JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Clemens Hofstadler

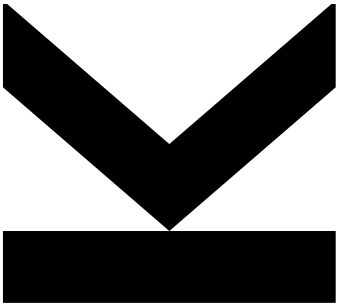
Submitted at
Institute for Symbolic Artificial Intelligence

Supervisor
Univ.-Prof.in Dr.in Martina Seidl

Co-Supervisor
DI Maximilian Heisinger

February 2022

Solving QBFs with AlphaZero and MCTS



Master Thesis
to obtain the academic degree of
Master of Science
in the Master's Program
Artificial Intelligence

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

The extension of propositional logic with quantifiers leads to the logic of *quantified boolean formulas (QBF)*. From a theoretical point of view, the decision problem of QBF, called *QSAT*, is PSPACE-complete. Nevertheless, QBFs have been used successfully to encode and solve problems coming from several different fields. To be able to solve larger and more complex problem instances, a lot of effort was put into the development of efficient solvers. A recent trend is to improve QBF solvers by (deep) machine learning. One such approach, which was presented by Xu and Lieberherr, uses the famous AlphaZero framework to solve QSAT problems.

AlphaZero is an algorithm that combines symbolic reasoning techniques in form of a *Monte Carlo tree search (MCTS)* with novel deep reinforcement learning techniques, called *self-play reinforcement learning*, to learn fully-observable, symmetric 2-player games. It has proven to be powerful by beating the best human players and computer programs in the games of chess, shogi and Go.

The key to using AlphaZero for QSAT is to consider QBF solving as a 2-player game where one player tries to make the formula true by assigning variables to truth values while the other player in the same way tries to make the formula false. The main idea is then to use the AlphaZero framework to learn two (perfect) players for this game and to use these (perfect) players to determine the truth value of QBFs.

In this thesis, we reproduce and analyze the results communicated by Xu and Lieberherr. In particular, we train two AlphaZero players for the game of QBF solving and compare them to several different variants of a standard MCTS and to a state-of-the-art QBF solver. We describe how to adapt the standard AlphaZero algorithm to the task of QBF solving. Finally, we also discuss how MCTS can be used as a preprocessing tool before applying a standard QBF solver.

Zusammenfassung

Die Erweiterung der Aussagenlogik mit Quantoren führt zur Logik der *quantifizierten booleschen Formeln (QBF)*. Aus theoretischer Sicht ist das Entscheidungsproblem für QBF, auch *QSAT* genannt, ein PSPACE-vollständiges Problem. Nichtsdestotrotz konnten QBFs erfolgreich verwendet werden um Probleme aus verschiedenen Bereichen zu kodieren und zu lösen. Um immer größere und komplexere Probleme lösen zu können, wurde in den letzten Jahren viel Aufwand betrieben, um die Entwicklung effizienter Lösungsverfahren voranzutreiben. Ein neuer Trend ist hierbei die Verfahren mit Hilfe von tiefem maschinellen Lernen zu verbessern. Vor Kurzem wurde von Xu und Lieberherr solch ein Ansatz vorgestellt, welcher das berühmte AlphaZero Modell nutzt um QSAT-Probleme zu lösen.

AlphaZero ist ein Algorithmus, der symbolische Schlusstechniken in Form einer *Monte Carlo Baumsuche (MCTS)* mit neuen Techniken des bestärkenden Lernens, auch *Selbst-Spiel-Lernen* genannt, verbindet, um vollständig beobachtbare, symmetrische 2-Spieler Spiele zu lernen. AlphaZero konnte seine Stärke bereits beweisen, indem es die besten menschlichen Spieler sowie Computerprogramme in den Spielen Schach, Shogi und Go schlug.

Der Schlüssel um AlphaZero für QSAT verwenden zu können, ist das Lösen von QBFs als 2-Spieler Spiel zu betrachten. In diesem Spiel versucht ein Spieler die Formel wahr zu machen, indem er den Variablen Wahrheitswerte zuordnet, während der zweite Spieler auf die gleiche Weise versucht die Formel falsch zu machen. Die Idee ist dann den AlphaZero Algorithmus zu verwenden um zwei (perfekte) Spieler für dieses Spiel zu trainieren und diese (perfekten) Spieler im Folgenden zu nutzen, um den Wahrheitswert von QBFs zu bestimmen.

In dieser Arbeit werden die Ergebnisse, die von Xu und Lieberherr kommuniziert wurden, reproduziert und analysiert. Genauer gesagt werden zwei AlphaZero Spieler für das QSAT-Spiel trainiert und dann mit mehreren Varianten einer klassischen MCTS sowie mit moderner QBF-Lösungssoftware verglichen. Bevor diese Experimente beschrieben werden, werden die Änderungen beschrieben, die nötig sind, um AlphaZero an die Aufgabe des QBF-Lösens anzupassen. Abschließend wird auch eine Methode vorgestellt, wie MCTS als Vorbearbeitungsschritt genutzt werden kann, bevor ein klassisches QBF-Lösungsverfahren angewandt wird.

Acknowledgements

First and foremost, I have to express my deepest gratitude to my supervisor Univ.-Prof.[in] Dr.[in] Martina Seidl who not only sparked my interest in the topic of symbolic artificial intelligence and in particular in QBF solving but who also continuously supported and guided me with all her knowledge on this subject. At the same time, she always granted me full independence and freedom to develop and pursue my own ideas, for which I am very thankful. I am also grateful for the numerous helpful discussions we had and for all the time she spent proofreading this thesis.

Additionally, I have to thank my co-supervisor DI Maximilian Heisinger for his valuable remarks and his careful reading of this work. I am very thankful for his indispensable support, especially for introducing me to the servers of the institute. I also thank the Institute for Symbolic Artificial Intelligence at JKU for allowing me to use their servers.

Finally, I want to express my sincerest thanks to my parents for supporting me throughout my whole life and to my girlfriend and my friends for their understanding and unconditional support over the last years.

Clemens Hofstadler
Linz, February 2022

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.
Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

.....	.....
Ort, Datum	Clemens Hofstadler

Contents

Chapter 1

Introduction

In the last 50 years, the formalism of propositional logic has sparked more and more (theoretical) interest. The *satisfiability problem of propositional logic (SAT)* is the problem of deciding for a given formula in propositional logic whether there exists an assignment of variables to the truth values *true* and *false* such that the formula evaluates to true. This problem was the first one to be proven NP-complete [Coo71], showing that there does not exist a polynomial-time algorithm to solve SAT (assuming that P $\neq$ NP). Despite the theoretical exponential complexity of SAT, impressive progress in this field has enabled modern SAT solvers to nevertheless efficiently tackle many problems coming from real-world applications.

Problems whose complexity is beyond NP require more powerful formalisms that play a similar role as SAT plays for solving problems in NP. One such formalism is the logic of *quantified boolean formulas (QBF)*, which can be considered as an extension of propositional logic with existential ($\exists$) and universal ($\forall$) quantifiers. While the addition of these quantifiers does not add expressiveness, it allows for more compact encodings than SAT. The decision problem of QBF, often called *QSAT*, is the prototypical example of a PSPACE-complete problem [MS73]. This complexity class contains many interesting reasoning problems of practical interest, which can all be encoded in terms of QBF. For example, QBF is used successfully to solve problems coming from formal verification, planning, and function synthesis. We refer to the recent survey [SBPS19] for further information on applications of QBF.

To be able to solve larger and more complex problem instances, a lot of effort was put into the development of efficient QBF solvers in the last years with annual competitions to objectively evaluate the state of the art (see e.g. [PS19] for competition reports of 2016 and 2017). Recently, many attempts were made to improve the process of QBF (and SAT) solving by machine learning. For example, the QBF solver QFUN [Jan18] is based on counterexample guided abstraction refinement extended with machine learning. QFUN progressively solves a QBF by learning "good" assignments of the variables using a decision tree classifier. Alternatively, the SAT solver NeuroSAT [SLB$^+$18] applies graph neural networks [LTBZ15] and single-bit supervised learning to solve SAT problems. In this solver, boolean formulas are encoded as graphs which form the input of a specially designed graph neural network. This network then outputs a single bit representing the truth value of the input SAT problem.

Recently, also the algorithm applied by the famous AlphaZero framework [SHS+18] was used to solve QSAT instances [XL21].

AlphaZero is an algorithm that has proven to be powerful by beating the best human players and computer programs in the the games of chess, shogi and Go. It combines a classical *Monte Carlo tree search (MCTS)* (see [BPW+12] for a recent survey) with newly developed deep reinforcement learning techniques, called *self-play reinforcement learning* [SSS+17]. AlphaZero is the successor of AlphaGo [SHM+16], the first computer program to beat a human professional in the game of Go. In contrast to its predecessor, which was developed purely for the game of Go, AlphaZero is a more general framework able to learn any fully-observable, symmetric 2-player game. One of the remarkable aspects of AlphaZero is that it does this without any human knowledge except the rules of the game (*tabula rasa learning*). It uses a MCTS guided by a neural network for move selection and learns, starting as a completely random player, by playing against itself (*self-play*).

In [XL21], QBF solving is considered as a 2-player game where one player tries to make the formula true by assigning variables to truth values while the other player in the same way tries to make the formula false. The main idea is then to use the AlphaZero framework (in [XL21] also called *neural MCTS*) to learn two (perfect) players for this game and to use these (perfect) players to determine the truth value of QBFs. It has to be noted that, in contrast to standard QBF solvers, the results produced by this approach are just predictions and not provably correct. Nevertheless, the authors report that their approach works well empirically, at least for problems within a limited size.

In this thesis, we reproduce and analyze the results communicated in [XL21]. In particular, we start from an open source implementation[1] of AlphaZero and adapt it so that it can be used for QBF solving. Then we use our AlphaZero adaptation to train two players on the game of QBF solving using randomly generated QBFs as training data. We evaluate our players on an independent test set and compare it to the state-of-the-art QBF solver DepQBF [LB10]. Additionally, we compare this AlphaZero approach to several different variants of a standard MCTS. Finally, we also show how MCTS can be used a preprocessing tool before applying a standard QBF solver. Our implementations together with all files needed to reproduce our experiments are available at

`https://github.com/ClemensHofstadler/MCTS4QBF`.

The remainder of this work is structured as follows. To keep this thesis self-contained, we first recall the the syntax and semantics of QBF as an extension of propositional logic in Chapter 2. Furthermore, we also give a brief summary of the most common solving approaches for QSAT. Following upon that, we discuss the AlphaZero framework in Chapter 3. In particular, we give a brief historic overview of the development of AlphaZero and on how it fits into this field of artificial intelligence that deals with teaching computers to play (board) games on a superhuman level. Then, after presenting the classical AlphaZero framework as introduced in [SHS+18], we discuss the relevant changes needed to adapt the algorithm for our problem of QSAT in Chapter 4. Finally, we present our experimental results. Based on the insights gained from these experiments, we describe a slightly different approach to

[1] available at `https://github.com/suragnair/alpha-zero-general`

QSAT in Chapter 5. Here, we use MCTS as a preprocessing tool to simplify QBFs before giving them as input to a standard QBF solver. We report on some experiments we conducted and on possible future work.

Chapter 2

Quantified Boolean Formulas

In this chapter, we recall the syntax and semantics of quantified boolean formulas (QBF). This is done in Section 2.1 and Section 2.2. Note that, as we introduce QBF as an extension of propositional logic, we first define the syntax of the latter. For a more extensive introduction of propositional logic and QBF, we refer to [BL99] and [BB09] respectively, which were also our main references for this chapter. Additionally, in Section 2.3, we present the two main approaches to QSAT that modern solvers implement, namely backtracking search and variable expansion. To end this chapter, we also discuss how to consider QBF solving as a 2-player game. This point of view will be of particular relevance for the forthcoming chapters.

2.1 Syntax

The syntax of a logic specifies how formulas can be structured. Only formulas which comply with the syntactic rules are part of the language of the logic. In this section, we define the syntax of QBF. To this end, we first recall the language of propositional logic.

2.1.1 Propositional Logic

The objects that we are dealing with in the language of propositional logic are called *propositional formulas*. The basic building blocks of these propositional formulas are *atomic propositions* (also called *variables*), *boolean connectives* and *parentheses* as well as the *truth constants* $\top, \bot$. The variables are taken from a countably infinite set $\mathcal{P} = \{x_1, x_2, x_3, \dots\}$. They represent propositions which can either be true or false. Truth constants represent propositions which are always true ($\top$) or always false ($\bot$). Finally, the boolean connectives *negation* $\neg$, *conjunction* $\wedge$ and *disjunction* $\vee$ together with parentheses),(allow to form more complex expressions from simpler ones. More formally, we have the following recursive definition of the syntax of propositional logic.

Definition 2.1. Let $\mathcal{P}$ be a countably infinite set of variables. The *language of propositional logic* $\mathcal{L}$ is the smallest set such that

1. $\top, \bot \in \mathcal{L}$;

2. $\mathcal{P} \subseteq \mathcal{L}$;

3. if $\phi \in \mathcal{L}$, then $(\neg\phi) \in \mathcal{L}$;

4. if $\phi, \psi \in \mathcal{L}$, then $(\phi \circ \psi) \in \mathcal{L}$, where $\circ \in \{\wedge, \vee\}$;

We will typically denote propositional variables by $x_1, x_2, x_3, \ldots$. However, in cases where only a few variables are needed, we might switch to denoting them just by lowercase letters $a, b, c, \ldots, x, y, z$. This avoids the usage of too many subscripts, and consequently, makes the formulas easier to read. Furthermore, writing formulas strictly according to the syntax introduced above will become cumbersome because of many parentheses. Therefore, we agree upon the notational convention to drop the outermost parentheses and we assume the following precedence of the logical connectives: $\neg$, $\wedge$, $\vee$. However, for better readability, we might still sometimes keep some of the parentheses.

Example 2.2. Using the notational conventions from above, we write $x \wedge \neg y \vee z$ instead of $((x \wedge (\neg y)) \vee z)$.

Apart from the boolean connectives introduced above, there exist further well-known operators such as *implication* $\rightarrow$, *equivalence* $\leftrightarrow$ and *exclusive disjunction* $\oplus$. These additional connectives can be introduced as abbreviations, as they can be expressed in terms of the previous ones. For example, we introduce implication $x \rightarrow y$ as an abbreviation for $\neg x \vee y$. Similarly, equivalence $x \leftrightarrow y$ expresses $(x \wedge y) \vee (\neg x \wedge \neg y)$ and $x \otimes y$ stands for $(x \wedge \neg y) \vee (\neg x \wedge y)$.

In the following, we recall some standard definitions in propositional logic. Given a formula $\phi \in \mathcal{L}$, we denote by $V(\phi)$ the set of variables occurring in ϕ. Furthermore, for a variable $x \in \mathcal{P}$, a *literal* is either x or its negation $\neg x$. A literal l is called *positive* if $l = x$ and *negative* if $l = \neg x$ for some $x \in \mathcal{P}$. The variable of l is denoted by $v(l)$ and is given by $v(l) = x$ if either $l = x$ or $l = \neg x$. A *clause* C is a disjunction of literals, that is $C = l_1 \vee \ldots \vee l_k$ for some $k \geq 0$. The quantity k is referred to as the *size* of the clause. If $k = 0$, then the clause C is called the *empty clause* and denoted by $C = \bot$. Clauses allow us to specify a certain class of propositional formulas which have a particular uniform structure, the so-called *conjunctive normal form (CNF)*.

Definition 2.3. Let $\phi \in \mathcal{L}$. Then ϕ is called in *conjunctive normal form (CNF)* if ϕ is of the form

$$C_1 \wedge \ldots \wedge C_n,$$

where $C_1, \ldots, C_n$ are clauses. The quantity n is called the *size* of ϕ. If $n = 0$, then ϕ is called the *empty CNF* and denoted by $\phi = \top$.

Example 2.4. The propositional formula $\phi = (x \vee y) \wedge (\neg x \vee y \vee z) \wedge (\neg y \vee \neg z)$ is in CNF. It consists of three clauses, two of size two and one of size three. The largest clause contains the positive literals y, z and the negative literal $\neg x$. Furthermore, we have $V(\phi) = \{x, y, z\}$.

We note that any formula $\phi \in \mathcal{L}$ can be transformed (in linear time) into a satisfiability equivalent formula $\phi' \in \mathcal{L}$ which is in CNF; see [BL99, Section 2.3] for an explicit description

of this transformation procedure. This is very useful in practice, as the set of rules stated in Definition 2.1 allows to build propositional formulas with arbitrary structure. In particular, there are no restrictions on how deep the propositional operators can be nested. In practical applications, however, it is often convenient to restrict oneself to only a particular class of formulas. Using a transformation into CNF, we can restrict ourselves to formulas of this specific form. In fact, CNF is widely used in the domain of automated reasoning and in this thesis we will consider QBF solving entirely in the context of a normal form based on CNF.

2.1.2 QBF

Based on the language of propositional logic developed in the previous section, we can now easily introduce the language of *quantified boolean formulas (QBF)*. QBF extends propositional logic by allowing variables to be associated with *universal* ($\forall$) and *existential* ($\exists$) *quantifiers*. Recall that $\mathcal{P}$ denotes a countably infinite set of variables and $\mathcal{L}$ denotes the language of propositional logic.

Definition 2.5. The *language of QBF* $\mathcal{Q}$ is the smallest set such that

1. $\mathcal{L} \subseteq \mathcal{Q}$;
2. if $\phi \in \mathcal{Q}$, then $(\neg\phi) \in \mathcal{Q}$;
3. if $\phi, \psi \in \mathcal{Q}$, then $(\phi \circ \psi) \in \mathcal{Q}$, where $\circ \in \{\wedge, \vee\}$;
4. if $\phi \in \mathcal{Q}$ and $x \in \mathcal{P}$, then $(Qx \,.\, \phi) \in \mathcal{Q}$, where $Q \in \{\forall, \exists\}$;

Note that Definition 2.5 allows quantifiers to be arbitrarily nested in a QBF. Following the same rules as in the case of propositional logic, we might drop some of the parentheses of a QBF in order to increase its readability. We extend the notation $V(\phi)$ for all variables appearing in ϕ from propositional formulas to QBFs $\phi \in \mathcal{Q}$. For $\forall x \,.\, \phi$ (respectively $\exists x \,.\, \phi$), the formula ϕ is called the *scope* of the quantified variable x. An occurrence of the variable x in ϕ is called *bound*. All non-quantified occurrences of a variable, that is, all occurrences which are not in the scope of a quantifier, are called *free* occurrences. A variable $x \in \mathcal{P}$ is called *bound* (respectively *free*) in a formula ϕ if there is a bound (respectively free) occurrence of x in ϕ. A QBF is called *closed* if it does not contain any free variables. In later parts of this work, we will restrict ourselves to closed QBFs.

In the previous section, we have introduced CNF as a uniform and structured way to represent propositional formulas. In a similar fashion, we now extend this normal form to the so-called *prenex conjunctive normal form (PCNF)* for QBFs.

Definition 2.6. Let $\psi \in \mathcal{Q}$. Then ψ is called in *prenex conjunctive normal form (PCNF)* if ψ is of the form

$$Q_1 x_1 \ldots Q_n x_n \,.\, \phi,$$

where $Q_1, \ldots, Q_n \in \{\forall, \exists\}$ and $\phi \in \mathcal{L}$ is a propositional formula in CNF. The string $Q_1 x_1 \ldots Q_n x_n$ is called the *prefix* and ϕ is called the *matrix* of ψ.

