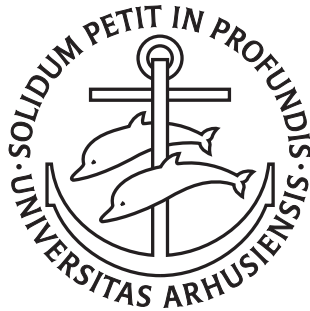# Concise Encodings for
Planning and 2-Player Games

Irfansha Shaik

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Concise Encodings for Planning and 2-Player Games

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Irfansha Shaik
August 25, 2023

Abstract

Given a description of some world, a planning problem is finding a sequence of actions that transform some initial state to some goal state. We focus on two planning problems, Classical Planning and 2-Player games. Classical planning is the simplest planning problem with a single initial state and deterministic actions. 2-Player games can be seen as planning with an adversary, where two players take turns to choose actions. In this work, we focus on providing compact encodings that preserve the structure and allow efficient performance from domain-independent solvers.

In classical planning, heuristic based approaches dominate when optimality is not needed. Propositional Satisfiability (SAT) based planning is a good alternative when optimality is needed. A planning problem can be translated into a propositional formula and solved using successful SAT solvers. However, Hard-to-ground problems with many action and predicate parameters can result in massive propositional formulas. Quantified Boolean Formula (QBF) Logic is an extension of propositional logic with existential and universal quantifiers that allow compact representations. We contribute a QBF encoding for the planning problem that generates a compact encoding which avoids the encoding-size blowup via propositional translation. Using our lifted encoding, we were able to solve Hard-to-ground Organic Synthesis problems that could not be handled using existing SAT/QBF based methods. We also consider an application, *Optimal Layout Synthesis* for Quantum Circuits and model it as a classical planning problem. By using efficient classical planners, we are able to outperform existing leading Satisfiability Modulo Theories (SMT) based exact methods.

In 2-player games, we focus on solving games and computing winning strategies where a player can win regardless of the opponent's moves. First, we consider positional games, a subset of games where one can only occupy an empty position on the board. We generate compact QBF encodings with implicit board and goal constraints compared to the existing encodings. Second, we provide a Board-Game Domain Description Language (BDDL) for allowing domain independent solving for a subset of games played on grid. We provide a general translation from BDDL to QBF so that one can use QBF solvers for generating winning strategies.

Finally, validation of QBF encodings is not trivial due to the presence of quantifier alternations. We propose interactive play with certificates/QBF solvers for the validation of QBF encodings. We also propose techniques for allowing equivalence checks between two similar QBF encodings with winning strategies. For scalability, we use partial certificates for validation and equivalence checking.

Resumé

►**Resumé på dansk**◄

Acknowledgments

Doing research and contributing to the scientific community has always been my dream. But dreams do not become reality without people guiding and supporting you. First and foremost, I would like to thank my PhD advisor Jaco van de Pol from the bottom of my heart. Thank you for always being there when I needed the most, and for being a good person to look up to. Thank you for trusting me and giving me freedom for exploring different ideas, the journey would have been infinitely less fun without you. I enjoyed the brainstorming sessions and leisure walks throughout the years. All the effort you put into your students and classes never goes unnoticed.

Next I would like to thank my professors Krishna Prasad and Krishna Mohan from Undergraduate degree for their support. I would like to thank my office mates Steffan Sølvsten, Sergei Stepanenko and others for bearing me all these years. I would also like to thank Mikael Dahlsen-Jensen for translating the abstract into Danish. Research papers and research vists are a big part of the PhD. I would like to thank my co-authors Martina Seidl, Maximilian Heisinger, Valentin Eichberger, and Abdallah Saffidine. Especially, Martina Seidl for hosting my research visits at JKU. I would like to thank administrative staff for all the help during the PhD, especially Henriette Farup, Sofia Rasmussen, Julie Rasmussen, and Sututhi Perrananthasivam.

Friends have always been a big part of my life. The PhD journey was by no means smooth, especially with all the lockdowns and quarantines. I would like to thank Priyanka Sadineni, Mahidhar Viswanadhapalli, Thomas van Veelan, and others for all the Zoom calls. I would like to thank Niklas Scheel, Ping'an Cheng, Mads Larsen, and others for all the games we played and the walks we went on. I am happy to have you all in my life, I would be bored without you.

I would like to thank my mother Shahanaz Shaik for all the financial support throughout the education. Despite the tough financial situation, you made sure to provide good education so thank you for that.

Finally, I would like to thank my partner, Hasti Dienar Uetami, for being there everyday. I feel content with you in my personal life, and it helps me focus on work without worry. Thank you for being patient with me, during thesis writing and busy paper deadlines.

Irfansha Shaik,
Aarhus, August 25, 2023.

Contents

Part I

Overview

Chapter 1

Introduction

1.1 Motivation For Automated Planning

Automated Planning is part of our everyday life, from scheduling to space exploration there are applications everywhere [49]. Given a description of a world with actions, initial states and goal states, automated planning is finding a sequence of actions such that the initial state is transformed to some goal state. Automated planning covers from the simplest planning problem, where the actions are deterministic and single initial state to planning with an adversary like 2-player games. Some standard variations of planning are conformant planning, contingent planning, temporal planning etc. Mission critical systems like autonomous cars, spacecrafts and rovers use planning modules of some form. Usually planning systems based on machine learning are used when scalability is important. While such techniques scale to real world scenarios, for example vision based planning, it comes at the cost of correctness. Provable and verifiable planning is of utmost importance in the applications with mission critical systems. Similarly, complete search space exploration is needed in any application where optimality is required. In this work, we focus on two planning problems classical planning and 2-player games. Especially, we focus on using techniques that ensure optimality when needed and provide proofs for the results.

1.2 Domain Independent Solving

The main success in several problems of automated planning such as classical planning and general game playing, is due to domain independent solving. Domain independent solving attempts to solve problems by separating the problem-solving into two main parts. As shown in Figure 1.1, first an application problem is encoded to some standard format which then is solved using standard solvers. This achieves two things, first encoding/formulating a problem in some representation is much easier than developing an optimized program. Second, since one fixes the representation, many solving techniques can be explored and highly optimized solvers can be developed. There has been great success in Classical Planning [49], SAT solving [12, 41], Model
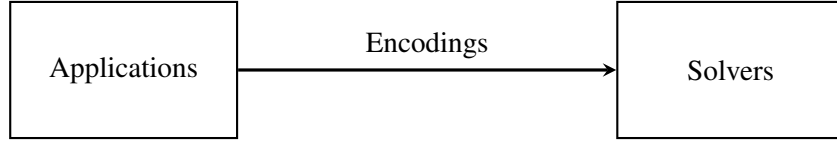
Figure 1.1: Domain Independent Solving

Checking [28] and General Game Playing [39, 46].

In planning problems, heuristic search based approaches dominate when optimal plans are not needed. However, when optimization problems are considered such search based methods can suffer from time and mainly memory bottlenecks. In such cases, Propositional Satisfiability (SAT) based methods can be considered, which are known for low memory usage. In propositional logic, a formula is made of boolean variables and connectives Negation ($\neg$), And ($\wedge$) and Or ($\vee$). A planning problem can be encoded as a propositional formula and optimized SAT solvers can be used for solving them. SAT solvers can also generate proofs for the results, which can be used for proving correctness. Quantified Boolean Formula (QBF) logic, extends propositional logic with universal and existential quantification over boolean variables. A SAT formula can be seen as a special case of QBF with only existentially quantified variables. One could use QBFs for compact encodings when SAT encodings become too large. More importantly, QBF allows natural representation of opponent moves using universal variables in 2-player games. In this work, we mainly focus on SAT and QBF solving for the considered planning problems. We briefly discuss SAT and QBF based solving in the following subsections.

1.2.1   Propositional Satisfiability

Given a boolean formula, a propositional satisfiability problem is to find if there exists an assignment of each variable such that the formula is satisfied. If no such assignment exists then the problem is unsatisfiable. For example, consider a formula $\phi_1 = \exists x \exists y \, x \leftrightarrow y$. By assigning x to True and y to True, we can satisfy the formula ϕ_1 and the formula is satisfiable. Figure 1.2 shows the search tree of the formula ϕ_1, where there exist two solutions (by looking at the leaves). A SAT solver only needs to find one such solution to show the formula is satisfiable. This problem is proved to be NP-complete by Stephen Cook [29] and Levin [76] independently.

Despite being an NP-complete problem, SAT solvers can handle instances with millions of variables and clauses in practice [12]. Especially, there has been great success with industrial problems such as Classical Planning [71], Bounded Model Checking [10] and Hardware verification [12]. In recent years, several open mathematical problems are solved using SAT solvers. The success mainly comes from extensive research in solving techniques and their improvements [12]. SAT competitions are held every year for the last 2 decades with many applications as benchmarks and new solvers. Recent focus in SAT solving has been in certifying the results and proving the correctness of the solvers themselves. [41] discusses the key aspects of SAT solving,
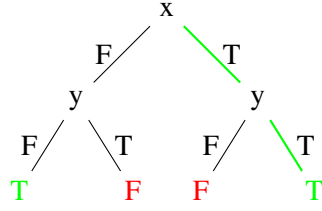
Figure 1.2: Search Tree for Formula ϕ_1, Satisfiable

its success and challenges.

While solving techniques and engineering of SAT solvers play a major role, better encoding techniques are essential too. It is consensus that SAT solvers are able to take advantage of the problem structure, and good encodings are crucial for successful SAT solving. One of the bottlenecks in encodings in SAT is their large encoding sizes, especially when the structure of the problems is not clear. In practice, we often see that adding auxiliary variables can decrease number of clauses and also improve performance. Alternatively, one could encode the same problems compactly using Quantified Boolean Formulas, Answer Set Programming and others. In the next subsection, we discuss QBF solving which we will mainly use in this work for compact encodings.

1.2.2 Quantified Boolean Formulas

Given a closed QBF i.e., a QBF with all variables quantified, the QBF problem is to check if the formula is a True formula or a False formula. For example, consider a formula $\phi_2 = \exists x \forall y \, x \leftrightarrow y$. The formula is a False formula as there is no assignment of x such that the formula is True for both assignments of y. Figure 1.3 shows the complete search tree of the formula ϕ_2. We can see that no value of x results in two satisfied leaves for y. One could encode the same problem in SAT by using two copies of y instead of one universal variable, for example:

$$\exists x \, \exists y_1, y_2$$
$$(x \leftrightarrow y_1) \wedge (x \leftrightarrow y_2) \wedge$$
$$(\neg y_1) \wedge (y_2)$$

The clauses $(\neg y_1)$ and (y_2) force variable y_1 to False and y_2 to True, thus explicitly representing 2 possible instantiations of the universal variable y. Similarly, every QBF can be expanded to a propositional formula with an exponential blowup in certain problems. Also, every propositional problem can be encoded into QBF, sometimes with exponentially compact encodings [73]. QBF solving being a relatively new field promises compact encodings compared to SAT and improved solving when SAT encodings get too large. Several applications exist for QBF solving such as Classical Planning [26, 91, 103], Conformant Planning [102, 104], Model Checking [10, 33] and Formal Verification [8].
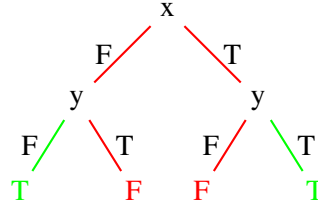
Figure 1.3: Search Tree for Formula ϕ_2, False formula

1.3   Overview of Structure of the Thesis

In this work, we consider two important planning problems, classical planning and 2-player games. We mainly focus on three objectives: (1) Generating concise encodings while preserving the structure of the problems; (2) Ensuring scalability for the encodings with existing SAT/QBF solvers; (3) Allowing validation of the encodings and verification of the results generated.

The thesis is divided into two parts. The first part consists of four chapters discussing and introducing the main topics of research. In each chapter, we first introduce the problem and give a brief overview of existing work. Then we specify our research questions more clearly and finally present an overview of our contributions. In Chapter 2, we consider classical planning problem mainly using SAT/QBF based techniques. In Chapter 3, we consider the problem of encoding 2-player games in QBF. In Chapter 4, we consider the problem of validation of QBF encodings. The second part consists of 5 chapters with 3 conference and 2 Arxiv papers listed below. The papers are added in the thesis with minor format adjustments to Chapters 5-9.

- *Classical planning as QBF without grounding.* Irfansha Shaik and Jaco van de Pol, In ICAPS, pages 329–337, AAAI Press, 2022 [111]. This paper is included in Chapter 5.

- *Optimal Layout Synthesis for Quantum Circuits as Classical Planning.* Irfansha Shaik and Jaco van de Pol, In ICCAD'23, IEEE/ACM, 2023 [113]. This paper is included in Chapter 6.

- *Implicit State and Goals in QBF Encodings for Positional Games.* Irfansha Shaik, Valentin Mayer-Eichberger, Jaco van de Pol, and Abdallah Saffidine, CoRR, abs/2301.07345, 2023 [115]. This paper is included in Chapter 7.

- *Concise QBF Encodings for Games on a Grid.* Irfansha Shaik and Jaco van de Pol, arXiv preprint arXiv:2303.16949, 2023 [112]. This paper is included in Chapter 8.

- *Validation of QBF Encodings with Winning Strategies.* Irfansha Shaik, Maximilian Heisinger, Martina Seidl, and Jaco van de Pol, In SAT 2023, volume 271 of LIPIcs, pages 24:1–24:10 [114]. This paper is included in Chapter 9.

Chapter 2

Classical Planning

Classical Planning is the simplest planning problem, where the actions are deterministic and there is only a single initial state. For example, consider a Blocks-World problem where blocks can either be on a table or on another block. There are two actions, `stack` for stacking a block onto another block and `unstack` for placing a block on the table. Figure 2.1 shows the initial state (on the left) of a sample problem and a plan with 2 actions that transform it to the goal state (on the right). We will use this example throughout the chapter for explaining the differences in encodings. While classical planning is a PSPACE-complete problem [24], subset of problems with polynomial length plans are still in NP. Despite the hardness of the problem in the worst case scenario, many practical problems can be solved using classical planning techniques. In this chapter, we focus on solving some challenges in SAT/QBF based planning and modelling problems in classical planning.

2.1 Existing Work

2.1.1 State of the Art in Classical Planning

The problems are modelled using variations of Planning Domain Definition Language (PDDL) [85]. The PDDL is an action based description of the world, where the state is represented using predicates, and actions change the world. The PDDL description
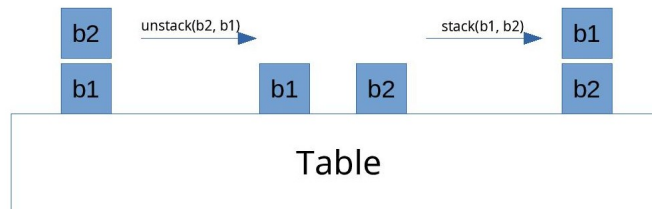

Figure 2.1: An example Blocks-World Problem, with a sample plan

---

Listing 1 blocks-world domain

```
1       (define (domain blocksworld)
2       (:predicates (clear ?y₁)(ontable ?y₁)(on ?y₁ ?y₂))
3       (:action unstack
4        :parameters (?x₁ ?x₂)
5        :precondition (and (clear ?x₁)(on ?x₁ ?x₂))
6        :effect (and (not (on ?x₁ ?x₂))(ontable ?x₁)(clear ?x₂)))
7        ...)
```

Listing 2 blocks-world problem

```
1       (define (problem BW_rand_2)
2       (:domain blocksworld)
3       (:objects b₁ b₂)
4       (:init (ontable b₁) (on b₂ b₁) (clear b₂))
5       (:goal (and (on b₁ b₂))))
```
---

is in (almost) first order representation with parameters, which can result in compact descriptions of the applications. For the blocks-world problem, Listing 1 shows a domain specification with predicates and a single action (out of 2). Here an action specifies parameters, preconditions for the actions to be applied, and effects after the action is applied. Listing 2 shows the problem specification with Initial state, Goal state, and Objects for the same problem. The complete PDDL specification is provided in the Listing 3 of Chapter 5. Given a description of a world in PDDL, the planners generate a sequence of actions which can transform the given initial state to some goal state. External plan validators like VAL [61] exist for validation of the plans generated.

Solver Paradigms   Mainly two solver paradigms exist, optimizing and satisfying solvers. Given some cost to actions, the goal of the optimizing solvers is to generate a plan with optimal cost. For returning an optimal plan, essentially the complete search space has to be explored. As expected, optimal planners can suffer from time and memory challenges especially for real-world problems. Satisfying solvers on the other hand focus on generating a valid plan quickly, mostly ignoring the plan cost. Some satisfying solvers such as Lama [100] and Fast Downward [57] can improve costs with more time.

Main Solving Approaches   There are three main solving approaches in classical planning, search, SAT, and symbolic based solvers. In search based solvers, the main idea is to explore search space using heuristics which are very effective in satisfying planning. In case of optimal solving, the whole search has to be explored where SAT and Symolic based solvers are effective. In SAT based planning, the planning problem is encoded as a bounded reachability problem in propositional logic and SAT solvers are used for finding plans. SAT solvers are known for lower memory usage and can be effective for optimization problems. Symbolic based solvers represent the

state space using compact data structures such as Binary Decision Diagrams (BDDs), enabling reasoning on set of states instead of exhaustive search. In recent years, Symbolic planning has proved to be an effective technique especially in cost-optimal planning [121, 134].

Lifted Planning   Many planners first translate the PDDL description in a propositional problem by a procedure called grounding. When grounding, one replaces the parameters of the actions and predicates with the corresponding objects. For example, in our blocks-world domain there are 2 objects b_1, b_2. As part of grounding, we generate 4 possible grounded actions for both the actions since they have two parameters. Similarly, predicates are also grounded resulting in 4 propositions for the predicate on and 2 propositions each for the predicates clear and ontable. This can blow up especially in real-world domains with many action and predicate parameters. In recent years, many lifted planning techniques are proposed where the grounding is avoided partially or completely [30, 31, 60, 75].

2.1.2   Classical Planning as SAT

Classical Planning is one of the first successful applications of the SAT solving. One could encode a classical planning problem as a bounded reachability problem in a graph [70]. Each state can be encoded using propositional variables, the standard way is to first ground the problem from PDDL. For finding a plan of length k, we generate $k + 1$ copies of a symbolic state where assignments of propositional variables results in all possible states. We encode the transition function between states using the actions described in the PDDL. For example, to encode our block-world instance with a plan of length 2, we specify 3 states. We generate constraints for Initial and Goal states, and 2 copies of transition functions at time steps $0, 1$. The following formula specifies the structure of a SAT encoding for a plan of length 2.

$$\exists S_0, S_1, S_2$$
$$I(S^0) \wedge T(S^0, S^1) \wedge T(S^1, S^2) \wedge G(S^2)$$

We can see that, problems with longer plans need several copies of the transition function. In §2.1.2, we provide a SAT encoding for a planning problem in more detail. In literature, several ideas were proposed to reduce encoding size and improved performance such as Invariants [105], Parallel Plans [72, 107], Operator Splitting [70], Frame Axioms [52], Symmetries [106] and planning specific heuristics [106]. We refer to Chapter 15 of the Handbook of Satisfiability [106] for an extensive discussion on Planning as SAT. In this Subsection, we briefly discuss two key techniques that are used in leading SAT based planners.

Invariants   It is possible to avoid some unreachable states in the search space by considering the properties of predicates specified. For example, we know a pair of predicates open and not-open can never be true in any state. One could generate

constraints based on domain specific information which may not be discovered by the
SAT solvers. For example, [105] discusses invariant computation which is used in a
leading SAT based planner Madagascar [101].

Parallel plans   One of the main bottlenecks in standard sequential encoding is
increasing encoding size. [14, 35] proposed to allow multiple independent actions at
each time step. The first encoding in parallel plans is called ∀-Encoding. Here the
idea is to allow two or more grounded actions that do not conflict, in other words
effects of one action must not conflict with preconditions of the other actions. Since,
the set of actions do not conflict each other any order of actions can result in a valid
plan. This restriction is relaxed a bit in the ∃-Encoding, where multiple actions are
allowed in the same time step as long as there exists one valid action sequence. These
two ideas were formalized and applied in the context of SAT based planning by [72]
and [107] respectively.

