute the reason for $f_{\mu;l}$

Input : The partial assigned formula $f_{\mu;l;S[f_{\mu;l}]}$ and its corresponding reason

Output: The reason for $f_{\mu;l}$

```
1  Function UNDO(f_{μ;l;S[f_{μ;l}]}, reason)
2      result := reason
3      ν := μ; l; S[f_{μ;l}]
4      while (f_ν ≠ f_{μ;l}) do
5          Let lit be the last assigned literal in ν
6          ν := ν \ {lit}
7          if (result ≠ NULL) then
8              if (lit is unit in f_ν) then
9                  if (f_ν is unsatisfiable and ¬lit ∈ result) then
10                     C := a clause in f such that lit is a unit w.r.t. C_{f_ν}
11                     result := resolve(f_ν, result, C, lit)
12     return result
```

As we can see from Algorithm 9, we repeatedly unassign the last assigned literal l 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 $f_{\mu;l}$. Once the reason for $f_{\mu;l}$ and $f_{\mu;\neg l}$ are derived, we can calculate the reason for f_μ based on the reason computation rules listed in table 1 with slight modification. In order to save memory, if a node is solved, we can remove its left and right children.

Algorithm 10: Backpropagation with Backjumping

Input: The node for update and the partial assigned formula

```
1  Function Backpropagate(node,f_μ)
2      v := node.variable
3      ret := arg min_{c∈node.child} dpn(c)
4      node.deep := ret.deep
5      if v is existential 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     fl := f_{μ;v;S[f_{μ;v}]}
12     fr := f_{μ;¬v;S[f_{μ;¬v}]}
13     if fl contains an empty clause then
14         node.left.reason := getReason(f, μ;v;S[f_{μ;v}])
15     if fr contains an empty clause then
16         node.right.reason := getReason(f, μ;¬v;S[f_{μ;¬v}])
17     left := undo(fl, node.left.reason)
18     right := undo(fr, node.right.reason)
19     process_reason(left, right,f_μ)
20     if (node is solve) then
21         node.left := node.right := NULL
```

The process of backpropagation is shown in Algorithm 10. Except line 11 to 19, the backpropagation procedure in DeepPN and Backjumping should behave exactly the same as the backpropagation procedure described in [8].

We will focus on how *process_reason*() works in backpropagation with backjumping. This is the process of calculating the reason for f_ν given the reason for $f_{\nu;l}$ and $f_{\nu;\neg l}$. Despite the order of exploring the children can be different in DeepPN, the way to compute the reason for a branching node is exactly the same as the rules in Table 1 in the QDLL-Backjumping scenario. There are 7 cases to be considered for the backpropagation in the backjumping scenario.

• If both the $left$ and $right$ are reason for satisfiability, and *current.variable* is a universal branching variable. The reason of the current node can be computed according to case 11, 13, and 14 in Table 1.

• If both $left$ and $right$ are reason for unsatisfiability, and *current.variable* is an existential branching variable. The reason of the current node can be computed according to case 6, 8, and 9 in Table 1.

• Only $left$ (resp. $right$) is a reason for satisfiability, 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 variable $left$ (resp. $right$).

• Only $left$ (resp. $right$) is a reason for unsatisfiability, and

current.variable is universal. This is equivalent to case 10 (resp. case 12) in Table 1 and the reason of the current node is the same as the reason stored in the variable $left$ (resp. $right$).

• Only $left$ (resp. $right$) is a reason for satisfiability and $right$ (resp. $left$) cannot be determined yet, and *current.variable* is universal, *current.variable* (resp. $\neg$*current.variable*) does not occur in the reason stored in the variable $left$ (resp. $right$). Similar to case 11 in the standard backjumping scenario, this is when the universal pruning occurs in DeepPN. 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 $left$ (resp. $right$) is a reason for unsatisfiability and $right$ (resp. $left$) cannot be determined yet, and *current.variable* is existential, $\neg$*current.variable* (resp. *current.variable*) does not occur in the reason stored in the variable $left$ (resp. $right$). Similar to case 6 in the standard backjumping scenario, this is when the existential pruning occurs in DeepPN. The reason of the current node is the same as the reason stored in the variable $left$ (resp. $right$), 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.

One might ask why on line 13 to 16 of the algorithm the reasons for $f_{\mu;v;S[f_{\mu;v}]}$ and $f_{\mu;\neg v;S[f_{\mu;\neg v}]}$ are recalculated when $f_{\mu;v;S[f_{\mu;v}]}$ or $f_{\mu;\neg v;S[f_{\mu;\neg v}]}$ contain an empty clause. A short answer to this question is this would ensure the reason stores the children of a node is still a valid reason, although this issue is going to be discussed in detail at the end of section 3.2.7. However, in the context of backjumping, these 4 lines of pseudocode can be removed if we enforce the following condition during simplification and *getReason*.

• In the procedure $simplify()$, if the partial assigned formula contains multiple unit literals, the unit literal with the least numeric order is going to be assigned first.

Note that because our solver targets input in QDIMACS format, the numeric order is just the numeric order we are familiar with. For example, when we detect unit literals -2 and 1, we would prioritize the unit propagation on -2. These conditions are used to ensure when a node in the search tree is visited more than once, the partial assigned formula corresponds to the node is always the same formula. As a result, the reason corresponds to a node is still a valid reason corresponds to that node.

3.1.5 Backtracking with reason

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 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 7). Note that

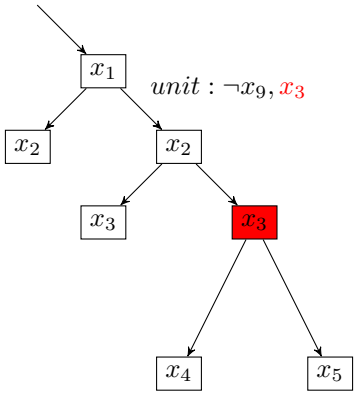

Figure 1: The issue with clause learning

since the reason calculation is done in the backpropagation procedure, we do not need to do any reason computation during backtracking.

3.2 DeepPN and QCDCL

In the context of depth-first search, any derived reason for unsatisfiability (resp. satisfiability) can be added to the formula conjunctively (resp. disjunctively) without affecting the correctness of the algorithm. However, in the context of proof number search, the difference between backjumping and QCDCL is much more than that.

3.2.1 Difficulty

The main difficulty of combining DeepPN and QCDCL is the "change" of the searching space after new clauses and cubes are added to the formula during the middle of the search. We will use an example to illustrate how clause learning might affect the proof number search solving procedure we discussed in 3.1. As we can see from figure 1, assume we have learned some new constraints on other parts of the search tree. After we assign $\neg x_1$, the literal x_3 that's originally not unit becomes unit. When we traverse down to the node with branching variable x_3, we run into a problem. Because according to the selection method discussed in the context of backjumping, we should branch on x_3. However, since the variable $|x_3|$ has been assigned, we can no longer branch on x_3. There are 2 "natural" options to deal with this case.

• We pick a new branching variable for the red node and delete the subtree of the red node.

• We postpone any additional unit propagation until the expansion of the most proving node.

Note that for the first approach, we would waste all the search that has been done in the subtree of the red node. For the second approach, although the

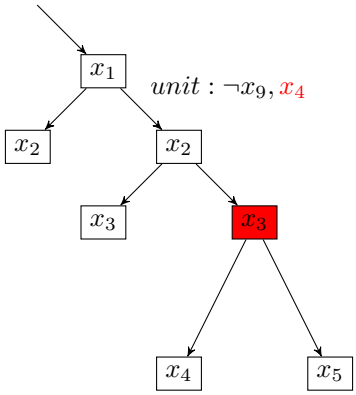

Figure 2: Root to node assignment

structure of the search tree is preserved, and the issue in figure 1 would never happen, in the ideal case, we do not want to postpone any unit propagation. The desired combination should contain the following features: 1) No unit propagation is postponed, 2) the search that has been done in earlier iterations should not be wasted. In 2017, Schloeter proposed a novel SAT-solving algorithm that combines Monte Carlo tree search and CDCL which addressed the above difficulty in the SAT world [25]. For the rest of section 3.2, we will discuss how Schloeter's method can be generalized from SAT to QBF. In backjumping, the search tree structure never changed, however, with the application of Schloeter's method, the search tree structure can be updated. For the convenience of later discussion, we would clarify the following terminologies in advance.

• All search tree nodes have a memory address during creation, when we say *a node is visited again during a later iteration*, we refer to the node with the **same memory address** is visited again during a later iteration, instead of the node with the same "location" in the tree is visited again.

• The partial assignment made from root to a node N refers to "during an iteration from root to N, all the literals assigned during this procedure". This includes all the branching literals in the ancestor of N (exclude the branching variable stored in N) and all the forced assigned unit literals. We would use figure 2 as an example, the partial assignment made from root to the red node is $\neg x_1; \neg x_9; \neg x_4; \neg x_2$.

3.2.2 DeepPN with CDCL and SBJ

Instead of considering the combination of DeepPN and QCDCL directly, let us start with a relatively simpler scenario. In this scenario, the desired solver consists of the following features.

- DeepPN
- Unit propagation
- Conflict Driven Clause Learning

• Solution Driven Backjumping

In other words, under this solver configuration, no reason for satisfiability is going to be added to the formula, and the formula is still in conjunctive normal form. In addition, pure literal elimination is deactivated. For simplicity, we assume **a learned clause is never going to be deleted**. Although we ignore SDCL in this solver configuration,we have already added solution driven backjumping to Schloeter's method, a technique that does not exist in the context of SAT. We denote this solver configuration as DeepPN-CDCL-SBJ. Under this solver configuration, the game state f under partial assignment μ can be written in form $[\Phi\ ,\ \theta\ ]_\mu$ (check equation (2) for the definition of Φ, θ). For each node in the search tree we store the exact same information as the backjumping scenario. 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. The main search loop of the DeepPN-CDCL-SBJ procedure is exactly the same as the DeepPN-Backjumping procedure that is described in Algorithm 5. In the following subsections, we will describe how initialization, selection, expansion, backpropagation, and backtracking should change in the context of DeepPN-CDCL-SBJ. We assume the formula f always has the form $[\Phi\ ,\ \theta]$ (i.e. $\Phi\ \wedge\ \theta$) during the solving procedure.

3.2.3 Initialization with clause learning

The initialization in DeepPN-CDCL-SBJ works almost exactly the same as the initialization method that is introduced in DeepPN-Backjumping. To avoid redundancy, we only focus on the difference between the initialization in DeepPN-CDCL-SBJ and DeepPN-Backjumping. The only difference between the two algorithms is the state of a partial assigned formula is not only determined by clauses of the original formula, but also the learned clauses. This implies that we must modify the evaluate method to Algorithm 11.