Example 2.7. The QBF $\psi = \forall x \exists y \,.\, (x \vee y) \wedge (\neg x \vee y \vee z) \wedge (\neg y \vee \neg z)$ is in PCNF. Its prefix is $\forall x \exists y$ and its matrix is the propositional formula $(x \vee y) \wedge (\neg x \vee y \vee z) \wedge (\neg y \vee \neg z)$ in CNF. Note that ψ is not closed because the variable z is free in ψ. The other two variables x and y are both bound. The scope of x is $\exists y \,.\, (x \vee y) \wedge (\neg x \vee y \vee z) \wedge (\neg y \vee \neg z)$ and the scope of y is $(x \vee y) \wedge (\neg x \vee y \vee z) \wedge (\neg y \vee \neg z)$.

We note that every propositional formula ϕ in CNF is also in PCNF, where the prefix is empty. Similar to the case of CNF transformation, also every QBF $\psi \in \mathcal{Q}$ can be transformed into an equivalent formula $\psi' \in \mathcal{Q}$ which is in PCNF. This transformation is again linear in the size of ψ; see [BB09] for further details. Using this transformation into PCNF, we can restrict ourselves to formulas of this specific form. In fact, we will consider QBF solving entirely in the context of this normal form in this thesis.

For a formula $\psi = Q_1 x_1 \dots Q_n x_n \,.\, \phi$ in PCNF, we typically abbreviate a subpart of the prefix of the form $Qx_i Qx_{i+1} \dots Qx_{i+k}$ with $Q \in \{\forall, \exists\}$, i.e., a block of consecutive equally quantified variables, by $Qx_i, x_{i+1}, \dots, x_{i+k}$. Note that $Qx_i, x_{i+1}, \dots x_{i+k}$ actually means $Q\{x_i, x_{i+1}, \dots x_{i+k}\}$ but we typically omit the parentheses. Hence, we can write $\psi = Q_1 B_1 \dots Q_k B_k \,.\, \phi$ where $Q_1, \dots, Q_k \in \{\forall, \exists\}$ such that $Q_i \neq Q_{i+1}$ for all $1 \leq i < k$ and $B_1, \dots, B_k \subseteq \mathcal{P}$. Furthermore, we can assume that there occurs at least one literal in the matrix ϕ for each quantified variable. Otherwise, this variable together with its quantifier can simply be removed.

The *quantifier blocks* $B_1, \dots, B_k$ induce a partial ordering on the literals of ψ as follows. Given literals l_i, l_j with $v(l_i) \in B_i$ and $v(l_j) \in B_j$, we have $l_i < l_j$ if and only if $i < j$. A quantifier block B_i is called *universal* (respectively *existential*) if $Q_i = \forall$ (respectively $Q_i = \exists$). Given a QBF with k quantifier blocks, there are $k - 1$ *quantifier alternations*. The first quantifier block B_1 and the last quantifier block B_k are called the *outermost*, respectively the *innermost*, quantifier block. For a literal l with $v(l) \in B_i$, the *quantifier type* of l is $q(l) \coloneqq Q_i$.

Example 2.8. The prefix $\exists w \forall x \exists y \exists z$ of the QBF

$$\psi = \exists w \forall x \exists y \exists z \,.\, (x \vee y) \wedge (\neg w \vee y \vee z) \wedge (\neg y \vee \neg z)$$

can be written more compactly as $\exists w \forall x \exists y, z$. Consequently, ψ has three quantifier blocks and two quantifier alternations. Its outermost quantifier block is the existential block $\exists w$ and its innermost quantifier block is the existential block $\exists y, z$. From this, we can see that w is the smallest variable appearing in ψ and, more precisely, that $w < x < y, z$. Note that y and z are incomparable in the partial ordering of variables. Furthermore, we for example have $q(y) = q(\neg y) = \exists$.

To end this section, we mention the QDIMACS format [QBF05] which provides a standardized format for QBFs. In particular, it sets a standard for how the input and output of QBF solvers should look like. We note that our implementations also expect the input to be given in this format.

2.2 Semantics

Now that we have defined the syntactic structure of propositional formulas and QBFs, we can address their semantic evaluation. Semantics provides a set of rules to assign meaning to formulas. In contrast to the case of syntax where we discussed the propositional case separately, we will now immediately introduce the semantics of QBF. The semantics of propositional logic can then be considered as a special case.

In QBF, a semantic evaluation consists of assigning the truth values *true* ($\top$) and *false* ($\bot$) to variables. The truth value of a formula $\psi \in \mathcal{Q}$ can then be determined by assigning truth values to all variables appearing in ψ and by simplifying the formula according to a set of semantic rules. Note that we agree upon the convention to use the symbols $\top$ and $\bot$ for both, the syntactic truth constants as well as for the semantic truth values.

In order to define the semantics of QBF following the standard recursive approach, we first need the notion of an *assignment*.

Definition 2.9. Let $\psi \in \mathcal{Q}$ and let $S \subseteq V(\psi)$. An *assignment* A of ψ is a function $A : S \to \{\top, \bot\}$ which assigns truth values to variables in ψ. An assignment A is *complete* if $S = V(\psi)$ and *partial* otherwise.

Given a formula $\psi \in \mathcal{Q}$ and $S = \{x_1, \ldots, x_n\} \subseteq V(\psi)$, we can represent an assignment $A : S \to \{\top, \bot\}$ of ψ as a set of literals $\{l_1, \ldots, l_n\}$ such that $l_i = x_i$ if and only if $A(x_i) = \top$ and $l_i = \neg x_i$ if and only if $A(x_i) = \bot$ for all $i = 1, \ldots, n$. Hence, the literals $l_1, \ldots, l_n$, with $v(l_i) \neq v(l_j)$ for all $i \neq j$, represent truth assignments to variables.

Example 2.10. We consider the QBF $\psi = \forall x \exists y \,.\, (x \vee \neg y) \wedge (\neg x \vee y)$. The partial assignment $A : \{x\} \to \{\top, \bot\}$ with $A(x) = \top$ of ψ can be represented by the set $\{x\}$. Similarly, the set $\{x, \neg y\}$ corresponds to the complete assignment $A : \{x, y\} \to \{\top, \bot\}$ with $A(x) = \top$ and $A(y) = \bot$ of ψ.

An assignment A of a QBF ψ allows us to define the *interpretation* of ψ under A, denoted by $\psi[A]$, as follows. Since every quantified boolean formula can be transformed into an equivalent formula in PCNF, we can assume w.l.o.g. that ψ is in PCNF.

First, we consider the case where $A = \{l\}$ is a partial assignment of a single variable. Then $\psi[\{l\}]$ is obtained from ψ as follows. First, $v(l)$ is deleted from the prefix, if it appears in it. That is, if $\psi = Q_1 B_1 \ldots Q_i (B_i \cup \{v(l)\}) \ldots Q_k B_k \,.\, \phi$, then $\psi[\{l\}] = Q_1 B_1 \ldots Q_i B_i \ldots Q_k B_k \,.\, \phi[\{l\}]$, where $\phi[\{l\}]$ is computed as described below.

If l is positive, each occurrence of $v(l)$ in ϕ is replaced by $\top$ and each occurrence of $\neg v(l)$ in ϕ is replaced by $\bot$. Dually, if l is negative, each occurrence of $v(l)$ in ϕ is replaced by $\bot$ and each occurrence of $\neg v(l)$ in ϕ is replaced by $\top$. Then ϕ is simplified by applying the following rewrite rules until a fixed point is reached. We note that these rules correspond to well-known equivalences of Boolean algebra.

$$\neg\top \rightsquigarrow \bot \qquad \neg\bot \rightsquigarrow \top \qquad \top \wedge \phi \rightsquigarrow \phi$$
$$\bot \wedge \phi \rightsquigarrow \bot \qquad \top \vee \phi \rightsquigarrow \top \qquad \bot \vee \phi \rightsquigarrow \phi$$

Under these rewrite rules, every formula has a unique normal form (up to reordering of literals and clauses). We define this normal form to be $\phi[\{l\}]$. Finally, variables which no longer appear in $\phi[\{l\}]$ are removed from the prefix $Q_1B_1 \dots Q_kB_k$. If this causes a block B_j to become empty, then the quantifier Q_j can be removed and the (equally quantified) blocks B_{j-1} and B_{j+1} can be merged together.

Additionally, we define $\psi[\emptyset] = \psi$ for the *empty assignment* $A = \emptyset$ and $\psi[\{l_1, \dots, l_n\}] = (\psi[l_1])[\{l_2, \dots, l_n\}]$. To keep the presentation simple, we agree upon the convention to omit parentheses in $\psi[\{l_1, \dots, l_n\}]$ and write $\psi[l_1, \dots, l_n]$.

To summarize, the interpretation of a formula under an assignment A is calculated from the evaluation of the variables of the formula given by A according to the above rewrite rules. We note that the notation $\phi[A]$ is also applicable to propositional formulas ϕ in CNF, as they can be considered as QBFs in PCNF with empty prefix. Furthermore, the interpretation of a formula under a complete assignment is always either $\top$ or $\bot$.

Example 2.11. We consider the QBF $\psi = \forall x \exists y \,.\, (x \vee \neg y) \wedge (\neg x \vee y)$. Then the interpretation of ψ under the partial assignments $\{x\}$ and $\{\neg y\}$ is $\psi[x] = \exists y \,.\, y$ and $\psi[\neg y] = \forall x \,.\, \neg x$ respectively. Furthermore, the complete assignment $\{x, \neg y\}$ yields $\psi[x, \neg y] = (\exists y \,.\, y)[\neg y] = \bot$.

The rewrite rules above justify our notation for the empty clause and the empty CNF. These rules also allow us to make certain assumptions about PCNFs. In particular, we can assume that no clause in a PCNF contains multiple or complementary literals of the same variable. A clause containing complementary literals reduces under any complete assignment to $\top$ and is therefore redundant and can be eliminated from a PCNF. Furthermore, we can assume that a non-empty clause never contains $\top$ or $\bot$, as they can always be removed by the above rules.

Based on the notion of interpretation, we can now state the following standard definition of *satisfiability* of QBFs based on recursive evaluation. Note that we first define satisfiability only for closed QBFs and then extend it to all formulas.

Definition 2.12. Let $\psi \in \mathcal{Q}$ be a closed QBF. Then ψ is called *satisfiable* if and only if one of the following holds:

- $\psi = \top$;
- $\psi = \neg\phi$ and ϕ is not satisfiable;
- $\psi = \phi \wedge \phi'$ and ϕ and ϕ' are satisfiable;
- $\psi = \phi \vee \phi'$ and ϕ or ϕ' is satisfiable;
- $\psi = \forall x \,.\, \phi$ and $\phi[x]$ and $\phi[\neg x]$ are satisfiable;
- $\psi = \exists x \,.\, \phi$ and $\phi[x]$ or $\phi[\neg x]$ is satisfiable;

A not necessarily closed QBF $\psi \in \mathcal{Q}$ is called *satisfiable* if and only if there exists an assignment A of the free variables in ψ such that the closed QBF $\psi[A]$ is satisfiable. If ψ is not satisfiable, then it is called *unsatisfiable*.

According to Definition 2.12, a QBF ψ with free variables $x_1, \ldots, x_n$ is semantically equivalent to the closed QBF $\psi' = \exists x_1, \ldots, x_n \psi$. Hence, for the rest of this work, we restrict ourselves to closed QBFs.

Example 2.13. The closed QBF $\psi = \forall x \exists y \,.\, (x \vee \neg y) \wedge (\neg x \vee y)$ is satisfiable. This can be seen by applying Definition 2.12 as follows

$$
\begin{aligned}
& \psi \text{ is satisfiable} \\
\iff & \Big(\exists y \,.\, (x \vee \neg y) \wedge (\neg x \vee y)\Big)[x] \text{ and } \Big(\exists y \,.\, (x \vee \neg y) \wedge (\neg x \vee y)\Big)[\neg x] \text{ are satisfiable} \\
\iff & \exists y \,.\, y \text{ and } \exists y \,.\, (\neg y) \text{ are satisfiable} \\
\iff & \Big(y[y] \text{ or } y[\neg y] \text{ is satisfiable}\Big) \text{ and } \Big((\neg y)[y] \text{ or } (\neg y)[\neg y] \text{ is satisfiable}\Big) \\
\iff & \Big(\top \text{ or } \bot \text{ is satisfiable}\Big) \text{ and } \Big(\bot \text{ or } \top \text{ is satisfiable}\Big).
\end{aligned}
$$

Now, the first part of Definition 2.12 shows that the last line is true.

Similarly, we can show that the non-closed QBF $\phi = \forall y \,.\, (x \vee \neg y) \wedge (\neg x \vee y)$ is unsatisfiable. To this end, we consider the closed QBF $\phi' = \exists x \forall y \,.\, (x \vee \neg y) \wedge (\neg x \vee y)$. Then the following holds:

$$
\begin{aligned}
& \phi \text{ is satisfiable} \\
\iff & \phi' \text{ is satisfiable} \\
\iff & \Big(\forall y \,.\, (x \vee \neg y) \wedge (\neg x \vee y)\Big)[x] \text{ or } \Big(\forall y \,.\, (x \vee \neg y) \wedge (\neg x \vee y)\Big)[\neg x] \text{ is satisfiable} \\
\iff & \forall y \,.\, y \text{ or } \forall y \,.\, (\neg y) \text{ is satisfiable} \\
\iff & \Big(y[y] \text{ and } y[\neg y] \text{ are satisfiable}\Big) \text{ or } \Big((\neg y)[y] \text{ and } (\neg y)[\neg y] \text{ are satisfiable}\Big) \\
\iff & \Big(\top \text{ and } \bot \text{ are satisfiable}\Big) \text{ or } \Big(\bot \text{ and } \top \text{ are satisfiable}\Big).
\end{aligned}
$$

Since $\bot$ is unsatisfiable, the last line is false. Consequently, ϕ is unsatisfiable.

Remark. As Example 2.13 shows, the prefix of a QBF can drastically influence its satisfiability. The two QBFs $\forall x \exists y \,.\, (x \vee \neg y) \wedge (\neg x \vee y)$ and $\exists x \forall y \,.\, (x \vee \neg y) \wedge (\neg x \vee y)$ share the same matrix but the different prefixes lead to different semantic evaluations.

2.3 Solving QBFs

In this section, we deal with the satisfiability problem for quantified boolean formulas, called *QSAT*. Like SAT for NP, QSAT is the prototypical example of a PSPACE-complete problem [MS73]. Nevertheless, there exist powerful approaches for solving QSAT instances in practice, that is, to decide for a given QBF whether it is satisfiable or not. In the following,

we give a brief overview of the two main techniques that modern QBF solvers implement, which are *backtracking search* and *variable expansion*. To end this section, we also discuss how QSAT can be considered as a game between two players, which will be particularly relevant for our approach involving AlphaZero. Since every QBF can be turned into PCNF, we will restrict ourselves to solving PCNFs in this section. Furthermore, recall that every QBF is semantically equivalent to a closed QBF. Consequently, we may assume that all formulas are closed.

2.3.1 Backtracking Search

The idea of backtracking search arises naturally from the semantic definition of QBFs. Definition 2.12 can be turned into a simple recursive algorithm, where rules for evaluating $\psi = \forall x.\, \phi$ and $\psi = \exists x.\, \phi$ correspond to case splits into subgoals $\phi[x]$ and $\phi[\neg x]$, respectively. This splitting of the proof into subgoals, also called *branching* or *decision making*, is done until either the empty clause $\bot$ is produced (in this case the sub-QBF is unsatisfiable) or the matrix becomes the empty CNF $\top$ (in this case sub-QBF is satisfiable). Then, on the basis of the satisfiability of the subgoals $\phi[x]$ and $\phi[\neg x]$, the satisfiability of ψ can be determined according to Definition 2.12. Once a subgoal is solved, the algorithm backtracks to the most recent unsolved subgoal and continues. This gives rise to a simple yet infeasible algorithm. However, by integrating the following observations into the solving process one can drastically improve the efficiency of this procedure in practice.

One can immediately conclude that a QBF $\psi \in \mathcal{Q}$ is unsatisfiable if the matrix contains a contradictory clause. A clause is called *contradictory* if it contains no existential literal. The empty clause $\bot$ is a special case of a contradictory clause.

The second improvement concerns so-called *unit literals* and *pure literals*.

Definition 2.14. Let $\psi \in \mathcal{Q}$ be in PCNF. An existentially quantified literal e with $v(e) \in V(\psi)$ is called *unit literal* of ψ if there exists a clause C in ψ such that

$$C = e \vee a_1 \vee \ldots \vee a_n,$$

where $a_1, \ldots, a_n$ are universally quantified literals and $e < a_i$ for all $i = 1, \ldots, n$.

Definition 2.15. Let $\psi \in \mathcal{Q}$ be in PCNF. A literal l with $v(l) \in V(\psi)$ is called *pure literal* of ψ if l occurs in ψ but $\neg l$ does not occur in ψ.

Example 2.16. The QBF $\psi = \forall x \exists y \forall z\,.\,(x \vee y) \wedge (x \vee \neg y \vee z) \wedge (\neg y \vee \neg z)$ contains the unit literal $\neg y$ and the pure literal x. Note that $\neg y$ is a unit literal of ψ because of the clause $\neg y \vee \neg z$ as $y < z$. Furthermore, x is a pure literal of ψ because $\neg x$ does not appear in the matrix.

The following two lemmas tell us that unit and pure literals force assignments of the respective variables.

Lemma 2.17. *Let $\psi \in \mathcal{Q}$ be in PCNF and let l be a unit literal of ψ. Then ψ is satisfiable if and only if $\psi[l]$ is satisfiable.*

Proof. See [CSGG02, Lemma 2.2]. □

Lemma 2.18. *Let $\psi \in \mathcal{Q}$ be in PCNF and let l be a pure literal of ψ. If $q(l) = \exists$, then ψ is satisfiable if and only if $\psi[l]$ is satisfiable. Dually, if $q(l) = \forall$, then ψ is satisfiable if and only if $\psi[\neg l]$ is satisfiable.*

Proof. See [CSGG02, Lemma 2.4, Lemma 2.5]. □

Example 2.19. We reconsider the QBF $\psi = \forall x \exists y \forall z \,.\, (x \vee y) \wedge (x \vee \neg y \vee z) \wedge (\neg y \vee \neg z)$ from Example 2.16. Since $\neg y$ is a unit literal of ψ, Lemma 2.17 implies that it suffices to consider

$$\psi' \coloneqq \psi[\neg y] = \forall x \,.\, x$$

to determine the satisfiability of ψ. Now, we can apply Lemma 2.18 to the pure literal x in ψ', which reduces ψ' to

$$\psi'' \coloneqq \psi'[\neg x] = \bot.$$

Since ψ'' is clearly unsatisfiable, we can deduce that so is ψ.

Detecting unit and pure literals and assigning the respective variables according to the lemmas above is called *unit literal elimination*, respectively *pure literal elimination*.

Finally, we note that the ordering in which variables are assigned in such a solving process can substantially influence the overall performance of a QBF solver. Since the variables within one quantifier block can be reordered, this allows to introduce some heuristics into the choice of the variable for branching.

Combining all the improvements mentioned above gives rise to Algorithm 1 for determining the satisfiability of QBFs based on backtracking search. This recursive algorithm is a generalization of the classical DPLL approach for propositional logic [DP60, DLL62] and was first described in [CGS98, CSGG02].