2.1.3   Classical Planning as QBF

Despite improvements like splitting actions, Invariants, and parallel plans, the SAT
encoding still needs copies of transition functions for each time step. If there are
many actions and actions with several parameters, the clauses to encode transition
functions can get quite large. Encoding in QBF is considered as an alternative, as
one can generate compact encodings compared to SAT encodings. While QBF based
planning is also used in the context of conformant planning for modelling uncertainty
in the initial state, we focus on QBF encodings for classical planning here. The main
existing idea is to avoid copies of transition functions using universal variables. For
example, let ϕ be some function and $\phi(a) \wedge \phi(b)$ be a very large propositional formula.
We could use universal quantifiers for reusing the formula instead of copying the
formula twice. For instance, one could use the following formula in QBF.

$$\forall X \left((X = a) \vee (X = b) \rightarrow \phi(X) \right)$$

In this subsection, we briefly discuss two main encodings that compress the plan
length using universal variables.

Flat Encoding   In Bounded Model Checking (BMC), an encoding that avoids the
transition function copies using ∃∀∃ quantifier alternation was proposed in [34, 69].
In Flat Encoding, also called iterative squaring, the idea of compression is used with
universal variables and the path is encoded recursively thus avoiding copies of the
transition function. [103] proposed a QBF encoding that compresses the path in the
context of classical planning. The following formula shows the structure of a QBF

encoding for a plan length 4.

$$\exists S_I, S_G$$
$$\exists S_{IG} \forall y_1 \exists S_1, S_2$$
$$\exists S_{12} \forall y_2 \exists X_1, X_2$$
$$(\neg y_1 \rightarrow (S_1 \leftrightarrow S_I \wedge S_2 \leftrightarrow S_{IG})) \wedge (y_1 \rightarrow (S_1 \leftrightarrow S_{IG} \wedge S_2 \leftrightarrow S_G)) \wedge$$
$$(\neg y_2 \rightarrow (X_1 \leftrightarrow S_1 \wedge X_2 \leftrightarrow S_{12})) \wedge (y_2 \rightarrow (X_1 \leftrightarrow S_{12} \wedge X_2 \leftrightarrow S_2)) \wedge$$
$$I(S_I) \wedge G(S_G) \wedge T(X_1, X_2)$$

Here we can see that the transition function is provided only once on symbolic states X_1, X_2. The universal variables y_1, y_2 have 4 instantiations, rest of the constraints specify which states to apply the transition function on. For example, when y_1, y_2 are set to False, the transition function corresponds to the first step of the plan i.e., from initial state to the next intermediate state. For a problem with plan length k, we only need $\log(k)$ universal variables and $\log(k) + 1$ quantifier alternations in Flat encoding.

Compact Tree Encoding   When the Flat Encoding is visualized as a tree, one could observe that the implicit copies of transition function exist at the leaves. All the inner nodes of such a tree are only used for keeping track of intermediate state variables. [26] proposed a Compact Tree Encoding (CTE) to specify copies of transition function also in inner nodes which can reduce the number of universal variables and quantifier alternations. The following formula shows the structure of the encoding for a plan length 8.

$$\exists S_I, S_G$$
$$\exists X_2 \forall y_2$$
$$\exists X_1 \forall y_1 \exists X_0$$
$$((\neg y_1 \wedge \neg y_2) \rightarrow T(S_I, X_0)) \wedge ((y_1 \wedge y_2) \rightarrow T(X_0, S_G)) \wedge$$
$$(\neg y_1 \rightarrow T(X_0, X_1)) \wedge (y_1 \rightarrow T(X_1, X_0)) \wedge$$
$$((\neg y_2 \wedge y_1) \rightarrow T(X_0, X_2)) \wedge ((y_2 \wedge \neg y_1) \rightarrow T(X_2, X_0)) \wedge$$
$$I(S_I) \wedge G(S_G)$$

Essentially, at the cost of logarithmic of path length copies of transition function, the new CTE encoding improves performance. Note that for the same number of universal variables in Flat encoding, CTE encodes a plan of length 8.

Techniques like splitting actions, invariants and parallel plans also apply for QBF based encodings. As long as the problem is grounded, invariants and parallel plans can be employed easily. In §5.6, we discuss an existing partial grounding encoding [27] and the challenges in detail. One key similarity in all the existing encodings is that there is either complete or partial grounding. Our main focus is to improve the bottleneck from grounding in hard-to-ground domains.

2.1.4   Application : Optimal Layout Synthesis for Quantum Circuits

We consider the problem Optimal Layout Synthesis (OLS) [123] for quantum circuits. A quantum circuit can always be translated to a circuit with single qubit gates and two qubit Conditional-NOT CNOT gates. To run a quantum circuit on a quantum computer, it needs to be compiled such that some constraints on the hardware are satisfied. There are three main constraints to be satisfied in the problem we consider:

- Qubits in the circuit (Logical qubits) are initialized on to the qubits of the quantum platform (Physical qubits) bijectively

- Gates are to be scheduled such that their order dependencies are satisfied

- To apply the two qubit gates on a quantum hardware, the physical qubits must be connected.

It is not always possible to satisfy all three constraints. The main challenge is from scheduling two qubit gates, as single qubit gates can always be scheduled anywhere. Modelers assume one-to-one connectivity of all physical qubits, but in reality only certain pairs are connected. One could use so-called *SWAP* gates to move around the qubits so that all the constraints are satisfied. The goal here is to minimize the number of swap gates, as error rate increases with the number of gates. In this Subsection, we briefly discuss the existing methods for heuristic and exact (or optimal) approaches.

Heuristic Approaches   There are many heuristics approaches which focus on specific quantum platforms (see §6.1 for specific references). There also exist general layout synthesis algorithms, for example like SABRE [78], which add swap gates greedily using heuristics where necessary. Some tools based on SAT [87, 131] and SMT [123] also generate near optimal mappings.

Exact Approaches   Exact approaches generate mappings with minimal number of swap additions when optimized for the swap count. Note that some tools also allow minimizing depth of a given circuit and the error rate of the gates used. While there has been plenty of research in heuristic approaches, exact approaches still suffer from scalability issues. Existing leading approaches are QMAP and OLSQ which are based on SMT. QMAP [131] generates propositional constraints for satisfying connectivity on the 2-qubit gates and uses cost for optimizing swap count. The bottleneck of QMAP is that the encoding uses an exponential number of variables in the number of physical qubits. The OLSQ [123] approach improves on QMAP by using a polynomial number of variables instead. Both approaches scale to circuits with a few qubits, thus improved encodings are needed for better scalability.

2.2 Research Challenges

In our research, we focus on two challenges.

2.2.1 Classical Planning

In recent years, the main focus of classical planning has been to solve problems with grounding bottleneck. The problem with grounding was evident, especially, with SAT based planners in 2018 IPC competitions. As the first challenge, we focus on generating lifted encodings that avoid grounding completely. We pose the following research questions:

RQ-CP1 How to avoid grounding in SAT/QBF based planning?
All existing SAT and QBF encodings apply grounding to some extent. SAT based planner like Madagscar apply complete grounding, which can help in computing strong invariants and efficient encodings. Main QBF encodings, flat encoding and compact tree encoding also completely ground the problem. While QBF encodings avoid copies of transitions functions, they at least need a single copy of grounded transition function. In [27], a partial grounding approach is proposed which still suffers with large encodings sizes. Avoiding grounding either by SAT or QBF encodings is necessary for solving hard-to-ground instances.

RQ-CP2 Can we solve the organic synthesis problems using SAT/QBF?
Organic Synthesis benchmarks are especially hard for SAT based techniques since actions can have up to 31 parameters and there can be up to 100 objects. In-equality constraints on the action parameters dominate in these instances. The challenge here is to encode such constraints efficiently while keeping the encoding itself compact.

2.2.2 Optimal Layout Synthesis

We explain the problem OLS in detail in Chapter 6. We pose the following research questions:

RQ-CP3 Can classical planning be a better alternative compared to temporal planning for OLS?
There exist temporal planning encodings for OLS focused on optimizing depth on specific quantum circuits [37, 128]. It was unclear whether classical planning can be a better alternative instead of temporal planning. It is consensus that classical planners are more matured and efficient compared to their counterparts. Despite the early negative results in literature, it is important to investigate if better encodings can be formulated for classical planners. Further, optimizing the number of swaps would be an additional criterion since existing temporal planning encodings only produce near optimal swaps.

RQ-CP4   How do classical SAT based encodings perform compared to existing OLS specific SMT encodings?

One of the advantages of domain independent solving, is separation of engineering solvers and encodings of the problems. Indeed, SAT based planners, for example Madagascar [101], are heavily engineered for optimal performance of planning problems. Further, the encodings are well studied with improvements like invariant computation and parallel plans. We observed that existing SMT encodings for OLS are not efficient, for example constraints in QMAP [131] use exponential number of variables. We conjectured that simply encoding OLS as a planning problem, one could readily use well studied encodings and highly optimized SAT based planners.

2.3   An Overview of Our Results

2.3.1   Key Ideas : Lifted Classical Planning as QBF

KI-CP1   Use universal variables to represent symbolic constraints of the actions and propositions

We generated a lifted encoding that encodes the problem directly in almost first order representation. Grounding essentially generates object combinations for predicate parameters. We propose to use universal variables to represent the object combinations. The following formula shows the general structure of our encoding.

$$\exists A^0, PM^0, \ldots, A^{k-1}, PM^{k-1}$$
$$\forall OC$$
$$\exists P^0, \ldots, P^k$$
$$I_u(P^0, OC) \wedge G_u(P^k, OC) \wedge \bigwedge_{i=0}^{k-1} T_u^i(P^i, P^{i+1}, OC, A^i, PM^i)$$

We propose to use binary representation for action variables (A^i) and parameter variables (PM^i). We use the universal variables (OC) to generate symbolic constraints for updating the state (P^i) after an action is applied. This ensures compact encoding size and results in logarithmic growth with the number of objects. The complete constraints and more explanation of our encoding is provided in §5.4.

KI-CP2   Constraining in-equalities on parameter variables directly

In our initial ungrounded encoding, the in-equality is simply a new predicate on two parameters. Encoding in-equalities using predicate variables pushes the constraints to the innermost quantifier layer. Since, the variables parameters are in the outermost quantifier layer, we propose to generate in-equality constraints directly on the corresponding variables. We propose to generate equality circuit between two sets of parameter variables and negate the output gate. We conjecture that this results in improved solving times, especially on the Organic synthesis benchmarks.

2.3.2 Key Ideas : Optimal Layout Synthesis as Classical Planning

KI-CP3   Encode OLS as a classical planning problem based on layers
We conjectured that a classical planning encoding for OLS would be more efficient than the existing temporal planning encodings. For a better encoding, we employ two main differences compared to existing temporal encodings [37, 128]. First, we compute dependency graph and the layers where independent CNOT gates are placed in the same layer using Qiskit [97]. Second, we constrain swap gates to be added separately at each time step to allow optimization of swap insertions. We modelled OLS as a classical planning problem using the following 4 actions to satisfy the three constraints described in §2.2.2.

- `map_initial` : to map logical qubit to physical qubits bijectively

- `apply_cnot` : to apply a cnot gate on correspoding physical qubits

- `swap` : to swap the logical qubits mapped to the physical qubits

- `move_depth` : to move to the next layers according to the dependencies

In this model, given a quantum circuit, the number of `map_initial`, `apply_cnot`, `move_depth` actions are fixed. Only the number of swap actions differ, so a plan with optimal length corresponds to optimizing the number of swaps. In this way, we use classical planners for generating mapped circuits with optimal number of swap gates.

KI-CP4   Avoid layers using local dependencies
In model from KI-CP3, the number of move_depth actions are equal to the number of layers in the quantum circuit. Using an action for moving to next layer results in longer plan length which can effect solving performance. In the worst case, the number of layers can be equal to the number of CNOT gates themselves. In our Local encoding, we propose to avoid the layers altogether and specify dependency constraints using propositions. Essentially, preconditions in each action ensure that the predecessor CNOTs are *done* before the current CNOT gate. So the new encoding only contains first 3 actions and results in shorter plans.

KI-CP5   Integrating initial map actions with CNOT actions
As discussed in KI-CP4, increase in plan length increases solving time for planners. In the Local encoding, we proposed to use one action map_initial for mapping one logical qubit to one physical qubit. This can be avoided by integrating the mapping into the CNOT actions. We propose to map an unmapped logical qubit to an unmapped physical qubit when a CNOT gate depends on the corresponding logical qubit. The new version of the encoding only has `apply_cnot` and `swap` actions, thus reducing the plan length further. Note that in both the variations of local encoding, optimal plan length still corresponds to the optimal number of swaps added.

2.3.3   Contribution

In Chapter 5, we address the research questions RQ-CP1 and RQ-CP2. We implemented the key ideas KI-CP1 and KI-CP2 in our open source tool Q-Planner[1] written in Python. Given a domain and problem description in PDDL, we generate a QBF instance encoding the planning problem for some given plan length k. Our QBF encoding grows linearly in the number of action names, predicates and the path length, and logarithmic in the number of objects. Our lifted encoding results in compact encoding size, compared to existing SAT/QBF encodings. Using our tool, we could solve Organic Synthesis instances optimally which could not be handled using existing SAT/QBF techniques. Our tool performed better than state-of-the-art planner Fast Downward on some hard-to-ground instances, as we avoid grounding completely. The key idea KI-CP2 indeed improved the performance of our tools as the QBF solvers can take advantages of efficient in-equality constraints.

In Chapter 6, we address the research questions RQ-CP3 and RQ-CP4. We provide two encodings, first a Global encoding with layer using KI-CP3. With the second Local encoding, we avoid layers reducing the plan length using KI-CP4 and KI-CP5. We implemented both encodings in our open source tool Q-Synth[2] in Python. Using Q-Synth with state-of-the-art classical planners, we perform 2 orders of magnitude faster than the current leading tools OLSQ [123] and QMAP [22]. We optimally mapped 9 qubit benchmarks on a 14 qubit platform, which could not be handled using existing tools. We submitted our domain to International Planning Competitions 2023, and got awarded *Outstanding Domain Submission*. Confirming our conjecture, SAT based planners outperform existing tools with OLS specific SMT encodings. We also showed that classical planners are a strong alternative to temporal planners for OLS.

2.4   Recent Developments

After our work, [59] presented a lifted SAT encoding that avoids the grounding. Their key idea is to use stateless encoding, where no state is represented but only conflicts between the actions are tracked. The authors presented better results on the Organic Synthesis domain. Better results are expected as SAT solvers perform better compared to QBF solvers. Their encoding grows quadratic in plan length and the clause sizes can increase with number of precondition conflicts in actions.

[79] proposed a followup encoding OLSQ2 improving on OLSQ encoding. The main improvement was removing so-called space variables in their encoding and allowing different optimization routines. The authors report improved solving by using different representations for integer variables in their encoding. We are working on a followup SAT encoding that is based on parallel plans in SAT based planning. Our main idea is to group CNOT gates in the same timestep when swap gates are not needed.

---

[1]https://github.com/irfansha/Q-Planner
[2]https://github.com/irfansha/Q-Synth

Chapter 3

2-Player Games

Two player games can be seen as a planning problem with an adversary. In particular, we consider 2-player, complete information, zero-sum and turn based games played on a board. The hardness of the problem increases with the number of turns and the general complexity is PSPACE-complete. General Game Playing is one of the main problems in Artificial General Intelligence. In General, games are specified in specifications like Game Description Language (GDL) [47] for formulating strong players. Machine learning techniques can be used for programming such strong players, where moves are generated with a high probability of winning a game. AlphaGo is such a successful strong player for the game Go, which was able to beat the best human players. There exist strong players for general game playing, and it is still possible for such strong players to lose a game.

In this work, we focus on solving the games. Given a description of a game, can a solver find if a game is winning and extract a winning strategy if exists. A player has a winning strategy if the player can always win regardless of the choice of the opponent moves. In the games we consider, there are three possibilities: (1) First player is winning; (2) Second player is winning; (3) Neither player is winning. For example, consider a classical game of Tic-Tac-Toe where 2 players take turns to draw symbols X and O on a 3×3 board. Here the goal of the game is to make a 3-in-a-line with their own symbols. Figure 3.1 shows a completed game where the player with symbol O won. For such a game, there is no winning strategy for either player and first player can always force a draw. Completely solving a game is mostly done by complete search space exploration. QBF solving allows a compact and natural representation of opponent moves with universal variables. Considering the success of SAT solvers in NP-complete problems, QBF solving is a potentially promising technique for solving 2-player games, a PSPACE-complete problem. Some small games have been encoded in QBF, which we will briefly discuss in the next section. Additionally, QBF solvers can also generate certificates which contain the winning strategies of the games solved. In this chapter, we discuss and address some bottlenecks in QBF solving for solving 2-player games.
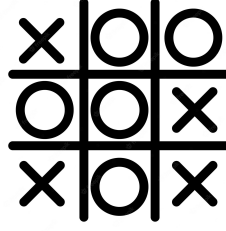
Figure 3.1: A completely filled Tic-Tac-Toe game
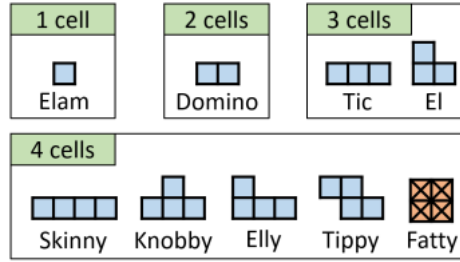
Figure 3.2: Shapes with 4 cells or less in Harary's Tic-Tac-Toe, from Figure 2 in [36]

3.1 Existing Work

One could see any QBF as a 2-player game between an existential player and a universal player. There exists some QBF solving techniques that take advantage of such structure [9]. In this work, we focus on the games that are encoded and solved as a QBF instance directly. Several games have been encoded into QBF, for example Petri games [42], Positional games [36, 84], Connect4 [48] and Evader-Pursuer [1, 3] etc. In this section, we briefly discuss two main subsets of 2-player games and the key ideas for encoding them.

3.1.1 Positional Games

In a positional game, two players take turn to occupy empty positions on the board. The goal of a positional game is to achieve a specified shape or a path with the player's own pieces. Depending on whether the winning condition is specifying a shape or connecting a path, the complexity of the goal can change. We briefly discuss two games, one with specifying a shape and another with connecting a path.

Our earlier example, Tic-Tac-Toe, is a positional game where the winning condition is to achieve the shape 3-in-a-line. There exists some variations of Tic-Tac-Toe such as Harary's Tic-Tac-Toe (HTTT) [54] and Generalised Tic-Tac-Toe (GTTT) [36]. In HTTT, a set of defined shapes, called polyominos, are specified as a goal condition. For example, Figure 3.2 shows some shapes with 4 cells or less. GTTT simply extends HTTT with multiple moves for each player in their own turn. In both variations of the games, the number of winning goal conditions are polynomial in the board size.
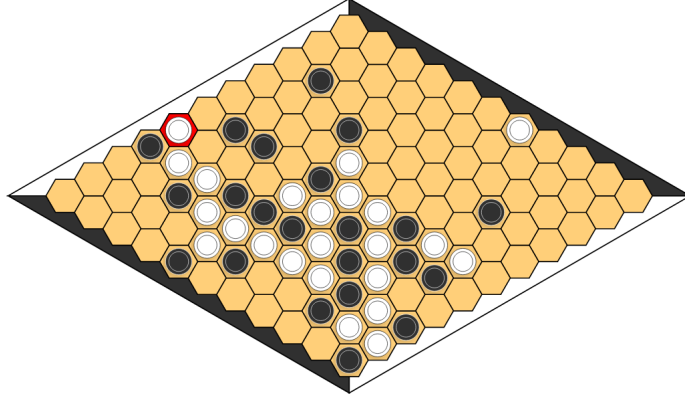
Figure 3.3: An example Hex Game where white player won

Hex, a popular positional game, invented by Piet Hein [55] is played on a board with hexagonal cells. The winning condition here is to connect opposite borders of the board with one's own pieces. Figure 3.3 shows a Hex game on a 11×11 board, where white player won the game. The number of winning conditions grow exponentially with the size of the board.