Algorithm 11: Evaluation in clause learning

Input : The partial assigned formula
Output: The evaluation result of the formula

```
1 Function evaluate_cdl([Φ , θ ]_μ)
2     if (Φ ∧ θ)_μ or has a contradictory clause then
3         return False
4     if all clauses in Φ_μ are satisfied then
5         return True
6     return Unknown
```

Despite this tiny change, the initialization method described in Algorithm 6 can still be reused. In other words, at the end of the initialization procedure, the newly created nodes still have their proof and disproof number initialized. More importantly, if a newly created node is a solved node, the corresponding reason

for unsatisfiability or satisfiability is still calculated by theorem 2.1, and 2.2 and stored in the node.

3.2.4 Selection with clause learning

Selection is the part of the algorithm that differs the most between DeepPN-Backjumping and DeepPN-CDCL-SBJ. We not only need to find the most proving node but also need to update the structure of the tree whenever the clause learning causes a branching variable to be assigned earlier. To ensure the correctness of our algorithm, we must ensure that after each branching variable assignment, we call the simplification method to eliminate **all** unit literals.

The pseudocode for the selection procedure is presented in Algorithm 12. The input of the method is a node of the search tree such that the most proving node is located in its subtree. The partial assigned QBF expression f of the form $[\Phi\ ,\ \theta\ ]_\mu$ is also passed in as the second parameter of this method. As we can see from Algorithm 12, if we have reached a node that is marked as solved during a previous iteration we would return NULL as well. In addition, if we have reached a node that is not marked as solved during a previous iteration but f_μ is evaluated to be True or False by the $evaluate\_cdl()$ method, we would initialize this node. Note that the above two base cases are unique to DeepPN-CDCL-SBJ, because in DeepPN-Backjumping we are guaranteed that no solved nodes can be visited again during the selection procedure. Same as backjumping, when we reach a node without children, then we return NULL. This simply means that we have reached the most proving node.

Algorithm 12: Select MPN with clause learning

```
Input  : The current node and the partial assigned formula
Output: A node which subtree contains the most proving node
Function selection_cdl(node, f_μ)
    if evaluate_cdl(f_μ) is not Unkown then
        node := initialize(f_μ)
        return node
    if node has no children or node is solved then
        return NULL
    v := node.variable
    if v ∈ μ or ¬v ∈ μ then
        if v ∈ μ then
            ret := node.left
            Link up the node.left with node.parent
            Remove node
        else
            ret := node.right
            Link up the node.right with node.parent
            Remove node
        return ret
    ret := arg min_{c∈node.child} dpn(c)
    if (ret is the left child ) then
        f := f_{μ;node.variable}
    else
        f := f_{μ;¬node.variable}
    simplify(f)
    return ret
```

If none of the above base cases happen, there are two cases we must consider. Firstly, if the branching variable stored in the current node has already been assigned due to additional unit propagation, we would check if the branching variable is assigned positively or negatively. If it has been assigned positively (resp. negatively), we would link up the left (resp. right) child of the current node with *current.parent*, and continue the most proving node selection procedure from the left (resp. right) child. Otherwise, we can still branch on the variable stored in the current node. In this case, we can simply follow the standard DeepPN selection procedure which is finding the child with the minimum dpn value, and keep searching the most proving node in the corresponding direction. If we revisit the example in section 3.2.1, as shown in figure 3, during the selection phase, if we have detected the branching variable $|x_3|$ has been assigned positively as unit when we reach the red node x_3, we would simply link up the parent of x_3 with its left child (represent as the green node), and continue the search from x_4. One remark is similar to the backjumping case, the

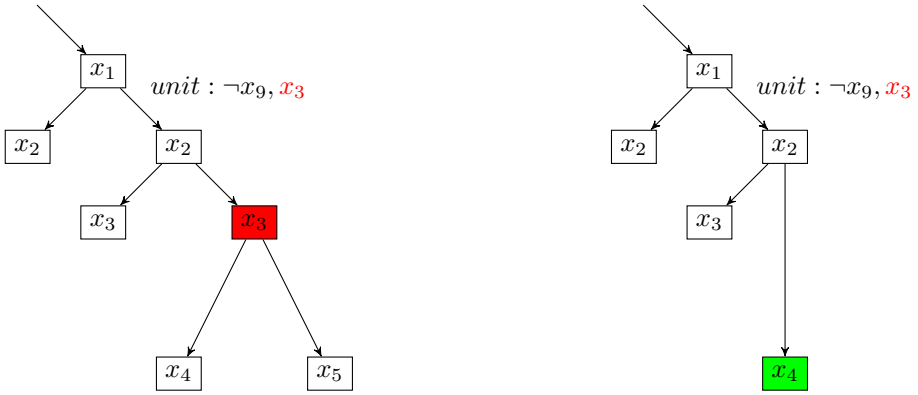

Figure 3: Schloeter's method

return value of the *selection_cdl*() method is a child of the current node which subtree contains the most proving node of the search tree. Therefore, a single call to the method is **not** going to return the most proving node. The most proving node should be obtained by repeatedly call the *selection_cdl*() method until NULL is returned (line 6, 7 of algorithm 5).

There are two important properties of this selection algorithm that were not explicitly shown in Schloeter's paper which are, however, critical for our later discussion.

Lemma 3.1. *Suppose a node N with branching variable v is visited at least twice during the solving procedure. Suppose during the ith visit of node N, the partial assignment made in the root to N path is μ such that $v \notin \mu$ and $\neg v \notin \mu$. For the (i+1)th visit, the partial assignment made in the root to N path is μ'. Let β be the* ***sequence*** *of all the branching literals in μ and β' be the* ***sequence*** *of all the branching literals in μ'. We also let $U(\beta)$ denote all the universal literals in β. We have: 1) $set(\beta') \subseteq set(\beta)$; 2) $U(\beta) = U(\beta')$.*

Proof. Assume during the ith visit of the node N the there are k branching literals assigned on the root to N path which means $\beta = [b_1, b_2, ..., b_k]$. If we examine Algorithm 12, we could easily see that during two consecutive visits of node N all the nodes that are branching during the ith iteration have been examined in the order from b_1 to b_k. The first part of the lemma almost trivially holds. During two consecutive visits of the same node with branching variable $|b_j|$ such that $1 <= j <= k$, there are only two cases.

1. The branching variable b_j has been assigned either positively or negatively. According to line 8 to 17 of the algorithm, we would simply skip the assignment of the branching variable for this case, and link up the parent of the current node with one of the children. In other words, during the (i+1)th visit of node N, the branching literal b_j has been removed from β.

2. The branching variable b_j has not been previously assigned. We would enter the standard case of proof number search, which is to branch on the

variable $|b_j|$. Note that in order to reach the node N, we must have assigned b_j instead of $\neg b_j$. In both cases, no additional branching literal is introduced. Thus, it is simple to conclude $set(\beta') \subseteq set(\beta)$.

Similarly, for the second part of the lemma, if we examine Algorithm 12, the only case a branching literal l such that $l \in \beta$ but $l \notin \beta'$ is when the branching variable $|l|$ has been assigned earlier. Note that this is only possible because clause learning causes either l or $\neg l$ to be assigned as unit. Since all nodes that store a universal branching variable have been examined, and there exists no universal unit propagation in DeepPN-CDCL-SBJ (i.e. we do not have cube learning in this solver configuration, hence unit literals can only be existential). As long as we can revisit N during a later iteration, we must have assigned exactly the same set of universal literals. This is equivalent to $U(\beta) = U(\beta')$. □

Lemma 3.2. *Suppose a node N has a branching variable v stored and is visited at least twice during the solving procedure. Suppose during the ith visit of node N, the partial assignment made on the root to N path is μ such that $v \notin \mu$ and $\neg v \notin \mu$. For the $(i+1)$th visit, the partial assignment made in the root to N path is μ'. Suppose during the $(i+1)$th visit, the formula with partial assignment μ' does not contain an empty clause, we have $set(\mu) \subseteq set(\mu')$.*

Proof. Firstly note that the condition $v \notin \mu$ and $\neg v \notin \mu$ is necessary, otherwise the node N would be removed after the ith visit. Suppose f is the QBF expression during the ith visit of N and f' be the QBF expression during the $(i+1)$th visit of N that contains a superset of clauses (the clauses of f' are the ones in f and the ones that are learned between the ith visit of N and the $(i+1)$th visit of N). Suppose during the ith visit of node N, the sequence of branching literals assigned from the root to N path are $b_1, b_2, ..., b_k$, and the nodes contain these branching literals are $N_1, N_2, , , , , N_k$; we also denote the partial assignment made on the root to N_i path as μ_i. Note that under this setting, the node N can also be represented as N_{k+1}. It can be easily seen from Algorithm 12 that all of the nodes N_1 to N_k are visited during the (i+1)th visit of node N although some of the nodes would be removed after detecting the branching literal b_i has been assigned already. Therefore, we denote the partial assignment made from the root to N_i path during the $(i+1)$th visit of N as μ'_i.

We would prove the lemma by induction.

Induction Base: According to the selection method defined in Algorithm 12, $set(\mu_j) \subseteq set(\mu'_j)$ when $j = 0$. Note that when $j = 0$, μ_0 corresponds to the partial assignment made from the root to root which is the empty set. Therefore, $set(\mu_0) \subseteq set(\mu'_0)$ obviously holds.

Induction hypothesis: According to the selection method defined in Algorithm 12, $set(\mu_j) \subseteq set(\mu'_j)$ for all $j <= k$.

Induction step: We need to show $set(\mu_{k+1}) \subseteq set(\mu'_{k+1})$. There are two cases.

1. $b_k \notin \mu'_k$. In this case, the search tree structure does not need to be updated, and $|b_k|$ is still a branching variable. Note that N_{k+1} is reachable

only if we assign b_k instead of $\neg b_k$ at node N_k. And by definition of forced assignment sequence (check definition 2.13) we have

$$\mu_{k+1} = \mu_k; b_k; S[f_{\mu_k;b_k}] \tag{9}$$

, and

$$\mu'_{k+1} = \mu'_k; b_k; S[f'_{\mu'_k;b_k}] \tag{10}$$

It is obvious that based on the induction hypothesis, we only need to show that for every $l \in S[f_{\mu_k;b_k}]$, we have $l \in \mu'_{k+1}$. Suppose $S[f_{\mu_k;b_k}]$ has length n, and the jth unit literal is e_j that is unit with respect to clause C_j. It is obvious that $e_j \in C_j$ for all j. We need to apply a very important property of unit propagation [1]. If a QBF expression g is in conjunctive normal form, and there exist more than one unit literals. If we simplify the formula until it contains no unit literals. Either the resulting formula contains a contradictory clause or it would result in the same formula g' regardless of the order of propagating those unit literals. This property allows us to only consider the case that the unit literals in $S[f'_{\mu'_k;b_k}]$ are propagated according to the following order: Unit literals that are in $S[f'_{\mu'_k;b_k}]$ and also in $S[f_{\mu_k;b_k}]$ are propagated first, according to their unit propagation order in $S[f_{\mu_k;b_k}]$; while unit literals that only exist in $S[f'_{\mu'_k;b_k}]$ but not in $S[f_{\mu_k;b_k}]$ are propagated afterward. We would apply proof by induction. Suppose for every $1 <= j <= n-1$, $e_j \in \mu'_{k+1}$, we need to show that $e_n \in \mu'_{k+1}$ as well. There are three cases:

1) e_n has been assigned as unit already, $e_n \in \mu'_{k+1}$ trivially holds.