Theorem 2.20. *Let $\psi \in \mathcal{Q}$ be a closed QBF in PCNF. Then Algorithm 1 returns $\top$ given ψ as input if and only if ψ is satisfiable, and $\bot$ otherwise.*

Proof. See [CSGG02, Section 3]. □

Example 2.21. In this example, we apply Algorithm 1 to the QBF

$$\psi = \exists w \forall x \exists y, z \,.\, (\neg w \vee \neg x \vee y) \wedge (\neg x \vee \neg y) \wedge (y \vee z) \wedge (w \vee \neg x \vee \neg z) \wedge (x \vee y \vee \neg z).$$

Since ψ does not contain a contradictory clause, a unit or pure literal, the algorithm reaches line 12. The only possible choice is $l = w$. Consequently, we have to recursively apply Algorithm 1 to $\psi[w]$ and $\psi[\neg w]$.

Case 1. $\psi[w]$ We have to recursively evaluate

$$\psi' \coloneqq \psi[w] = \forall x \exists y, z \,.\, (\neg x \vee y) \wedge (\neg x \vee \neg y) \wedge (y \vee z) \wedge (x \vee y \vee \neg z).$$

Since ψ' does not contain a contradictory clause, a unit or pure literal, the algorithm reaches line 12. The only possible choice is $l = x$. Consequently, we have to recursively apply Algorithm 1 to $\psi'[x]$ and $\psi'[\neg x]$.

Algorithm 1 QDPLL

Input: A closed QBF $\psi = \Pi.\phi$ in CNF with prefix Π and matrix ϕ
Output: $\top$ if ψ is satisfiable and $\bot$ otherwise

```
1:  if φ contains a contradictory clause then
2:      return ⊥
3:  if φ is empty then
4:      return ⊤
5:  if ψ contains a unit literal l then
6:      return QDPLL(ψ[l])
7:  if ψ contains a pure literal l then
8:      if q(l) = ∃ then
9:          return QDPLL(ψ[l])
10:     else
11:         return QDPLL(ψ[¬l])
12: l ← a literal of the outermost quantifier block of ψ
13: if q(l) = ∃ then
14:     return QDPLL(ψ[l]) ∨ QDPLL(ψ[¬l])
15: else
16:     return QDPLL(ψ[l]) ∧ QDPLL(ψ[¬l])
```

Case 1.1. $\psi'[x]$ We have to recursively evaluate $\psi'' \coloneqq \psi'[x] = \exists y, z \,.\, y \wedge (\neg y) \wedge (y \vee z)$. Now, ψ'' contains the unit literal y. Hence, ψ'' simplifies to $\psi''[y] = \bot$, which closes this subcase.

Case 1.2. $\psi'[\neg x]$ We have to recursively evaluate $\psi'' \coloneqq \psi'[\neg x] = \exists y, z \,.\, (y \vee z) \wedge (y \vee \neg z)$. Now, ψ'' contains the pure literal y. Hence, ψ'' simplifies to $\psi''[y] = \top$, which closes this subcase.

Since the two subcases $\psi'[x]$ and $\psi'[\neg x]$ are closed, Algorithm 1 continues by combining their results. More precisely, the results of the subcases are conjoined by $\wedge$ since x was universally quantified. This yields $\bot \wedge \top = \bot$ and closes Case 1.

Case 2. $\psi[\neg w]$ We have to recursively evaluate

$$\psi' \coloneqq \psi[\neg w] = \forall x \exists y, z \,.\, (\neg x \vee \neg y) \wedge (y \vee z) \wedge (\neg x \vee \neg z) \wedge (x \vee y \vee \neg z).$$

Since ψ' does not contain a contradictory clause, a unit or pure literal, the algorithm reaches line 12. The only possible choice is $l = x$. Consequently, we have to recursively apply Algorithm 1 to $\psi'[x]$ and $\psi'[\neg x]$.

Case 2.1. $\psi'[x]$ We have to recursively evaluate $\psi'' \coloneqq \psi'[x] = \exists y, z \,.\, (\neg y) \wedge (y \vee z) \wedge (\neg z)$. Now, ψ'' contains the unit literals $\neg y$ and $\neg z$. Hence, ψ'' simplifies to $\psi''[\neg y, \neg z] = \bot$, which closes this subcase.

Case 2.2. $\psi'[\neg x]$ We have to recursively evaluate $\psi'' \coloneqq \psi'[\neg x] = \exists y, z \,.\, (y \vee z) \wedge (y \vee \neg z)$. Now, ψ'' contains the pure literal y. Hence, ψ'' simplifies to $\psi''[y] = \top$, which closes this subcase.

Since the two subcases $\psi'[x]$ and $\psi'[\neg x]$ are closed, Algorithm 1 continues by combining their results. More precisely, the results of the subcases are conjoined by $\wedge$ since x was universally quantified. This yields $\bot \wedge \top = \bot$ and closes Case 2.

Finally, the algorithm combines the results of Case 1 and Case 2 to obtain the semantic evaluation of ψ. Since w was existentially quantified, the two sub-results are conjoined using $\vee$ which gives $\bot \vee \bot = \bot$. Consequently, the algorithm returns $\bot$ proving that ψ is unsatisfiable.

Remark. In the evaluation of Case 1 in Example 2.21, Algorithm 1 could have stopped after evaluating Case 1.1 because no matter what result r Case 1.2 would yield, we always obtain $\bot \wedge r = \bot$. The same also holds for the evaluation of Case 2.

One example of a state-of-the-art QBF solver that implements a search-based approach based on QDPLL is DepQBF [LB10]. This is also the solver that we will use for all our experiments.

Finally, we note that modern QBF solvers extend the basic QDPLL algorithm by more advanced techniques such as *conflict driven clause learning (CDCL)* [MSS99, ZM02]. This method uses Q-resolution [BKF95], an extension of resolution for propositional logic, to derive clauses from the original formula ψ. Adding such clauses to ψ does not change its semantics but should help the solver rule out certain assignments that are not satisfying anyway. In this way, the search process is guided away from regions of the search space that do not contain solutions.

2.3.2 Variable Expansion

An alternative approach to the one described in the previous section is variable expansion, sometimes also referred to as *expansion-based* solving. While backtracking search can be considered as a top-down approach that tries to construct a satisfying assignment for a given QBF $\psi \in \mathcal{Q}$, variable expansion tries to successively remove quantified variables from ψ until the formula eventually reduces to $\top$ or $\bot$. The following result builds the theoretical foundation for variable expansion. As in the previous sections, we restrict ourselves to formulas in PCNF.

Proposition 2.22. *Let $\psi = Q_1 B_1 \ldots Q_n B_n \cup \{x\} \,.\, \phi \in \mathcal{Q}$ be in PCNF. Then ψ is satisfiable if and only if*

$$Q_1 B_1 \ldots Q_n B_n \,.\, (\phi[x] \circ \phi[\neg x])$$

is satisfiable, where $\circ = \wedge$ if $Q_n = \forall$ and $\circ = \vee$ if $Q_n = \exists$.

So, to expand a variable x, we make two copies of the matrix ϕ and assign x to true in one copy and to false in the other copy. The two copies are then conjoined either by conjunction or by disjunction depending on the quantifier type of x.

Note that the expanded variable x is taken from the innermost quantifier block. This causes variables to be eliminated from right to left and stands in contrast to the search-based

approach where variables were assigned left to right. Therefore, variable expansion is also called a *bottom-up* approach. Furthermore, in this approach, the variable x is assigned to true and false *simultaneously*. This is another difference to the search-based approach where variable assignments are made tentatively and retracted if no solution can be found. We note that there are also generalizations of Proposition 2.22 that allow to expand a variable from an arbitrary quantifier block. In cases where a variable from a non-innermost quantifier block is expanded, all variables that are larger than the expanded variable have to be duplicated.

The main problem of the expansion-based approach is that each variable expansion step can cause the formula to double in size. One way to mitigate this increase in size is to only copy those parts of the matrix on which the expanded variable has an effect. We note that the relevant parts can be found for example by an approach called *mini-scoping* and refer to [Lon12, Section 3.2.2] for further details. Additionally, one can observe that universally quantified variables from the innermost quantifier block do not have to be expanded but can be eliminated directly. This process is called *universal reduction* (sometimes also called *forall reduction*) and was first introduced in [BKF95]. It is based on the following lemma.

Lemma 2.23. *Let $\psi \in \mathcal{Q}$ be in PCNF and let*

$$C = a \vee l_1 \vee \ldots \vee l_n$$

be a clause in ψ such that $q(a) = \forall$ and for all $i = 1, \ldots, n$ if $q(l_i) = \exists$ then $l_i < a$. Then a can be removed from C without changing the satisfiability of ψ.

Proof. This result follows from the principle of Q-resolution. See for example [BKF95] or Definition 23.5.2 and the subsequent discussion in [BB09]. □

Example 2.24. By applying Lemma 2.23, the QBF $\psi = \forall x \exists y \forall z \,.\, (x \vee y) \wedge (x \vee \neg y \vee z) \wedge (\neg y \vee \neg z)$ can be simplified to

$$\forall x \exists y \,.\, (x \vee y) \wedge (x \vee \neg y) \wedge (\neg y).$$

Additionally, as for the search-based approach, also in variable expansion the propagation rules for unit and pure literals can be applied. This gives rise to Algorithm 2 for determining the satisfiability of QBFs based on variable expansion.

Theorem 2.25. *Let $\psi \in \mathcal{Q}$ be a closed QBF in PCNF. Then Algorithm 2 returns $\top$ given ψ as input if and only if ψ is satisfiable, and $\bot$ otherwise.*

Proof. Follows from Lemma 2.17, Lemma 2.18, Proposition 2.22 and Lemma 2.23. □

We note that there is also a version of Algorithm 2 which expands universal variables and eliminates existential variables (see for example Figure 24.4 in [BB09]). For propositional formulas, this algorithm boils down to the Davis Putnam procedure [DP60].

Example 2.26. In this example, we apply Algorithm 2 to the QBF

$$\psi = \exists w \forall x \exists y, z \,.\, (\neg w \vee \neg x \vee y) \wedge (\neg x \vee \neg y) \wedge (y \vee z) \wedge (w \vee \neg x \vee \neg z) \wedge (x \vee y \vee \neg z).$$

Algorithm 2 QDP

Input: A closed QBF $\psi = Q_1 B_1 \dots Q_n B_n \cup \{x\} \,.\, \phi \in \mathcal{Q}$ in PCNF
Output: $\top$ if ψ is satisfiable and $\bot$ otherwise

```
1:  if φ contains a contradictory clause then
2:      return ⊥
3:  if φ is empty then
4:      return ⊤
5:  if ψ contains a unit literal l then
6:      return QDP(ψ[l])
7:  if ψ contains a pure literal l then
8:      if q(l) = ∃ then
9:          return QDP(ψ[l])
10:     else
11:         return QDP(ψ[¬l])
12: if B_n = ∃ then
13:     return QDP(Q_1 B_1 ... Q_n B_n . (φ[x] ∨ φ[¬x]))
14: else
15:     φ' ← remove all occurrences of x and ¬x in φ
16:     return QDP(Q_1 B_1 ... Q_n B_n . φ')
```

Note that this is the formula to which we also applied Algorithm 1 in Example 2.21.

Since ψ does not contain a contradictory clause, a unit or pure literal, the algorithm reaches line 12. We have to expand the formula as the innermost quantifier block is existential. In this step, we have the choice between expanding y or z. If the algorithm chooses y, then we have to recursively evaluate

$$\exists w \forall x \exists z \,.\, \big((\neg x) \wedge (w \vee \neg x \vee \neg z)\big) \vee \big((\neg w \vee \neg x) \wedge z \wedge (w \vee \neg x \vee \neg z) \wedge (x \vee \neg z)\big).$$

Note that this formula is not in PCNF yet. Hence, before we can proceed, this formula first has to be transformed into normal form. Doing this and removing all duplicates of literals within a clause and all clauses with complementary literals of the same variable yields

$$\psi' \coloneqq \exists w \forall x \exists z \,.\, (\neg w \vee \neg x) \wedge (\neg x \vee z) \wedge (w \vee \neg x \vee \neg z).$$

Now, ψ' does not contain any unit literals but it contains the pure literal $\neg x$. Consequently, ψ' is simplified to $\psi'' \coloneqq \psi'[x] = \exists w, z \,.\, (\neg w) \wedge z \wedge (w \vee \neg z)$. The resulting formula ψ'' contains the unit literals $\neg w$ and z. After setting z to $\top$ and w to $\bot$, the formula simplifies to $\bot$. Consequently, the algorithm returns $\bot$ proving that ψ is unsatisfiable.

Modern QBF solvers often implement variable expansion following the so-called *CEGAR (Counterexample-Guided Abstraction Refinement)* approach [CGJ+03, JKMSC12]. This approach is based on the observation that a partial existential expansion can prove a QBF to be satisfiable while a partial universal expansion can prove a QBF to be unsatisfiable. The goal of CEGAR-based approaches is to find such a partial expansion quickly. To this

end, for a given QBF ψ, an existential expansion and a universal expansion are computed simultaneously. An example of a CEGAR-based QBF solver is RAReQS [JK12].

2.3.3 Gamification of QBF Solving

In this section, we describe how determining the satisfiability of a QBF can be considered as a game between two players. This point of view will be particularly useful when using the AlphaZero framework for solving QBFs.

Determining the truth value of a QBF can be seen as a game between a *universal player* and an *existential player*. During the game, the universal player assigns truth values to the universally quantified variables and tries to make the resulting formula unsatisfiable, i.e., tries to obtain $\bot$. Dually, the existential player assigns truth values to the existentially quantified variables and tries to make the formula satisfiable, i.e., tries to obtain $\top$. A player can only assign a value to a variable if this variable is quantified in the outermost quantifier block. The game ends when the resulting formula is $\bot$ or $\top$, with the universal player winning in the first case and the existential player winning in the latter case. More formally, we have the following procedure to "play a QBF $\psi \in \mathcal{Q}$".

Algorithm 3 QSAT game

Input: A closed QBF ψ in PCNF

1: **while** $\psi \notin \{\top, \bot\}$ **do**
2: $x \leftarrow$ a variable of the outermost quantifier block of ψ
3: **if** $q(x) = \exists$ **then**
4: Existential player chooses assignment $A : \{x\} \to \{\top, \bot\}$ of x
5: **else**
6: Universal player chooses assignment $A : \{x\} \to \{\top, \bot\}$ of x
7: $\psi \leftarrow \psi[A]$
8: **if** $\psi = \top$ **then**
9: **return** Existential player wins
10: **else**
11: **return** Universal player wins

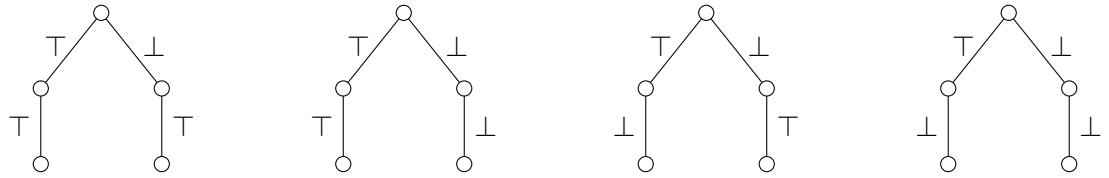
Note that a single game is in general not enough to determine the satisfiability of a given QBF. It is possible that the existential player wins such a game for an unsatisfiable formula, or dually, that the universal player wins a game for a satisfiable formula. This is witnessed by the following example.

Example 2.27. We consider the QBF $\psi = \forall x \exists y \,.\, (x \vee \neg y) \wedge (\neg x \vee y)$. Recall that in Example 2.13 we have seen that ψ is satisfiable. Now, we apply Algorithm 3 to this formula. Since the outermost variable x is universally quantified, the universal player is allowed to choose an assignment for x. If we assume that he decides to set x to $\top$, then ψ simplifies to $\psi' \coloneqq \psi[x] = \exists y \,.\, y$. Now, the outermost variable y is existentially quantified. Consequently, it is the existential player's turn to assign y to a truth value. Assuming that he sets y to

$\bot$, we obtain $\psi'' \coloneqq \psi'[\neg y] = \bot$. Then Algorithm 3 terminates and returns a win for the universal player.

However, we can still relate the satisfiability of a closed QBF to Algorithm 3 by considering *winning strategies*. Here, a wining strategy is a game tree that, for every possible move of the opponent, indicates how to proceed so as to guarantee a win. More precisely, given a closed QBF $\psi = Q_1 x_1 \dots Q_n x_n \,.\, \phi \in \mathcal{Q}$ in PCNF, a *universal strategy* for ψ is a tree of height $n+1$ where every node at level k, with $1 \leq k \leq n$, has a single child if $Q_k = \forall$ and two children if $Q_k = \exists$. If a node has two children, the two edges to these children are labelled by $\top$ and $\bot$, respectively. If a node has only a single child, the edge to this child is labelled by either $\top$ or $\bot$. *Existential strategies* are defined analogously, with the only difference being that that the roles of the quantifiers $\exists$ and $\forall$ are exchanged. More precisely, a node at level k has only one successor if $Q_k = \exists$ and two successors if $Q_k = \forall$.

Example 2.28. We consider the QBF $\psi = \forall x \exists y \,.\, (x \vee \neg y) \wedge (\neg x \vee y)$. There exist the following four existential strategies for ψ:

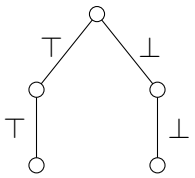


Additionally, there are two universal strategies:

Every path in a strategy for ψ yields a complete assignment $A : V(\psi) \to \{\top, \bot\}$ of ψ and corresponds to one particular game between the universal and the existential player. A universal strategy is a winning strategy (for the universal player) for ψ if all its paths lead to assignments A such that $\psi[A] = \bot$. Dually, an existential strategy is a winning strategy (for the existential player) for ψ if all its paths lead to assignments A such that $\psi[A] = \top$. This definition leads to the following central characterization for the satisfiability of a closed QBF.

Proposition 2.29. *Let $\psi \in \mathcal{Q}$ be a closed QBF in PCNF. Then ψ is satisfiable if and only if there exists an existential winning strategy. Dually, ψ is unsatisfiable if and only if there exist a universal winning strategy.*

Example 2.30. The existential strategy



is a winning strategy for the QBF $\psi = \forall x \exists y \,.\, (x \vee \neg y) \wedge (\neg x \vee y)$. To see this, we note that the left path yields the assignment $\{x, y\}$ and the right path yields $\{\neg x, \neg y\}$. The interpretation of ψ under these assignments is $\psi[x, y] = \psi[\neg x, \neg y] = \top$. Consequently, Proposition 2.29 implies that ψ is satisfiable.

Obtaining a winning strategy for one of the players is equivalent to QSAT, and hence, a hard problem. Even determining whether a given strategy is a winning strategy is computationally expensive as this in general requires to play exponentially many games (measured in the number of variables). However, assuming that we have two perfect players, that is, two players that always make the best possible move, we can deduce the following corollary of Proposition 2.29.

Corollary 2.31. *Let $\psi \in \mathcal{Q}$ be a closed QBF in PCNF and assume that we have a perfect existential and universal player. Then a single run of Algorithm 3 suffices to determine the satisfiability of ψ. More precisely, in this case, Algorithm 3 returns a win for the existential player if and only if ψ is satisfiable.*

Chapter 3

The AlphaZero Framework

In 2016, the British artificial intelligence startup DeepMind made headlines after its computer program AlphaGo [SHM+16] beat a human professional in the game of Go. It was the first computer program to do so. In the following years, AlphaGo was improved and generalized to AlphaZero [SHS+18], a program capable of learning any fully-observable, symmetric 2-player game. AlphaZero has proven to be a powerful framework by beating the best human players and computers in the games of chess, shogi and Go.