In recent years, several encodings were proposed for positional games. Since there is only a single move that depends on a single position of the board, it is possible to generate constraints taking advantage of such simple structure.

Encoding with Indicator Variables Inspired by a QBF encoding for Connect4 in [48], [36] proposed a QBF encoding for GTTT. The main challenge with QBF encodings for 2-player games is handling invalid opponent moves. Since one encodes opponent moves using universal variables, there are always some instantiations which correspond to invalid moves. The key idea proposed was to use indicator variables to explicitly specify an invalid move. Note that the indicator variables are needed for each move and at each time step, which can increase the number of variables and clauses significantly. The opponent moves with universal variables are encoded with one-hot encoding which can result in many invalid moves. For example, suppose we want to encode n number of moves. To encode in a one-hot encoding, we first define n variables $m_1, \ldots, m_n$. By using n universal variables, we essentially encode all 2^n possibilities. We use Exactly-One constraints to specify the valid moves. The following formula shows the general structure of the one-hot encoding for opponent moves.

$$\forall m_1, \ldots, m_n \text{ ExactlyOne}(m_1, \ldots, m_n) \to \Phi(m_1, \ldots, m_n)$$

The encoding was effective for smaller instances, however the encoding size can blow up quickly for board sizes greater than 5×5.

Corrective Encoding   [84] observed that for positional games one can avoid using indicator variables completely. In the corrective encoding, when the universal player makes an illegal move the existential player is allowed to change the board completely. By doing so, when the universal player (i.e., the opponent) makes an illegal move the existential player can simply satisfy the winning condition. Avoiding indicator variables simplifies the constraints to a great extent and seems to improve solver performance. Further, the opponent moves are encoded using binary encoding instead of one-hot encoding to reduce the number of illegal moves. For example, to encode n moves we only need $k = \lceil \log(n) \rceil$ variables. Let $\overrightarrow{M} = m_1, \ldots, m_k$ be the opponent move. To specify the valid moves, we use a less than comparator and use the following formula.

$$\forall \overrightarrow{M} \ (\overrightarrow{M} \leq n) \rightarrow \Phi(\overrightarrow{M})$$

The Corrective encoding could solve some Hex puzzles up to the board size 5×5.

Blocking Strategy + Corrective Encoding in HTTT   In games like HTTT, there exist some strategies that take advantage of the fixed goal conditions for efficient solving. The blocking strategy can be used for blocking shapes as in [53], the key observation is that a partially filled shape with an opponent piece can never be a winning shape. Thus, one could disable such shapes from the winning shapes. [20] encoded the blocking strategy in the Corrective encoding. The idea is to check if all the winning conditions are blocked already after playing n moves. Without a blocking strategy, one might need to play many moves to detect non-existence of a winning strategy. It outperforms the original corrective encoding with standard QBF solvers on the HTTT benchmarks. While it is an effective technique, one cannot use the same blocking strategy for other positional games like Hex and non-positional games.

3.1.2   Beyond Positional Games

Most games allow more than just occupying an empty position. There has been a limited research on QBF encodings of the non-positional games. The first QBF encoding for any game has been proposed by [48] in 2004, for the game Connect4. Connect4 is similar to the positional game 4-in-a-line. The main difference is, one cannot occupy a position if its neighbor below is empty. Solving the game on the 7×6 board was out of reach with that encoding. The opponent moves are encoded using one-hot encoding, thus many invalid moves are possible.

[1] proposed a QBF encoding for the game Evader-Pursuer. This is a simple version of chess to simulate attacking and defending. Each player has one piece and are placed at specified cells of a grid in the initial state. The first player tries to evade the second player to reach a specified goal cell. The second player tries to capture the first player, and wins when the first player is captured. An improved encoding was proposed for the same game by [3], focusing on reducing the search space to be pruned. Indicator variables are used, but they are grouped together based on some constraints to help the solver backtrack early.

3.2 Research Challenges

3.2.1 Positional Games

We look at positional 2-player games and their QBF encodings for winning strategies. While several encodings were proposed in the context of positional games as discussed in the previous section, games with complex goal constraints are still beyond reach. As board size increases the encoding size also increases resulting in large encodings. The main aim is to avoid this blowup by using universal variables and other techniques.

RQ-G1   How to avoid polynomial board constraints in the positional game encodings?
Existing encodings [20, 84] encode board constraints explicitly. After each move the state of each position on the board has to be updated. In an explicit encoding, one generates constraints for each position after each move. Encoding size can become too large even for moderate board sizes. For solving instances from large board sizes, a more concise encoding is needed.

RQ-G2   How to avoid exponential goal constraints in games like Hex?
For games like HTTT and GTTT, the winning condition is simply forming some fixed shape. As long as the number of pieces needed to form the shape is constant, the total number of goal conditions is polynomial in the size of the board. However, in case of games like Hex, the number of winning paths grow exponentially with the board size. Using the leading QBF encoding in [84], one could not generate QBF instances for Hex puzzles with board size more than 6×6. For a board size of 7×7, the number of winning conditions is more than a million. Clearly, an implicit goal encoding is needed to encode such winning conditions.

3.2.2 Beyond Positional Games

We now consider 2-player games beyond positional games. While there has been a lot of progress in positional games, the QBF encodings for non-positional games are limited. Formulating a QBF encoding for a game from scratch is challenging and time-consuming. Our focus here is to provide a domain independent way for modelling games and translation to QBF.

RQ-G3   How to simplify the process of encoding new games in QBF?
As discussed in §3.1.2, there only exist previous QBF encodings of two non-positional games for Connect4 [48] and Evader-Pursuer [1, 3]. Generating a new QBF encoding, and perhaps a corresponding tool, for a new game is challenging and time-consuming. Clearly, one needs a domain independent approach similar to PDDL in planning and GDL in general game playing. The existing GDL is too complex for a direct QBF translation. Our challenge is to find either a fragment of GDL or a new specification language.

RQ-G4   How to avoid large encoding sizes in the non-positional games?
Similar to positional games, encoding size for non-positional games can blowup
with larger board size. One main challenge is to generate compact encodings for
non-positional games. It is not trivial to encode board constraints of the non-positional
games symbolically. Unlike positional games, a move can depend on the states of
neighboring positions. After a move is played, a move can also update the state of
multiple neighboring positions.

3.3   An Overview of Our Results

3.3.1   Key Ideas : Lifted Encodings for Positional Games

KI-G1   Encoding symbolic board constraints using universal variables
In Chapter 2, in the context of planning, we observed that a lifted encoding can solve
the encoding size bottleneck to some extent. Since positional games have a single
move of occupying an empty position, we can encode constraints symbolically on a
single position. The main idea is to use universal variables to represent all positions
on the board. We propose to use binary representation for the symbolic positions.
Essentially, we enumerate the board positions. Each universal branch of the symbolic
position corresponds to one unique position on the board. If the number of positions
is a non-power of 2, we exclude the invalid positions from binary representation using
less than comparator.

KI-G2   Implicit goal representation with neighbor relation for a winning path
Using KI-G1 results in implicit board constraints. However, to address RQ-G2, i.e.,
to handle exponential goal conditions, we need to encode the goal implicitly as well.
One can see that the Hex-like-games have some structure on the board, i.e., paths are
connected by neighboring positions. We propose to encode the winning path using
the neighbor relation between positions. Briefly, we say there exists a winning path at
the end of every game play. We ensure that the path connects two opposite borders of
the board and adjacent path positions are neighbors.

KI-G3   Stateless constraints for simplifying board and goal constraints
In our work, we consider maker-breaker positional games where the opponent only
needs to stop the first player from satisfying the goal condition. In this subset of
games, one could avoid variables for maintaining state of the board. Essentially, we
ensure that all the moves played by both players are different. As long as the players
play on different positions, we do not need to keep track of the state of the board.
These constraints can be generated using in-equality clauses on the move variables
without extra variables. In a maker-breaker game, there is no notion of a draw. For
checking the winning condition, one only need to check if the first player achieved the
goal condition. We reuse the move variables, which specify the occupied positions, to
avoid extra variables in goal constraints. One drawback from symbolic encoding in

KI-G1 and KI-G2 is the extra quantifier alternation needed. Stateless encoding avoids this extra quantifier alternations at the cost of many move variable in-equalities. For shallow games, we observed that stateless encoding performs well with existing QBF solvers.

3.3.2 Key Ideas : Lifted Encodings for Grid Games

KI-G4 A new board game domain definition language (BDDL) for grid based 2-player games

We first consider a subset of games, that can be played on a grid and each player uses a single type of piece. Even with this restriction, our subset allows several classical games such as positional games, Connect-4, BreakThrough, KnightThrough, Domineering and many others. We propose a new definition language, Board Game Domain Definition Language (BDDL). BDDL specification is close to PDDL with two files, a domain file and a problem file, for each model. A domain file lists moves of both players whereas a problem file specifies initial state, goal conditions, board size and winning strategy depth. For example, Listing 3.1 specifies the domain file for any positional game. Preconditions and Effects specify when a position can be occupied and how the state of the position changes after occupying. Listing 3.2 specifies the problem file for the Tic-Tac-Toe game with an empty 3×3 board. Both Listings together encode the classical Tic-Tac-Toe game.

Listing 3.1: Positional games' domain, from §8.3.2

```
1   #blackactions
2   :action occupy
3   :parameters (?x,?y)
4   :precondition (open(?x,?y))
5   :effect (black(?x,?y))
6   #whiteactions
7   :action occupy
8   :parameters (?x,?y)
9   :precondition (open(?x,?y))
10  :effect (white(?x,?y))
```

Listing 3.2: Snippets of Tic-Tac-Toe problem instance

```
1   #boardsize 3 3
2   #init
3   #depth 9
4   #blackgoals
5   (black(?x,?y)black(?x,?y+1)black(?x,?y+2))
6   (black(?x,?y)black(?x+1,?y)black(?x+2,?y))
7   ...
8   #whitegoals
9   (white(?x,?y)white(?x,?y+1)white(?x,?y+2))
10  ...
```

KI-G5   Using relative indexing for representing neighbor positions

In RQ-G4, we pointed out that a move of a non-positional game can depend on the state of neighboring positions. In BDDL, we allow relative indexing of constraints to represent the neighbors using a single position as a reference. For example, in Listing 3.2 we specify the goal conditions of Tic-Tac-Toe game. To encode 3-in-a-line goal conditions, first we define a reference position (`?x,?y`) on the grid. Next, we use relative indices to specify the state of the neighboring positions (`?x,?y+1`) and (`?x,?y+2`). Similarly, complex moves can also be encoded using relative constraints. For example, in Connect4 game one can only occupy an open position if the neighbor below is not-empty. Listing 3.3 shows such a move specification with constraints using a relative index.

Listing 3.3: A Black move for Connect4

```
1  :action occupyOnTop
2  :parameters (?x,?y)
3  :precondition (open(?x,?y) NOT(open(?x,?y+1)))
4  :effect (black(?x,?y))
```

KI-G6   Using adder, comparator circuits and universal variables for compact encodings

For a concise encoding, we employ two ideas in our lifted encoding. First, we extend the idea of encoding state constraints symbolically using universal variables. Second, we take advantage of relative indexing of constraints as stated in KI-G5 for accessing neighbor positions. The following formula shows the prefix of our lifted encoding for games in BDDL, the full encoding with explanation is provided in §8.4.

$$\underbrace{\exists A^1, X^1, Y^1, gs^1}_{\text{Black move 1}} \quad \underbrace{\forall A^2, X^2, Y^2, gs^2}_{\text{White move 2}} \quad \underbrace{\exists lb^2, P^2}_{\text{White Indicators 2}} \quad \ldots \quad \underbrace{\exists A^d, X^d, Y^d, gs^d}_{\text{Black move d}}$$

$$\underbrace{\exists B_x, B_y, B_c}_{\text{Black goal}} \quad \underbrace{\forall W_x, W_y, W_c \, \exists W_{ce}}_{\text{White goal}} \quad \underbrace{\forall P_x, P_y}_{\text{symbolic pos}} \quad \underbrace{\exists o^1, w^1, \ldots, o^{d+1}, w^{d+1}}_{\text{state variables}}$$

The variables X^i, Y^i correspond to the reference positions for the moves. Similarly, we use reference positions for goal conditions, for example B_x, B_y variables for black goal. Now, we generate adder and subtractor circuits on the reference positions to access the neighboring positions. Together with symbolic universal position variables (P_x, P_y), we avoid explicit board and goal constraints.

3.3.3   Contribution

In Chapter 7, we address the research questions RQ-G1 and RQ-G2 on positional games. We formulated 7 different encodings with implicit goal and implicit state constraints. We implemented key ideas KI-G1 to KI-G3 in our open source tool Q-sage[1] in Python. Using new implicit encodings, we were able to encode Hex

---

[1] https://github.com/irfansha/Q-sage

puzzles of the board size greater than 6. Note that the goal constraints in Hex grow exponentially with board size. While solving 19×19 board (the standard board size for human play) is still out of reach, we could encode the complete games (with 361 moves) using our encoding. For experimentation, we considered human played games from recent (online) championship and generated a set of puzzles with end games. Note that such puzzles of end games have many already played positions. We also proposed some preprocessing techniques for our implicit encodings to simplify such game instances. Using implicit encodings and preprocessing techniques, we were able to solve some puzzles with the board size of 19×19. Our encodings with QBF solvers are 2 orders of magnitude faster than existing leading QBF encodings. Our QBF instances for Hex encodings were submitted at the QBFeval-2022 competitions.

In Chapter 8, we address the research questions RQ-G3 and RQ-G4 on grid based games. First, we proposed a Board-Game Domain Definition Language (BDDL) as in KI-G4 and KI-G5. Second, we provide a translation to QBF, which corresponds to a winning strategy of the game encoded. We use the key idea KI-G6 to keep our generated encoding compact. Our translation is implemented in Python and available in the open source tool Q-sage. The generated QBF encoding is linear in the size of the input BDDL model and depth of the winning strategy required. We were able to provide first QBF encodings for several non-positional games such as BreakThrough, KnightThrough and Domineering etc. Our QBF instances for BreakThrough and Connect4 were submitted at the QBFeval-2023 competitions.

Chapter 4

Validation of Encodings

One of main strengths of SAT based approaches is the ability to verify the results produced. In SAT solving, when a formula is satisfied, a complete assignment is generated which can be used to verify if the result is correct. In case of an unsatisfiable formula, a SAT solver produces a proof showing that the formula is unsatisfiable. There has been a major effort in proof generation in context of SAT solving [23]. On the other hand, certification in QBF solving still faces many challenges. Current state-of-the-art QBF solvers do not produce certificates, or only produce by disabling several solving techniques. While there is progress in certifying QBF solvers, validation of QBF encodings is largely unexplored. In this chapter, we focus on certificates in the context of QBF solvers.

4.1   Certificates for Correctness

When a QBF formula is solved, there are two main ways of ensuring correctness. Either the QBF solver is formally verified, or a proof is generated for the current run which can be verified. The QBF solvers take the second approach, usually solvers can generate a trace/proof which can be verified with the input instance. In this section, we briefly look at the generation of such traces and how they are used in the context of verification.

As an example, we consider a True formula $\phi = \forall x \exists y \, x \leftrightarrow y$. We can choose the same y value for any x value such that they are equal. Figure 4.1 shows the search tree for ϕ, and we can see that it is indeed a True formula. If we use one of the leading QBF solver, DepQBF [80], to solve ϕ, it returns satisfiable as expected. Figure 4.2a shows the encoding in QDIMACS format, which is the input format for the solver DepQBF. The first line "p cnf 2 2" specifies that the instance is in prenex conjunctive normal form with 2 variables and 2 clauses. Followed by quantifier blocks, "a" represent universal variables and "e" represents existential variables. Since we can only use integers as variables we use 1 for the variable x and 2 for the variable y in the formula ϕ. After quantifier blocks, each line represents a disjunction of variables and the line ends by 0. The lines `-1 2 0` and `1 -2 0` represent $(\neg x \vee y) \wedge (x \vee \neg y)$ of ϕ.
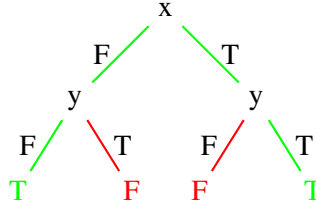
Figure 4.1: Search Tree for the formula ϕ

Skolem/Herbrand Functions   Certfication frameworks exists [7, 13, 89, 93, 98], which can generate Skolem/Herbrand functions. Given a True formula, for each existential variable one could generate a so-called *skolem* function with preceding universal variables as the input. The set of skolem functions for all existential variables forms a certificate for the corresponding True formula. Essentially, skolem functions can be used to compute the values of existential variables given the values of the preceeding universal variables. Since functions are not inherent to QBF, clauses are generated that correspond to the skolem functions. For example, Figure 4.2b is the skolem function for variable y and consequently for the formula ϕ. Figure 4.2b shows a boolean formula in the standard DIMACS format which is in CNF. Similar to the encoding format QDIMACS, except we do not need to specify the quantifier blocks as all the variables are existential. We can observe that the variables $1, 2$ from the original formula also exist in the clauses of the certificate. Suppose, we want to find the output of the skolem function i.e., the value of y variable when the x variable is set to True. In the certificate, we add an extra clause 1 0 to generate a new DIMACS instance which results in Figure 4.2c. If we use a SAT solver to solve this instance and extract the variable assignment for 2, we would get True which is the desired output for the variable y. So the idea is to add the input values of universal variables as assumptions and extract the resultant values of existential variables using a SAT call. Herbrand functions are dual to Skolem functions, here the functions take existential variables as inputs for the output universal variables. If a QBF is a False formula, then one can generate Hebrand functions and similar process can be used to extract the required universal assignments.

Certifying Solver Run   First application of certificate generation in QBF solvers is certifying the solver run. As discussed in [89], for certifying, first the existential/universal variables of the encoding are replaced with their corresponding Skolem/Hebrand functions. Now, the resulting propositional formula contains only universal/existential variables. For a True QBF formula, the resulting propositional formula must be a tautology and one could check it using a SAT solver. To check a tautology, one can negate the formula and check if the result is unsatisfiable. For False QBF formulas, the resulting propositional formula is checked if it is unsatisfiable. Only when the certificate generated is correct, the (un)satisfiability check will be verified.

```
p cnf 2 2
a 1 0
e 2 0
-1 2 0
1 -2 0
```

```
p cnf 3 4
-1 -3 0
1 3 0
-3 -2 0
3 2 0
```

```
p cnf 3 5
-1 -3 0
1 3 0
-3 -2 0
3 2 0
1 0
```

(a) Encoding   (b) Skolem Function/Certificate   (c) With an assumption

Figure 4.2: Skolem Function for ϕ from DepQBF and QRPcert

Winning Strategies   Second application of certificates is for extracting a winning strategy. A Quantified Boolean Formula can be seen as a 2-player game between existential and universal player. Skolem/Herbrand functions essentially contain the winning strategies of the first/second player. Instead of just the True/False result, a certificate/winning-strategy can contain interesting information that can be useful [122]. For example, one could use the information of a reachable bad state in Bounded Model Checking for Debugging.

4.2 Research Challenges

In our research, we focus on the validation of QBF encodings using certificates(or winning strategies). Due to the presence of quantifier alternations, it is not trivial to extract the inner variable assignments and even harder to check the correctness. As a first step towards complete validation of the QBF encodings, we investigate the following research questions:

RQ-V1   How to check if the problem is encoded correctly?
Usually an encoding has to satisfy some conditions to ensure correctness. In SAT based solving, given a propositional formula, a SAT solver returns a complete assignment if the formula is satisfiable. One can check if the assignment satisfies required properties of the problem encoded. In case of QBF, as discussed before, a winning strategy is in the form of a tree due to quantifier alternations. Most QBF solvers only provide assignment of the outermost quantifier variables. To validate general conditions on a problem, one might need the assignments of the inner quantifier variables as well. Here the main challenge is to allow extraction of inner variable assignments.