2) $\neg e_n$ has been assigned as unit already. Consider the clause C_n. Firstly, for all the universal variables in C_n, according to lemma 3.1, we know they must either have not been assigned yet, or assigned as the same value as they are in $\mu_k; b_k$. In other words, none of the universal variables in C_n is satisfied by the assignment set $set(\mu'_k) \cup \{b_k\} \cup set([e_1, ...e_{n-1}])$. Secondly, for all the existential literals in C_n that is not equal to e_n, because of $set(\mu_k) \subseteq set(\mu'_k)$ and for every $1 <= j <= n-1$, $e_j \in \mu'_{k+1}$, we know that these existential literals must be assigned as the same value as they are in $\mu_k; b_k$. Since e_n is unit with respect to C_n under the partial assignment $\mu_k; b_k; e_1, ...e_{n-1}$, we know that if we assign $\neg e_n$, all existential literals in C_n have been falsified. Therefore, the clause C_n is contradictory. Note this contradicts with the precondition of this lemma that $f'_{\mu'_{k+1}}$ does not contain an empty clause. Hence, $\neg e_n$ cannot be assigned as unit.

3) $|e_n|$ has not been assigned. We also consider the clause C_n. Similar to the previous case, it is not hard to derive that for this case, e_n must be unit with respect to C_n. Hence, $e_n \in S[f'_{\mu'_k;b_k}]$, and $e_n \in \mu'_{k+1}$.

Thus, we conclude that $e_n \in \mu'_{k+1}$. By induction, every $l \in S[f_{\mu_k;b_k}]$, $l \in \mu'_{k+1}$. Hence, we have shown that $set(\mu_{k+1}) \subseteq set(\mu'_{k+1})$.

[1] This property was explicitly mentioned in the following lecture note in the context of SAT `https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/solvers.html` while it is not too hard to show it also holds the context of QBF

2. $b_k \in \mu'_k$. In this case, the search tree structure is updated, and $\mu'_{k+1} = \mu'_k$. Note that μ_{k+1} would have exactly the same expression as equation (9). We must show that $l \in S[f_{\mu_k;b_k}]$, we have $l \in \mu'_k$. We use a very similar proof procedure as case 1, the notation C_j and e_j have exactly the same meaning as before. We also proof by induction. Suppose every $1 <= j <= n-1$, we have $e_j \in \mu'_k$. We need to show that $e_n \in \mu'_k$ as well. There are also three cases we need to consider:

1) e_n has been assigned as unit already, $e_n \in \mu'_k$ trivially holds.

2) $\neg e_n$ has been assigned as unit in μ'_k. By the induction hypothesis, we have

$$set(\mu_k) \subseteq set(\mu'_k) \tag{11}$$

, and

$$set([e_1, ...e_{n-1}]) \subseteq set(\mu'_k) \tag{12}$$

Hence,

$$set(\mu_k; b_k; [e_1, ...e_{n-1}]) \subseteq set(\mu'_k) \tag{13}$$

Therefore, if we replicate the reasoning procedure of case 1 2), it is not difficult to derive that C_n must be contradicted which contradicts the precondition of the lemma.

3) $|e_n|$ has not been assigned. The proof to this part is almost identical to the proof of case 1 3), and we can easily obtain that e_n must be unit in $f'_{\mu'_k}$. Since all unit propagations are done immediately, e_n must have already been assigned as unit in μ'_k. In conclusion, when $b_k \in \mu'_k$, $set(\mu_{k+1}) \subseteq set(\mu'_{k+1})$ also holds. Since we have proved the lemma holds in both cases, we have completed the proof of the lemma. □

3.2.5 Expansion with learning

Expansion in DeepPN-CDCL-SBJ is very similar to the expansion procedure introduced in the DeepPN-Backjumping (Algorithm 13). To avoid redundancy, we only explain the cases that are unique to DeepPN-CDCL-SBJ. There are two unique cases for DeepPN-CDCL-SBJ that does not exist in DeepPN-Backjumping. Firstly, in DeepPN-Backjumping, it is impossible to call expansion at a node that is solved or the partial assigned formula is evaluated to be True or False. In DeepPN-CDCL-SBJ however, because of the learned clauses such case is possible. Note that this case can happen only after line 5 to 6 of Algorithm 12 is executed. In other words, although we attempt to expand a solved node, its proof number, disproof number and reason for satisfiability/unsatisfiability has already been calculated. In this case, we would simply exit the expansion algorithm. The second case is when we try to create the left and right child of the current node, the branching variable has already been assigned due to additional unit propagation triggered by the learned clauses. If this is the case, before we do the standard node expansion, we must update the branching variable of the current node with an unassigned variable in the outermost quantifier block. If neither of the two cases happen, we would follow

the expansion procedure that we have discussed in the context of backjumping.

Algorithm 13: Expansion with Learning

```
   Input: The node for expansion and the partial assigned formula f_μ
1  Function expand(current, f_μ)
2      if current is solved then
3          return
4      if current.variable has been assigned then
5          current.variable := get_variable()
6      v := current.variable
7      fl := f_{μ;v}
8      simplify(fl)
9      L := initialize(fl)
10     fr := f_{μ;¬v}
11     simplify(fr)
12     R := initialize(fr)
13     L.parent := R.parent := current
14     L.depth := R.depth := current.depth + 1
```

3.2.6 Backpropagation and backtracking with learning

Backtracking and backpropagation in DeepPN-CDCL-SBJ are very similar to the ones discussed in the Backjumping and DeepPN section. During backtracking, we just need to unassign all the literals that are assigned between *current* and *current.parent*, and return the parent of the current node. As shown in algorithm 14, during backpropagation, we maintain the proof number, the disproof number, and the deep value the same way as before. We also calculate the reason for satisfiability or unsatisfiability of the current node by calling the *process_reason*() method. Note that the behavior of *process_reason*() is exactly the same as we have discussed in section 3.1.4. Because clause learning is enabled, we can add any reason for unsatisfiability to θ conjunctively. However, we can potentially learn exponential many clauses if we add all the clauses to the formula. Therefore, we follow the clause learning schema of the QBF solver Quaffle described in this paper [30] and would learn a clause only if the clause does not appear in Φ or θ (again we assume the formula f has form $\Phi \wedge \theta$).

Algorithm 14: Backpropagation with learning

Input: The node for update and the partial assigned formula

```
1  Function backpropagate_cdl(node,f_μ)
2      if node is solved then
3          return
4      v := node.variable
5      ret := arg min_{c∈node.child} dpn(c)
6      node.deep := ret.deep
7      if v is existential then
8          node.pn := min(node.left.pn, node.right.pn)
9          node.dn := node.left.dn + node.right.dn
10     else
11         node.dn := min(node.left.dn, node.right.dn)
12         node.pn := node.left.pn + node.right.pn
13     fl := f_{μ;v;S[f_{μ;v}]}
14     fr := f_{μ;¬v;S[f_{μ;¬v}]}
15     if fl contains an empty clause then
16         node.left.reason := getReason(f, μ; v; S[f_{μ;v}])
17     if fr contains an empty clause then
18         node.right.reason := getReason(f, μ; ¬v; S[f_{μ;¬v}])
19     left := undo(fl, node.left.reason)
20     right := undo(fr, node.right.reason)
21     process_reason(left,right,f_μ)
22     if (node is solve) then
23         node.left := node.right := NULL
24     if node is not solved and v is existential then
25         if left is not NULL and left is a reason for UNSAT then
26             learn(f, left)
27         if right is not NULL and right is a reason for UNSAT then
28             learn(f, right)
```

3.2.7 Correctness discussion

Although the proof number search with clause learning algorithm is almost identical to Schloeter's method if f is a QBF with no universal variables, the correctness of this algorithm in the context of general QBF does not trivially hold. There are two problems we need to solve before we confirm that Schloeter's method can be generalized to QBF. One problem we need to consider is the reusability of the previously derived proved and disproved nodes. For example in figure 4, assume we have previously derived a proved node (i.e. the green node). After the solver learned some additional constraints on other parts of the search tree, an additional unit propagation x_7 is triggered. The problem is if

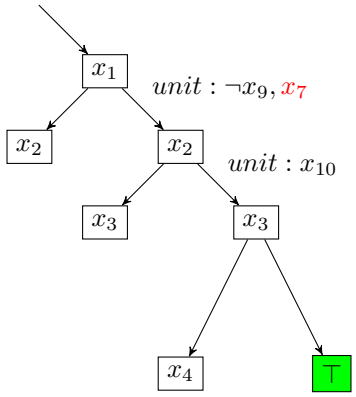

Figure 4: Reusability of solved nodes

the learned constraints and additional unit propagation change the truth value of a proved node to disproved node (e.g. $\neg x_7$ was used during the derivation of the truth value of the green node). Note that in the context of SAT, such problem does not exist because:

• Clause learning is not going to change a derived disproved node to be a proved node, and in the context of SAT, there is no proved node, because once a proved node is found, the program would terminate.