AlphaZero combines a *Monte Carlo tree search (MCTS)* [BPW+12] with a newly developed deep reinforcement learning technique called *self-play reinforcement learning* [SSS+17]. In particular, AlphaZero applies a special variant of MCTS called *PUCT (Predictor + Upper Confidence Bound)* [Ros11] for its move selection. This variant uses a so-called predictor to guide the tree search. In AlphaZero, this predictor is realized in form of a neural network that computes confidence scores for all possible moves and an overall confidence score for the current board configuration. Before an AlphaZero agent can be employed in a competitive environment, the predictor has to be trained to learn which moves and board configurations it has to consider as favorable. This is done by a novel reinforcement learning technique called self-play reinforcement learning. Starting as a completely random player, AlphaZero learns by playing games against itself. By learning from the previous games, the predictions of the predictor network improve with each game, which in turn improves the move selection via the MCTS. This improved move selection then leads to a stronger and more competitive player (and opponent). In this way, the algorithm receives a gradually improving training signal.

Following the successes of AlphaZero in the games of chess, shogi and Go, the framework was exported to several other fields. For example, it has been applied to other games, in particular to wargames [MS19, Blo20], as well as to solve complex military planning problems [ZYY+20]. Furthermore, on the more theoretical side, AlphaZero was used in combinatorial optimization [XL20] and to tackle problems coming from first-order logic [XKL21].

In this chapter, we first recall some recent milestones of this field of artificial intelligence that deals with teaching computers to play (board) games on a superhuman level in Section 3.1. On the one hand, this shall help to form a better understanding of the accomplishments achieved by AlphaZero and on the other hand, this allows to highlight the

differences between AlphaZero and its predecessors and successors. Then, in Section 3.2, we describe the original AlphaZero framework as introduced in [SHS+18]. In particular, we discuss the two central concepts applied in the AlphaZero algorithm – Monte Carlo tree search and self-play reinforcement learning.

3.1 Historical Context

We start this brief history overview in 1996, when IBM's chess computer DeepBlue [CHJH02] first played against chess world champion Garry Kasparov in a six-game match and lost 4 – 2. At this time, chess was considered to be *the* challenge for (symbolic) artificial intelligence. For its move selection, DeepBlue used custom hardware to run an optimized minimax algorithm with alpha-beta pruning [KM75] in a highly parallel fashion. This approach can be summarized as pure brute force with DeepBlue using a specially crafted evaluation function to evaluate 50 – 100 million chess board positions per second [CHJH02]. Additionally, DeepBlue had hardcoded data bases with dedicated opening and endgame moves. After the defeat in 1996, several improvements were made to DeepBlue in preparation for a rematch with Kasparov in 1997. Using advanced hardware, a completely redesigned evaluation function and dedicated evaluation tuning software, DeepBlue was able to win the rematch with a score of 3.5 – 2.5. This win allowed to rule off the quest for teaching computers the game of chess and researchers moved on to the next challenge, which was found in the traditional Chinese board game Go.

Go is a symmetric 2-player board game, where players take turns placing stones on the board aiming at surrounding more territory than their opponent. Despite its relatively simple rules, Go is way more complex than chess. This is mainly because of two reasons. First of all, the average branching factor for Go, that is, the average number of legal moves a player can execute at each turn during a game of Go, is 250, compared to 35 for chess [MIG96]. This leads to an astronomically huge number of possible configurations of the board in Go. Additionally, in contrast to chess, there is often no easy way to evaluate these board configurations. Due to this, the brute force approaches that barely worked for chess are practically useless for Go.

This is why it took until the beginning of the last decade that the first Go programs [OK09, Cou11] emerged that could beat human professionals when the humans were given a handicap. These programs replaced exhaustive brute force techniques by a stochastic Monte Carlo tree search. However, it was not until October 2015 that the first Go program was able to beat a human expert without handicap. This breakthrough was achieved by DeepMind's AlphaGo [SHM+16] when it defeated the European Go champion Fan Hui 5 – 0. Similar to the other Go programs, also AlphaGo replaced brute force approaches by a Monte Carlo tree search. However, in contrast to its competitors, AlphaGo supported its MCTS with deep learning techniques. In particular, a neural network guided the tree search by identifying the best moves and the winning percentages of these moves. This neural network was trained from human expert moves. Following the win against Fan Hui, AlphaGo was improved further and played against Lee Sedol, one of the best Go players, in March 2016. Out of five games, AlphaGo could win four. At this time, AlphaGo was running on a cluster consisting

of 1920 CPUs and 280 GPUs.

In the following years, the AlphaGo framework was improved even further. First, AlphaGo Zero [SSS+17] was presented, a version which could learn without any expert data just by playing against itself. It was stronger than any previous AlphaGo version. Additionally, AlphaGo Zero required only four TPUs and could therefore run on a single machine. Following upon that, in December 2017, DeepMind published a paper announcing AlphaGo Zero's successor, named AlphaZero [SHS+18]. AlphaZero generalized its predecessors framework from the game of Go to any fully-observable, symmetric 2-player game. It was able to achieve superhuman level of play in the games of chess, shogi and Go within 24 hours of training and could beat the strongest version of AlphaGo Zero with a score of 60 – 40.

While AlphaZero is limited to conventional board games, its successor MuZero [SAH+20] extends the framework also to other domains. MuZero was presented by DeepMind at the end of 2020 and is considered by many as a significant step towards general artificial intelligence. It combines the planning approach from AlphaZero with ideas from model-free reinforcement learning. This leads to an improved performance in classical planning regimes, such as Go, while also allowing to handle domains with much more complex inputs, such as visual video games. In particular, the new framework can be applied to environments where the underlying dynamics of the system are unknown, such as Atari games (*learning without knowing the rules*).

3.2 AlphaZero in General

In this section, we describe the classical AlphaZero framework as introduced in [SHS+18]. This framework can be considered as a planning algorithm. Such an algorithm deals with finding a strategy for a sequential task for one or more decision makers [LaV06]. In general, this means finding the best possible sequence of *actions* given the current *state* of an environment. In AlphaZero's original field of application, that is, in board games, the current state is the current configuration of the board and an action corresponds to a legal move by one of the players. Often, in planning an optimal action is chosen based on imaginary trajectories (*simulations*). To this end, AlphaZero uses the following two key components:

1. A *Monte Carlo tree search* which is used to generate data in form of simulations.
2. A *deep neural network* which is used to learn from the data and to guide the tree search towards winning positions.

This general approach allows AlphaZero to learn any fully-observable, symmetric 2-player game. Often, such games are also referred to as symmetric *combinatorial games* [ANW19]. They usually share the following characteristics [ZY20]:

- The game contains two players (e.g. chess, Go). Also certain one player games (such as Sudoku or Solitaire) can be considered as combinatorial games between the player and the game designer. Games with more than two players are not regarded combinatorial due to the possibility of coalitions forming.

- The game does not contain any factors of chance that could influence the outcome (such as dice).
- The game is symmetric, in the sense that both players pursue the same goal. More precisely, the outcome of playing a particular strategy depends only on the opponent's strategy, and not on who is playing it.
- The game is fully-observable, often also referred to as a *perfect information game*. This means that at each point in time each player can determine the state of the game; there is no hidden information (as in poker for example). Additionally, each player is perfectly informed about all events that have occurred before.
- The players perform actions in a turn-based manner. Furthermore, at each point in time each player only has finitely many legal moves at their disposal, in other words, the action space is finite. Moreover, also the state space, that is, the set of all possible game configurations, is finite.
- The game ends after finitely many steps either with a win for one of the players or with a draw.

3.2.1 Monte Carlo Tree Search

Monte Carlo methods [MU49] are computational algorithms that use randomness to solve deterministic problems. In particular, they use repeated random sampling to obtain an estimate of a value based on stored statistics such as means and variances. They are particularly useful for very complex problems where it is difficult or even impossible to use deterministic approaches. For example, Monte Carlo methods are used in numerical integration, for physical simulations, or to evaluate business risks [Ham13].

In planning, Monte Carlo methods typically appear in form of a Monte Carlo tree search (MCTS) [BPW$^+$12]. MCTS refers to a family of Monte Carlo algorithms where, given the current state of an environment, random sampling is used to evaluate nodes in a tree search and subsequently use these values to select a promising action in the current state. It is based on the assumption that random sampling can approximate the true value of an action, and that the mean outcome of random simulations forms precisely this approximation. A MCTS is particularly useful for problems with a large branching factor (i.e., a large action space) where exhaustive methods are computationally infeasible. In this section, we describe the variant of MCTS applied by AlphaZero for its move selection. To this end, we first recall the standard MCTS.

Standard MCTS

A MCTS includes two main parts, the *search tree* and the *search method*. The search tree is a data structure consisting of nodes connected by edges that holds all relevant information. Each node represents one state of the environment and each edge between two nodes corresponds to the action that transforms the parent state into the child state. For example, in

terms of board games, each node represents one board configuration and each edge represents a legal move for one of the players starting from the board represented by the parent state. In the following, we may use the terms *node* and *state* as well as *edge* and *action* interchangeably.

We denote the nodes of the search tree by $s_0, \ldots, s_m$ (respectively by s if we only talk about a single node) and the edges by $a_1, \ldots, a_n$ (respectively by a if we only talk about a single edge). Besides the correspondence to a state, a node s in the search tree also holds information about its *visit count* $n(s)$. This number keeps track of how often each node is traversed during the search and is initialized with 0. For a node s with outgoing edge a, the pair (s, a) is called a *state-action pair*. For each state-action pair (s, a), we also keep track of the visit count $n(s, a)$ (initially set to 0). Furthermore, to each such pair, we associate its *value* $Q(s, a)$. The value of a pair (s, a) encodes the average reward that all simulations involving s and a yield. It is initialized with 0 as well. One important aspect to keep in mind is that, for our application of MCTS to 2-player games, the tree contains two perspectives, one for each player. This means that the information at each node is either from the perspective of player A or from the perspective of player B.

In the beginning of the MCTS, the search tree is initialized with a root node representing the state of the environment for which a promising action should be found. Then the following four steps are repeated until a termination criterion is met:

1. **Selection**: Actions are selected from the root node to a leaf node following the search method.

2. **Expansion**: Child nodes are added to the current leaf node and one child node is selected.

3. **Simulation**: A (random) simulation is generated starting from the selected child node. The simulation ends when an end state is reached (e.g. when the game ends with a win for one of the two players).

4. **Backpropagation**: The result of the simulation is backpropagated through the tree to update the information at each node that was traversed during this iteration.

With every iteration of the MCTS the search tree is extended by adding the children of a leaf node. As the tree grows in this way, the values of the state-action pairs, which are updated after each simulation, become more and more accurate. The iterations are usually halted before these values converge to an optimum with standard termination criteria being pre-defined time limits or iteration numbers. Finally, the action a from the root node s_0 with either the highest visit count $n(s_0, a)$ or the highest value $Q(s_0, a)$ is chosen as the most promising action. In this choice, ties are broken arbitrarily. In the following, we discuss the four main steps of the MCTS in more detail.

Selection. A crucial part of the MCTS is the search method that is used in the selection phase to traverse the search tree. The aim of the search method is to explore promising regions of the search space and thereby estimate the state-action values $Q(s_0, a)$ of the root

node s_0. The most commonly used strategy is acting greedily with respect to the *upper confidence bound for trees (UCT)* [KS06], which is based on the upper confidence bound algorithm UCB1 [ACBF02] for classical multi-armed bandit problems.

The UCT algorithm provides a good tradeoff between *exploitation*, that is, reusing actions that have proven to be good in the past, and *exploration*, that is, trying out new actions. In UCT, exploitation is included via the values $Q(s,a)$ of the state-action pairs (s,a) and exploration is added through the visit counts $n(s)$ and $n(s,a)$. These two aspects are balanced by an *exploration constant* γ, typically chosen such that $1 \leq \gamma \leq 2$ with the theoretical optimum being $\gamma = \sqrt{2}$. The UCT of a state-action pair (s,a) in the search tree is given by

$$\mathrm{UCT}(s,a) = Q(s,a) + \gamma\sqrt{\frac{\log n(s)}{n(s,a)}} \tag{3.2.1}$$

if $n(s,a) > 0$ and by $\mathrm{UCT}(s,a) = \infty$ otherwise. The first term $Q(s,a)$ measures the average reward of simulations involving the state-action pair (s,a) and encourages the exploitation of high-reward actions. The second term $\sqrt{\frac{\log n(s)}{n(s,a)}}$ weighs the total number of times the state s was visited against the number of times the action a was chosen after visiting s, and thereby drives exploration of rarely visited states.

Using UCT as a search method, the selection phase traverses the search tree until a leaf node is reached always following the edge from a state s which maximizes $\mathrm{UCT}(s,a)$ among all outgoing edges a of s. During this traversal, ties are broken arbitrarily. More precisely, the selection phase with UCT can be described by the following algorithm.

Algorithm 4 Selection

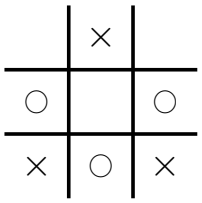
Input: A search tree t
Output: A leaf node s of t

1: $s \leftarrow$ root of the search tree
2: **while** s is not a leaf node **do**
3: $\quad a_1, \ldots, a_n \leftarrow$ outgoing edges of s
4: $\quad$ compute $\mathrm{UCT}(s,a_i)$ for all $i = 1, \ldots, n$
5: $\quad$ select $a \in \{a_1, \ldots, a_n\}$ such that

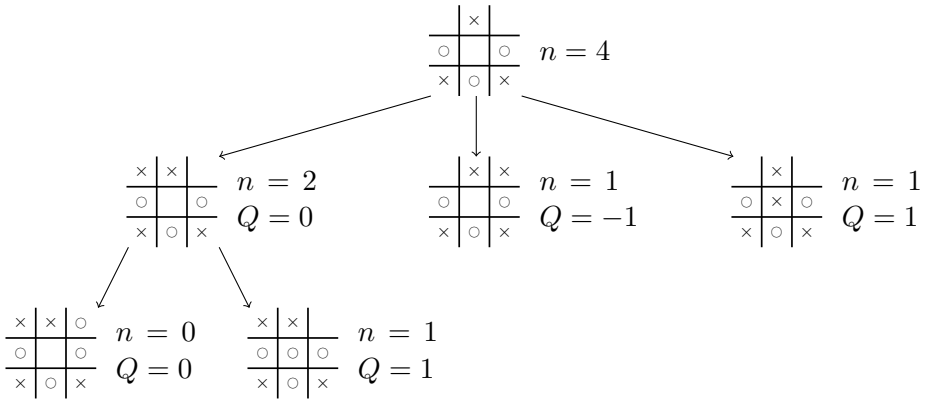
$$\mathrm{UCT}(s,a) = \max_{i \in \{1,\ldots,n\}} \mathrm{UCT}(s,a_i)$$

6: $\quad s \leftarrow$ the child of s reached by a
7: **return** s

Example 3.1. To illustrate the main steps of a MCTS, we accompany our descriptions with a running example. We consider the following ongoing game of tic-tac-toe:



Next up, is player ×'s turn. Using a MCTS, we will determine which action this player should choose given the current state of the board. We assume that a few iterations of the MCTS have already been executed resulting in the following search tree:



We represent each node in the tree by the respective state of the board. The root of the tree is the state for which we want to find a promising action. We have added to each node s its visit count $n(s)$ and to each child s' of s the value $Q(s,a)$ of the state-action pair (s,a), where a is the action the leads from s to s'. We also note that the visit count $n(s,a)$ of (s,a) is given by $n(s')$.

Using UCT as a search method, we start at the root node s_0 and compute for all its three children the UCT. For the sake of simplicity, we use the exploration constant $\gamma = 1$. Denoting the outgoing edges of s_0 from left to right by a_1, a_2, a_3 this yields

$$\mathrm{UCT}(s_0, a_1) = 0 + \sqrt{\frac{\log 4}{2}} \approx 0.83,$$

$$\mathrm{UCT}(s_0, a_2) = -1 + \sqrt{\frac{\log 4}{1}} \approx 0.18,$$

$$\mathrm{UCT}(s_0, a_3) = 1 + \sqrt{\frac{\log 4}{1}} \approx 2.18.$$

Consequently, the selection phase follows the edge a_3 to the rightmost child of s_0. Since this child is a leaf node, the selection phase terminates and returns this node.

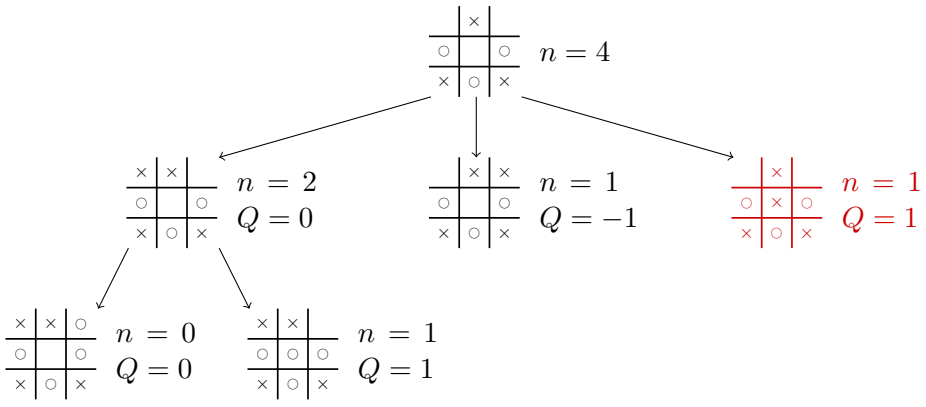
Expansion. After the selection phase, we end up with a leaf node s of the search tree. If this leaf node has not been visited before and is not the root, or if s is an end state, then we can skip the expansion phase and immediately start a simulation starting from s. Otherwise, that is, if s is either the root node or a non-end state that has been visited at least once, we expand s. This means that we add all possible children $s_1, \ldots, s_m$ of s to the search tree. In case of a board game, those are all possible board configurations that are reachable from the board state represented by s by a single legal move. Then one of these children s_i is selected arbitrarily. Starting from this child, a simulation will be started in the next phase. The expansion phase is summarized in the following algorithm.