RQ-V2   Do encoding variations preserve equivalence?
When encoding some problem, it is common to have several encoding variations of the same problem. It is a consensus that simple and natural encodings do not perform well with current QBF solvers. Thus, modelers *optimize* encodings to improve the target solver performance. This results in a challenging validation task, i.e., to check if the changes preserve equivalence with the original encoding. Complete equivalence check is impractical even for small formulas, one needs to relax some conditions for practical purposes.

RQ-V3   How to scale QBF validation?

Skolem/Herbrand functions can be extracted from certificates i.e., from certifying QBF solvers. From QBF competitions [95], it is clear that certifying solvers can be much slower compared to non-certifying solvers. The main penalty in certifying solvers comes from turning off some key solving techniques. One could not use most preprocessing techniques if the certificates are to be preserved. Further, the generated certificate size can grow exponentially with the instance depth and the number of variables. One needs to avoid these bottlenecks for any practical validation of QBF encodings.

4.3   An Overview of Our Results

4.3.1   Key Ideas

KI-V1   Interactive play with a Certificate

In RQ-V1, we mentioned that inner variables assignments are masked due to quantifier alternations. In §4.1, we mentioned that a certificate is simply a set of Skolem/Herbrand functions. The key idea here is to use a certificate for extracting inner variable assignments. The main aim of validation is to find errors if they exist. We propose to play interactively with the certificate i.e., the winning strategy, as an opponent to explore different parts of the winning strategy. By playing as an opponent, we try to find a path which can lead to errors. We also propose to use a random player for automation of exploring different paths.

KI-V2   Interactive play with a QBF solver

While KI-V1 solves the problems of extracting inner variable assignments, the certificate sizes can exceed GB even for small QBF instances. Alternatively, we also propose to use a QBF solver as an oracle. All QBF solvers produce outer variable assignments, so the idea is to modify the original formula to extract inner variable assignments. For example, consider a QBF formula ϕ_1 with prefix $\exists a_1, a_2 \forall b_1, b_2 \exists c_1, c_2$. Let us suppose we need to extract the values of c_1, c_2 when b_1, b_2 are set to False. We transform the formula ϕ_1 to ϕ_2, where b_1, b_2 are changed to existential variables and unit clauses are added to force them to False. Now the outermost layer of ϕ_2 also contains our target variables c_1, c_2, thus we can extract their assignments. This procedure can be extended to arbitrary quantifier alternations.

KI-V3   Using assertions in interactive play for validation

Interactive play allows us to extract the inner variables assignments. However, to ensure correctness, we need to check if the required conditions hold in the winning strategy. We propose to use assertions i.e., a set of clauses in CNF format which must hold in every possible game play. The assertions can be with new variables or the variables already used in the QBF encoding. This flexibility allows us to encode complex correctness condition checks. For validation, we combine the interactive

play with assertion checks. If the encoding is incorrect, there exists some game play (in the search tree) where some assertion is violated.

KI-V4   Using common winning strategies for equivalence check of similar encodings

The equivalence check for arbitrary encodings needs exploration of all assignments of variables, which can be impractical. Instead, we constrain the equivalence check between two similar encodings i.e., the encodings which share some common variables. For example, in case of 2-player games one could consider two encodings which share the same move variables. We also relax the equivalence check to checking if two instances share winning strategies, one could call it solution-equivalence. These two relaxations allow us to practically check for solution-equivalence between two encodings. First, we cannot simply use validation with interactive play for checking solution-equivalence between similar encodings. Instead, we observe that if two QBF instances encode the same problem then the winning strategies for one must be winning for the another. Checking for all winning strategies is not practical, instead we define one such equivalence check. The key idea is to rewrite the winning strategy of the first instance to avoid variable clashes and check if it is also winning for the second instance. Applying the solution-equivalence check on many QBF instances for the same encodings increase the correctness confidence. Similarly, checking if winning strategies from different QBF solvers hold for the same two QBF instances can also increase the confidence.

KI-V5   Partial certificates for scaling validation

Both validation and equivalence check still suffer from memory and time bottlenecks. In case of validation, the large certificate size is the bottleneck when using certificates for interactive play. On the other hand, a single iteration of interactive play with a QBF solver requires solving a QBF instance. To overcome both the memory and time bottlenecks, we propose to use partial certificates. Essentially, we extract the Skolem/Herbrand functions of only first k layers of the QBF instance. The interactive play now consists of two parts, first with the partial winning strategy and then with the QBF solver as an oracle. The idea of using partial certificates can also be used in case of equivalence check. The key observation is that, we do not need Skolem/Herbrand functions for all variables in the QBF instance. Instead, we only need such functions for the common variables which form the winning strategy.

4.3.2 Contribution

In Chapter 9, we investigated the research questions RQ-V1, RQ-V2 and RQ-V3. We discussed the key ideas from KI-V1 to KI-V5 and implemented them in our tool SQval (Scalable QBF validator). SQval[1] is an open source tool written in python, which does both the validation and equivalence checks of QBF instances. The tool also produces a trace when assertion is violated, thus helping in retracing the bug. In the paper, we conducted a case study on 2-player QBF encodings presented in [115]. We showed that validation and equivalence checks are feasible on such encodings. Further, we used partial certificates for scaling to deeper instances which could not be handled before.

The encodings for both positional games in Chapter 7 and grid games in Chapter 8 are complex. It is easy to miss bugs and many variations of the encodings make it a much harder task. To overcome this, we use similar interactive play with goal assertion checks. In our tool Q-sage, we also visualize the winning strategy of the QBF encoding for manual validation. In practice, we were able to find some bugs easily using this approach.

---

[1]https://github.com/irfansha/SQval

Part II

Publications

Chapter 5

Classical planning as QBF without grounding

Abstract

Most classical planners use grounding as a preprocessing step, essentially reducing planning to propositional logic. However, grounding involves instantiating all action rules with concrete object combinations, and results in large encodings for SAT/QBF-based planners. This severe cost in memory becomes a main bottleneck when actions have many parameters, such as in the Organic Synthesis problems from the IPC 2018 competition. We provide a compact QBF encoding that is logarithmic in the number of objects and avoids grounding completely, by using universal quantification for object combinations. We show that we can solve some of the Organic Synthesis problems, which could not be handled before by any SAT/QBF based planners due to grounding.

5.1   Introduction

Automated planning has many real-world applications, such as Space Exploration and Robotics, cf. the book by Ghallab et al. [49]. The three main research directions in automated planning are heuristic-based state-space search, symbolic search and propositional satisfiability (SAT) based solving. While heuristic-based search often finds some plan quickly, it may not guarantee to search the whole search space. SAT-based solvers, on the other hand, can also be used to prove the non-existence of plans up to a bounded length and finding optimal plans. For some applications, quickly falsifying the existence of a plan or finding provably optimal plans can be useful. Classical planning is the most simple problem; its aim is to find a valid sequence of actions from a single initial state to some goal state, where the state is completely known and the effect of all actions is deterministic. Kautz and Selman [70] reduced the planning problem to the bounded reachability problem and provided a corresponding SAT encoding for a plan of length k.

Classical planning domains are usually defined by logical rules that describe the pre-conditions and effects of applying actions. Usually, a single action involves

multiple objects, corresponding to the arity of that action. The pre-conditions and effects contain predicates, which also refer to multiple objects. A concrete planning problem defines a universe of concrete objects, and an initial and goal condition. Many planning tools, both SAT-based and heuristic, first apply a grounding step: the action rules are instantiated for all combinations of objects. However, for domains with actions involving many objects, the grounded specification is very large, so memory becomes the main bottleneck. Despite several improvements, such as action splitting [70], explanatory frame axioms [52], invariants [105] and parallel plans [107, 108], the memory still remains a bottleneck for domains with large action arity.

QBF (Quantified Boolean Formula) encodings are known for being more compact than SAT encodings, so they are considered as an alternative when SAT encodings suffer from memory problems. Dershowitz et al. [34] and Jussila and Biere [69] proposed a QBF encoding with $\exists\forall\exists$ quantifier alternation for reachability in Bounded Model Checking (BMC). It generates only a single copy of the transition function, instead of k copies in SAT, by using quantification over state variables. Rintanen [103] proposed a reachability QBF encoding that is logarithmic in the length of the plan, and uses only one transition function. Cashmore et al. [26] proposed the Compact Tree Encoding (CTE), which improves upon the logarithmic encoding by Rintanen (in the context of planning) by efficient traversal of the search tree. Although these QBF encodings are more concise than the SAT encodings, it has been reported that the SAT encodings can usually be solved faster than the QBF encodings by the current solvers [26]. More importantly, the QBF encodings mentioned above still require the problem to be grounded first, so the memory bottleneck for actions that involve many objects has not been solved.

Matloob and Soutchanski [83] proposed Organic Synthesis benchmarks and showed that SAT encodings could not solve any of them. This is consistent with the findings from the IPC-2018 planning competition, in which SAT based planners performed poorly on instances from Organic Synthesis. These benchmarks contain actions that manipulate up to 31 objects, so the grounding step exhausts the memory. Even heuristic planners that employ grounding will exhaust the memory for this domain. Thus, to solve these problems, there is a need for encodings that avoid grounding altogether.

Our Contribution. In this paper, we propose a QBF encoding of quantifier structure $\exists\forall\exists$ which completely avoids grounding, by using universal variables to represent combinations of objects. The encoding grows linearly in the number of action names, predicates, and path length, and logarithmic in the number of objects. We provide an open-source implementation[1] of the encoding. We evaluate our QBF encoding on 18 domains from previous IPC competitions and on 10 hard-to-ground domains, and try to solve them with QBF solver CAQE. We compare the size of the encoding and the time needed for solving them with the best SAT-based solver Madagascar, and two of the best heuristic solvers.

---

[1]https://github.com/irfansha/Q-Planner

5.2 Preliminaries

We introduce the main notions of the classical planning problems and QBF. We also present a running example of the blocks-world domain (Listing 3) in PDDL, the Planning Domain Definition Language [44].

5.2.1 Classical Planning

Definition 1. *A* planning signature $\Sigma = \langle P, A, O, \mathrm{arity} \rangle$ *is a 4-tuple where* P *is a set of predicate symbols,* A *is a set of action names,* O *is a set of objects, and the function* $\mathrm{arity} : (P \cup A \to \mathbb{N}_0)$ *indicates the expected number of arguments.*

In the example of Listing 3, $P = \{\mathrm{clear}, \mathrm{ontable}, \mathrm{on}\}$, $O = \{b_1, b_2\}$, $A = \{\mathrm{unstack}, \mathrm{stack}\}$, and $\mathrm{arity}(\mathrm{on}) = 2$. A schematic action corresponds to an action name with parameters, e.g. $\mathrm{stack}(?x_1, ?x_2)$. It can be grounded to a set of concrete actions, by replacing the parameters by concrete object combinations. Predicate symbols are also equipped with parameters, that can be grounded by concrete objects. To facilitate our encodings, we unify the parameter names of all actions to a fixed sequence of *x*'s, and the formal predicate parameters to a sequence of *y*'s. Note that adding dummy parameters does not change the problem. The fixed predicate parameter names will be used later for universal object combination variables in the QBF encoding. Since we use a sequential plan semantics, a single set of action and predicate parameter variables is sufficient to represent any chosen action at each time step. Each action is specified schematically by preconditions and effects.

Definition 2. *Given a signature, we define the set of atoms and grounded atoms (or fluents) as:*

$$\mathrm{Atom} = \{p(\overrightarrow{x}) \mid p \in P, |\overrightarrow{x}| = \mathrm{arity}(p)\}$$
$$F = \{p(\overrightarrow{o}) \mid p \in P, o_i \in O, |\overrightarrow{o}| = \mathrm{arity}(p)\}$$

Definition 3. *Given a signature* $\Sigma = \langle P, A, O, \mathrm{arity} \rangle$. *The planning problem* $\Pi = \langle I, G, \mathrm{pre}^+, \mathrm{pre}^-, \mathrm{eff}^+, \mathrm{eff}^- \rangle$ *is a 6-tuple where*

- *Initial state* $I \subseteq F$ *and*

- *Goal condition* $G = (g^+, g^-)$, *where* $g^+, g^- \subseteq F$.

- *for each action* $a \in A$, *we have:*

  - *positive preconditions* $\mathrm{pre}^+(a) \subseteq \mathrm{Atom}$
  - *negative preconditions* $\mathrm{pre}^-(a) \subseteq \mathrm{Atom}$
  - *positive effects* $\mathrm{eff}^+(a) \subseteq \mathrm{Atom}$
  - *negative effects* $\mathrm{eff}^-(a) \subseteq \mathrm{Atom}$

In the example, (on $?x_1$ $?x_2$) is a positive precondition for action `unstack`, while (ontable $?x_1$) is a negative effect for action `stack`. We assume that the variables used in preconditions and effects are (non-dummy) action parameters.

Listing 3 blocks-world domain and problem

```
1   (define (domain blocksworld)
2   (:predicates (clear ?y₁)(ontable ?y₁)(on ?y₁ ?y₂))
3   (:action unstack
4    :parameters (?x₁ ?x₂)
5    :precondition (and (clear ?x₁)(on ?x₁ ?x₂))
6    :effect (and (not (on ?x₁ ?x₂))(ontable ?x₁)(clear ?x₂)))
7   (:action stack
8    :parameters (?x₁ ?x₂)
9    :precondition (and (clear ?x₁)(clear ?x₂)(ontable ?x₁))
10   :effect (and (not(clear ?x₂))(not(ontable ?x₁))(on ?x₁ ?x₂))))
11  (define (problem BW_rand_2)
12  (:domain blocksworld)
13  (:objects b₁ b₂)
14  (:init (ontable b₁) (on b₂ b₁) (clear b₂))
15  (:goal (and (on b₁ b₂))))
```

5.2.2   SAT Encoding

The planning problem is encoded as finding a path (of length k) in a graph, whose nodes consist of assignments to the fluents and whose edges are defined by the conditions and effects of grounded actions. Usually, a SAT encoding uses propositional variables encoding the k actions and $k+1$ states in the path. A direct encoding would use k variables for each grounded action, but since we use sequential plan semantics (in which only one action occurs at the time), this can be encoded by a number of variables that is logarithmic in the number of grounded actions (still linear in k).

We will present the constraints in circuit format (with nested $\wedge$ and $\vee$), which can be transformed to CNF clauses by a standard procedure. We will write $\bigwedge$ and $\bigvee$ for conjunctions and disjunctions over finite (indexed) sets.

5.2.3   Quantified Boolean Formulas

Quantified Boolean Formulas (QBF) extend propositional formulas in SAT with universal quantification over some variables. A QBF formula in *prenex normal form* is of the shape $\Pi\phi$ where ϕ is called the matrix, which is a propositional formula, and $\Pi = Q_1X_1\ldots Q_nX_n$ is the prefix with alternating existential and universal quantifiers $Q_i \in \{\forall, \exists\}$ and disjoint sets of variables X_i. The Q_i specify the quantification of each variable that occurs in ϕ. We consider only closed formulas, i.e., the quantification of all variables is known. A QBF is evaluated to either True or False and its truth value can be computed by recursively solving the formula over each outermost variable. A formula $\exists x\phi$ is true if and only if $\phi[\top/x]$ or $\phi[\bot/x]$ is true. A formula $\forall x\phi$ is true if and only if both $\phi[\top/x]$ and $\phi[\bot/x]$ are true. The order of the variables within each quantified block does not matter.

5.3 Intermediate SAT Encoding

We now provide the intermediate SAT encoding of planning problems, as an introduction to our QBF encoding. Throughout the section, we assume a fixed planning problem as in Definition 3. Recall that we introduced a fixed sequence of action parameters x.

For the encoding, we generate copies of variables for each time step (represented by superscript) for a path of length k. We use action variables $A^i = \{a_b^i \mid 1 \leq b \leq \sigma\}$ for each $0 \leq i \leq k-1$ and $\sigma = \lceil \log(|A|) \rceil$; here a_b^i represents the b-th bit of a logarithmic encoding of the action name that occurs at time i in the plan. The action parameter variables are $PM^i = \{x_{j,b}^i \mid 1 \leq j \leq \zeta, 1 \leq b \leq \gamma\}$ for each $0 \leq i \leq k-1$, where ζ is the maximum action arity and $\gamma = \lceil \log(|O|) \rceil$. Here $x_{j,b}^i$ represents the b-th bit of parameter x_j of the action scheduled at time i in the plan. We will write $\overrightarrow{x_j^i}$ (sequence of γ variables) for the parameter x_j at time i. The state variables are $F^i = \{f_{p(\overrightarrow{o})}^i \mid p(\overrightarrow{o}) \in F\}$, i.e., one variable for every fluent and for each $0 \leq i \leq k$.

The corresponding SAT encoding is based on these variables, and constraints for the initial condition I, goal condition G, transition function T_g, and a domain restriction RC.

$$\exists A^0, PM^0, \ldots, A^{k-1}, PM^{k-1} \exists F^0, \ldots, F^k$$

$$I(F^0) \wedge G(F^k) \wedge \bigwedge_{i=0}^{k-1} T_g^i(F^i, F^{i+1}, A^i, PM^i) \wedge$$

$$\bigwedge_{i=0}^{k-1} RC(A^i, PM^i)$$

In the constraint for the initial state, each variable is positive if the proposition is in the set I and negative if it is not. For the goal state, the variable is positive if the proposition is in g^+ and negative if it is in g^-. Note that this provides a unique initial state, but there may be multiple states that satisfy the goal condition. We group the constraints (throughout the article) based on the predicates for the sake of easy understanding of the correctness proof.

Definition 4. *Initial constraint* $I(F^0) =$

$$\bigwedge_{p \in P} \bigwedge_{p(\overrightarrow{o}) \in F} \{f_{p(\overrightarrow{o})}^0 \mid p(\overrightarrow{o}) \in I\} \wedge$$

$$\bigwedge_{p \in P} \bigwedge_{p(\overrightarrow{o}) \in F} \{\neg f_{p(\overrightarrow{o})}^0 \mid p(\overrightarrow{o}) \notin I\}$$

Definition 5. *Goal constraint* $G(F^k) =$

$$\bigwedge_{p \in P} \bigwedge_{p(\overrightarrow{o}) \in F} \{f^k_{p(\overrightarrow{o})} \mid p(\overrightarrow{o}) \in g^+\} \wedge$$

$$\bigwedge_{p \in P} \bigwedge_{p(\overrightarrow{o}) \in F} \{\neg f^k_{p(\overrightarrow{o})} \mid p(\overrightarrow{o}) \in g^-\}$$

To encode the transition function, we will generate five constraints for each proposition variable. These constraints define the value of all fluents and time stamps, just before or after some action occurs. The first four constraints correspond to the positive preconditions, negative preconditions, positive effects and negative effects, respectively. The last constraint corresponds to the frame axiom, which indicates that untouched propositions should not change.