• Resolution would not reduce the number of SAT solutions. However, in QBF, since we are using Q-resolution during clause learning, the number of solution nodes can decrease. This is mainly because of the universal reduction step in Q-resolution. Therefore, although the correctness of Schloeter's method trivially holds in SAT, we need to show its correctness in the context of QBF. To justify the correctness of the DeepPN-CDCL-SBJ algorithm, we must prove clause learning is not going to change a previously derived proved node to a disproved node. For the convenience of the proof, we introduce the following notations. We define $Con(f, \mu)$ as the QBF obtained by concatenating all literals in μ with clauses in f conjunctively while replacing the quantifiers of variables in μ with existential quantifiers. For example, let f be the QBF expression

$$f \;=\; \exists\, x_1\, \forall\, y_1\, \exists\, x_2 \quad (y_1 \;\vee\; x_2) \;\wedge\; (\neg x_1 \;\vee\; y_1 \;\vee\; \neg x_2) \tag{14}$$

Then, we have

$$Con(f, \neg y1; x_2) \;=\; \exists\, x_1 \exists\, y_1 \exists\, x_2 \;\; \neg y_1 \wedge x_2\, (y_1 \vee x_2) \wedge (\neg x_1 \vee y_1 \vee \neg x_2) \tag{15}$$

At the same time, we define $Assign(f, \mu)$ as the QBF expression obtained by assigning all literals in μ to f. By the soundness of unit propagation it is not hard to see that $Assign(f, \mu) = Con(f, \mu)$. Note that the notation f_μ is different to $Assign(f, \mu)$, because for the notation f_μ, the partial assignment μ must be a valid **assignment**, but for $Assign(f, \mu)$, μ can be any sequence of literals. However, when μ is a valid assignment of f, $f_\mu = Assign(f, \mu)$

Theorem 3.3. *Suppose the clauses of the QBF f are stored conjunctively in Φ, and the branching variable stored in the current node N is v. During a previous selection procedure, the partial assignment made on the root to N path is ν, such that $f_{\nu;v}$ was proved to have a truth value R_1. Suppose during a later selection procedure, the partial assignment made in the root to N path is ν', and the clauses learned between the two visits of N are stored conjunctively in θ, and the QBF expression is now f'. Let the truth value of $f'_{\nu';v}$ be R_2, then $R_1 = R_2$.*

Proof. Let $\mu = \nu; v$ and $\mu' = \nu'; v$, and β be the sequence of all branching variables in μ, and β' represents the sequence of all branching variables in μ', then according to lemma 3.1 and lemma 3.2, we have

$$set(\beta') \subseteq set(\beta) \tag{16}$$

, and

$$set(\nu) \subseteq set(\nu') \tag{17}$$

Hence,

$$set(\mu) \subseteq set(\mu') \tag{18}$$

By the relationship between branching literals and partial assignment, it is trivial that

$$set(\beta) \subseteq set(\mu) \tag{19}$$

, and

$$set(\beta') \subseteq set(\mu') \tag{20}$$

Using our newly defined notations, we have

$$R_1 = Con(f, \mu) = Assign(f, \mu) \tag{21}$$

, and

$$R_2 = Con(f', \mu') = Assign(f', \mu') \tag{22}$$

By the soundness of unit propagation, we have

$$Assign(f, \mu) = Assign(f, \beta) \tag{23}$$

, and

$$Assign(f', \mu') = Assign(f', \beta') \tag{24}$$

Since we have shown $set(\beta) \subseteq set(\mu)$, and $set(\mu) \subseteq set(\mu')$, we have $set(\beta) \subseteq set(\mu')$. Thus, it is not hard to show that for all literals $l \in \beta$ and $l \notin \beta'$, we have l assigned as unit in μ'. Thus, because of the soundness of unit propagation, we have

$$Assign(f', \beta') = Assign(f', \beta) \tag{25}$$

Note that f has form Φ while f' has the form $\Phi \wedge \theta$. Since all clauses in θ are derived from Q-resolution of the clauses in Φ and θ, and the fact that Q-resolution is sound, we have

$$Con(f', \beta) = Con(\Phi \wedge \theta, \beta) = Con(\Phi, \beta) = Con(f, \beta) \tag{26}$$

Note that $Con(\Phi, \beta) = Assign(\Phi, \beta)$ and $Con(\Phi \wedge \theta, \beta) = Assign(\Phi \wedge \theta, \beta)$. Therefore, by combining equation (21), (22), (23), (24), (25), (26), we have $R_1 = R_2$ which completes the proof. □

Another correctness issue actually underlies within the backpropagation procedure. In backpropagation, we have assumed that the method $process\_reason()$ would calculate the reason for $f'_{\mu'}$ given the reason for $f'_{\mu';l;S[f'_{\mu';l}]}$ and the reason for $f'_{\mu';\neg l;S[f'_{\mu';\neg l}]}$. However, one might note that in our design, we do not recalculate the reason for $f'_{\mu';l;S[f'_{\mu';l}]}$ and the reason for $f'_{\mu';\neg l;S[f'_{\mu';\neg l}]}$. Instead, we use the reason that is previously stored in the left and right child as the reason for $f'_{\mu';l;S[f'_{\mu';l}]}$ and the reason for $f'_{\mu';\neg l;S[f'_{\mu';\neg l}]}$. One might doubt the correctness of such simplification. Indeed, the reason we previously stored in the a child is derived based on a previous formula f and a different partial assignment ν. Suppose we want to reuse the reason for a node N during a later iteration, the formula during this iteration is f', and the partial assignment from root to N is ν'. There are two cases. Firstly, $f'_{\nu'}$ contains an empty clause. Then, we can simply recalculate the reason for $f'_{\nu'}$ based on Algorithm 3. Otherwise, the node N stores a reason that has been previously derived. We should note that if the reason is a reason for unsatisfiability, the reason is a ν-contradicted clause, otherwise, it is a ν-truth cube. Because of lemma 3.1, and lemma 3.2, we can easily show that $set(\nu) \subseteq set(\nu')$, and a ν-contradicted clause is a ν'-contradicted clause, and a ν-truth cube is a ν'-truth cube. This simply means that the reason stored in N can be reused in this case. Be aware that we cannot ignore the first case, because lemma 3.2 does not hold when $f'_{\nu'}$ contains an empty clause. This also answers the question on why we need to recalculate the reason at the end of section 3.1.4 if we do not enforce the order of performing unit propagation during the simplification procedure.

Since $resolve()$ in backpropagation is implemented using the $rec - resolve$ procedure in [6], according to lemma 4 in that paper, if we can associate each assignment μ with a μ-contradicted clause (resp. μ-satisfied cube), the resolution procedure is sound and terminating. Hence, the reason that is stored in the node N can also be used for Q-resolution. As a result, our reason derivation method described in the backpropagation procedure is sound and terminating.

3.2.8 Complications in solution driven cube learning

Solution driven cube learning is a very important component in a QCDCL solver. One might think it is very simple to convert backjumping to cube learning just by applying Schloeter's method. In this section, we would discuss the feasibility of a "full" proof number search based QCDCL solver that features:

- DeepPN
- Unit propagation
- Conflict Driven Clause Learning
- Solution Driven Cube Learning

Since in QCDCL, we must associate each satisfied node with a μ-satisfied

cube, and each contradicted node with a μ-contradicted clause, we should show whether such task is feasible in the context of proof number search and Schloeter's method. The main difference between solution driven cube learning and solution driven backjumping is with the learned cubes, the concept of universal unit propagation comes in. It is widely believed that universal unit propagation can affect existential unit propagation [9]. In other words, depending on the order of propagating the unit existential and universal literals, the resulting formula might be different. This would make lemma 3.1 and the proof of lemma 3.2 no longer holds. At the same time, even if this superset-property can be ensured, because of the existence of universal unit propagation, a μ-contradicted clause might not be μ'-contradicted even if $set(\mu) \subseteq set(\mu')$. Recall the definition of μ-contradicted clauses, these are clauses that has all existential literals falsified by μ while no universal literals are satisfied by μ. In other words, there might be universal literals in the μ-contradicted clause but not in the assignment μ. If the negation of such universal literals become unit in μ', such clause can be satisfied, and can no longer be used for future resolutions. One possible solution to this situation is we postpone new unit propagation. In other words, during expansion, we associate each node with its forced assignment sequence. For any selection procedure, we only assign the branching literals and its forced assignment sequence while ignoring the potential additional unit propagation caused by newly learned clauses and cubes. We should be aware that this approach is exactly the same as the second "natural" solution we discussed at the start of section 3.2.1. Let us use the search tree described in figure 5, as an example. During the initial selection procedure on the $\neg x_1$ to x_3 path, $\neg x_9$ and x_{10} have been propagated as unit literals. During a later visit to x_3, we detect an additional existential unit literal x_7, and an additional universal unit literal y_1. Instead of propagating these two literals immediately, we wait until we have reached the most proving node x_3. After x_3 has been reached, the proof number search procedure would expand the most proving node, and we check if y_1 and x_7 are still unit. If yes, we would propagate these additional unit literals during the expansion phase of the algorithm.

However, it is widely believe that postponing unit propagation can significantly weaken the performance of a QBF solver [7]. Furthermore, if we implement such an approach, we are not using Schloeter's method anymore, because the key benefit of Schloeter's method is to ensure no unit propagation needs to be postponed. In summary, because of the difficulty of combining solution driven cube learning with proof number search without postponing some unit propagation, solution driven cube learning is not featured in our current proof number search based QBF solver.

3.3 Complications of pure literals

We have emphasised at the start of section 3 that in this section, the simplification procedure would turn off pure literal elimination. However, this is an undesired behaviour in the context of QBF [16]. Note that in backjumping, such deactivation is unnecessary. This is because in backjumping, the search

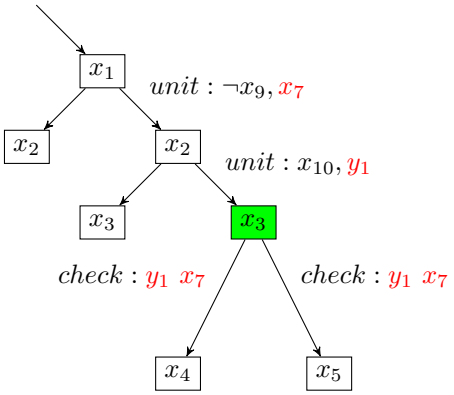

Figure 5: Deal with additional unit propagation in SDCL

space is not changed and no additional clauses are added to the formula during the search procedure. As long as we define an order of unit propagation and pure literal elimination, if we return to the same node in the searching space twice, the partial assigned formula corresponds to the node is guaranteed to be the same. In our implementation, the following order of unit propagation and pure literal elimination is defined.

• If f contains unit literals, we propagate the unit literal with the least lexicographical order.

• If f does not contain unit literal, but contains pure literals, we propagate the pure literal with the least lexicographical order.

However, with clause learning enabled such technique can no longer be applied. The newly learned clauses might transform an original pure literal to be non-pure [7]. This would falsify lemma 3.2, and all the correctness discussion based on this lemma no longer holds. Note that in SAT, the fix to a literal l changed from pure to non-pure is simple, we can simply change $|l|$ from pure to the next branching variable. For example in figure 6, if we detect the literal $\neg x_4$ is no longer pure, we can make the variable $|x_4|$ to be the next branching variable. However, in QBF, since we must assign the variables based on quantifier order, this simple fix does not work. For example in figure 6, assume the formula is f, it is possible that the variable $|x_4|$ is not in the outermost quantifier block of $f_{x_1;x_3}$. As a result, branching on x_4 is invalid, and the method we proposed no longer holds. Consider the fact that unit propagation is much more useful than pure literal elimination to QBF solving, pure literal elimination was not implemented in any clause learning solvers for this thesis project.