Algorithm 5 Expansion

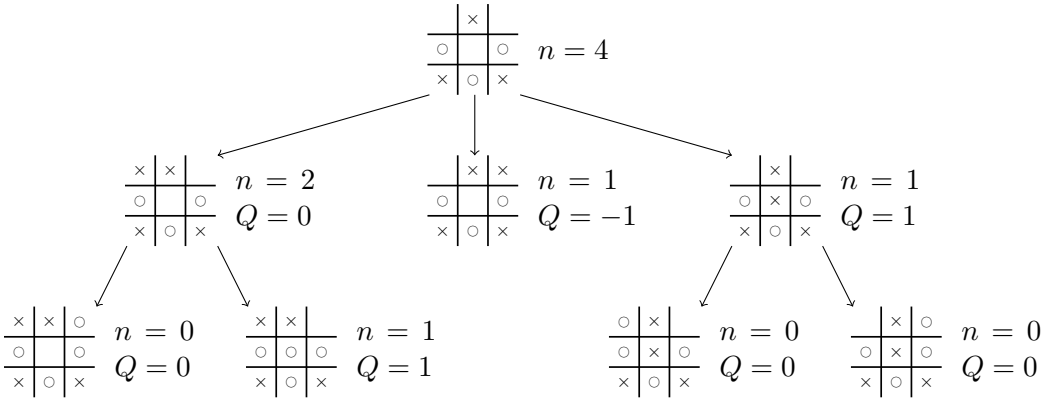
Input: A search tree t and a leaf node s of t
Output: The (expanded) search tree t' and a leaf node s' of t'

```
if n(s) = 0 and s is not the root node then
    return t and s
else if s is an end state then
    return t and s
else
    t' ← add all possible children s_1, ..., s_m of s to t
    choose s' ∈ {s_1, ..., s_m} in some way
    return t' and s'
```

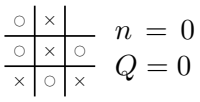
Example 3.1 (continuing from p. 26)**.** Recall that the selection phase with UCT selected the leaf node highlighted in red in the following search tree:



Since this node has already been visited and is not an end state (the game has not ended yet), the expansion phase expands this node by adding all its children to the search tree. This yields the following extended tree:



Then one of the newly added nodes is selected for the simulation phase. If we use a left-to-right selection strategy for this, the node



will be returned by the expansion phase.

Simulation. Once the expansion phase returns a state s from the (extended) search tree, a simulation (also called *playout* or *rollout*) is started from this state. During this simulation, a *simulation strategy* chooses (pseudo-)random moves for both players in a self-play style until an end state is reached.

Often, the simulation strategy is chosen to be purely random. In this case, the simulations are also referred to as *light* simulations. If domain knowledge is available, it can be advantageous to include certain heuristics to influence the choice of moves. In this case, the simulations are also called *hard* simulations. Using an adequate simulation strategy can improve the performance of the MCTS significantly [Cha10]. However, while adding knowledge to a strategy can lead to more accurate and reliable simulations, it can also cause a computational overhead. This might lead to a lower number of iterations per second compared to using a random (and fast) simulation strategy, which in turn might reduce the overall accuracy of the MCTS. Consequently, there is always a trade-off between search and knowledge when selecting a suitable simulation strategy. Additionally, a second trade-off between exploration and exploitation has to be taken into account. A random simulation strategy can lead to too much exploration causing the simulations to become unrealistic as too many weak actions are chosen. On the other hand, if the strategy is too deterministic (i.e., if the same moves are repeated over and over again) too much exploitation takes place. This leads to biased simulations and can again decrease the effectiveness of the MCTS.

Once the simulation reaches an end state, the *reward* of this simulation is recorded and returned. In combinatorial games between a player A and a player B, this reward is always from the perspective of one of the two players and depends on which player has initiated the MCTS. If the MCTS was started to find a promising action for player A, then the reward is

counted positively whenever the simulation yields a win for player A and negatively whenever the simulation yields a win for player B. Usually, the values $+1$ and -1 are chosen for this. Draws lead to the reward 0.

The following algorithm describes the simulation phase for a combinatorial game with rewards 0 and ± 1.

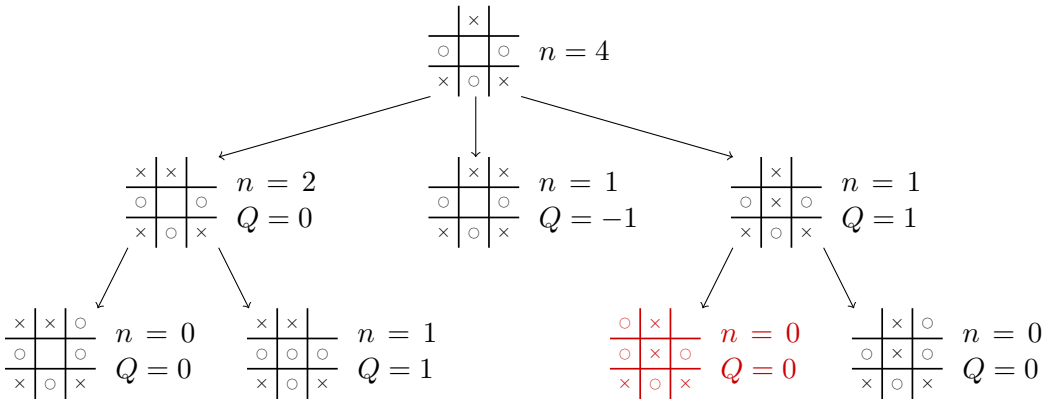
Algorithm 6 Simulation

Input: A state s of the search tree
Output: Reward of a simulation starting from s

```
while s is not an end state do
    a ← select a legal action for s following the simulation strategy
    s ← apply action a to update the state s
if s is a draw then
    return 0
else if the player who initiated the MCTS won the game then
    return +1
else
    return −1
```

Note that the states encountered during the simulation are not added to the search tree. Only the reward of the simulation is recorded.

Example 3.1 (continuing from p. 27)**.** Recall that the expansion phase has returned the node highlighted in red in the following search tree:



Now, starting from this node a game is played according to the simulation strategy. In this case, there is only one possible move left. So, every simulation strategy would yield the same result, which is a win for player $\times$. Consequently, after executing this move, the simulation phase returns $+1$ indicating that player $\times$, who initiated the MCTS, won the game.

Backpropagation. During the backpropagation phase, the reward v obtained from the simulation is used to update the information held in the search tree. In particular, for each state-action pair on the path from the simulated node to the root node, the visit count is increased by one and the reward v is used to update to the value of the pair. The visit counts of the nodes themselves are also increased accordingly.

When updating the value of a state-action pair, one has to take the different perspectives of the players into account. The reward of the simulation is always from the perspective of the player who initiated the MCTS but some nodes in the search tree correspond to states where the opponent is acting. From the perspective of the opponent, the reward of the simulation is not v but $-v$. Assuming player A has initiated the MCTS, then the value $Q(s, a)$ of every state-action pair (s, a) where s is a state in which player A has to act are updated using v. Conversely, the value $Q(s, a)$ of every state-action pair (s, a) where s is a state in which player B has to act are updated using $-v$. This gives rise to the following algorithm.

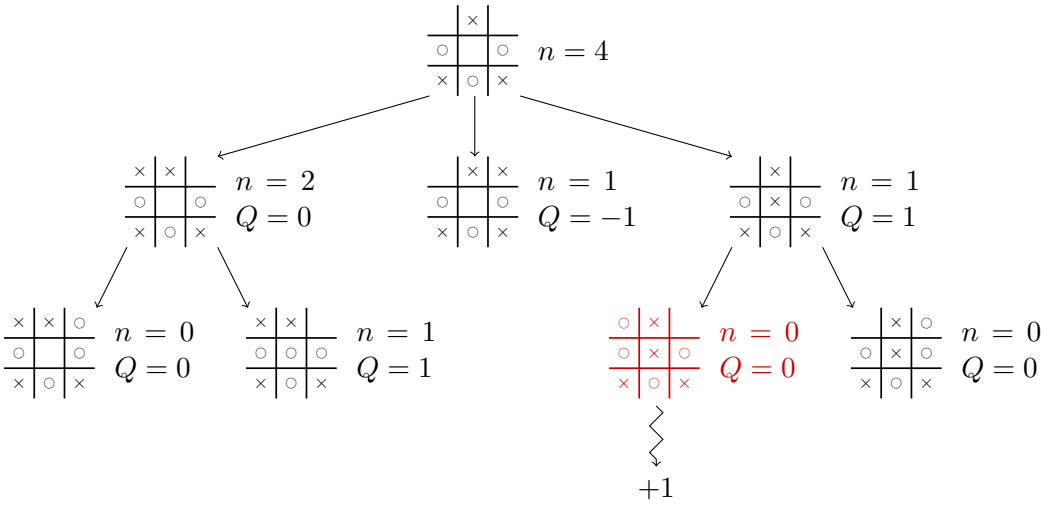
Algorithm 7 Backpropagation

Input: A search tree t, a leaf node s' of t, the reward v of a simulation starting from s'
Output: An updated version of the search tree t

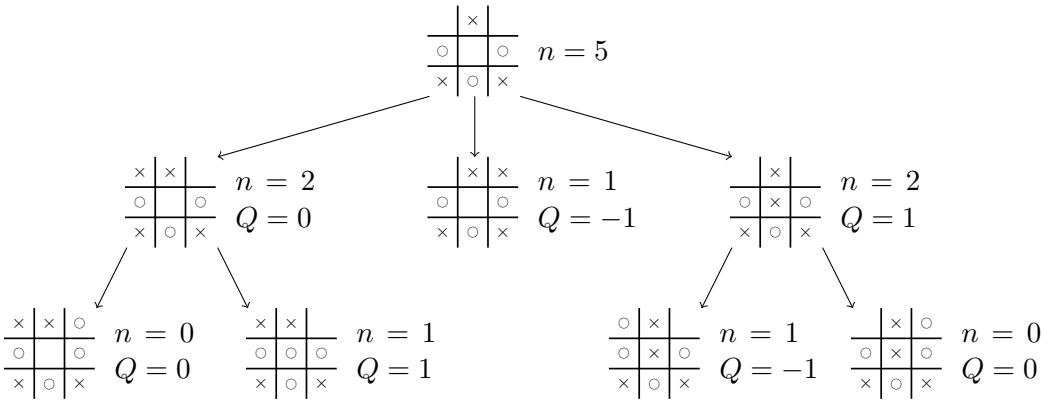
$n(s') \leftarrow n(s') + 1$ ▷ as a side effect t is updated
while s' is not the root node **do**
  $s \leftarrow$ the parent of s'
  $a \leftarrow$ the action that leads from s to s'
  if s is a state where the player who initiated the MCTS has to act **then**
    $Q(s, a) \leftarrow \frac{n(s,a)Q(s,a)}{n(s,a)+1} + \frac{v}{n(s,a)+1}$ ▷ as a side effect t is updated
  else
    $Q(s, a) \leftarrow \frac{n(s,a)Q(s,a)}{n(s,a)+1} - \frac{v}{n(s,a)+1}$ ▷ as a side effect t is updated
  $n(s, a) \leftarrow n(s, a) + 1$ ▷ as a side effect t is updated
  $n(s) \leftarrow n(s) + 1$ ▷ as a side effect t is updated
  $s' \leftarrow s$
return t

We note that for evenly alternating combinatorial games, the backpropagation procedure can also be phrased in a slightly different way. In these cases, the reward can be negated when moving from one state-action pair to the next and then the values can be updated by always adding v. This is the way the backpropagation phase is usually presented. Our presentation, however, has the advantage that it is more flexible and also allows variable move orders, i.e., situations where one player is allowed to make several consecutive moves. This is particularly relevant for our application of QSAT, where games are not necessarily evenly alternating.

Example 3.1 (continuing from p. 29)**.** Recall that the simulation phase started a simulation from the node highlighted in red in the following search tree and returned the value $+1$ indicating a win for player $\times$.



Following Algorithm 7 for the backpropagation, we obtain the following updated search tree:



Note that the value of the last state-action pair has been updated by subtracting $v = +1$ because it was player $\bigcirc$'s turn, while the value of the first state-action pair was updated by adding v as it was player $\times$'s turn.

If the MCTS was stopped at this point and the value of the state-action pairs was used as the final decision criterion, then the action to put the cross in the middle of the tic-tac-toe board would be chosen.

Bringing the four sub-algorithms together gives the following procedure for a MCTS with UCT as search method for a combinatorial game with results 0 and ± 1. Note that in this algorithm the final action is chosen based on the visit counts of the state-action pairs.

Algorithm 8 MCTS

Input: A state s_0 of the environment
Output: A promising action a for state s_0

$t \leftarrow$ initialize search tree with root node s_0
while within computational budget **do**
    $s \leftarrow \text{Selection}(t)$
    $t', s' \leftarrow \text{Expansion}(t, s)$
    $v \leftarrow \text{Simulation}(s')$
    $t \leftarrow \text{Backpropagation}(t', s', v)$
$a_1, \ldots, a_n \leftarrow$ outgoing edges of s_0
select $a \in \{a_1, \ldots, a_n\}$ such that $n(s_0, a) = \max_{i \in \{1,\ldots,n\}} n(s_0, a_i)$
return a

MCTS in AlphaZero

The variant of MCTS that AlphaZero applies for its move selection differs slightly from the standard MCTS described above. More precisely, AlphaZero uses *PUCT (Predictor + UCT)* [Ros11] as a search method during the selection phase. In this variant of UCT, a *predictor* is used in combination with UCT to traverse the search tree. The predictor can be considered as an expert that guides the tree search towards promising actions, which reduces the search breadth. In AlphaZero, this predictor is realized in form of a two-headed deep neural network parameterized by certain parameters θ. The first head π_θ of this network maps states s of the environment to *action probabilities* $\pi_\theta(s)$. These action probabilities are then used to improve the selection phase in the MCTS.

Additionally, the second head ν_θ of the neural network predicts for a given state s the *estimated reward* $\nu_\theta(s)$ of this state. This prediction replaces the reward of the simulation in the simulation phase. So, effectively AlphaZero replaces the simulation phase by an *evaluation phase* where a deep neural network is used to evaluate a state directly. This is particularly useful for large search spaces where (random) simulations do not provide much information due to high variance. By replacing these sub-trees with a single prediction from a neural network, the search depth is reduced.

In the following, we describe how the standard MCTS selection and simulation phase have to be adapted to fit into the AlphaZero framework.

Selection. The output of the first neural network π_θ, also called the *policy network*, is a discrete probability distribution over all legal actions of the currently acting player. This means that the action probabilities $\pi_\theta(s)$ of a state s computed by π_θ form a vector with components $\pi_\theta(a|s) \coloneqq \pi(s)_a \in [0, 1]$ for each state-action pair (s, a) such that $\sum_a \pi_\theta(a|s) = 1$. These values represent the probability of selecting each action a in the given state s and are integrated into the selection phase via the following search formula:

$$\text{PUCT}(s, a) = Q(s, a) + \gamma \cdot \pi_\theta(a|s) \frac{\sqrt{n(s)}}{n(s, a) + 1}. \tag{3.2.2}$$

Note that there are several differences but also certain similarities between the UCT (Equation 3.2.1) and the equation above. First of all, the exploitation term $Q(s,a)$ remains the same. The exploration term $\pi_\theta(a|s)\frac{\sqrt{n(s)}}{n(s,a)+1}$, however, has been adapted to integrate the action probabilities $\pi_\theta(a|s)$. Moreover, there is an additional exploitation bias in the denominator of this term. By adding a plus one to the counts $n(s,a)$ of all possible actions, the immediate expansion of all unvisited nodes is prevented.

Remark. Theoretically the square root in the exploration term in Equation 3.2.2 should cover the whole fraction, i.e. it should be $\sqrt{\frac{n(s)}{n(s,a)+1}}$. However, AlphaZero uses the variant from Equation 3.2.2 without giving any explanation why. In [XL21], the authors claim to have used both versions with the AlphaZero version performing better. Consequently, in our implementations we will also work with Equation 3.2.2.

The selection phase in AlphaZero then follows Algorithm 4 except for the fact that $\mathrm{UCT}(s,a)$ is replaced by $\mathrm{PUCT}(s,a)$.

Evaluation (replaces Simulation). The expansion phase in AlphaZero is done as described in Algorithm 5. Once this process finishes, we obtain a leaf node s of the search tree, from which a standard MCTS would start a simulation. If the state s is an end state, then in AlphaZero the result of this end state is used as the reward for the backpropagation (0 in case of a draw, $+1$ if the player who initiated the MCTS won, and -1 otherwise). We note that this is also what happens in a standard MCTS when a simulation is started from an end state. However, if the state s returned by the expansion phase is not an end state, then the procedure in AlphaZero really differs from the standard MCTS. In such a case, AlphaZero replaces the simulation by a call to the neural network ν_θ, also called the *value network*. The output $\nu_\theta(s)$ of this neural network is a scalar value in the interval $[-1,1]$ which forms an estimate of the probability of the current player winning from state s. Here, -1 represents absolute certainty that the current player will lose and $+1$ represents absolute certainty that the current player will win. Note that the value $\nu_\theta(s)$ is always from the perspective of the player acting in state s. To use the backpropagation procedure described for the standard MCTS, this value has to be multiplied by -1 in case the active player in state s is not the player who has initiated the MCTS. Then Algorithm 7 can be used for the backpropagation.

At the end of the MCTS in AlphaZero, the final action is chosen based on the visit counts of the state-action pairs.

3.2.2 Self-play Reinforcement Learning

For the MCTS in AlphaZero to work properly, it is essential that the neural network π_θ knows which actions to consider as useful in a given state s and that the neural network ν_θ knows how to relate states to (likely) winning positions. This is something which has to be learned. To this end, a new reinforcement learning technique called *self-play reinforcement learning* [SSS+17] is used to adapt the parameters θ of the networks in such a way that the MCTS performs well.

The general idea of self-play reinforcement learning is to collect data by letting AlphaZero play games against itself and to then use this data to update the parameters θ in order to improve the performance of AlphaZero in subsequent games. More precisely, the parameters θ are initialized randomly and then the following two steps are alternated for a fixed number of iterations (also called *epochs*):

1. Generate data by self-play games
2. Update parameters θ

In the data generation phase, the current version of AlphaZero plays several games against itself. During each game, for each state s of the game, a MCTS is done as described in the previous section. However, in this case, the final action in the MCTS is not chosen deterministically but sampled randomly from a distribution computed from the visit counts of the state-action pairs. More precisely, in the learning phase, the MCTS in AlphaZero returns a discrete probability distribution p over all legal actions a for the player acting in the root state s. Each component p_a of this distribution is computed as

$$p_a = \frac{n(s,a)^{1/\tau}}{\sum_a n(s,a)^{1/\tau}},$$

where τ is a temperature parameter that regulates exploration. For the first moves of a game it is set to $\tau = 1$. This causes actions to be selected proportionally to the visit counts of the state-action pairs in the MCTS and leads to more exploration. As the game progresses, τ is decreased and finally set to an infinitesimal value, resulting in a distribution that almost always chooses the action a with the highest visit count $n(s,a)$. Given the distribution p, an action a is then sampled and the state s is updated according to this action. Once this is done, the next AlphaZero MCTS is started to find a promising action for the updated state. This process is continued until an end state is reached.

After each game, the game trace is stored as a set $\{(s_1, p_1, v_1), \ldots, (s_t, p_t, v_t)\}$ of tuples (s_i, p_i, v_i), where s_i is the state of the game at time i, p_i is the distribution computed by the MCTS with s_i as a root state and v_i is the final result of the game. The result v_i is 0 in case the game ended in a draw, $+1$ if the player acting in state s_i ultimately won the game and -1 otherwise. Note that v_i is always from the perspective of the player acting in state s_i.

Once the final game ends, all the stored games traces are merged into one big set $\{(s_1, p_1, v_1), \ldots, (s_T, p_T, v_T)\}$ for some $T \in \mathbb{N}$. This data is then used to update the parameters θ of the neural networks by performing gradient descent to minimize the loss function

$$L(\theta) = \sum_{t=1}^{T} \Big((v_t - \nu_\theta(s_t))^2 - p_t^T \cdot \log \pi_\theta(s_t) \Big) + c\|\theta\|.$$

The first part $(v_t - \nu_\theta(s_t))^2$ of this loss function is a mean-squared error that shall improve the predictions of the value network ν_θ. In particular, the value network should learn to associate the state s_t to the end result v_t of the game where s_t appeared in. In this way, over time, the network learns what states lead to wins (and losses).