Definition 6. *The **grounded transition function** is:*

$$T^i_g(F^i, F^{i+1}, A^i, PM^i) = \bigwedge_{p \in P} \bigwedge_{p(\overrightarrow{o}) \in F} PC^i_{p(\overrightarrow{o})}$$

*where the **Proposition Constraint** $PC^i_{p(\overrightarrow{o})}$ =*

$$(AC_{p(\overrightarrow{o})}(A^i, PM^i, pre^+) \implies f^i_{p(\overrightarrow{o})}) \wedge$$

$$(AC_{p(\overrightarrow{o})}(A^i, PM^i, pre^-) \implies \neg f^i_{p(\overrightarrow{o})}) \wedge$$

$$(AC_{p(\overrightarrow{o})}(A^i, PM^i, eff^+) \implies f^{i+1}_{p(\overrightarrow{o})}) \wedge$$

$$(AC_{p(\overrightarrow{o})}(A^i, PM^i, eff^-) \implies \neg f^{i+1}_{p(\overrightarrow{o})}) \wedge$$

$$((f^i_{p(\overrightarrow{o})} = f^{i+1}_{p(\overrightarrow{o})}) \vee AC_{p(\overrightarrow{o})}(A^i, PM^i, eff^+) \vee$$

$$AC_{p(\overrightarrow{o})}(A^i, PM^i, eff^-))$$

Consider the positive precondition constraint: in plain words it expresses that if some action occurs at time step *i*, and some fluent (grounded predicate) is in the positive precondition of that action, then the corresponding variable is true at time step *i*. Similarly, the proposition variables corresponding to the positive (negative) effects should be set to true (false) at time step $i+1$. The last constraint encodes the frame axiom: it expresses that either the value of a proposition variable stays the same, or some positive or negative effect occurs that defines this proposition.

We still need to encode when a positive/negative condition/effect Φ is associated with the current action. We also need to define the set of grounded instances of a predicate.

Definition 7. *Given an action* $a \in A$, *a set of atoms* $\Phi \in \{pre^+, pre^-, eff^+, eff^-\}$, *and sequence of objects* $\overrightarrow{o} \in O^n$ *with* $n = \text{arity}(a)$, *its **grounding** $GD(\Phi, a, \overrightarrow{o}) \subseteq F$ is a set of fluents, defined* $GD(\Phi, a, \overrightarrow{o}) := \{\phi[\overrightarrow{x}/\overrightarrow{o}] \mid \phi \in \Phi(a)\}$.

The grounding of an action with given object parameters returns the corresponding grounded preconditions/effects. In the running example, the positive precondition of action stack for object parameters (b_1, b_2) is grounded as $\text{GD}(\text{pre}^+, \text{stack}, (b_1, b_2)) = \{\text{clear}(b_1), \text{clear}(b_2), \text{ontable}(b_1)\}$.

Below, we use "bin" to express the logarithmic encoding of objects by γ bits and of action names by σ bits. We also use "=" to denote a bit-wise conjunction of bi-implications.

Definition 8. *Given the set of action names* A *and a set of atoms*
$\Phi \in \{\text{pre}^+, \text{pre}^-, \text{eff}^+, \text{eff}^-\}$ *and a fluent* f,
*the **grounded action constraint*** $\text{AC}_f(A^i, \text{PM}^i, \Phi) =$

$$\bigvee_{\substack{a \in A}} \bigvee_{\substack{n=\text{arity}(a) \\ \overrightarrow{o} \in O^n \ s.t. \\ f \in \text{GD}(\Phi, a, \overrightarrow{o})}} (\overrightarrow{A^i} = \text{bin}(a)) \wedge \bigwedge_{j=1}^{n} (\overrightarrow{x_j^i} = \text{bin}(o_j))$$

In the grounded action constraint, for each grounded action $a(\overrightarrow{o})$ that contains fluent f in its grounded precondition/effect Φ, we generate equality constraints for action and parameter variables. For example, for fluent $\text{clear}(b_1)$, $\Phi = \text{pre}^+$ and grounded action $\text{stack}(b_1, b_2)$, we generate a constraint $\overrightarrow{A^i} = \text{bin}(\text{stack}) \wedge \overrightarrow{x_1^i} = \text{bin}(b_1) \wedge \overrightarrow{x_2^i} = \text{bin}(b_2)$.

Finally, due to the use of logarithmic variables, invalid actions are possible when the number of objects and actions are non-powers of 2. To maintain consistency we restrict the invalid action and parameter values by the RC-constraint. Here $<$ denotes the bit-blasted comparison operator on binary numbers.

Definition 9. *Restricted constraints* $\text{RC}(A^i, \text{PM}^i) =$

$$(\overrightarrow{A^i} < |A|) \wedge \bigwedge_{1 \leq j \leq \zeta} (\overrightarrow{x_j^i} < |O|)$$

In practice, this constraint can still be refined if one has type information on the objects, as in typed PDDL domains.

5.4 New Ungrounded QBF Encoding

Similar to the intermediate SAT encoding, the QBF encoding of a plan of length k corresponds to finding a path of length k from the initial state to some goal state. The SAT encoding used propositional variables for each object-instance of predicate symbols and each time step. Note that the values of these propositional variables only depend on the action variables, i.e., given a sequence of actions and (complete) initial state, the intermediate states until the final state are completely determined. From the

constraints on the propositional variables, one can also observe that their values are independent of each other, so these constraints can be enforced independently, at each time step and for each action.

For the QBF encoding, the idea is to avoid grounding by representing all object combinations as universal variables and generate constraints on the predicate variables directly. Essentially the universal object combination variables and existential predicate variables replace the existential propositional variables in the intermediate SAT encoding.

In the sequel, we assume a fixed planning problem as in Definition 3, and a fixed planning signature as in Definition 1. Recall that we introduced fixed variables *x* for formal action parameters and fixed variables *y* for formal predicate variables. These variables are used in the transformation.

The QBF encoding will use the same action variables (A) and action parameter variables (PM) as the SAT encoding. We define object combination variables and predicate variables separately. The object combination variables are $OC = \{y_{j,b} \mid 1 \le j \le \eta, 1 \le b \le \gamma\}$, where η is the maximum arity of predicates and γ is as defined before. Here, $y_{j,b}$ represents the *b*-th bit of the *j*th object variable and $\overrightarrow{y_j}$ (sequence of γ variables) represents the *j*th object variable. The (*ungrounded*) predicate variables are $P^i = \{q_p^i \mid p \in P\}$ for each $0 \le i \le k$.

The corresponding Ungrounded Encoding is:

$$\exists A^0, PM^0, \ldots, A^{k-1}, PM^{k-1} \,\underline{\forall}\, OC \,\exists\, P^0, \ldots, P^k$$

$$I_u(P^0, OC) \wedge G_u(P^k, OC) \wedge$$

$$\bigwedge_{i=0}^{k-1} T_u^i(P^i, P^{i+1}, OC, A^i, PM^i) \wedge \bigwedge_{i=0}^{k-1} RC(A^i, PM^i)$$

In the sequel, we will define the auxiliary constraints for the initial and goal states and the transition functions. The domain restriction RC does not depend on the universal variables OC, so it remains unchanged from the SAT encoding (Definition 9). We refer to the example in Figure 5.1 for an illustration of the encoding running example of Listing 3. At the end of this section, we comment on the equivalence of the SAT and the QBF encoding. Basically, when the universal variables in the ungrounded QBF encoding are expanded, the resulting conjunction of constraints is equivalent to the constraints in the intermediate SAT encoding.

The initial and goal constraints must apply to particular object combinations, so for each predicate, we specify in which for-all branches, i.e., for which objects instantiations, they are positive or negative:

Definition 10. *The **Ungrounded Initial** constraint is defined as* $I_u(P^0, OC)$*:*

$$\bigwedge_{p \in P} \left(\bigvee_{p(\overrightarrow{o}) \in I} \bigwedge_{j=1}^{|\overrightarrow{o}|} (\text{bin}(o_j) = \overrightarrow{y_j}) \right) \iff q_p^0$$

Prefix:

$\exists \overrightarrow{A^0} \exists \overrightarrow{x_1^0} \exists \overrightarrow{x_2^0} \quad \exists \overrightarrow{A^1} \exists \overrightarrow{x_1^1} \exists \overrightarrow{x_2^1}$

$\forall \overrightarrow{y_1} \forall \overrightarrow{y_2}$

$\exists q_{clear}^0 \exists q_{ontable}^0 \exists q_{on}^0$

$\exists q_{clear}^1 \exists q_{ontable}^1 \exists q_{on}^1$

$\exists q_{clear}^2 \exists q_{ontable}^2 \exists q_{on}^2$

Initial state:

$((\text{bin}(b_2) = \overrightarrow{y_1}) \iff q_{clear}^0) \wedge$

$((\text{bin}(b_1) = \overrightarrow{y_1}) \iff q_{ontable}^0) \wedge$

$((\text{bin}(b_2) = \overrightarrow{y_1}) \wedge (\text{bin}(b_1) = \overrightarrow{y_2}) \iff q_{on}^0) \wedge$

Goal state:

$((\text{bin}(b_1) = \overrightarrow{y_1}) \wedge (\text{bin}(b_2) = \overrightarrow{y_2})) \implies q_{on}^2 \wedge$

For time steps i = 0,1:

clear:

$((\overrightarrow{A^i} = \text{bin}(\text{stack}) \wedge (\overrightarrow{x_1^i} = \overrightarrow{y_1} \vee \overrightarrow{x_2^i} = \overrightarrow{y_1})) \vee$

$\quad (\overrightarrow{A^i} = \text{bin}(\text{unstack}) \wedge \overrightarrow{x_1^i} = \overrightarrow{y_1})) \implies q_{clear}^i \wedge$

$(\overrightarrow{A^i} = \text{bin}(\text{unstack}) \wedge \overrightarrow{x_2^i} = \overrightarrow{y_1}) \implies q_{clear}^{i+1} \wedge$

$(\overrightarrow{A^i} = \text{bin}(\text{stack}) \wedge \overrightarrow{x_2^i} = \overrightarrow{y_1}) \implies \neg q_{clear}^{i+1} \wedge$

$((q_{clear}^i = q_{clear}^{i+1}) \vee (\overrightarrow{A^i} = \text{bin}(\text{unstack}) \wedge \overrightarrow{x_2^i} = \overrightarrow{y_1})$

$\quad \vee (\overrightarrow{A^i} = \text{bin}(\text{stack}) \wedge \overrightarrow{x_2^i} = \overrightarrow{y_1})) \wedge$

ontable:

$(\overrightarrow{A^i} = \text{bin}(\text{stack}) \wedge \overrightarrow{x_1^i} = \overrightarrow{y_1}) \implies q_{ontable}^i \wedge$

$(\overrightarrow{A^i} = \text{bin}(\text{unstack}) \wedge \overrightarrow{x_1^i} = \overrightarrow{y_1}) \implies q_{ontable}^{i+1} \wedge$

$(\overrightarrow{A^i} = \text{bin}(\text{stack}) \wedge \overrightarrow{x_1^i} = \overrightarrow{y_1}) \implies \neg q_{ontable}^{i+1} \wedge$

$((q_{ontable}^i = q_{ontable}^{i+1}) \vee (\overrightarrow{A^i} = \text{bin}(\text{unstack}) \wedge \overrightarrow{x_1^i} = \overrightarrow{y_1})$

$\quad \vee (\overrightarrow{A^i} = \text{bin}(\text{stack}) \wedge \overrightarrow{x_1^i} = \overrightarrow{y_1})) \wedge$

on:

$(\overrightarrow{A^i} = \text{bin}(\text{unstack}) \wedge \overrightarrow{x_1^i} = \overrightarrow{y_1} \wedge \overrightarrow{x_2^i} = \overrightarrow{y_2}) \implies q_{on}^i \wedge$

$(\overrightarrow{A^i} = \text{bin}(\text{stack}) \wedge \overrightarrow{x_1^i} = \overrightarrow{y_1} \wedge \overrightarrow{x_2^i} = \overrightarrow{y_2}) \implies q_{on}^{i+1} \wedge$

$(\overrightarrow{A^i} = \text{bin}(\text{unstack}) \wedge \overrightarrow{x_1^i} = \overrightarrow{y_1} \wedge \overrightarrow{x_2^i} = \overrightarrow{y_2}) \implies \neg q_{on}^{i+1} \wedge$

$((q_{on}^i = q_{on}^{i+1}) \vee (\overrightarrow{A^i} = \text{bin}(\text{stack}) \wedge \overrightarrow{x_1^i} = \overrightarrow{y_1} \wedge \overrightarrow{x_2^i} = \overrightarrow{y_2})$

$\quad \vee (\overrightarrow{A^i} = \text{bin}(\text{unstack}) \wedge \overrightarrow{x_1^i} = \overrightarrow{y_1} \wedge \overrightarrow{x_2^i} = \overrightarrow{y_2}))$

Figure 5.1: Ungrounded Encoding for blocks-world for $k = 2$

Definition 11. *The **Ungrounded goal** constraint is defined as* $G_u(P^k, OC) = G^+ \wedge G^-$, *where*

$$G^+ = \bigwedge_{p \in P} (\bigvee_{p(\overrightarrow{o}) \in g^+} \bigwedge_{j=1}^{|\overrightarrow{o}|} (\mathrm{bin}(o_j) = \overrightarrow{y_j})) \implies q_p^k,$$

$$G^- = \bigwedge_{p \in P} (\bigvee_{p(\overrightarrow{o}) \in g^-} \bigwedge_{j=1}^{|\overrightarrow{o}|} (\mathrm{bin}(o_j) = \overrightarrow{y_j})) \implies \neg q_p^k$$

For each predicate, if any of the action constraints is true in the preconditions or effects, then the corresponding variable is constrained to the appropriate value. We will expand on the action constraints AC_p^u in the definition 13.

Definition 12. *The **ungrounded transition function***

$$T_u^i(P^i, P^{i+1}, OC, A^i, PM^i) = \bigwedge_{p \in P} UPC_p^i$$

*where the **ungrounded predicate constraint*** $UPC_p^i =$

$$(AC_p^u(A^i, PM^i, OC, \mathrm{pre}^+) \implies q_p^i) \wedge$$
$$(AC_p^u(A^i, PM^i, OC, \mathrm{pre}^-) \implies \neg q_p^i) \wedge$$
$$(AC_p^u(A^i, PM^i, OC, \mathrm{eff}^+) \implies q_p^{i+1}) \wedge$$
$$(AC_p^u(A^i, PM^i, OC, \mathrm{eff}^-) \implies \neg q_p^{i+1}) \wedge$$
$$((q_p^i = q_p^{i+1}) \vee AC_p^u(A^i, PM^i, OC, \mathrm{eff}^+) \vee$$
$$AC_p^u(A^i, PM^i, OC, \mathrm{eff}^-))$$

The constraints are similar to the SAT encoding. For example, the positive precondition constraint states that a predicate must have been true at step i, whenever some action with this predicate in its precondition and matching action parameters occurs at time step i. The final constraint is called the frame axiom; it says that either the value of predicate stays the same, or some matching positive or negative effect has occurred. Next, we define in which OC-branch action A^i with parameters PM^i matches condition/effect Φ.

Definition 13. *Given* $\Phi \in \{\mathrm{pre}^+, \mathrm{pre}^-, \mathrm{eff}^+, \mathrm{eff}^-\}$, *the **ungrounded action constraint***

$$AC_p^u(A^i, PM^i, OC, \Phi) = \bigvee_{a \in A} ((\overrightarrow{A^i} = \mathrm{bin}(a)) \wedge$$

$$\bigvee_{p(x_{l_1}, \dots, x_{l_n}) \in \Phi(a)} \bigwedge_{j=1}^{n} (\overrightarrow{x_{l_j}^i} = \overrightarrow{y_j}))$$

Similar to the initial and goal constraints, the value of the predicate variables must be constrained in each for-all branch. As we discussed before, each for-all branch corresponds to one instantiation of object combinations. The ungrounded action constraint specifies which branches are evaluated to true. In plain words, for every for-all branch where the predicate parameters variables are equal to the universal variables, the ungrounded action constraint is evaluated to true. Note that the predicate parameters from the preconditions and effects are simply a subset of the action parameter variables, by the static semantics of PDDL.

Regarding the frame axioms, we use the following performance optimizations in our implementation: We do not propagate static predicates, i.e., predicates that do not appear in any of the action effects. Instead, we introduce only one instance, reused across all time steps. In particular, the type predicates in typed planning domains are handled as static predicates. Equality-predicates are treated specially, since they impose constraints between action parameters only, independent of the universal variables. We simply generate (in)equality constraints between action parameters directly, similar to the RC-constraint. We discuss this in more detail in the technical report [110].

5.4.1 Correctness

We demonstrate correctness, by showing how the SAT encoding can be stepwise transformed into the equivalent QBF encoding. Let $\delta = 2^{\gamma \times \eta}$ be the number of object combinations, and $n = |P|$. Then the constraints on the propositions in time step i and $i+1$ in the SAT encoding are:

$$\exists f^i_{p_1(\overrightarrow{o_1})} \ldots \exists f^i_{p_1(\overrightarrow{o_\delta})} \quad \cdots$$
$$\exists f^i_{p_n(\overrightarrow{o_1})} \ldots \exists f^i_{p_n(\overrightarrow{o_\delta})}$$
$$\bigwedge_{p \in P} \bigwedge_{p(\overrightarrow{\sigma}) \in F} PC^i_{p(\overrightarrow{\sigma})}$$

Here the existential blocks of propositional variables are grouped based on the predicate symbols. Since each of the propositions appears in only one constraint, we can push the existential variables inside the conjunction. Thus the constraints are equivalent to the following groups of constraints in non-prenex form:

$$\exists f^i_{p_1(\overrightarrow{o_1})} \ldots \exists f^i_{p_1(\overrightarrow{o_\delta})} \bigwedge_{p_1(\overrightarrow{\sigma}) \in F} PC^i_{p_1(\overrightarrow{\sigma})}$$
$$\wedge \cdots \wedge$$
$$\exists f^i_{p_n(\overrightarrow{o_1})} \ldots \exists f^i_{p_n(\overrightarrow{o_\delta})} \bigwedge_{p_n(\overrightarrow{\sigma}) \in F} PC^i_{p_n(\overrightarrow{\sigma})}$$

Since all the constraints have a uniform shape, we can abbreviate the conjunction by a for-all quantification in QBF. So the previous formula is equivalent to the grouping

with object combination variables in the QBF encoding:

$$\forall OC \; (\exists q^i_{p_1} \, UPC^i_{p_1} \wedge \cdots \wedge \exists q^i_{p_n} \, UPC^i_{p_n})$$

Remember that the action and parameter variables are the same for both SAT and QBF encoding. This proves the equivalence between the SAT and the QBF encodings.

5.5   Implementation and Experiments

We implemented our encoding of PDDL problems to QBF problems in QCIR format as a Python program, which is available online.[2] The tool handles domains with types and equality predicates by treating them as static predicates; it also handles negative preconditions and constants, but it cannot yet handle domains with conditional effects. To solve the encoded problems for a given length k, we transform them to QDIMACS format, and use the QBF solver CAQE [98] with the internal preprocessor Bloqqer [11]. Some initial experiments with other preprocessors and solvers indicated that CAQE+Bloqqer is a very good combination; we leave a systematic comparison as future work. Finally, our tool extracts a concrete plan from CAQE's output, and validates that the plan is indeed correct. In the sequel, we refer to the combination of our encoding, CAQE and Bloqqer as "Q-Planner".

We performed two experiments: in the first experiment, we run Q-Planner on a number of Hard-to-Ground (htg) domains. These include planning problems from 4 Organic Synthesis domains: 2 from IPC-18 (non-split Satisfying and Optimizing track) and the 2 original benchmarks (Alkene and Mitexams) [82]. The simplified domains submitted to IPC-18 received an outstanding domain submission award. In addition, we also consider other Hard-to-ground domains (htg): Genome-edit-distance (without costs), Pipesworld from Corrêa et al. [31] and Childsnack, Visitall, Blocks and Logistics from Lauer et al. [75]. For these domains, we consider a subset of instances which capture the hardness in grounding. We compare Q-Planner with 3 state-of-the-art planners, specified below.

In the second experiment, we encoded all problems from previous IPC planning competitions that our translation can handle, resulting in 18 domains. We are mainly interested in the number of instances that can be solved in each domain (within a given time and memory limit). In this experiment, we compared Q-Planner with Madagascar, using a simple SAT encoding (no invariants or parallel plans) to compare the effect of our ungrounded encoding on the encoding size, solving time, and memory usage.