Figure 6: Deal with pure literal in SAT

4 Implementation and Experimental Analysis

4.1 Implementation details

To implement backjumping or QCDCL, we need to have access to an efficient data structure that supports:

- Get a variable in the outermost quantifier block
- Assign a variable to be true or false
- Unassign a variable
- Detect a contradictory clause or a satisfied term
- Detect unit-literal
- Add a clause/term to the original formula during backtracking

It has been experimented in both the SAT world [19] and the QBF world [4] that lazy data structures are much more efficient than non-lazy data structures for such tasks. Two-watched literal data structure [4] is a lazy data structure, one benifit of such data structure is during backtracking, the only task we need to do is to mark a variable from assigned to unassigned. As its name suggests, two literals in each clauses are watched. When we assign the literal l, instead of iterating over all clauses that contains the variable $|l|$, we only iterate over the clauses that has $\neg l$ as watched literal (and cubes that has l as watched literal if we are doing QCDCL). This significantly reduces the number of clauses iterated per variable assignment, and therefore increases the efficiency of the solver.

In the thesis project, the well-known two-watched literal data structure was implemented, and for backjumping based solvers, we also implement watched-clause data structure [4] to detect and process pure literals. For branching heuristics, although VSIDS [22] is frequently used by many search-based QBF solvers, for simplicity we only implement a simple prefix level (outermost to innermost) and variable identifier (smallest first) heuristic [1]. For clause (resp. cube) learning strategy, we implement the classic 1-UIP learning schema. However, unlike many solvers that delete previously added clause in later stage of the search [6], we follow the learning schema of the QBF solver Quaffle which is once the clause (resp. cube) is added to the formula, it is never going to be deleted [30]. We have discussed in section 2.3 that in DeepPNS there is a parameter R in the most proving node selection formula that is used to control seesaw effect. In the original DeepPNS paper [8], the R value for a specific

game is normally set as a fixed constant between 0 and 1, however, we should note that the parameter R is not going to affect the correctness of the DeepPNS algorithm. In other words, it is possible to use different R value for different iterations during the DeepPNS solving procedure. For our implementation, the parameter R is updated every 5000 iterations, when the number of iterations is divisible by 5000, we set the parameter R as follows

$$R = \begin{cases} 0 & \text{if } iter \ / \ 5000 \text{ is odd} \\ 0.5 & \text{otherwise} \end{cases} \tag{27}$$

The intuition of dynamically changing the R value is: during $R = 0.0$, the solver can reach more conflict or solution nodes than $R = 0.5$. In contrast, during $R = 0.5$, we force the solver to perform heuristic search, and allow it to search in a different searching space if the formula corresponds to the current searching space is too "difficult" to be solved (e.g. the pn or dn is too large). Since we want the solver to be able to both reaching significant many solved nodes, and not getting trapped on one part of the searching space, we let the DeepPNS solver switch between pure depth first search (i.e. R =0.0) and proof number search (i.e. R =0.5).

4.2 Experimental Results

The aim of the experiment is to compare the performance of depth first search based solver and proof number search based solvers. Since we want to reduce the effect of variable selection heuristics, and clause learning strategies to solver's performance, we implement our own depth first search based QBF solver. This would ensure that the only difference between the depth first search based solvers and the proof number search based solvers is the search strategy. To indicate our depth first search based QBF solver's implementation is efficient, we also list out the performance of three 2004 QBF solvers QUBE-BJ, QUBE-LRN, and semprop on the same set of instances. Here QUBE-BJ [5] implements depth first search based backjumping, QUBE-LRN [6] and semprop [14] implements standard QCDCL. The reason of not using state of art search-based solvers such as DepQBF as a reference solver is because DepQBF contains much more QBF solving techniques than backjumping and QCDCL (e.g. integrating the QBF preprocessor blocker into the solver). Although QCDCL is still the cored QBF solving technique in current state of art search based QBF solvers, it alone is not longer competitive. Since in this thesis, we are only up to a point to be able to combine proof number search with cored QBF solving techniques such as backjumping and QCDCL, it is more feasible to compare with depth first search based solvers that use these QBF techniques alone. Our experiment condition is not ideal, because QUBE-LRN, and semprop contain solution driven cube learning while our solver only contains solution driven backjumping. Thus the main purpose of listing the results of the 2004 solvers is to ensure our depth first search based solvers were implemented efficiently enough such that the

comparison between depth first search based solvers and proof number search based solvers is meaningful.

In this experiment, we compare the performance of the depth first search based backjumping solver (DB), depth first search based CDCL-SBJ solver (DL), DeepPNS based backjumping solver (PB), and DeepPNS based CDCL-SBJ solver (PL) on some QBFEVAL benchmarks. Since pure literal elimination would not affect the correctness of the proof number search based backjumping solver, we also provide the result of depth first search based backjumping solver with pure literal elimination activated (DBP), and DeepPNS based backjumping solver with pure literal elimination activated (PBP). In table 2, we list the number of instances per family solved by these 6 solvers. The instances for the experiment are a subset of the QBFEVAL-2004 instances and 96 corrective encoded gttt4x4 instances [21], the time limit per instance is 900s. As we have discussed just now, we also list the total number of instances solved by three strong 2004 QBF solvers QUBE-BJ (Q*J), QUBE-LRN (Q*N), and semprop (s*p) on the 20 families of QBFEVAL-2004 instances [2].

Family	**#**	**Q*J**	**Q*N**	**s*p**	**DB**	**DBP**	**DL**	**PB**	**PBP**	**PL**
blocks	8	3	5	3	2	3	3	0	1	1
chain	8	6	5	8	8	8	8	7	7	6
counter	8	4	4	4	3	3	4	2	2	2
k_d4_n	8	0	0	1	0	0	0	0	0	0
k_d4_p	8	7	1	7	8	8	8	1	1	8
k_ph_n	8	6	8	6	4	4	6	4	4	6
k_ph_p	8	2	3	2	2	2	2	2	2	2
k_lin_n	8	4	8	8	3	3	5	2	2	4
k_lin_p	8	8	8	2	2	3	8	1	1	3
k_dum_n	8	5	4	8	3	4	6	2	2	5
k_dum_p	8	4	3	8	3	3	7	1	1	7
k_path_n	8	5	3	8	3	3	5	3	3	3
k_path_p	8	3	3	5	3	3	3	3	2	3
k_poly_n	8	2	1	8	1	1	3	1	1	2
k_poly_p	8	8	1	8	8	8	8	8	8	8
k_t4p_n	8	1	0	2	0	0	1	0	0	1
k_t4p_p	8	2	1	2	1	1	2	1	1	2
logn	2	1	2	2	2	1	2	0	0	2
toilet	8	4	6	6	6	6	6	7	7	7
tree	8	8	8	8	6	6	8	6	6	8
Total	154	83	74	106	68	70	95	51	51	80

Table 2: number of instances solved by different methods on a subset of the 2004 QBFEval instances

[2]The results are retrieved from the QBFEVAL website`http://www.qbflib.org/solver_view_domain.php?year=2004&track=1`

Family	#	DB	DBP	DL	PB	PBP	PL
gttt4x4	96	14	41	16	19	39	21

Table 3: number of solved simplified gttt4x4 instances by different methods

Before we compare our depth first sesarch based solvers and proof number search based solvers, it can be seen from Table 2 that the performance of our implementation of the depth first search based CDCL-SBJ solver is between QUBE-LRN and semprop. This indicates that although we deactivate solution driven cube learning, on this specific subset of instances, the loss of performance is not too much. For backjumping based solvers however, our two depth first search based backjumping solvers (DB and DBP) perform worse than QUBE-BJ. While this may because of the use of different branching heuristics, it might also because of the fact that QUBE-BJ was problematic during QBFEVAL 2004 [1]. One evidence is the instance tree-exa2-45 in the family tree should be unsolvable with backjumping solvers [14] but QUBE-BJ solved it within 1 second. Nevertheless, this comparison between our depth first search based solvers and 3 strong 2004 solvers shows that our depth first search based QBF solvers are implemented efficiently, such that the comparison between depth first search based solving and proof number search based solving is meaningful.

If we compare our 3 depth first search based solvers and 3 proof number search based solvers, we can make the following 3 statements based on Table 2 and 3.

• Solvers with clause learning perform significantly better than solvers with backjumping alone regardless of whether the searching strategy is depth first search or proof number search.

• Pure literal elimination can improve the solver's performance in the context of both depth first search (i.e. DB vs DBP) and proof number search (i.e. PB vs PBP).

• Except toilet and gttt4x4, when the proof system is the same (i.e. DB vs PB, DBP vs PBP, and DL vs PL), depth first search based solvers perform not worse than proof number search based solvers on all the other families of instances.

The first statement matches the related research on clause learning. It is widely believed that clause learning is indispensable for search-based solvers, simply executing backjumping without actually caching the resolvent is insufficient for QBF solving [14]. Our work has confirmed that by replacing depth first search with proof number search, the same result still applies. For depth-first search based solvers, the instances solved by DL form a superset of the ones solved by DB, and for proof number search based solvers, the instances solved by PL form a superset of the ones solved by PB except one instance from the CHAIN family, because it is believe that clause learning is not beneficial for this family [7].

Similarly, the second statement matches the related work on pure literal elimination. Although whether pure literal elimination is helpful is disputed in SAT, it is classified as a beneficial simplification technique in QBF [16]. Our work has showed that pure literal elimination is useful to proof number search based backjumping solver. Especially on the family gttt4x4, DBP and PBP even outperform the two clause learning solvers DL and PL that do not contain pure literal elimination.

The third statement however, is the main concern of this thesis. It can be interpreted as replacing depth first search by proof number search in both the context of backjumping and clause learning is **not** beneficial for most family of benchmarks. To provide a better explanation about the third statement, we would present some additional insights into the performance difference between the depth first search based solvers and proof number search based solvers. For simplicity, we only explain the performance difference between depth first search based CDCL-SBJ solver (DL), and proof number search based CDCL-SBJ solver (PL). Firstly, we plot out the number of instances solved by the two solvers under a certain amount of time [3]. As we can see from figure 7 that there are about 80 instances require similar amount of the for DL and PL to solve, however, as the instances get harder, under the same amount of time (e.g. time limit is over 200s), the number of instances that can be solved by DL is more than the number of instances solved by PL. To analyse the reason for the above

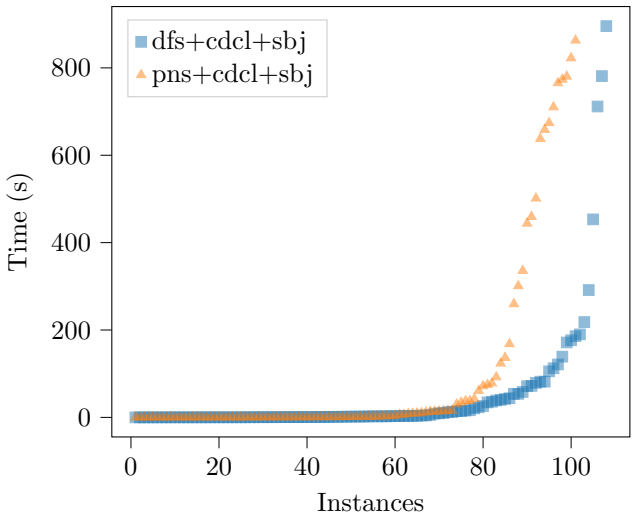

Figure 7: Number of instances solved by DL and PL under a certain amount of time

[3]This is a classic type of plot for QBF solvers, a data point at (x, y) means there are x instances can be solved under y seconds, so in general the solver corresponds to the outer curve performs better

phenomenon, we pick a subset of representative instances from from Table 2, and present the following information:

• The visit ratio of a node of proof number search. This indicates the average number of times a node is visited during the proof number search solving procedure. During depth first search each state is always visited once, so the visit ratio for DL is always 1.