The second term $p_t^T \cdot \log \pi_\theta(s_t)$ is a cross-entropy loss and affects the policy network. We note that we use vector notation here. More precisely, p_t and $\pi_\theta(s_t)$ are both column vectors (of the same length). Then p_t^T denotes the transpose of p_t and is therefore a row vector and $\log \pi_\theta(s_t)$ stands for a component-wise application of the logarithm. Finally, $p_t^T \cdot \log \pi_\theta(s_t)$ gives the usual vector product and results in a scalar value. With this loss term, the parameters θ are updated so that the distribution $\pi_\theta(s_t)$ computed by the policy network π_θ gets closer to the improved distribution p_t obtained from the MCTS.

Finally, the last part $c\|\theta\|$ of the loss function is an L_2 weight regularization term that shall prevent overfitting. The value c is a hyperparameter that controls the level of this regularization.

To summarize, the results of the MCTS are used to update the parameters θ and thereby improve the predictions of the neural networks. These improved predictions, in turn, help to guide the MCTS in the next self-play games to more promising actions and consequently lead to better MCTS results. This creates a constant learning stimulus and allows the AlphaZero player to continuously improve.

Chapter 4

AlphaZero for QBF Solving

As done in [XL21], we will apply the AlphaZero framework (in [XL21] called *neural MCTS*) to solve QSAT problems. The foundation to do this is to consider QSAT as a 2-player game as discussed in Section 2.3.3. More precisely, the crucial observation is Corollary 2.31. Based on this, the idea is to use the AlphaZero framework to train two (perfect) players, one existential player $P_\exists$ and one universal player $P_\forall$, for the QSAT game described by Algorithm 3. To then determine the satisfiability of a given QBF $\psi \in \mathcal{Q}$, we use $P_\exists$ and $P_\forall$ as players in Algorithm 3 and give ψ as input. If our two players are indeed perfect, then based on Corollary 2.31 the outcome of this game determines the satisfiability of ψ.

However, to apply the AlphaZero framework to the problem of QSAT, the algorithm has be adapted slightly. First of all, we have to find a suitable representation of QBFs that is accessible for AlphaZero and that can be used as an input for a neural network. We present one such representation in Section 4.1. Going along with this, we also discuss what network architecture can be used for the policy and value network. Additionally, the 2-player game derived from QSAT as presented in Algorithm 3 is not symmetric. The two players have different goals; one player tries to make the formula true while the other tries to make the formula false. In Section 4.2, we describe how to adapt the AlphaZero framework to this situation. For all adaptions described in this chapter, we follow the suggestions in [XL21].

It has to be noted that, since the AlphaZero framework is ultimately a black box, we cannot determine whether our players are really perfect. Consequently, the results produced in this way are not provably correct (like the results of standard QBF solving techniques) but should rather be considered as predictions. Nevertheless, we can empirically evaluate the accuracy of the AlphaZero players, which will be done in Section 4.3.

4.1 QBF Graphs and GGNNs

In the classical application of AlphaZero to board games such as chess, shogi and Go, a state of the game can be described essentially by taking a bird's eye view picture of the board. Computational, this "picture" is encoded in form of a multidimensional tensor with one "pixel" (entry) for each position on the board. The policy and value networks are then realized in form of two heads of a deep convolutional neural network (CNN) [KSH12] that

takes these multidimensional tensors as input. The advantage of CNNs is that they allow to efficiently work with high-dimensional data. Furthermore, CNNs also take into account the spatial structure of the input, which is particularly useful for board games.

In case of the QSAT game, each state is a QBF (in PCNF). While one could still encode such a formula in form of a "picture" and use CNNs, this does not seem like the best approach since spatial information is not as relevant for QBF solving as it is for playing chess or other board games. For example, the clauses of a QBF in PCNF as well as the variables within each clause can be permuted without changing the underlying formula (and its semantic evaluation). More precisely, QBFs exhibit a lot of symmetries which arise from the semantics of propositional logic [KS18]. Exploiting these symmetries can (hopefully) improve the learning efficiency. To this end, we consider a QBF encoding that preserves the symmetries of the formula.

Let

$$\psi = Q_1 B_1 \dots Q_k B_k \,.\, C_1 \wedge \dots \wedge C_m \in \mathcal{Q}$$

be a given closed QBF in PCNF with alternating quantifiers $Q_1, \dots, Q_k \in \{\exists, \forall\}$, variable blocks $B_1, \dots, B_k \subseteq \mathcal{P}$ such that $\bigcup_{i=1}^{k} B_i = \{x_1, \dots, x_n\}$ and clauses $C_1, \dots, C_m \in \mathcal{L}$. We represent ψ in form of an undirected graph $G_\psi = (V_\psi, E_\psi)$ with vertices V_ψ and edges E_ψ. This graph contains one vertex for each variable x_i, one vertex for the negation of each variable $\neg x_i$ and one vertex for each clause C_j, that is,

$$V_\psi = \{x_1, \dots, x_n, \neg x_1, \dots, \neg x_n, C_1, \dots, C_m\}.$$

Furthermore, the set of edges consists of the following four pairwise disjoint subsets:

1. The set of edges $E_{\exists,\forall}$ between every consecutive existential and universal literal, i.e.,

$$E_{\exists,\forall} = \{\{l, l'\} \mid l, l' \text{ literals}, q(l) = \exists, v(l) \in B_i, v(l') \in B_{i+1} \text{ for some } 1 \leq i < k\}.$$

2. The set of edges $E_{\forall,\exists}$ between every consecutive universal and existential literal, i.e.,

$$E_{\forall,\exists} = \{\{l, l'\} \mid l, l' \text{ literals}, q(l) = \forall, v(l) \in B_i, v(l') \in B_{i+1} \text{ for some } 1 \leq i < k\}.$$

3. The set of edges E_C between every literal and each clause it appears in, i.e.,

$$E_C = \{\{l, C\} \mid l \text{ literal}, C \text{ clause}, l \text{ appears in } C\}.$$

4. The set of edges E_X between every variable and its negation, i.e.,

$$E_X = \{\{x_1, \neg x_1\}, \dots, \{x_n, \neg x_n\}\}.$$

The set of edges E_ψ of G_ψ is then given by

$$E_\psi = E_{\exists,\forall} \cup E_{\forall,\exists} \cup E_C \cup E_X.$$

Example 4.1. To illustrate the encoding of QBFs in form of undirected graphs, we consider the formula

$$\exists x \forall y \exists z \,.\, (x \vee y \vee z) \wedge (y \vee \neg z) \wedge (\neg x \vee z),$$

which is represented by the graph in Figure 4.1. To help distinguish between the four different edge types, we have used different line styles. Edges in $E_{\exists,\forall}$ are depicted in form of dashed lines, edges in $E_{\forall,\exists}$ in form of dotted lines, edges in E_C in form of solid lines, and edges in E_X in form of double lines. Also, note that we represent the vertices in two different styles. Vertices representing literals are depicted as circles and vertices representing clauses are depicted as rectangles.

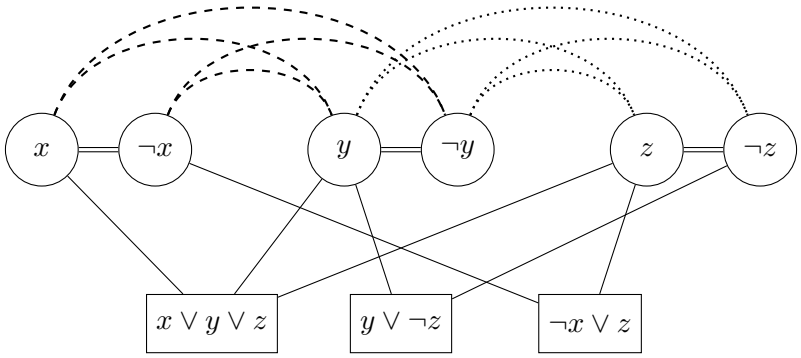


Figure 4.1: Encoding of a QBF in form of an undirected graph

The authors of [XL21] justify this encoding of QBFs in form of undirected graphs with the following three reasons. First of all, the prefix of a QBF holds essential information about the semantics of the formula. Two QBFs with the same matrix but different prefixes can have drastically different semantic evaluations. Therefore, the edges in $E_{\exists,\forall}$ and $E_{\forall,\exists}$ are used to track the sequential information stored in the prefix. Furthermore, while the variables only appear as positive literals in the matrix, they can appear both positively and negatively in the matrix. This naturally leads to representing each variable x_i by two vertices, representing the pair of complementary literals x_i and $\neg x_i$. Finally, the reflexive edges in E_X allow to model the inherent connection between a literal and its negation.

After encoding a QBF in form of an undirected graph, a special *gated graph neural network (GGNN)* [LTBZ15] is used to obtain the predictions in the AlphaZero MCTS. While there are also other possible network architectures that allow to work with graphs [GSR$^+$17, BHB$^+$18], the advantage of GGNNs is their straight-forward implementation.

To use a GGNN, we associate to each vertex $v \in V_\psi$ a *hidden representation*. This hidden representation is a vector of real numbers that numerically quantifies each vertex. Depending on whether a vertex represents a literal or a clause, this hidden message is initialized differently. We provide more information on this initialization in Section 4.3. To then propagate information through the graph, the hidden representation of each vertex v is combined linearly with the hidden representations of all neighbors of v to form *the message*

from vertex v. In this process, the different edge types are treated differently. The computed message is then used to update the hidden representation of v via a gated recurrent unit (GRU) [CVMBB14]. Gated recurrent units, like LSTM cells [HS97], introduce a gating mechanism to a neural network that allows to capture temporal information. This allows the hidden representations of each node, over several iterations of the message passing process just described, to capture the global structure and information of the entire input graph. Finally, after a fixed number of iterations T (the so-called *message passing time*), the final hidden messages are combined by two standard feedforward neural networks to obtain on the one hand the action probabilities (policy network) and on the other hand the winning probability (value network).

Equation 4.1.1 summarizes how the hidden representations and messages are updated at each time step $0 \leq t < T$. In this description, the vector $h_v^{(t)}$ denotes the hidden representation of the vertex v at time t and $m_v^{(t)}$ is the message from v. Furthermore, we denote by $E(v)$ the edges of the vertex v and A_1, A_2, A_3, A_4 are matrices (the so-called *edge-weight matrices*).

$$\begin{aligned} m_v^{(t+1)} &= \sum_{\substack{\{v,w\} \\ \in \\ E(v) \cap E_{\exists,\forall}}} A_1 h_w^{(t)} \; + \sum_{\substack{\{v,w\} \\ \in \\ E(v) \cap E_{\forall,\exists}}} A_2 h_w^{(t)} \; + \sum_{\substack{\{v,w\} \\ \in \\ E(v) \cap E_C}} A_3 h_w^{(t)} \; + \sum_{\substack{\{v,w\} \\ \in \\ E(v) \cap E_X}} A_4 h_w^{(t)} \\ h_v^{(t+1)} &= \mathrm{GRU}(h_v^{(t)}, m_v^{(t+1)}) \end{aligned} \tag{4.1.1}$$

We note that first the messages $m_v^{(t+1)}$ of all vertices $v \in V_\psi$ are computed before the hidden messages $h_v^{(t+1)}$ are updated. Furthermore, the output $h_v^{(t+1)} = \mathrm{GRU}(h_v^{(t)}, m_v^{(t+1)})$ of the gated recurrent unit is computed as follows:

$$\begin{aligned} x &= \sigma(W_x m_v^{(t+1)} + U_x h_v^{(t)} + b_x), \\ r &= \sigma(W_r m_v^{(t+1)} + U_r h_v^{(t)} + b_r), \\ h &= \tanh(W_h m_v^{(t+1)} + U_h(r \odot h_v^{(t)}) + b_h), \\ h_v^{(t+1)} &= (1 - x) \odot h_v^{(t)} + x \odot h, \end{aligned}$$

where $W_x, W_r, W_h, U_x, U_r, U_h$ are matrices, b_x, b_r, b_h are vectors, $\odot$ denotes the Hadamard product (component-wise multiplication) of two vectors and $\sigma(x) = 1/(1+e^{-x})$ is the sigmoid function, which, like tanh, is applied to a vector component-wise.

After the final iteration, the hidden messages are combined as follows to obtain the action probabilities π and the winning probability ν:

$$\begin{aligned} \pi &= \mathrm{softmax}\Big(\sigma\big(f_\pi(h_v^{(T)}, h_v^{(0)})\big) \odot g_\pi(h_v^{(T)})\Big), \\ \nu &= \tanh\Big(\sigma\big(f_\nu(h_v^{(T)}, h_v^{(0)})\big) \odot g_\nu(h_v^{(T)})\Big), \end{aligned}$$

where $f_\pi, f_\nu, g_\pi, g_\nu$ are feedforward neural networks and $\mathrm{softmax}(x)$ normalizes a vector $x = (x_1, \ldots, x_n)$ so that $\mathrm{softmax}(x)_i = e^{x_i} / \sum_{j=1}^{n} e^{x_j}$. In Section 4.3, we will specify more precisely how the feedforward neural networks look like.

To end this section, we make a few observations about GGNNs:

- The networks f_π and g_π together with the shared computation of the hidden representations $h_v^{(T)}$ form the policy network. Analogously, the networks f_ν and g_ν together with the shared computation of the hidden representations $h_v^{(T)}$ form the value network.
- The parameters θ of a GGNN that are updated during training are given by the edge-weight matrices A_1, A_2, A_3, A_4, the parameters $W_x, W_r, W_h, U_x, U_r, U_h,\ b_x, b_r, b_h$ in the GRU and the parameters of the neural networks $f_\pi, f_\nu, g_\pi, g_\nu$.
- The only fixed hyperparameters of such a network are the message passing time T as well as the size of the hidden representations $h_v^{(t)}$ and messages $m_v^{(t)}$. In particular, the size of the graph, that is, the number of vertices and edges, is unrestricted. This allows to use a single GGNN to process all different graph sizes, and consequently all QBFs.
- The computation within a GGNN is invariant under graph isomorphisms, which helps to capture the symmetries inherent to QBFs.

4.2 Asymmetry of QSAT

Since the self-play phase in the AlphaZero framework requires the algorithm to play against itself, it is essential that the underlying game is symmetric. The QSAT game, however, is asymmetric in terms of its winning condition. The existential player tries to make the formula true while the universal player tries to make the formula false. Theoretically this still allows to apply the standard AlphaZero framework to the QSAT game, however, as reported in [XL21] this leads to a very poor performance. The authors believe that this comes from the network getting confused as it has to learn two opposite tasks simultaneously. To solve this problem, we follow the approach taken in [XL21] and use a separate policy and value network for each player. More precisely, we have a policy network $\pi_{\theta_\exists}$ and value network $\nu_{\theta_\exists}$ for the existential player $P_\exists$ and a completely independent policy network $\pi_{\theta_\forall}$ and value network $\nu_{\theta_\forall}$ for the universal player $P_\forall$.

We consider each QSAT instance as an individual game. Given a QBF, we first encode the formula in form of a graph as described in the previous section. Then, the AlphaZero MCTS is used for the action selection of each player based on the player's value and policy network. In each state of the game, that is, for each QBF graph, each player can choose between two possible actions, either setting the outermost variable to $\top$ or to $\bot$. The MCTS takes as input the current state of the game (the QBF graph) and either uses $\pi_{\theta_\exists}$ and $\nu_{\theta_\exists}$ or $\pi_{\theta_\forall}$ and $\nu_{\theta_\forall}$ for the selection and evaluation. After the MCTS returns a promising action (or, during training, a probability distribution over the two possible actions), the QBF graph is updated according to the chosen variable assignment. This process continues until the underlying formula simplifies to $\top$ or $\bot$ with $P_\exists$ winning in the first case and $P_\forall$ winning in the latter case.

Due to the adaption of the AlphaZero framework to integrate two players, we also have to change the learning procedure. In particular, during the self-play data generation phase, we now let $P_\exists$, using the networks $\pi_{\theta_\exists}$ and $\nu_{\theta_\exists}$, play against $P_\forall$, using the networks $\pi_{\theta_\forall}$ and $\nu_{\theta_\forall}$. For each game played, we now store two separate game traces. One for $P_\exists$ consisting of all triples (s_i, p_i, v_i), where s_i is the state of the game, i.e., the QBF graph, at time i, p_i is the distribution computed by the MCTS with s_i as a root state and v_i is the final result of the game from the perspective of $P_\exists$. So, v_i is $+1$ if $P_\exists$ won this game, and -1 otherwise. We note that we only store a triple (s_i, p_i, v_i) for $P_\exists$ if it was the existential player's turn in state s_i. Similarly, we store a second game trace for $P_\forall$ consisting of all triples (s_j, p_j, v_j) with states s_j, MCTS distributions p_j and final results v_j. For the universal player, the result is $+1$ if $P_\forall$ won, and -1 otherwise. Dually to the other game trace, we only store a triple (s_j, p_j, v_j) for $P_\forall$ if it was the universal player's turn in state s_j.

After generating this data, we update the parameters of both players' networks separately using their individual game traces. Then, before starting the next iteration, we also apply the *arena phase* that was used in AlphaGo Zero [SSS+17]. In this arena phase, we evaluate the most recent parameter update. This is done by comparing the performance of the updated existential player $P'_\exists$ to the best previous existential player $P^{\text{best}}_\exists$. In particular, we let $P'_\exists$ play against the previous best universal player $P^{\text{best}}_\forall$ on a random subset of the training data consisting only of satisfiable QBFs. Analogously, we also let $P^{\text{best}}_\exists$ play against $P^{\text{best}}_\forall$ on the same subset of the training data. Finally, we compare the accuracy achieved by $P'_\exists$ to the one of $P^{\text{best}}_\exists$. If $P'_\exists$ wins this comparison, we set $P^{\text{best}}_\exists := P'_\exists$, otherwise we discard the most recent update and set $P'_\exists := P^{\text{best}}_\exists$. Then we repeat the same procedure to compare $P'_\forall$ to $P^{\text{best}}_\forall$, in this case only using unsatisfiable training examples, before starting the next data generation phase.

4.3 Experimental Results

We have implemented an AlphaZero framework for QBF solving following the description of the previous section. Our starting point for this was an open source Python implementation of a vanilla AlphaZero algorithm available at

`https://github.com/suragnair/alpha-zero-general`.

In particular, we extended the implementation with a data structure for QBFs representing the graph encoding from Section 4.1. Additionally, we used the Python machine learning framework PyTorch [PGM+19] to implement gated graph neural networks. Finally, we also implemented Algorithm 3 and integrated it into the adapted learning procedure for AlphaZero. We note that our implementation together with all files necessary to reproduce our experiments is available at

`https://github.com/ClemensHofstadler/MCTS4QBF`.

We used our implementation to train two players for the QSAT game. To this end, we fixed the following specifications of the GGNN.

For each call to the neural network, the hidden representation of each vertex $v \in V_\psi$ of a QBF graph $G_\psi = (V_\psi, E_\psi)$ has to be initialized. This hidden representation depends on what v represents with all hidden representations sharing the same *hidden dimension* $N = 128$. In particular, if v represents an existentially quantified literal its hidden representation is initialized by $h_v^{(0)} = (1, 0, 0, 1, 0, 0, 1, \ldots, 0, 1, 0)$. Similarly, we set $h_v^{(0)} = (0, 1, 0, 0, 1, 0, 0, 1, \ldots, 0, 0, 1)$ if v represents a universally quantified literal and $h_v^{(0)} = (0, 0, 1, 0, 0, 1, \ldots, 0, 1, 0, 0)$ if v represents a clause. This requires the edge-weight matrices A_1, A_2, A_3, A_4 to be of size $N \times N$.