5.5.1   Experimental Setup

For the first experiment, we compare Q-planner (QP) with 3 other planners: (1) Madagascar (version M) [101] with relaxed existential step encoding and invariant synthesis, a SAT based planner based on grounding; (2) Fast Downward Stone Soup

---

[2]https://github.com/irfansha/Q-Planner

	Domains (Total)	QBF/SAT		Non-SAT	
		QP	M	FDS	PL
LA	Alkene (18)	18	1	18	18
	OS-Sat18 (20)	12	-	3	15
	OS-Opt18 (20)	18	-	9	20
	MitExams (20)	6	-	1	12
	Ged (20)	7	5	20	20
	Pipesworld (20)	16	9	19	18
LP	Childsnack (16)	6	11	16	15
	Visitall (18)	2	-	6	18
LO	Blocks (12)	6	4	12	11
	Logistics (12)	-	-	12	12
	Total (176)	91	30	116	159

Table 5.1: Instances solved with 300 GB memory and 3 hour time limit. Running 4 tools on 10 Hard-to-ground domains, among them 6 large action arity (LA), 2 large predicate arity (LP), and 2 large objects (LO) domains.

2018 (FDS) [57], a non-SAT based planner which was the winner of IPC-2018 competitions in the satisfying track; and (3) Powerlifted (PL) [30], a non-SAT based planner which avoids grounding and is the state-of-the-art for Organic Synthesis. We use recommended configurations for all three planners for fair comparisons. We allow 300GB main memory and 3 hours per instance.

For the second experiment, we want to compare the ungrounded QBF encoding (using Q-Planner) with the grounded SAT encoding (using Madagascar). To eliminate other factors, we use Madagascar in its simplest configuration without invariants and without parallel plans (here called M-simple), i.e., a standard sequential SAT encoding with direct encoding of objects and actions. Since the CAQE solver calls SAT-solver Crypto-minisat [120], we also use Crypto-minisat to solve the SAT encodings in Madagascar. We allow 8GB main memory and 5000 seconds time for solving each instance.

For both experiments, we increase the path length in steps of 5 (consistent with Madagascar). We ran all computations for our experiments on the Grendel cluster.[3]

5.5.2 Results

We now discuss the results of the experiments.[4] For the first experiment (Table 5.1), Q-Planner solves 54 Organic Synthesis instances where as Madagascar only solves 1

---

[3]`http://www.cscaa.dk/grendel/`, each problem uses one core on a Huawei FusionServer Pro V1288H V5 server, with 384 GB main memory and 48 cores of 3.0 GHz (Intel Xeon Gold 6248R).

[4]All benchmarks, logs and statistics are avaialable at Zenodo, `https://doi.org/10.5281/zenodo.6367523`.
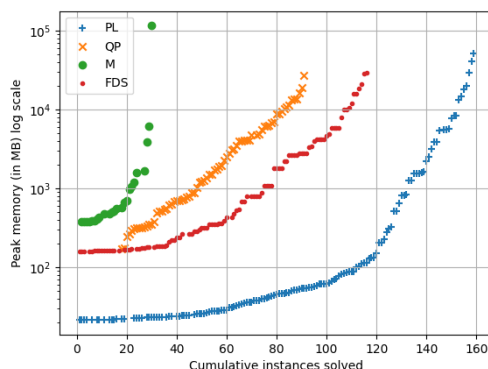
Figure 5.2: htg instances solved in given peak memory.
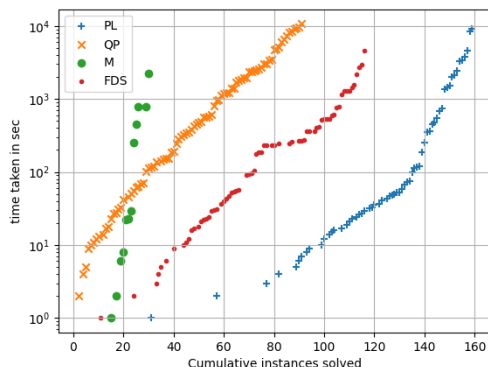
Figure 5.3: htg instances solved in given time.

using 115GB memory. These results are consistent with the IPC-2018 competitions, where 2 variants of Madagascar could not solve any of the (non-split) organic synthesis benchmarks in the satisfying track based on SAT solving. This is due to the presence of actions with many parameters (up to 31), thus even with action splitting and other optimizations the grounded encodings are too big. In total, Q-Planner solves 91 instances whereas Madagascar only solves 30 instances. For the Childsnacks domain Madagascar performs better than Q-Planner. This happens only for instances with a few action parameters, where Madagascar finds long plans quickly, using parallel plans. In most cases, Madagascar exhausts all 300GB memory before reaching the time limit, highlighting the memory bottleneck in htg domains.

We also provide cactus plots showing how many instances can be solved within a given memory (Fig. 5.2) or time limit (Fig. 5.3). In both aspects, Q-Planner clearly outperforms Madagascar significantly in the htg domains.

Concerning the non-SAT based planners: FDS performs well, solving 116 instances. On most of the remaining instances it exhausts all memory in the grounding phase. The Powerlifted planner solves 159 instances. Its excellent performance is as expected, since it also avoids grounding on actions while using optimized database queries on grounded states. Still, there are some Organic Synthesis instances where

	Domains	Q-planner		M-simple	
		#solved	size	#solved	size
Typed IPC domains	Blocks	28	0.14	**30**	4.8
	DriverLog	**11**	0.27	**11**	5.7
	Elevator	34	0.20	**43**	2.1
	Hiking	**9**	0.49	6	75.8
	Satellite	**6**	0.23	**6**	1.7
	Tidybot	1	1.7	**13**	519.9
	Termes	2	0.68	**3**	14.3
	Thoughtful	0	1.7	**5**	857.5
	Visitall	7	0.41	**13**	2.5
Untyped IPC domains	Blocks-3op	17	0.16	**19**	63.2
	Movie	**30**	0.15	**30**	0.28
	Depot	4	0.32	**6**	70.9
	Gripper	3	0.19	**5**	0.80
	Logistics	**12**	0.34	**12**	4.0
	Mprime	17	0.36	**34**	1954
	Mystery	3	0.36	**13**	399.1
	Grid	**2**	0.52	**2**	47.9
	Freecell	2	1.1	**6**	631.1
	Total sum	188	9.32	257	4656

Table 5.2: Instances solved within 5000s and 8GB internal memory. We also show the size (in MB) of the encoding for the maximum instance after preprocessing (if possible).

Powerlifted exhausts all memory. Q-Planner, on the other hand, might exceed the time-limit, but never used more than 28GB, even for refuting the existence of a plan of shorter length, when it could not solve the instance completely. The technical report contains more experiments and information on optimal plans [110]. Interestingly, Q-Planner solves one instance uniquely in the hardest set of benchmarks, i.e., MitExams.

Concerning the results of the second experiment (Table 5.2), the Ungrounded QBF Encoding never exceeds 1.7MB, so it is orders of magnitude more compact than the simple SAT encoding, which sometimes exceeds 1950MB. When comparing the performance of solving, the situation is different: In most domains, Madagascar solves more instances than Q-Planner, using less time and memory, despite the larger encodings. This is consistent with the observations of Cashmore et al. [26] on SAT vs QBF.

From the results on 18 IPC domains, it is clear that (at least with the current QBF solvers) our QBF encoding is not a replacement for SAT encodings, but rather a complement for domains where grounding is hard.

5.6   Related Work

We first discuss some approaches in the literature that avoid grounding in SAT encodings by allowing quantification, using EPR, CP, or SMT. Then we discuss related work on QBF.

Navarro Pérez and Voronkov [88] presented an encoding of planning problems in EPR (Effectively Propositional Logic). While EPR can encode conditional effects, satisfiability is NEXPTIME-complete [77]. Our QBF encoding stays in the 3rd level of the polynomial hierarchy, combining a compact encoding with more efficient solving.

Other approaches, such as numerical planning as SMT (Satisfiability Modulo Theories) by Bofill et al. [15] and numerical planning as CP (Constraint Programming) by Espasa et al. [40], also avoid grounding for compact encodings. It would be interesting to apply those approaches to Organic Synthesis.

Cashmore et al. [27] proposed a QBF encoding with *partial grounding*, which uses universal variables for a subset of ungrounded propositions only. They use *universal variables for only one object*. This leads to problems with propagating untouched propositions. To avoid incorrect plans, they need to introduce the restriction that predicates can have at most one ungrounded parameter. This is costly: if a predicate parameter is grounded, several action parameters must also be grounded.

We see our work as a generalization of their technique. We use *object combinations* in the universal layer of the encoding, thus avoiding grounding completely. This results in more compact QBF encodings. The software for partial grounding encoding is not available for comparison. Partial grounding of some parameters is trivial in our approach (it can be done on the domain file directly). However, we expect that partial grounding would be suboptimal. In case of organic synthesis, partially grounding some predicates still results in very large encodings.

The main focus of compact QBF encodings for reachability, synthesis and planning has been on reducing the number of transition function copies [26, 103]. While avoiding unrolling reduces the encoding size, it comes at the cost of losing inference between different time steps. On the other hand, our approach using universal quantification to avoid grounding, while keeping the inference between time steps, might be a better way to use the advances in the QBF solving for planning problems. Since the encodings are generated directly from (restricted) First-order Logic representations, the resulting encodings are compact and more scalable.

5.7   Conclusion and Future Work

In this paper, we propose the Ungrounded QBF Encoding for classical planning problems, which results in compact encodings by avoiding grounding completely. The size of this QBF encoding grows linearly in the number of action names, predicates and the path length, and logarithmic in the number of objects. We provide an open source implementation of the Ungrounded Encoding, translating classical planning

problems in PDDL to QBF problems in QCIR format. The resulting QBF formulas can be preprocessed and solved by existing QBF tools (we used Bloqqer and CAQE).

The experiments show that our encoding effectively avoids the memory bottleneck due to grounding. This is relevant for domains with actions that involve many objects, such as Organic Synthesis. Our main result is to solve many instances of Organic Synthesis, which so far could not be handled by any SAT/QBF based planner. We solve 54 problems in the 4 Organic Synthesis domains. We compare our implementation with the state-of-the-art SAT based planner Madagascar, which only solves 1 instance, using 115GB memory. Even on other htg domains, we outperform Madagascar when grounding is the bottleneck. In total we solve 91 instances where as the Madagascar solves 30 instances. We also compare with the non-SAT based planners Fast Downward Stone Soup 2018 (FDS) and Powerlifted, which perform better than any SAT/QBF approach.

Future Work.   There are several research directions from here. One direction is to extend the Ungrounded Encoding to planning problems with uncertainty or planning with adversaries. QBF encodings have been demonstrated to be useful for such domains, such as conformant planning [102, 104]. To this end, the Ungrounded Encoding must be extended, in order to handle some dependencies between state predicates, and to handle conditional effects.

Another direction is to improve the efficiency of the Ungrounded Encoding: One could consider to incorporate automatically generated grounded invariants [105] or lifted invariants [43] to speed up the search. Similarly, one could consider extending the Ungrounded Encoding with compact QBF encodings, such as the non-copying Iterative Squaring Encoding [103] and the Compact Tree Encoding [26]. These encodings could in principle lead to an even more concise encoding (logarithmic in the length of the plan), but we expect less spectacular benefits than compared to SAT encodings, since our encoding is already quite small. It would also be interesting to investigate ungrounded versions of *parallel plans* [107], but currently the sequentiality of plans is deeply embedded in our encoding.

Finally, we hope to inspire new research on pre-processing and solving QBF problems, by submitting our problems to the QBFeval competition. Our encodings are very concise, but the current QBF solvers do not yet take advantage of the structure of the Ungrounded Encoding. While SAT solvers are rather mature, the field of QBF solvers is still improving rapidly. An advance in QBF solvers could make the Q-Planner approach competitive to the SAT approach for general classical planning problems.

Chapter 6

Optimal Layout Synthesis for Quantum Circuits as Classical Planning

Abstract

In Layout Synthesis, the logical qubits of a quantum circuit are mapped to the physical qubits of a given quantum hardware platform, taking into account the connectivity of physical qubits. This involves inserting SWAP gates before an operation is applied on distant qubits. Optimal Layout Synthesis is crucial for practical Quantum Computing on current error-prone hardware: Minimizing the number of SWAP gates directly mitigates the error rates when running quantum circuits.

In recent years, several approaches have been proposed for minimizing the required SWAP insertions. The proposed exact approaches can only scale to a small number of qubits. Proving that a number of swap insertions is optimal is much harder than producing near optimal mappings.

In this paper, we provide two encodings for Optimal Layout Synthesis as a classical planning problem. We use optimal classical planners to synthesize the optimal layout for a standard set of benchmarks. Our results show the scalability of our approach compared to previous leading approaches. We can optimally map circuits with 9 qubits onto a 14 qubit platform, which could not be handled before by exact methods.

6.1 Introduction

Quantum algorithms can speed up certain computational problems. For example, Shor's Algorithm [116] provides an exponential speed-up for factorization, and Grover's algorithm [51] provides a quadratic speed-up for database search. In recent years, quantum computing is envisaged as a way to solve hard problems [5], out of reach for classical computers. Formulating new algorithms to solve more problems

faster on quantum computers, and enabling compilation of those algorithms on actual quantum hardware is of great interest.

In this paper, we focus on the latter part, especially mapping a quantum circuit to an arbitrary quantum architecture. Following [124], we distinguish logical synthesis and layout synthesis for quantum circuits. *Logical synthesis* constructs an (efficient) quantum circuit of quantum gates on a number of logical qubits. Following [2, 32, 86], arbitrary quantum circuits can be represented by a few simple gates. We assume that the resulting quantum circuit consists of any standard unary gates and binary Controlled-NOT (CNOT) gates. A CNOT gate acts on two qubits. If the control qubit is true, then the target qubit is flipped. After logical synthesis, the gates and the order of their application to the logical qubits is fixed.

We focus on *Layout synthesis*, mapping the logical qubits onto the qubits of some physical hardware platform. High level algorithms assume that every pair of (logical) qubits is connected. However, in actual hardware platforms, often only a subset of (physical) qubit connections are realized. So one needs to compile the logical circuit to a physical hardware platform, such that every CNOT gate is applied only on neighboring qubits. It is not always possible to find a fixed mapping from logical qubits to physical qubits, such that every CNOT is applied on neighbors. In such cases, SWAP gates can be inserted, to swap the values of two connected qubits. A SWAP gate can be built from 3 CNOT gates. The goal of layout synthesis is to insert a minimal number of SWAP gates into the circuit, to move logical qubits to physical qubits, satisfying all neighbor constraints on subsequent CNOT gates.

State-of-the-Art and Related Work   Layout Synthesis has many names, for example, Transpilation, Compilation, Routing problem etc. An extensive overview and comparison of existing approaches is provided in [124], discussing both exact and heuristic approaches. Several heuristic approaches exist, for instance based on Dynamic Programming [119], A* search [133], and Temporal Planning [18, 37, 128, 129]. Concrete tools include Qiskit's SABRE [78] and Hardware-Aware (HA) [90]. While the state-of-the-art heuristic approaches are fast and scalable, it has been observed that the solutions are non-optimal [131]. In fact, [124] observed large optimality gaps, up to 1.5-12x and 5-45x on average on major quantum computing platforms.

For practical quantum computing on current NISQ machinery (noisy, intermediate scale quantum computers), the primary goal is to reduce the error rate. Since with each new gate the error rate increases, it is essential to minimize the number of additional SWAP gates in the Layout Synthesis. Therefore, in this paper we will focus on exact approaches.

Recently, several exact approaches have been proposed for optimal Layout Synthesis. An exact approach based on Dynamic Programming was proposed in [63], but it scales only to small circuits. Layout Synthesis was formulated as an SMT encoding in [131], where propositional constraints are given for satisfying connectivity and integer costs are assigned to additional swap gates. The authors report improved optimal plans compared to heuristic approaches. They use an exponential number of variables

to represent all permutations for the initial mapping. In a followup work [92], optimal subarchitectures are considered to reduce the number of permutations by mapping to a subset of physical qubits. In [123], the tool OLSQ is presented, based on an improved SMT encoding of permutations using a polynomial number of variables in the number of qubits. Those authors also allow optimizing error rates when individual error rates are given for qubits and gates. In [87], it is proposed to solve Layout Synthesis as a MAXSAT problem.

In theory, both SMT and MAXSAT approaches can prove lower bounds on the number of SWAP gates, thus guaranteeing optimality. However, optimal Layout Synthesis is an NP-complete problem [19]. In practice, most tools make some approximations. For instance, they limit the number of added swap gates, or they switch to heuristic search after some time limit. For example, the tool `satmap` by [87] only produces near optimal solutions, due to a restriction on the number of swaps. Even with these restrictions, all three approaches report improved optimality compared to heuristic approaches. However, the improved (near) optimality comes at the cost of time and memory. We aim at improving the performance, while preserving the optimality of the exact approaches.

Contribution   In this paper, we provide the first encoding of Optimal Layout Synthesis as a classical planning problem. The possible actions in this planning problem are: 1) to map a logical qubit to an initial physical qubit; 2) to apply a CNOT gate to the proper qubits; or 3) to insert an additional SWAP gate. In the latter case, we allow swapping logical qubits, as well as swapping a logical qubit with an extra ancillary qubit, since this can lead to even shorter plans. The conditions on the actions ensure that the solution satisfies all constraints. The shortest plan corresponds to a layout with the minimal number of SWAP gates. This enables the use of optimal planning tools to find the optimal layout.

In fact, we provide two encodings. The first one is based on *global levels* to resolve dependencies. Viewing these levels as discrete time stamps, this encoding is close to the temporal planning approach [18, 37, 128, 129], but we use the simpler fragment of classical planning, and moreover our encoding is exact, while their approach minimizes makespan, but not necessarily the number of SWAP gates. Our second, more efficient but still exact encoding, avoids the explicit representation of time steps and is based on the relative order induced by *local dependencies* between CNOT gates.

We implement both encodings in the open source translator Q-Synth (`https://github.com/irfansha/Q-Synth/`), written in Python. It takes two inputs: a quantum circuit with unary and binary CNOT gates in the standard OPENQASM 2.0 format [32], and a directed graph, modeling the connectivity graph of a hardware architecture platform. We use Qiskit libraries [97] for parsing, printing and extracting layers from a given circuit. Initially, we ignore unary gates and compute the dependencies between the CNOT gates. The output of the translator is a planning problem represented in PDDL (Planning Domain Definition Language) [85]. This

enables to use off-the-shelf classical planners for computing optimal plans for these PDDL instances. The optimal plan is translated back to an OPENQASM circuit by re-inserting the unary gates.

We experimented with the planner Fast Downward [57] (with multiple backends [38, 118]) and with Madagascar [101], which is based on an encoding into SAT (propositional satisfiability). Other planners, like those based on QBF (Quantified Boolean Logic) [111] could be used as well. We report the performance of our two approaches on some standard benchmarks and compare them to a number of existing exact (QMAP [22, 92] and OLSQ [123]) and heuristic approaches (QISKIT's SABRE [78]). As reported before, the heuristic approach SABRE is very fast, but often uses more SWAP gates than necessary. In all solved cases, we computed the correct minimal number of required SWAP gates.

We demonstrate that the performance of our approach is better than previous exact approaches. Moreover, our Local encoding performs much better than our Global encoding. The existing tool QMAP performed better than OLSQ in our experiments on a 5 qubit platform. On the other hand, OLSQ performed better than QMAP on a 14 qubit platform. Our Local approach is up to two orders of magnitude faster than OLSQ and solves 8 unique instances that OLSQ could not solve in 3 hours. Our approach finds optimal mappings of 9-qubit circuits onto a 14-qubit platform, where previous exact tools, OLSQ and QMAP, could only map 5-qubit and 4 qubits circuits respectively on the same platform.