• The total number of terminal states. In the context of depth first search, this is equivalent to the number of times the search reaches a conflict or solution state. In the context of proof number search, we define the number of terminal states as the number of times the search reaches a conflict or solution state during the selection and expansion procedure (i.e. not backpropagation).

• The percentage of expanded nodes that are terminal.

• The run time for both solvers to solve the instance.

• The number of instances that is uniquely solved by PL but not DL on the **family** the instance belongs, and the number of instances that is uniquely solved by DL but not PL on the family the instance belongs. We make the assumption that instances of the same family have some similar structures, so that the statistics of the solver's performance on one instance of that family might be used to analyse the solver's performance on the entire family. The family a instance belongs to can be inferred easily from the prefix of the name of the instance.

An instance is called representative only if there exists some instance of the same family that is solved by one of PL or DL, but not the other.

Instance	**Visit**	DL				PL			
	ratio	Time	Terminal	%	U.	Time	Terminal	%	U.
blocks3ii.4.3	47.96	1.52	969	10.28	2	71.40	2248	3.78	0
chain19v.20	37.89	54.48	524288	50.00	2	506.29	524322	49.06	0
toilet7.1.iv.14	19.73	88.18	67707	17.00	1	3.50	2074	20.39	2
k_path_n-3	14.70	0.89	3917	11.80	2	35.03	47197	7.49	0
k_lin_n-3	20.38	0.90	3050	15.47	1	10.68	6526	10.29	0
k_lin_p-5	18.35	1.11	1566	10.33	5	54.88	5234	4.72	0
k_dum_n-5	7.13	4.43	58781	21.11	1	246.27	412381	13.51	0
k_dum_p-16	8.16	39.86	178380	14.80	1	87.72	111936	9.37	1
gttt_2_2_0010 20_*_tor*_w-0	13.32	41.90	12841	28.41	0	16.37	3770	18.55	5

Table 4: Detail statistics of representative instances including how many instances are uniquely (U.) solved by DL or PL. on the same family

For completeness, although we would not provide the detailed statistics as in Table 4, we will list out the number of instances that is uniquely solved by the depth first search based backjumping solver (DB) but not the proof number search based backjumping solver (PB), as well as the number of instances that is uniquely solved by the solver PB but not DB. Similar comparison result is going to be provided for the solver DBP and PBP.

It can be observed from Table 4 that the visit ratio of proof number search

Family	DB	PB
blocks	2	0
chain	1	0
counter	1	0
k_d4_n	0	0
k_d4_p	7	0
k_ph_n	0	0
k_ph_p	0	0
k_lin_n	1	0
k_lin_p	1	0
k_dum_n	1	0
k_dum_p	2	0
k_path_n	0	0
k_path_p	0	0
k_poly_n	0	0
k_poly_p	0	0
k_t4p_n	0	0
k_t4p_p	0	0
logn	2	0
toilet	1	2
tree	0	0
gttt4x4	1	6

Family	DBP	PBP
blocks	2	0
chain	1	0
counter	1	0
k_d4_n	0	0
k_d4_p	7	0
k_ph_n	0	0
k_ph_p	0	0
k_lin_n	1	0
k_lin_p	2	0
k_dum_n	2	0
k_dum_p	2	0
k_path_n	0	0
k_path_p	1	0
k_poly_n	0	0
k_poly_p	0	0
k_t4p_n	0	0
k_t4p_p	0	0
logn	1	0
toilet	1	2
tree	0	0
gttt4x4	4	2

Table 5: Number of instances that are uniquely solved DB vs PB and DBP vs PBP

ranges from 7.13 to 47.96 on the subset of instances. Since time is normally used as the evaluation metric for QBF solvers, repeatedly visiting the same node is undesired, because given the same amount of time, the solver is able to expand much fewer different states compared to the depth first search based solver. For example the family CHAIN, a family that both backjumping and conflict-driven clause learning have no benefits. Interestingly, the number of node expansions for both depth first search based solvers and proof number search based solver is approximately 2^x if the instance has x universal variables. As we can see from the table the visit ratio for DL on CHAIN19v.20 is 37.89. Since both solvers requires to expand very similar number of nodes for solving this instance family, a high visit ratio would have a significant drawback for the solver.

Besides the visit ratio, the percentage of terminal nodes is also a factor that affects the solver's performance. Since the reasoning techniques we implemented are backjumping and clause learning, during the search, we need significant amount of conflicts and solutions as the initial reasons for the Q-resolution. For the family blocks, a family that was used to demonstrate the effect of backjumping [5], DL is able to solve 2 instances that PL cannot solve. As we can see from Table 4, for the block3ii.4.3 instance that both solvers can solve, the percentage of terminal nodes reached by DL is 10.28%, while for PL this value is only 3.78%. Similarly, for the instance k_lin_p-5, the percentage of terminal nodes reached by DL is 10.33%, while for PL, this value is only 4.72%. In contrast, for the family toilet, a family that proof number search based solver is responsible for solving more instances than depth first search based solver,

the percentage of terminal nodes reached by PL (20.39%) is even higher than DL (17.00%).

To understand why the percentage of terminal nodes for proof number search based solver is lower than depth first search based solver, we plot the change of the difference between the proof number and disproof number at the root of the search tree with respect to the number of iterations for the instance blocks3ii.4.3. At the same time, we plot the total number of terminal nodes generated with respect to the number of iterations on the same figure. As we

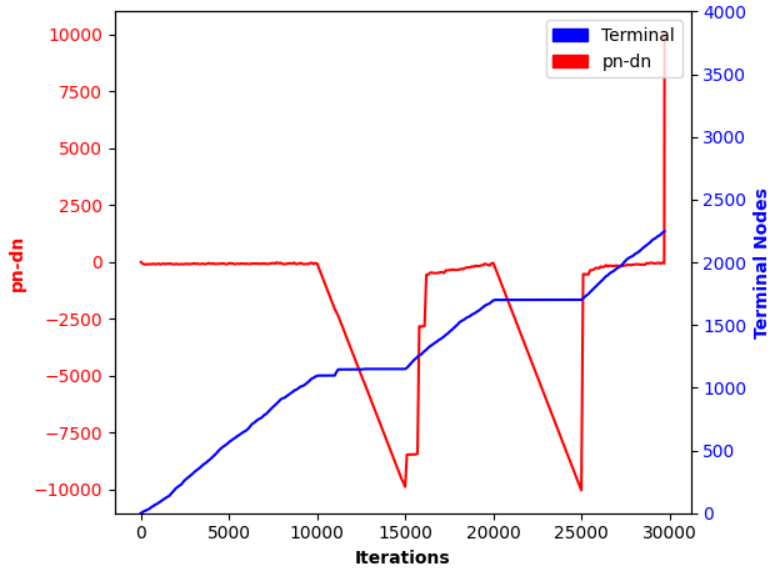

Figure 8: Change of pn-dn and total terminal nodes with respect to the number of iterations on instance blocks3ii.4.3

can see from figure 8, it takes PL approximately 30,000 iterations to solve the instance. Since the instance has truth value UNSAT, the expected difference between proof number and disproof number at the end of the solving procedure should be $+\infty$. However, we can easily observe that on this particular instance, the difference between pn and dn does not become positive until the very last moment. What makes things even worse is between iteration 11,000 to 15,000, and 20,000 to 25,000, the solver reached 0 terminal nodes. In other words, the search algorithm generates 10,000 new nodes between these iterations but none of these new nodes are terminal. This illustrates that the solver is severely affected by seesaw effect on this particular family of instances even if we applied DeepPNS which is designed to reduce seesaw effect. Note that the seesaw effect is not unique to the blocks family, our experiments have shown that the proof number search based solver also suffers seesaw effect on the family counter, and

logn. However, for instances from the family logn, because clause learning is too effective, the number of instances solved by the proof number search based solver is not going to decrease. If we plot the same statistics according to the

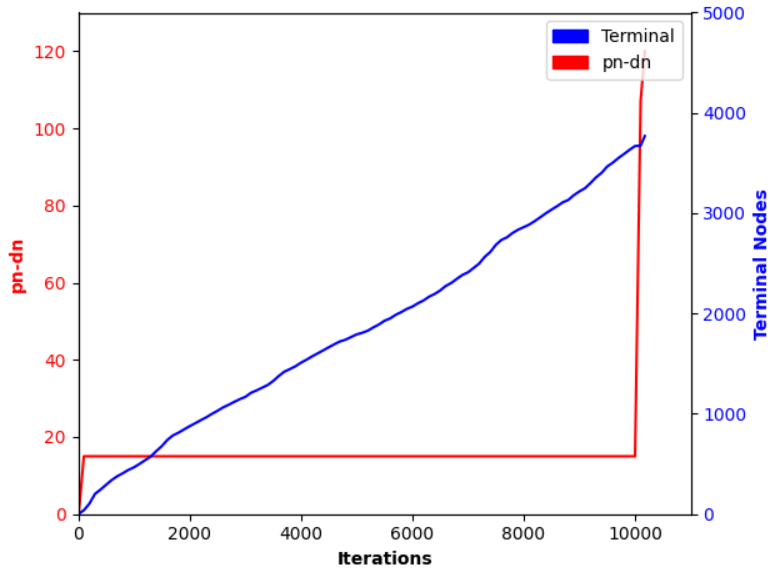

Figure 9: Change of pn-dn and total terminal nodes with respect to the number of iterations on instance gttt_2_2_001020_4x4_torus_w-0

instance gttt_2_2_001020_4x4_torus_w-0, an instance with truth value UNSAT, we can see from figure9 that on this instance the difference between the proof number and the disproof number is always positive, and the number of terminal nodes increased consistently. On this particular instance the proof number and disproof number is informative, and this also the reason why PL can solve 5 more instances than DL on gttt4x4.