Furthermore, the feedforward networks $f_\pi, f_\nu, g_\pi, g_\nu$ all consist of two fully-connected layers with tanh activation function. The input dimension of f_π and f_ν is $2N$ and the input dimension of g_π and g_ν is N. We note that the networks f_π and f_ν have the input size $2N$ because we concatenate the two inputs $h_v^{(T)}$ and $h_v^{(0)}$ to one vector of size $2N$ and then pass this vector through these networks. The output of the policy network f_π, g_π is a vector of size two, while the output of the value network f_ν, g_ν is a single scalar value. Additionally, we fix the message passing time T to be $T = 10$, which is the same value that was also used in [XL21]. All parameters θ of the GGNN that are updated during training are initialized randomly.

We also set the following hyperparameters for the AlphaZero MCTS. As done in [XL21], we limit the number of MCTS iterations to 25. During the selection phase, we use PUCT as stated in Equation 3.2.2 with the exploration constant $\gamma = \sqrt{2}$. As described in Section 3.2.2, during learning, the AlphaZero MCTS returns a probability distribution over the legal actions computed from the visit counts. We set the temperature parameter τ used in this computation to $\tau = 1$ for the whole game.

As training data, we generate 200 random QBFs (100 satisfiable, 100 unsatisfiable) in PCNF. Each formula has between 19 and 23 bound (and no free) variables, roughly half of which are existentially quantified. Furthermore, the matrix of each formula consists of 7 to 11 clauses, with clause sizes ranging from 2 to 5. While the prefix size and the number of clauses are sampled uniformly from the respective ranges, the clause sizes are sampled from a skewed distribution that favors clauses of larger size.

We execute a total of 30 epochs. After this number of iterations we stop the training as then learning seems to have stalled (see also Figure 4.2). In each epoch, we let the two players play 40 games to generate data. More precisely, we select 40 QBFs randomly from the training data and use these formulas during the self-play phase. Here we deviate slightly from the experiment described in [XL21] where the training data only consists of 20 formulas and each of these formulas is used in every epoch. We decided not follows this approach since it might more likely lead to overfitting than our approach. We note that we also tried different numbers of games but 40 turned out to provide the best compromise between improvements in learning and computational overhead.

The data collected during these games is then used to minimize the loss function with a variant of stochastic gradient descent. In particular, we apply the Adam optimizer [KB14] with an initial learning rate of 10^{-3} and a batch size of 32. The regularization constant c in the loss function is set to $c = 0$. After updating the parameters, we use 20 randomly selected formulas from the training data to compare the updated players to the best previous ones.

Finally, after each epoch, we evaluate the resulting players on an independent evaluation set consisting of 50 QBFs following the same distribution as the training data.

In Figure 4.2, we show how the accuracy of the two players on the evaluation data evolves over the course of the training procedure. Additionally, we also plot the linear regression line as an indicator of the overall trend. As this trend line shows, the accuracy constantly increases, which indicates that the players continuously learn and improve. It is interesting to note that, although the networks are initialized randomly, the players initially perform better than random. In general, one would expect the players to reach an accuracy of roughly 50% in epoch 0 (by epoch 0 we denote the initial evaluation benchmark before starting training). However, the (random) players perform way better than that reaching an accuracy of 70%. This shows that, even with random predictions by the networks, the MCTS can still extract useful information from the game. This comes from the fact that the value obtained in end states during the MCTS, which is independent of the network predictions, is also backpropagated through the search tree.

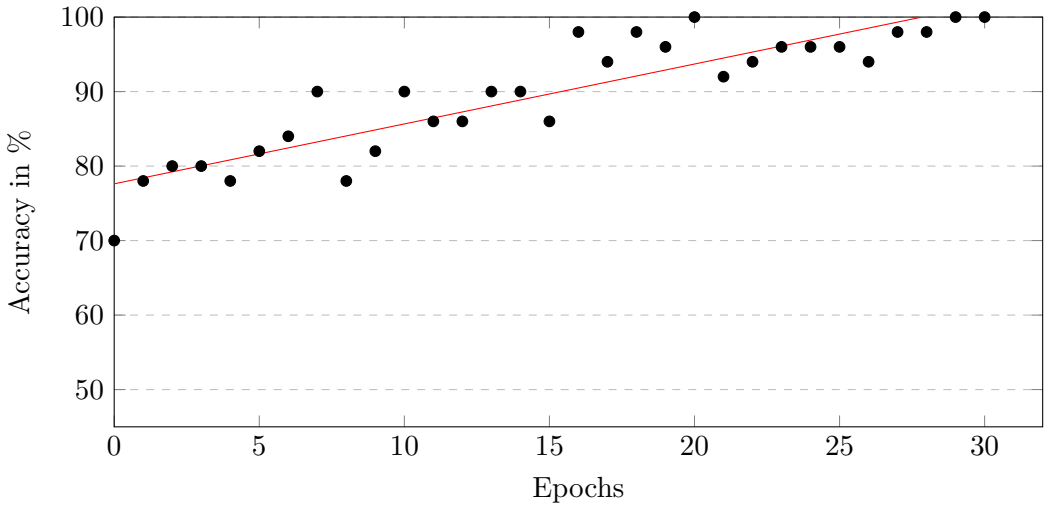


Figure 4.2: Accuracy on the evaluation data during training

In Figure 4.3 and Figure 4.4, we also illustrate the accuracies reached by the players during the comparison phase. As these plots show, the new player does not always outperform the best previous player. Consequently, by adding this comparison phase, we ensure that we always continue working with the best currently available player. This helps to increase the learning speed and efficiency.

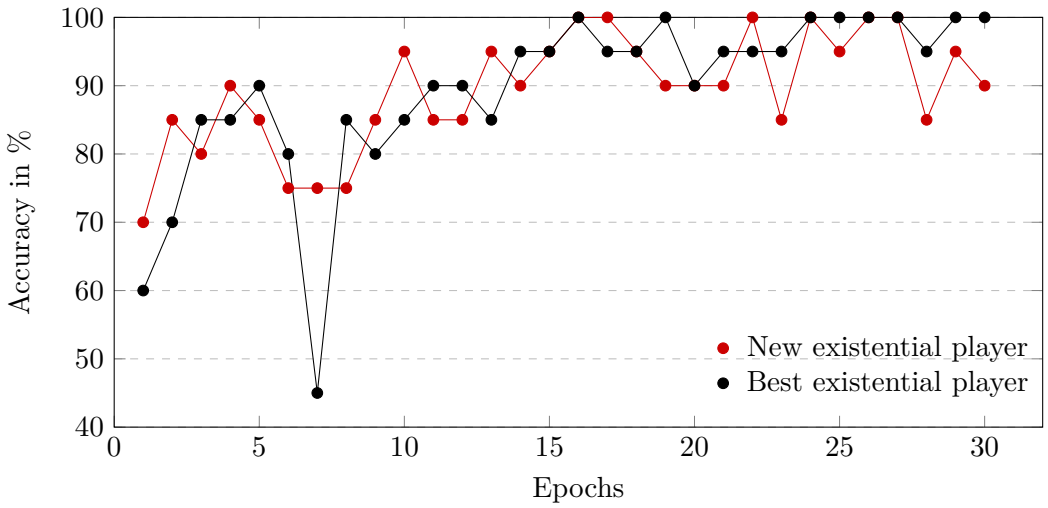


Figure 4.3: Comparison of the existential players

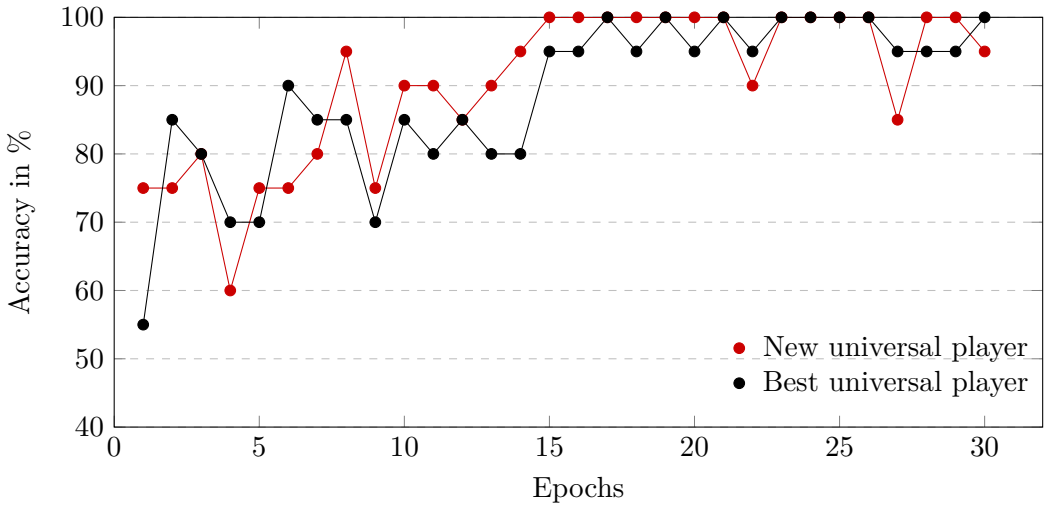


Figure 4.4: Comparison of the universal players

The performance of the final players after 30 epochs on the evaluation dataset is flawless. They reach an accuracy of 100%. To additionally verify this performance on independent data, we use a test dataset consisting of 100 QBFs (50 satisfiable, 50 unsatisfiable) following the same random distribution as the training and evaluation data. On this data, the final players reach an accuracy of 94%. More precisely, of the 6 wrongly classified formulas 3 are false positives, i.e., unsatisfiable formulas that are predicated to be satisfiable, and 3 are false negatives, i.e., satisfiable formulas that are predicted to be unsatisfiable. This evaluation first and foremost shows that, while the AlphaZero players are clearly not perfect players, they

nevertheless provide a fairly accurate tool to determine the satisfiability of QBFs. We note that these results also match what the authors in [XL21] report.

The main drawback of the AlphaZero approach, despite the fact that results are not provable correct, is the computation time. Evaluating the 100 test formulas takes a total of 828 seconds, which is a lot, especially when compared to the 1 second that DepQBF needs for the same task (see also Figure 4.5). Here, however, we have to note that this comparison is in fact not totally fair. DepQBF is a highly optimized software written in C with the main priority being speed, while our AlphaZero implementation is written in Python and uses data structures that are more catered towards flexibility than speed. Furthermore, all computations are done on a CPU. Using a GPU for the neural network computations within AlphaZero could drastically reduce the computation time. Nevertheless, even if our software was fully optimized, it would most likely still be a lot slower than a standard state-of-the-art QBF solver. The main reason for this is that the message passing process in the GGNN (see Equation 4.1.1), which has to be executed for every vertex over several time steps, is computationally very intense. Consequently, each call to the neural network is very expensive.

This brings up the question, whether it is really worth to do these costly computations, or if we can achieve similar results with a standard MCTS that replaces the expensive neural networks with cheap (random) simulations. To answer this question, we separated the two main parts of the AlphaZero framework, the neural networks and the MCTS, and evaluated each of these parts separately on the test data. In particular, in one experiment, we only used the predictions of the policy network of our trained players for the move selection (without any tree search), and in several additional experiments we used different MCTS variants with (semi-)random simulations instead of neural network predictions for the move selection. The results of these experiments are summarized in Figure 4.5.

More precisely, Figure 4.5 plots the time needed to evaluate all 100 test formulas against the achieved accuracy. The data points correspond to the following approaches for the move selection in Algorithm 3:

- **AlphaZero**: Using the trained AlphaZero players after 30 epochs.
- **AlphaZero NN**: Using only the policy networks of the trained AlphaZero players for the move selection without any tree search. Given a state of the game, i.e., a QBF graph, first, the action probabilities are computed with the policy network of the currently acting player. Then the action with the highest probability is chosen and executed.
- **MCTS 25**: Using a standard MCTS with UCT as search method and random simulations (each MCTS consists of 25 iterations).
- **MCTS 50**: Using a standard MCTS with UCT as search method and random simulations (each MCTS consists of 50 iterations).
- **MCTS 2/3**: Using a standard MCTS with UCT as search method and random simulations (each MCTS consists of 25 iterations). In this experiment, the players play in

a best-of-three manner, that is, for each formula several games (at most 3) are played until one of the players gets two wins.

- **PUCT**: Using a MCTS with PUCT as search method and random simulations (each MCTS consists of 25 iterations). In this experiment, Equation 3.2.2 is used as a search method but the action probabilities $\pi_\theta(a|s)$ are replaced by data from random simulations. In particular, in each state s 10 random rollouts are performed for each of the two possible actions ($a_0 = \bot$ and $a_1 = \top$) and the action probability $\pi_\theta(a_i|s)$ is replaced by the relative frequency of wins of the player acting in state s in the 10 random simulations after performing action a_i.
- **MCTS+**: Using a standard MCTS with UCT as search method and semi-random simulations (each MCTS consists of 25 iterations). In this experiment, domain knowledge is integrated into the simulation phase of the MCTS. In particular, unit and pure literal elimination (Lemma 2.17 and 2.18) are applied whenever possible. Variables that cannot be assigned by unit or pure literal elimination are still assigned randomly.
- **DepQBF**: For the sake of completeness, we also add a data point for the performance of DepQBF.

We note that due to the stochastic nature of the standard MCTS variants, the respective data points in Figure 4.5 correspond to the mean accuracy over 10 runs. Additionally, we plot error bars showing the best and worst achieved result. All experiments were performed on a laptop equipped with a 2.7 GHz Quad-Core Intel Core i7-6820HQ Processor with 16 GB of RAM running macOS version 12.0.1. We note that all our Python implementations use Python 3.

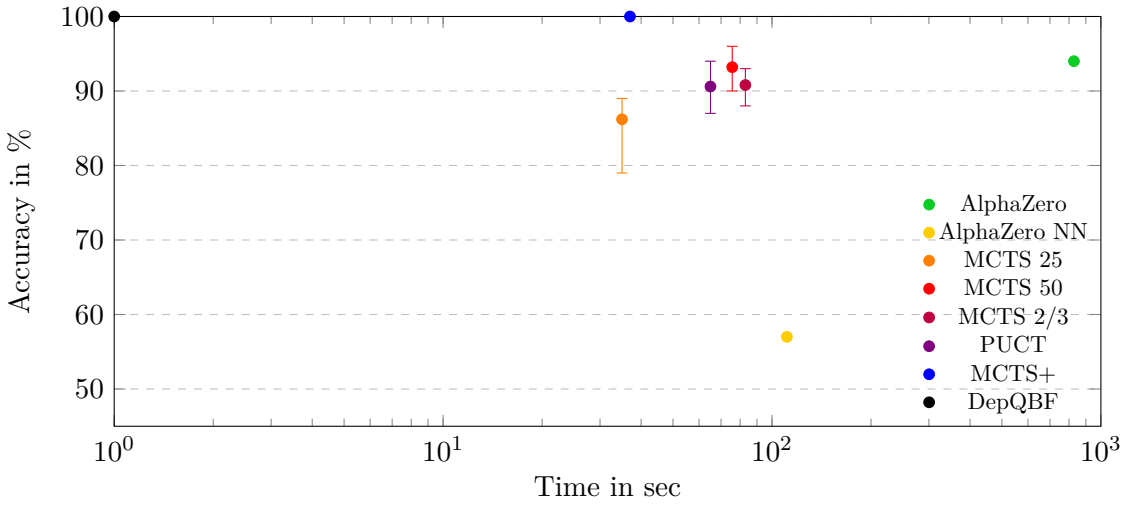


Figure 4.5: Accuracy and timings for solving 100 random QBFs

These experiments show that while the AlphaZero algorithm reaches a slightly higher accuracy than a standard MCTS with 25 iterations (94% vs. 86% on average), the latter

clearly outperforms the former in terms of computation time (828 sec vs. 35 sec). Surprisingly, using the neural networks without any tree search leads to a very poor performance that is just slightly better than random (57%). This shows that the main driver of the good performance of the AlphaZero framework – at least for the problem of QSAT – is the MCTS.

It is noteworthy that the three MCTS variants **MCTS 50**, **MCTS 2/3** and **PUCT** perform almost identically in terms of average accuracy (93% vs. 91% vs. 91%), with the PUCT variant being clearly the fastest of these three (65 sec vs. 76 sec respectively 83 sec). This indicates that it could be worth looking into finding good predictor heuristics to replace the expensive neural network predictions. In this way, one could achieve a similar (or even better) accuracy with drastically less computational effort. Maybe, some of the heuristics that are already successfully used in SAT solving could be used here as well.

The experiments also clearly show that applying domain knowledge in the simulations is definitely advantageous for QSAT, as the MCTS variant with unit and pure literal elimination reaches an accuracy of 100% while only taking marginally longer than the standard MCTS (37 sec vs. 35 sec). Here, it is clearly helpful that these techniques are computationally very cheap to apply. In future work, one could investigate whether it is also beneficial to use more expensive QBF reasoning techniques such as CEGAR-based expansion or QCDCL.

All in all, our experiments show that the AlphaZero framework can be used as a fairly accurate tool to predict the satisfiability of QBFs, thereby confirming the results communicated in [XL21]. However, the expensive neural network computations lead to a long overall computation time and consequently make this approach impractical for any real world applications. This problem can be overcome by replacing the neural network predictions by a standard MCTS with domain knowledge integrated into the simulations. In this way, the computations not only become more efficient but also more accurate results can be achieved.

Finally, we note that all the approaches in our experiments are ultimately black boxes that produce only predictions and in general not provably correct results. Consequently, these techniques alone are insufficient for any real world application where one has to correctly determine the satisfiability of QBFs. In the following chapter, we present a way to integrate a MCTS into a standard QBF solver in such a way that the correctness of the obtained results is ensured.

Chapter 5

MCTS for QBF Solving

The experiments from the previous section have shown that a standard MCTS (extended with some basic variable elimination techniques) provides a fast and accurate way to determine the satisfiability of QBFs. Based on these results, we test a different way to integrate a MCTS into the QBF solving process which ensures correctness of the obtained results.

In particular, we want to use a MCTS as a preprocessing tool to simplify a formula before handing it to a standard QBF solver. The idea here is to use the MCTS to obtain a good assignment of the variables in the outermost quantifier block. If such an assignment can be found, then we can simplify the formula according to it and solve the simplified formula with the QBF solver. We note that this approach only works for satisfiable QBFs with an existential outermost quantifier block, or dually, for unsatisfiable QBFs with a universal outermost quantifier block. It is based on the following proposition which follows directly from the semantic definition of QBF (Definition 2.12).

Proposition 5.1. *Let $\psi = Q_1 B_1 \,.\, \phi \in \mathcal{Q}$ be a QBF with outermost quantifier $Q_1 \in \{\forall, \exists\}$ and $\phi \in \mathcal{Q}$. If $Q_1 = \exists$, then ψ is satisfiable if and only if there exists an assignment $A : B_1 \to \{\top, \bot\}$ such that $\phi[A]$ is satisfiable. Dually, if $Q_1 = \forall$, then ψ is unsatisfiable if and only if there exists an assignment $A : B_1 \to \{\top, \bot\}$ such that $\phi[A]$ is unsatisfiable.*

This proposition leads to the following procedure to determine the satisfiability of a QBF $\psi = Q_1 B_1 \,.\, \phi \in \mathcal{Q}$. First, we play one QSAT game following Algorithm 3 using a standard MCTS for the move selection of both players. We assist the MCTS by universal reduction and apply unit and pure literal elimination whenever possible. During this game, we keep track of the variable assignments. In particular, we collect the assignments of the variables in the outermost quantifier block B_1 in a partial assignment $A : B_1 \to \{\top, \bot\}$. If the game ends with a win for the existential player and $Q_1 = \exists$, we give the simplified QBF $\phi[A]$ as input to a standard QBF solver. If the solver returns that $\phi[A]$ is satisfiable, then based on Proposition 5.1, the original formula ψ must be satisfiable as well. Dually, we can also give the simplified QBF $\phi[A]$ as input to the standard solver, if the QSAT game ends with a win for the universal player and $Q_1 = \forall$. If the solver returns that $\phi[A]$ is unsatisfiable, then so must be ψ according to Proposition 5.1.