6.2 Preliminaries

6.2.1 Layout Synthesis for Quantum Circuits

Layout Synthesis is mapping a logical circuit to some physical quantum computer architecture. Higher level algorithms make some assumptions, for example one-to-one connectivity is assumed among all qubits. However, physical architectures can have many restrictions, for example limited connectivity between qubits where binary gates can be applied. Particular qubits could also have different error rates and time durations. One takes into account such considerations when mapping to some architecture. In case no mapping exists such that the logical circuit can be mapped, SWAP gates can be added between neighbors to move the logical qubits to connected qubits. Note that using ancillary qubits can reduce the number of required SWAP gates. In this work, we focus on minimizing the number of SWAP gates and consider the use of planning for error rate optimization as future work.

We illustrate layout synthesis on an adder circuit, a standard example from [123]. The traditional circuit representation of the circuit is shown in Fig. 6.1. It contains 10 binary CNOT gates and 13 unary gates. Fig. 6.2 shows the adder circuit in OPENQASM format (with gate numbers as comments).

Figure 6.2 also provides a DAG representation with dependencies between gates, as computed by Qiskit [97]. The green nodes (inputs) are logical qubits, whereas the red nodes represent the (binary) CNOT gates, and the other nodes are unary gates. We

use the gate numbers to represent the dependencies between gates in the DAG. There are 11 layers in total, if we do not count the top qubit layer (the inputs are not part of the circuit). All gates in the same layer act on independent qubits, so they can be executed independently. We use this fact in §6.3.1 to handle dependencies between CNOT gates.

Figure 6.3 illustrates the IBM platform IBM-QX2 with 5 physical qubits. In general, the connections between qubits are directed, and our approach can handle directed coupling graphs. However, other exact tools can only handle symmetric connections, so for a fair comparison we only consider bidirectional platforms in this paper.

To map the adder circuit onto any platform, one needs 4 physical qubits connected as a square since it has CNOT gates on the qubit pairs $(q0, q1), (q1, q2), (q2, q3)$, $(q3, q0)$. However, IBM-QX2 has no 4 qubits that are connected in the form of a square, thus SWAP gates are needed. We observe from our results in Table 6.1 that 1 SWAP insertion is sufficient. The circuit in Figure 6.5 shows the mapped circuit with 1 additional SWAP gate. First, the logical qubits $q0, q1, q2, q3$ of the circuit are mapped to physical qubits $q0, q1, q2, q3$. For the mapped circuit, the order of gates g11 and g12 is reversed, and a SWAP gate on qubits q2 and q3 is added in between. Since g11 and g12 are in the same layer, they are independent, and this change preserves the correctness. All gates after g11 and g12 must be mapped to the swapped physical qubits: Essentially, from this point on, q2 and q3 are interchanged.

The Melbourne platform (Fig. 6.4) contains a subset of 4 qubits that form a square. Consequently, the adder circuit does not require any additional SWAP gates when mapped on Melbourne. We can indeed find a mapping without any SWAP gates as shown in Table 6.2.

6.2.2 Classical Planning

One of the main applications of Automated Planning [49] is scheduling. In §6.3, we will model Layout Synthesis as a planning problem. Classical Planning is finding a valid sequence of deterministic actions from a single initial state to some goal state, where the initial state and the effect of all actions are completely known. The *Planning Domain Definition Language* (PDDL) [85] is a standard domain-specific language for classical planning problems used in International Planning Competitions (IPC). As shown in Listing 6.1: The domain file specifies types, predicates and actions. For predicates one can optionally specify types, as we will later see in §6.3. PDDL is an
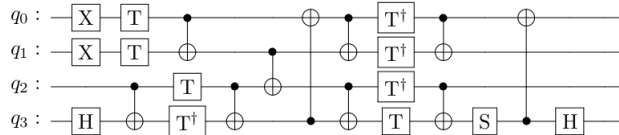
Figure 6.1: Adder circuit before mapping to IBM-QX2.

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[4];
x q[0];//g1
x q[1];//g2
h q[3];//g3
cx q[2], q[3]; //g4
t q[0];//g5
t q[1];//g6
t q[2];//g7
tdg q[3];  //g8
cx q[0], q[1]; //g9
cx q[2], q[3]; //g10
cx q[3], q[0]; //g11
cx q[1], q[2]; //g12
cx q[0], q[1]; //g13
cx q[2], q[3]; //g14
tdg q[0];  //g15
tdg q[1];  //g16
tdg q[2];  //g17
t q[3];//g18
cx q[0], q[1]; //g19
cx q[2], q[3]; //g20
s q[3];//g21
cx q[3], q[0]; //g22
h q[3];//g23
```
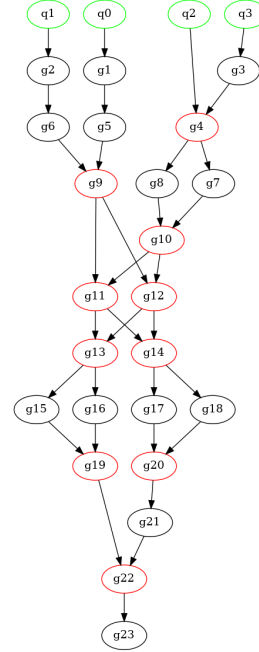

Figure 6.2: Adder circuit in OPENQASM format and in DAG format, showing the gate dependencies.
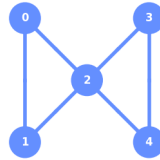
Figure 6.3: Coupling map for IBM-QX2 (Tenerife)

action-centered language; the actions essentially specify how the world changes by preconditions and effects. Action conditions and effects are represented similar to First-Order-Logic, using parameters to achieve compact descriptions. An example for an action is map_initial with parameters ?l and ?p in §6.3.1. The problem file specifies objects, initial state and goal conditionas shown in Listing 6.2 Initial and Goal states are represented using a set of (negated) propositions.

Given a planning problem in PDDL format, one can use off-the-shelf planners such as Fast Downward, Madagascar, etc. A planning problem description can also assign costs to actions. In Planning competitions, planners are categorized based on
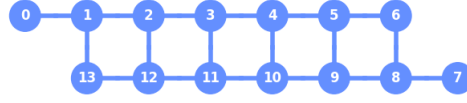
Figure 6.4: Coupling map for IBM Melbourne
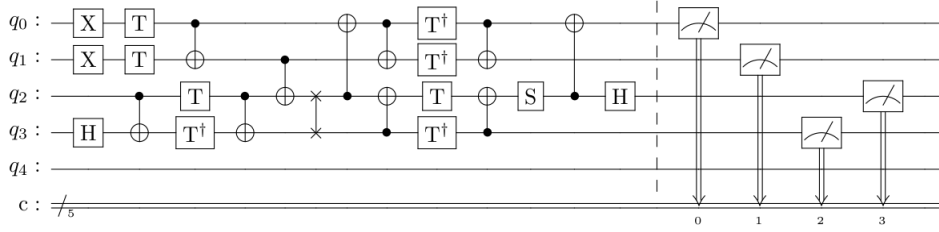
Figure 6.5: Adder circuit after mapping to IBM-QX2, with 1 optimal swap insertion. Measurement gates indicate the final location of the logical qubits.

how they handle these costs. Optimal Planners return only plans with optimal cost; Satisfying Planners, on the other hand, return some plan quickly, regardless of the cost. Nevertheless, given enough time, some satisfying planners such as Fast Downward Stone Soup (FDSS) [57] can optimize the cost to a large extent. Note that if cost is not specified, each action is considered to have cost 1.

Listing 6.1: Domain file structure

```
1   (define (domain Quantum)
2    (:requirements :typing)
3    (:types ...)
4    (:predicates ...)
5    (:action ...) ...)
```

Listing 6.2: Problem file structure

```
1   (define (problem example)
2    (:domain Quantum)
3    (:objects ...)
4    (:init ...)
5    (:goal (and ...)))
```

6.3 Layout Synthesis as Classical Planning

In this section, we describe two encodings of layout synthesis as a classical planning problem, represented in PDDL.[1] Both encodings include the following actions:

- apply_cnot, to apply the CNOT gate of two logical qubits on two physical qubits.

---

[1]Our Quantum Circuit Layout domain received the Outstanding Domain Submission Award at the International Planning Competition IPC 2023.

- `swap`, to insert a SWAP gate. We allow swapping two mapped logical qubits, as well as swapping a mapped logical qubit with an ancillary qubit.

Both encodings include the following predicates, to model the state after a partial plan:

- `mapped(?l,?p)`, to keep track on which physical qubit a logical qubit is currently mapped.

- `rcnot(...)`, keeping track which CNOT gates are still required (all CNOT gates are still required in the initial state, none should be required in the goal state).

The planning problem defines the following constraints on valid plans:

- Initial Mapping: The initial mapping from logical to physical qubits must be injective.

- Valid Gate Dependencies: Gates must be applied to the correct qubits in proper order.

- Valid CNOT and SWAP gates: All CNOT and SWAP gates are only applied on neighboring physical qubits.

As a consequence, since all CNOT gates must be applied anyway, the shortest valid plan will correspond to the correct layout, with the minimal number of SWAP gates inserted.

Since the hardware platform only constrains 2-qubit gate connectivity, layout synthesis can focus on the CNOT gates and their dependencies only. The unary gates can be reinserted later. Ignoring the unary gates, every CNOT gate has at most two direct dependencies. We illustrate this again on the DAG of the adder-circuit of Fig. 6.2. There, $g4$ depends on the input gates $g2$ and $g3$; gate $g10$ only depends on $g4$; and gate $g11$ depends on the CNOT gates $g9$ and $g10$. In general, a CNOT gate could also depend on one input qubit and one CNOT gate, but this does not happen in the example of Fig. 6.2.

We parse the circuit and extract the CNOT gates with their dependencies. After finding a solution, we reconstruct the circuit including single qubit gates. We first encode n logical qubits as objects $\{l_0, \ldots, l_{n-1}\}$ of type `lqubit`, and m physical qubits using objects as $\{p_0, \ldots, p_{m-1}\}$ of type `pqubit`. In our experiments, we do not specify costs, so each additional SWAP gate increases the cost by 1. Alternatively, one could specify an explicit cost for SWAP gates; for example a cost of 3, representing 3 underlying CNOT gates.

The first encoding (Section 6.3.1) is based on *global levels*, corresponding to the layers with CNOT gates in the dependency graph. This encoding uses two extra actions: `map_initial`, mapping a logical qubit to a physical qubit in the first time slice, and action `move_depth`, to advance to the next time slice. The second, more efficient encoding (Section 6.3.2), avoids the explicit representation of time steps and

is based on the relative order of *local dependencies* between CNOT gates (and logical input qubits).

6.3.1 Global Level Based Model

Domain Specification

In this section, we specify the predicates and actions required for the domain specification. Note that a domain file specifies the predicates and the actions of the problem domain. The predicates describe the state of the world, whereas the actions describe the change from one state to another.

Valid Initial Mapping   For modelling the first constraint i.e., mapping logical qubits to physical qubits, we define three predicates and one action. The predicate `mapped` specifies that the logical qubit `?l` is mapped to the physical qubit `?p`. The two auxiliary predicates `mapped_pq/mapped_lq` describe if a physical/logical qubit is already mapped, since one needs an injective mapping. The parameter `?p` (`?l`) in these predicates is of type `pqubit` (`lqubit`). To obtain propositional instances, one substitutes all objects with corresponding types in the predicates. This process is called grounding. It translates the PDDL description to a (potentially very large) propositional problem.

We use the `map_initial` action to map a logical qubit `?l` to a physical qubit `?p`. One can apply the action only if `?l` and `?p` are both not mapped. Once the action is applied, `?l` is mapped to `?p`.

<div align="center">Listing 6.3: predicates and action for initial mapping</div>

```
1  (:types lqubit pqubit)
2  (:predicates (mapped ?l - lqubit ?p - pqubit)
3               (mapped_lq ?l - lqubit)
4               (mapped_pq ?p - pqubit))
5  (:action map_initial
6   :parameters (?l - lqubit ?p - pqubit)
7   :precondition (and
8     (not(mapped_lq ?l)) (not(mapped_pq ?p)))
9   :effect (and (mapped ?l ?p)
10     (mapped_lq ?l) (mapped_pq ?p)))
```

Valid Gate Dependencies   To enforce gate dependencies in this model, we use the layers as generated by Qiskit.[2] As long as gates are executed layer by layer, their dependencies will be respected. When modelling layout synthesis as planning, we only consider the layers with CNOT gates. To represent layers in our first planning model, we introduce objects of type `depth`, representing discrete depths, i.e., one object for each layer that contains a CNOT gate. In the DAG of the adder example in Figure 6.2, only the layers at depths $d2, d3, d4, d5, d6, d8, d10$ contain CNOT gates.

---

[2]This may introduce spurious dependencies, when independent gates end up in different layers. This sub-optimality is solved in our Local model.

To ensure the circuit is executed layer by layer, we maintain a predicate `current_depth` to specify the current layer, and use a static predicate `next_depth` to relate the current and the next layer. We use the action `move_depth` to move from one layer to another layer in a plan. Every valid plan has exactly *k* `move_depth` actions, where *k* is the number of layers in the DAG with at least one CNOT gate.

Using layers provides two advantages over placing each instruction in its own layer, following the OPENQASM input order: (1) There are fewer dependencies, potentially reducing the number of swaps; (2) There are fewer `move_depth` actions in the plan, which typically decreases the solving time.

Listing 6.4: predicates and action for valid gate dependencies

```
1  (:types depth)
2  (:predicates (current_depth ?d - depth)
3               (next_depth ?d1 ?d2 - depth))
4  (:action move_depth
5   :parameters (?d1 ?d2 - depth)
6   :precondition (and (current_depth ?d1)
7     (next_depth ?d1 ?d2))
8   :effect (and (not(current_depth ?d1))
9     (current_depth ?d2)))
```

Valid CNOT and SWAP gates   For the final constraint, we need to specify when CNOT and SWAP gates can be applied. To model that a CNOT gate must be applied on specific qubits at a certain depth, we use predicate `rcnot` (required-cnot) in Listing 6.5. Predicate (`rcnot ?l1 ?l2 ?d`) indicates that a CNOT gate must still be applied between two logical qubits ?l1, ?l2 at the layer ?d. Here ?l1, ?l2, ?d are parameters that can be substituted with objects of the corresponding types to generate (grounded) propositions.

To apply a CNOT gate, the corresponding logical qubits must be mapped to some physical qubits; these physical qubits must be connected; and the current depth must match a required CNOT gate. The action `apply_cnot` ensures that all the above conditions are satisfied. The effect of this action removes one required CNOT operation (`rcnot`).

Finally, we model the action `swap` to allow swapping of logical qubits when required. We allow two types of swaps:

- Swapping two mapped logical qubits.

- Swapping a mapped logical qubit with an ancillary qubit.

Note that we call both actions `swap`, although their parameters and conditions are different. In the first case, similar to a CNOT gate, both logical qubits must be mapped to connected physical qubits. The effect will update the mapping from logical to physical qubits. In the second case, a logical qubit ?l1 must be mapped to a physical qubit ?p1, which should be connected to a free (not mapped) physical qubit ?p2. Once swapped, ?l1 is now mapped to ?p2, and ?p1 becomes free again (not mapped).

If the coupling graph is *not* bidirectional, we need a symmetric version of the ancillary swap as well (see Appendix B. for full details).

Listing 6.5: predicates and actions for valid CNOT and SWAP gates

```
1  (:predicates
2      (rcnot ?l1 ?l2 - lqubit ?d - depth)
3      (connected ?p1 ?p2 - pqubit))
4
5  (:action apply_cnot
6   :parameters (?l1 ?l2 - lqubit
7      ?p1 ?p2 - pqubit ?d - depth)
8   :precondition (and (connected ?p1 ?p2)
9      (mapped ?l1 ?p1) (mapped ?l2 ?p2)
10     (rcnot ?l1 ?l2 ?d) (current_depth ?d))
11   :effect (and (not(rcnot ?l1 ?l2 ?d))))
12
13  (:action swap ;; with another mapped qubit
14   :parameters (?l1 ?l2 - lqubit
15               ?p1 ?p2 - pqubit)
16   :precondition (and (connected ?p1 ?p2)
17      (mapped ?l1 ?p1) (mapped ?l2 ?p2) )
18   :effect (and
19      (not(mapped ?l1 ?p1)) (mapped ?l1 ?p2)
20      (not(mapped ?l2 ?p2)) (mapped ?l2 ?p1)))
21
22  (:action swap ;; with an ancillary qubit
23   :parameters (?l1 - lqubit ?p1 ?p2 - pqubit)
24   :precondition (and (connected ?p1 ?p2)
25      (mapped ?l1 ?p1) (not(mapped_pq ?p2)) )
26   :effect (and
27      (not(mapped ?l1 ?p1)) (mapped ?l1 ?p2)
28      (not(mapped_pq ?p1)) (mapped_pq ?p2)))
```

Problem Specification

In the problem file, we need to specify the initial and goal states of the planning problem. For the adder example, the initial depth is the first layer with a CNOT, `d2`. We specify the coupling map of the architecture using connected predicates. For CNOT mapping, we specify all the required CNOT gates at specific depths. Finally, the `next_depth` predicate specifies a total ordering of the layers. Note that the initial state specification is complete, i.e., all propositions not specified are negated. In case of the goal specification, one only needs to specify the propositions that must be achieved. Propositions that are not specified are open, resulting in more than one goal state. In our model, we specify that all logical qubits are mapped and all required CNOT gates are performed (i.e., each `rcnot` is negated). Listings 6.6 and 6.7 present snippets of initial and goal specification of our adder example.

Listing 6.6: Adder initial state snippets

```
1 (:init
2   (current_depth d2)
3   (connected p1 p0) (connected p1 p2) ...
4   (next_depth d2 d3) (next_depth d3 d4) ...
5   (rcnot 12 13 d2) (rcnot 10 11 d3) ... )
```

Listing 6.7: Adder goal state snippets

```
1 (:goal
2  (and
3    (mapped_lq 10) ... (mapped_lq 13)
4    (not (rcnot 12 13 d2)) ... ))
```

6.3.2   Local Dependency Based Model

In the Global model, using explicit actions for moving to the next depth results in almost doubling the plan length in the worst case. At least, the plan length increases by the number of layers in the circuit with CNOT gates. It is well known that increased plan length can result in increased solving time. Additionally, we need initial mapping actions for mapping logical qubits to physical qubits. This further increases the plan length. These initial mapping steps also create a large search space, which is not a priori pruned by the needs of the logical circuit. Moreover, since the layers introduce spurious dependencies, the resulting number of SWAPs could be suboptimal.

To overcome these problems, we make three improvements to our Global encoding. First, we drop the layers and handle local dependencies directly when applying CNOT gates. Second, we integrate the initial mapping into the corresponding CNOT gates. Third, as a final optimization, we partially ground the gates in the `apply_cnot` actions to reduce the number of parameters. We now describe these improvements in detail.

Avoiding Layers (Local/Initial)

In all our actions and predicates, we drop the depth parameters, and we drop the action `next_depth`. Instead, we will specify gate dependencies locally. Recall that during planning we only consider the binary CNOT gates, and we completely ignore the unary gates. Then a CNOT gate C_1 depends on at most two preceding CNOT gates C_2 and C_3. A gate could also directly depend on a logical input qubit. As long as C_2 and C_3 are executed before C_1, we preserve the correctness.

To model local dependencies (cf. Listing 6.8), we introduce a static predicate (`cnot ?l1 ?l2 ?g1 ?g2 ?g3`), which indicates that the logical circuit contains a gate `?g1` on logical qubits `?l1` and `?l2`, that depends on previous gates `?g2` and `?g3`. To avoid exceptions for gates that directly depend on the input qubits, we view `lqubit` as a subtype of `gate`. We introduce predicate `done` to keep track of the CNOT gates that have already been applied. Using `done`, we don't require `mapped_lq` anymore, and we rename `mapped_pq` to `occupied`. The predicates `mapped` and `connected` are similar as before.

Listing 6.8: Local/Initial Domain: types and predicates