One of the main reason that makes the proof number and disproof number uninformative on some families is because of the ratio between the existential variables and universal variables is extremely imbalanced. Although QBF can be interpreted as a two-player and-or strategy game, it is different from many two-player board games. Since most QBF has a prefix of only a few quantifier blocks, it is usually the case that one of the player (either existential or universal) takes many consecutive moves instead of both players take moves in turn during the and-or strategy game. For example, in the family blocks, the ratio between existential variables and universal variables can be up to 100:1, and the existential player normally takes more than 30 consecutive moves on such family of instance before the universal player takes a move. In contrast, for instances from the family gttt4x4, the ratio between existential variables and universal

variables is only about 15:1, and the consecutive moves made by the existential player is approximately 10:1. Because of the fact that we initialize the proof and disproof number of newly created nodes based on the default mobility initialization heuristic, which is actually designed for the standard and-or strategy games, it can be expected that for games that one player takes many consecutive moves, such initialization heuristic would not perform well. Furthermore, mobility initialization is irrelevant to the actual complexity of the formula unless the node is solved, we can expect that such heuristic is not ideal for QBF. Note that since we are using mobility initialization, it is very important to introduce the technique of dynamically changing the parameter R in DeepPNS during the solving procedure as shown in equation (27). Because for families that have extremely imbalance existential variables and universal variables such as blocks or logn, if we use the mobility initialization heuristics, unless R is extremely close to 0 (i.e. the search is almost pure depth first search), it is very unlikely for the search to reach terminal nodes within a feasible number of iterations. For algorithms like backjumping or clause-learning, such phenomenon is undesired. The detail discussion on why in our current version of the solver, we do not use a fixed parameter R in the DeepPNS algorithm is quite long, and we will put it in Appendix A.

Nevertheless, if we summarize table 2, 4, and 5 we can see that out of the 26 instances that are solved by only one of DL and PL, PL is responsible for 8 (31%); out of the 28 instances that are solved by only one of DB and PB, PB is responsible for 8 (28%); out of the 29 instances that are solved by only one of DBP and PBP, PBP is responsible for 4 (14%). This shows that although under our current initialization heuristics, proof number search based solvers cannot outperform depth first search based solvers using the same proof system on most instances, it can solve instances that depth first search are unable to solve because of dfs is trapped on the wrong side of the searching space.

5 Related Work

One pieces of work that is heavily related to this thesis is Schloeter's MCTS based CDCL SAT solver [25]. In this thesis, we have proved the idea in that paper can be generalized to the context of QBF and implemented a proof number search based QBF solver that features conflict driven clause learning and solution driven backjumping. Despite Schloeter's method was originally proposed in SAT, one of the main difference between Schloeter's method and our work is in Schloeter's method, none of the reasons are stored in the search tree nodes. Instead the reason for unsatisfiability is added to the formula immediately when we backtrack to a node and find the reason contains the branching variable (this is literally the 1-UIP learning schema). Since the selection procedure is going to maintain the search tree consistent with the formula with the learned clauses, the behaviour of Schloeter's method and our method is going to be essentially the same if all reasons for unsatisfiability at 1-UIP are learned. However, if one of such reasons at 1-UIP is not added to the formula conjunctively (it might

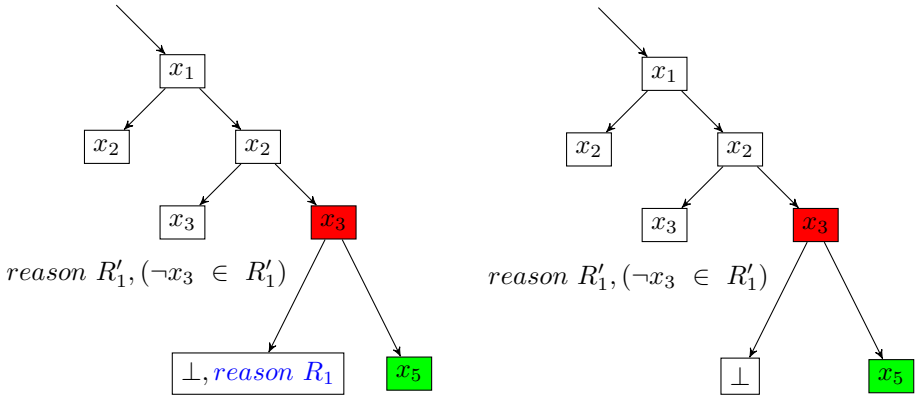

Figure 10: Similarity and Difference between our approach (left) and Schloeter's method (right)

because the length of the reason is too long), for our method, since we store the reason in a tree node (let us denote the node as N), the reason can still be used for Q-resolution on the current root to N path. In comparison, for Schloeter's method, if one reason at 1-UIP is discarded, when we require the reason for unsatisfiability of a node in a future iteration, the reason corresponds to the node is unknown. As a result, many Q-resolutions on the root to N path can no longer take place. In addition, since we have implemented solution driven backjumping instead of cube learning, no reason for satisfiability is added to the formula disjunctively. In the context of backjumping, the backjumping is not going to be enforced implicitly unless the reason is stored in the tree nodes. Do not store the reason in tree node would significantly weakened the effect of backjumping.

We would provide an example to illustrate the similarity and difference between Schloeter's method and our approach that we have discussed just now. As we can see from figure 10, suppose the left child of the red node is disproved during iteration i, the reason for unsatisfibility is R_1, and the partial assignment made from root to the red node is μ. Assume the reason R'_1 for $f_{\mu;x_3}$ contains $\neg x_3$. Then, we know the backjumping is blocked, and the right child of the red node cannot be pruned. Since the reason is a $(\mu; x_3)$-contradicted clause, it is not hard to show that after we unassign x_3, $\neg x_3$ is unit with respect to R'_1. Let us consider two possible cases, 1) the $(\mu; x_3)$-contradicted clause R'_1 is added to the originally formula conjunctively immediately, and 2) the $(\mu; x_3)$-contradicted clause R'_1 is not added to the original formula. Assume in both cases, the variable $|x_3|$ has not been assigned on the path from root to the red node, the formula f has been extended to f' with new clauses, and the partial assignment made from root to the red node is μ'. For the first case, both our method and the original Schloeter's method, when the red node is visited during iteration j (j > i), because of lemma 3.2, we know that $\neg x_3$ must become unit.

Thus, according to Schloeter's method. The parent of the red node is linked up with its right child and we continue the search from its right child. In this case, our method and Schloeter's approach would behave essentially the same. However, if case two happens, for example we think the $(\mu; x_3)$-contradicted clause R'_1 is too long. In our approach, during the selection phase of the jth iteration, we would know the left child of the red node is solved and continue the selection from the right child of the red node. For Schloeter's method, because it knows the left child of the red node is solved, it would continue the search from the right child as well. However, the two methods would be different during backpropagation. Assume the green node has been disproved, and reason for $f'_{\mu';\neg x_3}$ unsatisfiability contains x_3. This means the reason associated with the red node can only be calculated with the resolution of the reason for $f'_{\mu';x3}$ unsatisfiability and the reason for $f'_{\mu';\neg x_3}$ unsatisfiability. We have discussed in section 3.2.7, that the reason for $f'_{\mu';x_3}$ can be derived from the reason R_1. Thus for our method, the reason associated with the red node can be derived and used for future Q-resolutions. In the original Schloeter's method however, because the reasons are never explicitly stored in the tree nodes, the reason assicated with the red node cannot be derived by Q-resolution. If the reason associated with the red node cannot be derived, the reason associated with some ancestors of the red node cannot be derived as well. And this could reduce the chance of backjumping pruning, and weaken the solver's performance. Although in the actual Schloeter's implementation, every reason at 1-UIP is learned, this discussion shows that the performance of our approach (store reason associated with a node in the tree node) is less likely to be affected by not learning a clause at 1-UIP. Therefore, although we haven't done any experiments on this, our approach is going to work descent under other clause learning schema (e.g. size learning of order n and relevance learning of order n [6]) as well.

6 Conclusion and Future work

Using heuristic-search algorithms to solve QBF is not a complete new idea, for example, researchers who are inspired by the success of AlphaZero have attempted to use the neural MCTS algorithms to solve QBF [27]. However, this thesis is the first attempt on solving QBF using a heuristic-search algorithm that combines classic reasoning techniques such as backjumping and clause learning. We have proved that a technique proposed by Schloeter that combines MCTS and CDCL in the SAT world can be generalized to the context of QBF when the solver contains conflict driven clause learning and solution driven backjumping. We have also discussed the obstacles of enabling either solution driven cube learning or pure literal elimination for a proof number search based QBF solver. During the thesis project, we have implemented 3 types of proof number search based QBF solvers that include proof number search based backjumping solver, proof number search based backjumping solver with pure literal elimination enabled, and proof number search based conflict driven clause learning with

solution driven backjumping solver [4]. Our experimental results showed that in the context of proof number search, the learned clauses can significantly improve the solvers performance on the 2004 QBFEVAL benchmarks. However, because of the nature of the proof number search algorithm that each search state might be visited significantly more than once, and our current node initialization heuristic is not ideal, in both the context of backjumping and clause learning, none of the proof number search based solvers can outperform the traditional QDLL based solvers that uses the same proof system except a few families of instances.

The following three tasks should be addressed in the future if possible. Firstly, it is tempting to investigate if pure literal elimination can be activated for the proof number search based clause learning solver. Secondly, it is undesired to replace solution-driven cube learning with solution-driven backjumping especially based on the fact that the importance of cube learning has exponentially increased to search-based QBF solvers after the dynamic QBCE algorithm was proposed [15]. We should investigate how solution-driven cube learning can be enabled in the context of proof number search not necessarily using a modification of Schloeter's method. Thirdly, we have explained in section 4.2 that our current proof number search initialization heuristics is ineffective for QBF because unless the newly created node is solved, the proof and disproof number is irrelevant to the structure of the partial assigned formula corresponds to the node. Similar to Schloeter who defined a better initialization heuristics for MCTS in the context of SAT three years after he proposed the MCTS based CDCL SAT solver [11], it is desired to define a heuristics for the "game" QBF in the future.

References

[1] 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.

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

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

[4] Ian Gent, Enrico Giunchiglia, Massimo Narizzano, Andrew Rowley, and Armando Tacchella. Watched data structures for qbf solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 25–36. Springer, 2003.

[4] The thesis project source code is available at `https://github.com/hharryyf/Thesis`.

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

[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. On q-resolution and cdcl qbf solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 402–418. Springer, 2016.

[10] 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.

[11] Oliver Keszocze, Kenneth Schmitz, Jens Schloeter, and Rolf Drechsler. Improving sat solving using monte carlo tree search-based clause learning. In *Advanced Boolean Techniques*, pages 107–133. Springer, 2020.

[12] 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.

[13] 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.

[14] 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.

[15] Florian Lonsing, Fahiem Bacchus, Armin Biere, Uwe Egly, and Martina Seidl. Enhancing search-based qbf solving by dynamic blocked clause elimination. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 418–433. Springer, 2015.