However, in all other cases nothing about the satisfiability of the original formula ψ can

be said. This, in particular, includes the case when the result of the game does not match the prefix structure of ψ, i.e., when the existential player wins but $Q_1 = \forall$, or dually, when the universal player wins but $Q_1 = \exists$, or when the result of the QBF solver does not agree with the result of the game, that is, when the existential player wins but $\phi[A]$ turns out to be unsatisfiable, or dually, when the universal player wins but $\phi[A]$ turns out to be satisfiable. We note that the latter situation can appear for example when the partial assignment A computed during the execution of Algorithm 3 is not satisfiability preserving. In all these cases, the standard QBF solver has to be called again with ψ as input. We have summarized this procedure in form of a flowchart diagram in Figure 5.1.

We note that in an unsuccessful attempt of the approach described above, two calls to the QBF solver are needed, and consequently, time is lost compared to directly giving ψ as input to the solver. However, the hope is that in most cases the approach is successful and allows to save some time. The underlying motivation behind this is that one run of Algorithm 3 with an efficient MCTS should be quicker than the procedures that QBF solvers apply. Consequently, this leads to the idea of outsourcing some of the computations to determine the satisfiability of ψ from the standard QBF solver to Algorithm 3. In particular, we hope that the simplified formula $\phi[A]$ can be solved by the solver substantially faster than the original formula ψ. In fact, ideally we would hope that the MCTS is efficient and fast enough so that the overall computation time of running Algorithm 3 and solving the simplified formula $\phi[A]$ is lower than the time the solver would need to determine the satisfiability of ψ directly.

We have implemented the approach described above in the C program MCTSsolve. It is based on the code of the QBF preprocessor Bloqqer [BLS11] and is available together with our other implementations at

`https://github.com/ClemensHofstadler/MCTS4QBF`.

We tested our implementation on 100 random QBFs generated by the QBF fuzzer BlocksQBF[1] which generates random formulas in PCNF according to the model described in [CI05]. Each of these formulas is of the form $\exists B_1 \forall B_2 \exists B_3 \,.\, \phi$ where $|B_1| = 7$, $|B_2| = 12$, $|B_3| = 12$ and ϕ is a propositional formula in CNF consisting of 470 clauses, each with 3 literals from the outermost quantifier block and 2 literals from both the middle and innermost quantifier block. Of the 100 instances 58 are satisfiable.

The QBF solver DepQBF takes a total of 4.37 seconds to evaluate all 100 formulas. In contrast to that, our implementation needs 17.2 seconds for the same task if we do 1000 MCTS iterations for each move selection. Almost 79% (13.6 sec) of this time is spent on the MCTS. Out of the 58 satisfiable formulas, our approach manages to correctly solve 33 using the partial assignment from the MCTS. Additionally, we note that it never happens that a second call to the solver is necessary. However, for precisely a quarter of the formulas the prediction from Algorithm 3 does not match the actual semantic evaluation of the formula. In Figure 5.2, we plot for each formula the logarithm of the ratio of the solving time needed by our approach compared to DepQBF. As this figure shows, only three out of the 100 formulas can be solved quicker using our approach. For all other instances, our approach is slower than DepQBF and for 15 formulas it is even more than 10-times slower. While it is

[1] available at `http://fmv.jku.at/blocksqbf/`

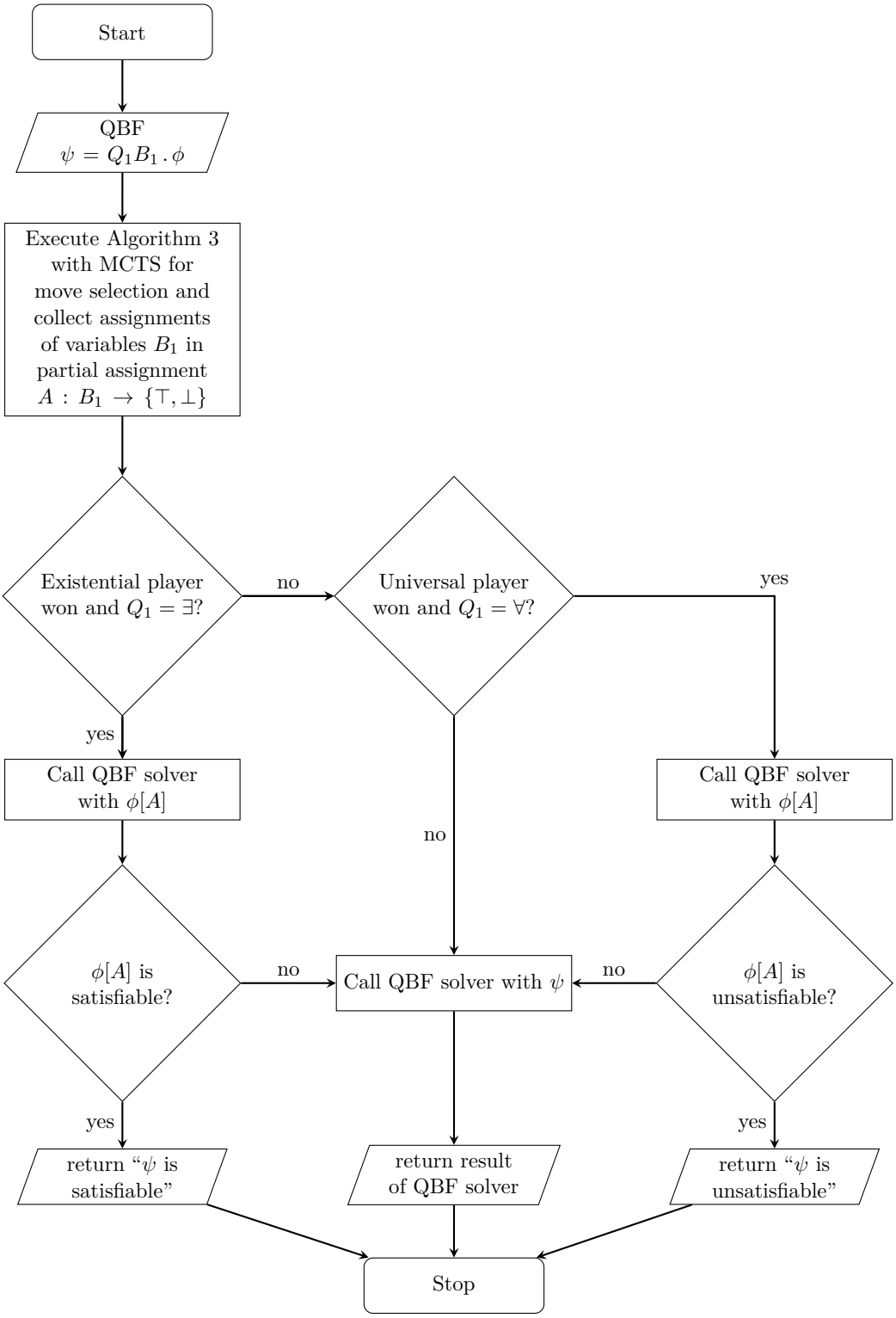


Figure 5.1: Flowchart of our approach to QSAT involving MCTS

clear that our approach cannot be faster than DepQBF for the unsatisfiable instances (this follows from the prefix structure of the formulas), it is surprising that the same apparently also holds for the satisfiable formulas. It is particularly notable that the biggest discrepancies in solving time are for satisfiable instances as can be seen in Figure 5.2. This data clearly shows that our approach is not competitive against state-of-the-art QBF solvers.

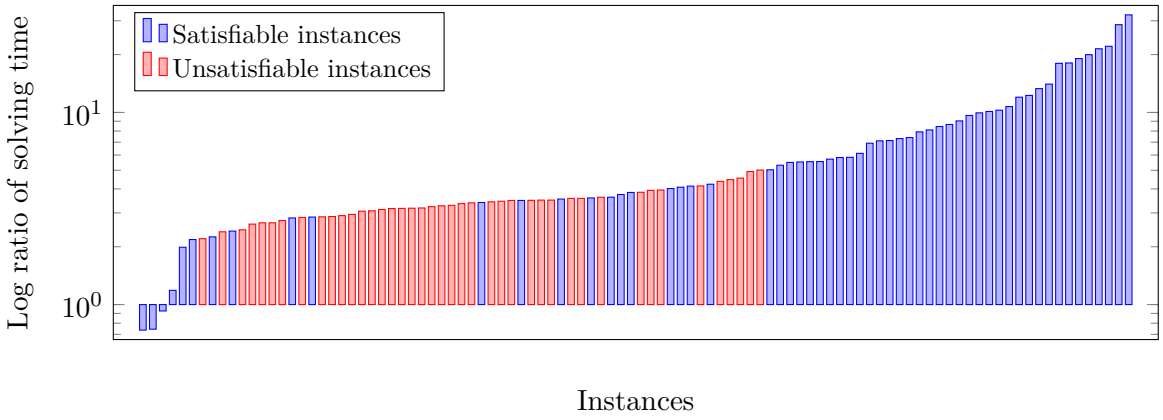


Figure 5.2: Log ratios of solving times for 100 random formulas

We note that we also applied our approach to the QBFs from the crafted instances track of QBFEVAL'20 but for each of these formulas either the prefix structure did not match with the semantic evaluation (existential outermost quantifier block but unsatisfiable formula or universal outermost quantifier block but satisfiable formula) or Algorithm 3 yielded a wrong prediction. Consequently, also for all these instances our approach was slower than DepQBF.

Although our previous experiments did not lead to competitive results, we want to further explore the possibility of including a MCTS into the QBF solving process. One idea is to utilize information about an input formula $\psi \in \mathcal{Q}$ gained during an execution of Algorithm 3 with MCTS to determine when to switch from one solving approach to another. A MCTS allows to compute for each variable (that cannot be assigned by unit or pure literal elimination) an estimate of the winning probability when setting this variable to $\top$ or $\bot$. We want to use these probabilities as a stopping criterion for the splitting phase in the cube and conquer approach implemented in the distributed solver Paracooba [HFB20]. In particular, we want to investigate whether it is beneficial to stop the splitting of a given formula once these probabilities strongly deviate from $50 : 50$. We suspect that such a deviation could indicate that the formula becomes easy to solve, and hence, can be solved directly without any further splitting. Additionally, we also want to explore whether the percentage of the search tree covered by the MCTS can be used as a metric to determine when to switch from splitting a formula to solving the subformulas.

Chapter 6

Conclusion and Outlook

To summarize, in this work we adapt the famous AlphaZero framework to the application of QSAT following the approach described in [XL21]. To this end, we consider the satisfiability problem of QBF as a game between an existential and a universal player. We train two players for this game using the adapted AlphaZero algorithm, where QBFs are encoded in form of undirected graphs and these graphs are used as input to gated graph neural networks.

One of our main contributions in this work is to test and evaluate this approach. We show empirically that the trained AlphaZero players provide a very accurate, yet not perfect tool to determine the satisfiability of QBFs. This coincides with the results reported in [XL21]. However, our experiments also reveal a main drawback of this approach: long computation times caused by expensive calls to the neural networks. In further experiments, we verify that a classical MCTS (without any neural networks) can achieve similar results to the AlphaZero players while only needing a fraction of the computation time. We note that different variants of MCTS perform very similarly on our task of QSAT. Furthermore, if assisted by some basic variable elimination techniques, the classical MCTS can even achieve 100% accuracy on our test data.

A second problem of the AlphaZero approach is that it is ultimately a black box algorithm that cannot produce provably correct results. To overcome this problem, we propose a different way to include MCTS into the QBF solving process as the second main contribution of this work. More precisely, we use the QSAT game with a classical MCTS for move selection as a preprocessing step before handing a QBF over to a standard QBF solver. During the game, we collect the assignments of the variables in the outermost quantifier block. If the result of the QSAT game matches the prefix structure of the formula, we use this partial assignment to simplify the formula and then solve the simplified formula with the standard solver. We test this approach on randomly generated QBFs. Our experiments show that, while for many of the formulas a correct partial assignment can be found that allows to simplify the formula, our approach is still almost always slower than giving the formula directly to the QBF solver without any MCTS preprocessing.

In future work, we want to explore further possibilities of supporting a standard QBF solver by MCTS. In particular, we want to use MCTS to gain information and gather statistics about an input formula and then use this data to guide the solver.

Bibliography

[ACBF02] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine learning*, 47(2):235–256, 2002.

[ANW19] Michael H. Albert, Richard J. Nowakowski, and David Wolfe. *Lessons in Play: an Introduction to Combinatorial Game Theory*. CRC Press, 2019.

[BB09] Hans Kleine Büning and Uwe Bubeck. Theory of Quantified Boolean Formulas. In *Handbook of Satisfiability*, pages 735–760. IOS Press, 2009.

[BHB+18] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[BKF95] Hans Kleine Büning, Marek Karpinski, and Andreas Flogel. Resolution for Quantified Boolean Formulas. *Information and computation*, 117(1):12–18, 1995.

[BL99] Hans Kleine Büning and Theodor Lettmann. *Propositional Logic: Deduction and Algorithms*, volume 48. Cambridge University Press, New York, USA, 1999.

[Blo20] Erik Blomqvist. Playing the Game of Risk with an AlphaZero Agent. Master's thesis, School of Electrical Engineering and Computer Science, Sweden, 2020. Available at `https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1514096&dswid=7969`.

[BLS11] Armin Biere, Florian Lonsing, and Martina Seidl. Blocked Clause Elimination for QBF. In *International Conference on Automated Deduction*, pages 101–115. Springer, 2011.

[BPW+12] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[CGS98] Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. *AAAI/IAAI*, 98:262–267, 1998.

[Cha10] Guillaume Maurice Jean-Bernard Chaslot. *Monte-Carlo Tree Search*. PhD thesis, Maastricht University, Netherlands, 2010.

[CHJH02] Murray Campbell, A. Joseph Hoane Jr., and Feng-hsiung Hsu. Deep Blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[CI05] Hubie Chen and Yannet Interian. A Model for Generating Random Quantified Boolean Formulas. In *IJCAI*, pages 66–71, 2005.

[Coo71] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[Cou11] Rémi Coulom. Crazy Stone Go program. `https://www.remi-coulom.fr/CrazyStone/`, 2011. Accessed: 03.12.2021.

[CSGG02] Marco Cadoli, Marco Schaerf, Andrea Giovanardi, and Massimo Giovanardi. An Algorithm to Evaluate Quantified Boolean Formulae and Its Experimental Evaluation. *Journal of Automated Reasoning*, 28(2):101–142, 2002.

[CVMBB14] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[GSR+17] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural Message Passing for Quantum Chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[Ham13] John Hammersley. *Monte Carlo Methods*. Springer Science & Business Media, 2013.

[HFB20] Maximilian Heisinger, Mathias Fleury, and Armin Biere. Distributed Cube and Conquer with Paracooba. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 114–122. Springer, 2020.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[Jan18] Mikoláš Janota. Towards generalization in QBF solving via machine learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[JK12] Mikoláš Janota and William Klieber. RAReQS - Recursive Abstraction Refinement QBF Solver. `http://sat.inesc-id.pt/~mikolas/sw/areqs/`, 2012.

[JKMSC12] 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.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KM75] Donald E. Knuth and Ronald W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[KS18] Manuel Kauers and Martina Seidl. Symmetries of Quantified Boolean Formulas. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 199–216. Springer, 2018.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[LaV06] Steven M. LaValle. *Planning algorithms*. Cambridge university press, 2006.

[LB10] Florian Lonsing and Armin Biere. DepQBF: A Dependency-Aware QBF Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):71–76, 2010.

[Lon12] Florian Lonsing. *Dependency Schemes and Search-Based QBF Solving: Theory and Practice*. PhD thesis, Johannes Kepler University Linz, Austria, 2012. Available at `http://www.florianlonsing.com/diss/index.html`.

[LTBZ15] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[MIG96] Hitoshi Matsubara, Hiroyuki Iida, and Reijer Grimbergen. Chess, Shogi, Go, natural developments in game research. *ICCA journal*, 19(2):103–112, 1996.

[MS73] Albert R. Meyer and Larry J. Stockmeyer. Word Problems Requiring Exponential Time. In *5th Symposium on the Theory of Computing*, pages 1–9. Association for Computing Machinery, New York, 1973.

[MS19] Glennn Moy and Slava Shekh. The Application of AlphaZero to Wargaming. In *Australasian Joint Conference on Artificial Intelligence*, pages 3–14. Springer, 2019.

[MSS99] Joao P. Marques-Silva and Karem A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[MU49] Nicholas Metropolis and Stanislaw Ulam. The Monte Carlo Method. *Journal of the American statistical association*, 44(247):335–341, 1949.

[OK09] Yoji Ojima and Hideki Kato. Zen Go program. `https://senseis.xmp.net/?ZenGoProgram`, 2009. Accessed: 03.12.2021.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[PS19] Luca Pulina and Martina Seidl. The 2016 and 2017 QBF solvers evaluations (QBFEVAL'16 and QBFEVAL'17). *Artificial Intelligence*, 274:224–248, 2019.

[QBF05] QBFLIB. QDIMACS Standard v1.1. `http://www.qbflib.org/qdimacs.html`, 2005.

[Ros11] Christopher D. Rosin. Multi-armed Bandits with Episode Context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

[SAH+20] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[SBPS19] Ankit Shukla, Armin Biere, Luca Pulina, and Martina Seidl. A Survey on Applications of Quantified Boolean Formulas. In *Proceedings of the 31st International Conference on Tools with Artificial Intelligence*, pages 78–84, 2019.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[SHS+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[SLB+18] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT Solver from Single-Bit Supervision. *arXiv preprint arXiv:1802.03685*, 2018.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the Game of Go without Human Knowledge. *nature*, 550(7676):354–359, 2017.

[XKL21] Ruiyang Xu, Prashank Kadam, and Karl Lieberherr. First-Order Problem Solving through Neural MCTS based Reinforcement Learning. *arXiv preprint arXiv:2101.04167*, 2021.

[XL20] Ruiyang Xu and Karl Lieberherr. Learning self-play agents for combinatorial optimization problems. *The Knowledge Engineering Review*, 35, 2020.

[XL21] Ruiyang Xu and Karl Lieberherr. Solving QSAT problems with neural MCTS. *arXiv preprint arXiv:2101.06619*, 2021.

[ZM02] 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.

[ZY20] Hongming Zhang and Tianyang Yu. AlphaZero. In *Deep Reinforcement Learning: Fundamentals, Research and Applications*, pages 391–415. Springer, 2020.

[ZYY+20] Xiaofei Zou, Ruopeng Yang, Changsheng Yin, Zongzhe Nie, and Huitao Wang. Deploying tactical communication node vehicles with AlphaZero algorithm. *IET Communications*, 14(9):1392–1396, 2020.