[16] Florian Lonsing and Armin Biere. Depqbf: A dependency-aware qbf solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):71–76, 2010.

[17] 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.

[18] Florian Lonsing, Uwe Egly, and Allen Van Gelder. Efficient clause learning for quantified boolean formulas via qbf pseudo unit propagation. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 100–115. Springer, 2013.

[19] Inês Lynce and Joao Marques-Silva. Efficient data structures for backtrack search sat solvers. *Annals of Mathematics and Artificial Intelligence*, 43(1):137–152, 2005.

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

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

[22] 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.

[23] Jakub Pawlewicz and Łukasz Lew. Improving depth-first pn-search: 1+ ε trick. In *International Conference on Computers and Games*, pages 160–171. Springer, 2006.

[24] 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.

[25] Jens Schlöter. A monte carlo tree search based conflict-driven clause learning sat solver. In *GI-Jahrestagung*, 2017.

[26] Masahiro Seo, Hiroyuki Iida, and Jos WHM Uiterwijk. The pn$\check{2}$217-search algorithm: Application to tsume-shogi. *Artificial Intelligence*, 129(1-2):253–277, 2001.

[27] Ruiyang Xu and Karl Lieberherr. Solving qsat problems with neural mcts. *arXiv preprint arXiv:2101.06619*, 2021.

[28] Lintao Zhang. Solving qbf with combined conjunctive and disjunctive normal form. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 21, page 143. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[29] 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.

[30] Lintao Zhang and Sharad Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In *International Conference on Principles and Practice of Constraint Programming*, pages 200–215. Springer, 2002.

A Dynamic R value in DeepPNS

In QBF, quantifiers are grouped in blocks. Some families contain only 2 or 3 blocks of quantifiers, while a single existential block might contain more than 30 quantified variables. We mainly focus on a single instance $block3ii.4.3$ for this section. This instance is from the blocks family, which proof number search with mobility initialization does not perform well. The instance contains 3 quantifier blocks, the first existential block contains 54 variables, while the second universal block contains only 4 variables, and the final existential block contains 189 variables. We only focus on the proof number search based CDCL-SBJ solver. We plot the change of the difference between the proof number and disproof number, and also the number of terminal nodes reached with respect to the total number of iterations. Note that for this experiment, we only plot the statistics for the first 100,000 iterations. Instead of letting the R value to change according to equation (27), we consider the R value is fixed during the solving procedure. We would consider three fixed R values, 0.1, 0.5, and 1.0. We can see from figure 11 that for all three settings, the difference between the proof and disproof number is always negative, and the number of terminal states reached by the search is very little. Note that by using the R value setting in equation (27), we can reach more than 2000 terminal nodes within 30,000 iterations. In comparison, by using a fixed R, even if R is as small as 0.1, the total number of terminal nodes can be reached within 100,000 iterations is only 57. The reason for this is mainly due to the mobility initialization. Recall that in mobility initialization, if a newly expanded node is undetermined, for an existential (resp. universal) node, we set its proof (resp. disproof) number to be 1 and disproof (resp. proof) number to be 2. However, for the instance $block3ii.4.3$, since the first existential block is large, it is often the case that the existential "player" takes more than 30 consecutive moves before the universal "player" takes a move. In other words, a path in the search tree normally starts with more than 30 existential nodes before the first universal nodes. What makes the situation even worse for this instance is: a terminal state can rarely be reached with only existential assignments. For simplicity, let us only consider how the search tree would change in the case of pure proof number search (i.e. $R = 1.0$). Consider an example instance with the following properties:

- The first quantifier block contains more than 30 existential variables.
- No universal variable assignment can happen before we assign at least 30 existential branching variables in the first quantifier block.
- No terminal node can be reached by without assigning a universal variable in the first quantifier block.

Since the proof number of an undetermined universal node is initialized as 2, and the proof number of an undetermined existential node is initialized as 1. Note that by the property of the proof number search algorithm, a universal leaf node that has a proof number 2 (according to equation (5) and (6), its dpn value is 0.5) is never going to be selected as the most proving node as long as the search tree contains at least one root to leaf (N) path such that all the nodes on the path are existential and the leaf N is undetermined. This is because under

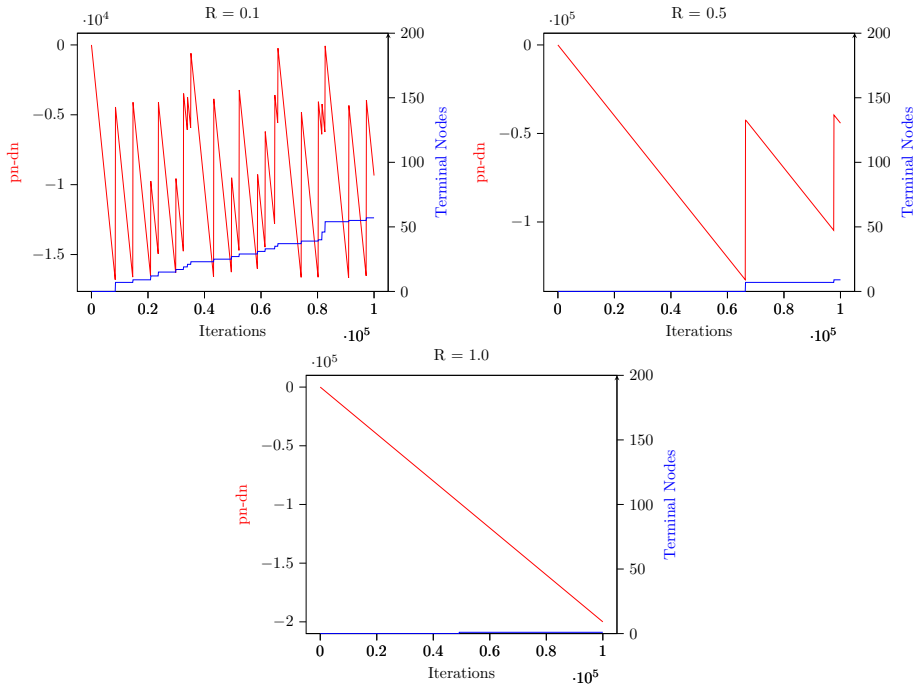

Figure 11: Change of pn-dn and total terminal nodes with respect to the number of iterations on instance blocks3ii.4.3 for different R value in DeepPNS

mobility initialization, all undetermined existential leaf are initialized with a proof number 1. If there exists a root to leaf path such that all nodes on the path are existential, then by the equation (7) for backpropagation it is not hard to show that all the nodes on this path have their proof number equal to 1 (according to equation (5) and (6), all nodes on this path have a dpn value 0). By the DeepPNS selection formula that the most proving node is in the subtree of a child with the smallest dpn value, it is obvious that an existential leaf is going to be the most proving node instead of some universal leaf in the search tree. For the example instance, in order to reach a terminal node, we must assign at least one universal variable. Consider the first time a universal variable u is assigned. Then, it is not hard to show that the node contains u must become the most proving node. However, when u becomes the most proving node, there must not exist a root to leaf path such that all nodes are existential. By the property of this example instance, it is not hard to show that all leaves in the search tree must be undetermined universal nodes or terminal nodes. Since no universal variable assignment can happen before we assign at least 30 existential branching variables, and no terminal node can be reached without assigning a universal variable, at the time u becomes the most proving node, the tree must contain at least as many nodes as a full binary tree with 31 levels. Also by the third property that no terminal nodes can be reached without

assigning a universal variable. Before u is considered as the most proving node, no terminal node can be reached. This means that no terminal node can be reached without creating around 2^{32} nodes. Since the algorithms we studied for this thesis are backjumping and QCDCL, both require a significant amount of terminal node in order to calculate the reason for internal nodes, we can expect that proof number search with mobility initialization would work poorly on this imbalanced instance.

Although the above example is just an extreme case, its property is quite similar to the instance *block*3*ii*.4.3. The analysis above also explains why when $R = 1.0$, on the instance *block*3*ii*.4.3, only 1 terminal node is reached after 100,000 iterations. One must note that using default initialization (i.e. both pn and dn are set to 1 for undetermined new nodes) instead of mobility initialization cannot solve the above issue although some universal nodes can become the most proving node earlier, it is not too difficult to show that when the ratio between the existential variables and universal variables is very imbalance, and terminal nodes only exist at a very deep level, both initialization heuristics would run into similar issues. In addition, it is not too hard to show that even if we use DeepPNS with $R \neq 1$, as long as we use the mobility initialization and a fix R value, the above phenomenon cannot be avoided unless R is extremely small. However, for the families that proof number search is beneficial (e.g. gttt4x4), a very small R value would remove the effect of proof number search, and the chance of getting trapped on the wrong side of the searching space would increase. This is the reason why we let the R value change between 0 and 0.5 such that the algorithm can both generate sufficient number of terminal nodes, and not get trapped on one side of the searching space. Although a more ideal solution is to design a more informative proof number search initialization heuristic that can enable the algorithm to deal with 2-player games that one player can take many consecutive moves, because the duration of the thesis is only one year, we use the technique of switching the R value between 0 and 0.5 as an expediency.

B Original QDLL algorithm

Algorithm 15: Original QDLL algorithm

```
1  Function QDLL(f, μ)
2      if f_μ has a contradictory clause then
3          return False
4      if all clauses in f_μ are satisfied then
5          return True
6      if f_μ has no universal variables then
7          return SAT(f_μ)
8      G := the formula obtained by removing all universal literals
        from f_μ
9      if SAT(G) then
10         return True
11     if l is unit in f_μ then
12         return QDLL(f, μ; l)
13     if l is pure in f_μ then
14         return QDLL(f, μ; l)
15     l = get_literal(f, μ)
16     if l is existential in f then
17         return QDLL(f, μ; l) or QDLL(f, μ; ¬l)
18     return QDLL(f, μ; l) and QDLL(f, μ; ¬l)
```

C The pseudocode for resolve

Algorithm 16: The method resolve [6]

Input : The QBF expression f and a partial assignment μ, reason for $f_{\mu;l}$ and reason for $f_{\mu;\neg l}$, literal l. Note that the truth of $f_{\mu;l}$ and $f_{\mu;\neg l}$ must be the same, both R and R' must contain the variable $|l|$

Output: The resolvent of R and R' (i.e. the reason for satisfiability or unsatisfiability of f_μ)

```
1  Function resolve(f_μ, R, R', l)
2      if ∃ l', l' ≠ l such that l ∈ R and ¬l in R' then
3          l' := a literal in R such has the lowest decision level
4          R" := the clause or cube l' is unit with respect to
5          R := Q-resolution(R, R", l')
6          return resolve(f_μ, R, R', l)
7      return Q-resolution(R, R', l)
```