```
1  (:types
2     pqubit gate − object
3     lqubit − gate)
4  (:predicates
5     (cnot ?l1 ?l2 − lqubit ?g0 ?g1 ?g2 − gate)
6     (done ?g − gate)
7     (mapped ?l − lqubit ?p − pqubit)
8     (occupied ?p − pqubit)
9     (connected ?p1 ?p2 − pqubit))
```

The actions `map_initial` and `apply_cnot` are adapted as in Listing 6.9. Note that an `lqubit` becomes done when it is mapped, while a CNOT gate becomes done when it is applied. Also note that in the latter case, we require that the dependent CNOT gates (or logical input qubits) are already done. The `swap`-actions are similar to the global encoding in Listing 6.5.

Listing 6.9: Local/Initial Domain: actions `map_initial` and `apply_cnot`

```
1  (:action map_initial
2   :parameters (?l − lqubit ?p − pqubit)
3   :precondition
4      (and (not(done ?l)) (not(occupied ?p)))
5   :effect (and (done ?l)
6      (mapped ?l ?p) (occupied ?p)))
7
8  (:action apply_cnot
9   :parameters (?l1 ?l2 − lqubit
10                 ?p1 ?p2 − pqubit
11                 ?g0 ?g1 ?g2 − gate)
12   :precondition (and
13      (cnot ?l1 ?l2 ?g0 ?g1 ?g2)
14      (connected ?p1 ?p2)
15      (mapped ?l1 ?p1) (mapped ?l2 ?p2)
16      (done ?g1) (done ?g2) (not(done ?g0)))
17   :effect (and (done ?g0)))
```

Integrating the Initial Mapping (Lifted/Compact)

Recall that not all CNOT gates have two CNOT dependencies; for example gate *g*4 in Figure 6.2 depends directly on two logical (input) qubits `l2` and `l3`. To avoid explicit initial mapping actions, we integrate the mapping of these input qubits directly into the CNOT actions.

In this lifted compact model, we drop the `map_initial` action entirely. Instead, we split the `apply_cnot` action in four cases, depending on whether this gate depends on previous gates or on logical input qubits. In the latter case, we immediately map the logical qubits to any unmapped, connected physical qubits. In Listing 6.10 we show two cases, where gate `?g0` depends on two gates, or on two inputs, respectively. Note that in the latter case we can drop the test for (`done ?l1`) and (`done ?l2`),

since in this model a logical qubit can only be the input to a single CNOT gate
(ignoring intermediate unary gates). The full model also includes mixed cases for
`apply_cnot_gate_input` and `apply_cnot_input_gate`.

Listing 6.10: Lifted/Compact Domain: `apply_cnot` actions

```
 1  (:action apply_cnot_gate_gate
 2    :parameters (?l1 ?l2 - lqubit
 3                 ?p1 ?p2 - pqubit
 4                 ?g0 ?g1 ?g2 - gate)
 5    :precondition (and
 6        (cnot ?l1 ?l2 ?g0 ?g1 ?g2)
 7        (connected ?p1 ?p2)
 8        (mapped ?l1 ?p1) (mapped ?l2 ?p2)
 9        (done ?g1) (done ?g2) (not (done ?g0)))
10    :effect (and (done ?g0)))
11
12  (:action apply_cnot_input_input
13    :parameters (?l1 ?l2 - lqubit
14                 ?p1 ?p2 - pqubit ?g0 - gate)
15    :precondition (and
16        (cnot ?l1 ?l2 ?g0 ?l1 ?l2)
17        (connected ?p1 ?p2)
18        (not (occupied ?p1)) (not (occupied ?p2))
19        (not (done ?g0))
20    )
21    :effect (and (done ?g0)
22        (mapped ?l1 ?p1) (occupied ?p1)
23        (mapped ?l2 ?p2) (occupied ?p2)))
```

Grounding the Gate Names (Local/Compact)

In both local models, the `cnot`-predicate and the `apply_cnot` action depend on
many parameters. This could slow down the planning tools, since they have to
instantiate these actions for all possible object combinations. Note that for each gate,
the dependencies and the logical qubits they are applied to are known at compile time.
Therefore, as a final optimization step, in our Local/Compact model we specialize the
`apply_cnot` actions in the domain specification for each gate.

   The domain file now depends on the logical circuit instance, since it refers to
its gates and qubits. This requires the introduction of constants with the new type
`gateid` (one for each CNOT gate), as well as constants for the logical qubits. The
mapped physical qubits are not known at compile time, but will be determined by the
planner tool.

   For example, the CNOT gate $g11$ in the adder circuit in Fig. 6.2 is represented by
the constant `g11`, and it is handled by the action `apply_cnot_g11`. It is applied to
the logical qubits `l3` and `l0`. This gate depends on two CNOT gates `g9` and `g10`, so
the precondition also requires that the dependent CNOT gates `g9` and `g10` are already
done. The complete action specification for this gate becomes:

Listing 6.11: A partially grounded CNOT action for the adder circuit

```
1  (:types gateid lqubit pqubit)
2  (:constants g4 g9 g10 g11 ... g22 - gateid
3              10 11 12 13 - lqubit)
4  (:action apply_cnot_g11
5   :parameters (?p1 ?p2 - pqubit)
6   :precondition (and (not (done g11))
7     (done g9) (done g10)
8     (mapped 13 ?p1) (mapped 10 ?p2)
9     (connected ?p1 ?p2))
10  :effect (and (done g11)))
```

The CNOT gate *g*4 depends on two inputs, so the corresponding action `apply_cnot_g4` will integrate two initial mappings as follows:

Listing 6.12: CNOT gate with integrated initial mapping

```
1  (:action apply_cnot_g4
2   :parameters (?p1 ?p2 - pqubit)
3   :precondition (and (not (done g4))
4     (not(occupied ?p1)) (not(occupied ?p2))
5     (connected ?p1 ?p2))
6   :effect (and (done g4)
7     (mapped 12 ?p1) (occupied ?p1)
8     (mapped 13 ?p2) (occupied ?p2)))
```

A similar integration is also applied to CNOT gates which depend only on one other CNOT gate. We provide the full specification for the Adder example in Appendix A.

The Local Compact encoding avoids spurious dependencies due to using fixed levels. It also results in shorter plans, since we avoid the `next_depth` and `map_initial` actions. Searching for shorter plans should be more efficient. We also expect that the efficiency is further improved since the actions have fewer parameters (due to grounding) and logical qubits will only be mapped to connected physical qubits, as required by the initial CNOT gates. The following section compares the performance of the "Global", the (lifted) "Local Initial" and the (grounded) "Local Compact" encodings experimentally.

6.4 Experimental Evaluation

6.4.1 Experiment Design

We have implemented the translation of quantum circuits into PDDL instances in our tool Q-Synth (Quantum Synthesizer).[3] For an experimental evaluation, we consider a standard set of 16 benchmark circuits from [123], and we map them onto IBM platforms Tenerife (5 qubits) and Melbourne (14 qubits). The first experiment maps onto platform Tenerife (also called IBM-X2). We consider those 6 out of 16 circuits that use at most the 5 physical qubits available in Tenerife. For this experiment, we

---

[3]Q-Synth is available open source at `https://github.com/irfansha/Q-Synth/`.

give a 300 seconds time limit and a 16-GB memory limit. The second experiment maps onto platform Melbourne. We try all 13 out of 16 circuits that use at most 14 qubits. We give a 3-hour time limit and a 48-GB memory limit for all 13 instances. In both cases, we measure and report the CPU time taken by the whole tool chain (including parsing, encoding, searching for an optimal plan, and extracting and validating the mapped circuits).

Q-Synth requires a planning tool to solve the generated PDDL instances. We consider the Fast Downward planner (FD) with two planner configurations, the Big Joint Optimal Landmarks Planner (BJOLP) [38] and Merge-and-Shrink (MS) [118], and the SAT based planner Madagascar (M) [101]. For both experiments, we apply FD+BJOLP on the Global models (column G-bj) and Lifted Initial models (column LI-bj). On the Local Compact models, we report results from all three planners FD with BJOLP (L-bj), FD with MS (L-ms), and Madagascar (L-M). Note that Madagascar is a satisfying planner. We increment the plan length by 1 until we find a plan. This ensures optimality of the plans generated by Madagascar.

We compare our results with three existing tools. First with Qiskit's heuristic approach SABRE [78]. We use the first 100 seeds for SABRE Layout and take the minimum number of swaps produced by any seed. In every instance, this approach with 100 runs takes approximately 2 minutes.

Then we compare with two leading exact approaches QMAP [22, 92] and OLSQ [123]. Although there are heuristic versions of both tools, we only consider the exact versions of them. We use the latest version OLSQ `0.0.4.1` installed using pip. Note that OLSQ only uses bidirectional coupling graphs, so for sake of uniformity we change the coupling maps of Tenerife and Melbourne for all experiments. For QMAP, we use the latest version `mqt.qmap 2.1.3`. By default, it uses subset optimization (QMAP-S) as in [22]. However, QMAP-S was shown to be suboptimal for certain architectures [92]. Essentially, QMAP-S maps a given circuit with k logical qubits to a subset of k physical qubits. Instead, QMAP-SA uses optimal subarchitectures that also allow ancillary qubits. We compare Q-Synth with the exact approaches QMAP (without subarchitectures) and QMAP-SA (with optimal subarchitectures) as described in [92]. We also experiment with QMAP-S on the Melbourne platform, but for a fair comparison, in this case we also disable ancillary swaps in Q-Synth, run with planner FD+BJOLP (L-bj-na).

6.4.2 Validation

Validating if our mapped circuits are optimal and correct is non-trivial. We use the following four measures to ensure correctness and optimality.

- When a circuit is mapped, we try to recover the original circuit by reversing the swaps and using the reverse initial mapping. The resultant circuit must be exactly the same as the original circuit, which we compare using Qiskit.

- We check if all the 2-qubit gates are applied on the connected qubits. One could alternatively use the checkmap function from QISKIT.

- We compare the equivalence of the mapped circuit with the original circuit with at least 1 million simulations in Qiskit.

- Finally, for optimality we cross compare with the results obtained from other exact tools.

Note that with SAT based planners, one could extract certificates when refuting an instance. This provides provable lower bounds on the number of swaps.

6.4.3 Results

In Table 6.1 (platform Tenerife), Q-Synth with both models and all planner configurations, and both existing tools QMAP and OLSQ, solve all the instances. Overall, for the benchmarks on this small platform, Q-Synth performs similar to QMAP, whereas OLSQ is considerably slower, especially on the larger instances. All exact approaches compute the minimum number of swaps required for `mod5mils_65` as 2. On the other hand, SABRE returns 3 swap insertions, which is non-optimal.

Table 6.2 presents the results on platform Melbourne with 14 qubits. Our Q-Synth with Local Initial models performs much better than with Global models. Further, Q-Synth with Local Compact models performs much better than with Local Initial models, especially on 9 qubit instances. Q-Synth solves most instances with planner FD with MS (L-ms) or BJOLP (L-bj): 11 out of 13 within the time and memory limits; with Madagascar it solves 8 out of 13 instances (L-M); surprisingly L-M times out on one 5-qubit instance. Even our Global encoding with FD+BJOLP (G-bj) solves 6 out of 13 instances, which is more than previous exact methods. Other planning tools on our Global models did not perform that well (timings are not reported here). Of the previous exact approaches, OLSQ solves only 3 out of 13 instances in total, while QMAP-SA solves 2 out of 13. Q-Synth is 2 orders of magnitude faster than OLSQ on the `or` instance. On the other hand, QMAP (without subarchitectures) runs out of memory for all instances. In case of solving without ancillary qubits, QMAP-S solves 7 instances, including one 7-qubit instance. Q-Synth with Local models without ancillary swaps (L-bj-na) still solves the same 11 instances optimally (although this combination is suboptimal in general). It runs up to ten times faster than L-bj (with ancillary swaps). So considering ancillary swaps can increase the solving time considerably.

We don't know the minimal number of SWAP gates required for the largest benchmarks, since none of the exact tools could solve them. On the other hand, with SABRE we could compute an upper bound on this number. One might also find such plans by using QMAP and OLSQ in satisfying mode, or running the planning tools on our encoding in satisfying mode, but this has not been the focus of the research reported here. Note that for 5 instances, SABRE gives non-optimal swap insertions (this phenomenon was also observed in [124]). For example, for `4gt13_92`, SABRE gives 3 additional swaps, which can be implemented by 9 additional CNOT gates. Within 2 minutes, our approach L-ms computed the minimum number of swaps needed, which then also provides a guaranteed lower bound. This demonstrates the

feasibility of classical planning for exact optimal layout synthesis of quantum circuits of moderate size.

Table 6.1: Platform Tenerife or IBM-QX2 (5 qubits), q: Number of logical qubits, c: Number of CNOT gates, +s: Number of swaps added, *: non-optimal count. We specify the time taken by all exact tools in seconds.

Circuit	Q	C	+S	Our tool Q-Synth (exact)					Previous Exact Tools			SABRE
				G-bj	LI-bj	L-ms	L-bj	L-M	QMAP	QMAP-SA	OLSQ	+S
or	3	6	0	4.9	4.3	4.1	4.0	3.9	4.0	3.9	6.7	0
adder	4	10	1	4.4	4.2	4.4	4.0	4.7	3.9*	4.1*	40.3	1
qaoa5	5	8	0	3.9	4.1	4.6	4.2	4.0	4.0	3.9	12.6	0
4mod5-v1_22	5	11	1	4.7	4.1	4.7	4.1	4.1	4.0	3.9	24.2	1
mod5mils_65	5	16	2	4.2	4.0	5.3	4.3	5.4	4.0	4.0	107	3
4gt13_92	5	30	0	4.3	4.2	6.4	4.1	5.0	4.1	4.4	136	0

Table 6.2: Platform Melbourne (14 qubits), q: Number of logical qubits, c: Number of CNOT gates, +s: Number of swaps added, TO/MO: Time/Memory Out, *: suboptimal count. We specify the time taken by all exact tools in seconds.

Circuit	Q	C	+S	Our tool Q-Synth (exact)					Previous Exact Tools			Without ancillary qubits		SABRE
				G-bj	LI-bj	L-ms	L-bj	L-M	QMAP	QMAP-SA	OLSQ	L-bj-na	QMAP-S	+S
or	3	6	2	4.9	**4.1**	7.6	**4.1**	4.3	MO	5.5	517	4.2	2.8	2
adder	4	10	0	5.5	4.5	15.1	**4.2**	4.6	MO	6.5	30.5	4.3	2.4	0
qaoa5	5	8	0	5.0	4.5	23.7	4.4	**4.1**	MO	TO	39.5	4.2	2.6	0
4mod5-v1_22	5	11	**3**	115	29.2	25.6	**5.1**	7.4	MO	TO	TO	4.7	10.1	4
mod5mils_65	5	16	6	1825	57.8	33.7	**7.0**	16.3	MO	TO	TO	4.6	18.6	6
4gt13_92	5	30	**10**	TO	280	**85.8**	121	TO	MO	TO	TO	11.5	183*	13
tof_4	7	22	1	5831	2716	125	**4.7**	12.3	MO	TO	TO	5.1	6734	1
barenco_tof_4	7	34	**5**	TO	10643	184	29.9	**27.7**	MO	TO	TO	8.4	TO	6
tof_5	9	30	1	TO	TO	386	**5.0**	449	MO	MO	TO	4.7	TO	1
mod_mult_55	9	40	**7**	TO	TO	**2316**	7710	TO	MO	MO	TO	761	TO	8
barenco_tof_5	9	50	**6**	TO	TO	634	**187**	TO	MO	MO	TO	23.2	TO	7
vbe_adder_3	10	50	-	TO	TO	TO	TO	TO	MO	MO	TO	TO	TO	8
rc_adder_6	14	71	-	TO	TO	TO	TO	TO	MO	MO	TO	TO	MO	12
Total number of instances solved:				6	8	11	11	8	0	2	3	11	7	13

6.5   Comparison with Related Work

Comparison to QMAP and OLSQ

Our tool Q-Synth with the Local model outperforms QMAP significantly: Q-Synth solves 9 instances uniquely. The difference is clear when the number of qubits is more than 4: all the instances timed out for QMAP after 3 hours. Note that Q-Synth solves all three 9-qubit instances (two even within 4 minutes). QMAP uses an exponential number of variables in the number of qubits to represent the permutations. We conjecture that is the reason for poor performance on the 14 qubit platform. Even with sub-architecture optimization, scaling to 14 qubit platform seems difficult, and we expect the same challenge for even larger platforms. A similar bottleneck for QMAP is reported in [87] with platform Tokyo (20 qubits). The number of SWAPs in QMAP depends on the order of the input circuit, which can result in suboptimal solutions.

For example, QMAP and QMAP-SA give 2 additional swaps for the `adder` instance, instead of 1 optimal swap (* in Table 6.1).

Even our Global encoding outperforms QMAP and solves three 5-qubit instances and one 7-qubit instance. Avoiding the layers in Local Initial already improves the performance up to an order of magnitude. Integrating the initial maps with CNOT actions improves the performance even further. This shows the strength of planning based approaches, and the effect of modelling the problem (Global vs Local) on performance.

OLSQ (run in exact mode for optimal solutions) scales poorly with the number of qubits and CNOT gates compared to Q-Synth. Between OLSQ and QMAP-SA, QMAP-SA performs better on a 5-qubit platform whereas OLSQ performs better on a 14-qubit platform. These results are consistent with the exponential dependency of QMAP on the target platform. Note that other approaches could also benefit from the optimal subarchitectures computed by QMAP-SA. Further, when optimizing for the number of swaps, the solutions from OLSQ are not necessarily optimal. OLSQ optimizes swap count for each depth iteratively and stops at the first depth where a mapping is found. This can be suboptimal since larger depth mapped circuits are not explored. In our experiments, the swap counts generated by OLSQ were optimal.

Both tools and also `satmap` by [87] provide heuristic approaches that produce near optimal solutions and scale much better. In this paper, we focus only on exact approaches where the solutions are truly optimal in the count of swap insertions. Proving that there exists no mapping with less SWAP gates is much harder. For larger circuits, one could apply satisfying planners to our models, to find near-optimal solutions.

Without Ancillary Qubits QMAP-S performs significantly better than QMAP-SA, consistent with the theoretical considerations in [92]. For example, consider the instance `qaoa5`, QMAP-S tries to map onto a 5-qubit subset of the physical qubits, while QMAP-SA tries to map onto a 7-qubit subset, i.e., it allows 2 additional ancillary qubits. Further, QMAP-S imposes a restriction on the number of swaps in front of each gate. In case of `qaoa5`, the swap limit is 3, which significantly reduces the solving time. However, it is not clear to us if this technique preserves the optimality. Note that, for the same instance, the swap limit with QMAP-SA is 7. We observed that QMAP-S returns 11 additional swaps for the instance `4gt13_92` (* in Table 6.2), instead of 10 swaps as reported by Q-Synth. OLSQ in heuristic mode (OLSQ-TB) also gives 10 additional swaps for the same instance, confirming an upper bound of 10 swaps. Our approach without ancillary swaps still reports the correct minimal swap insertions in all instances. However, [92] provides an example where ancillary swaps are essential to obtain an optimal solution.

Comparison to Temporal Planning

The authors of [128] proposed to use temporal planners for Layout Synthesis on QAOA problems. In Section 4 [128], classical planning is mentioned as a potential

alternative for temporal planning. However, they use time steps for gates to avoid dependencies, similar to our Global level-based encoding, but without grouping the CNOT gates in layers. This can almost double the plan lengths, and it blurs the look-ahead information due to lack of dependencies. In their paper, poor preliminary results were reported with the SAT based planner Madagascar (M/Mp) with parallel plans, which is consistent with our observations. Another key issue with their use of temporal planning is the lack of optimality: one can only obtain plans optimal in makespan. This issue is similar to parallel plans in classical planning: the number of swaps added need not be optimal and can be worse in practice.

In [37], layout synthesis for the QAOA algorithm for graph coloring is split into two stages: Qubit Initialization (QI) and Routing. Classical planning is considered for QI, where it is only used for initializing logical qubits on physical qubits. The addition of SWAP gates and generating actual plans is still handled by temporal planners. Compared to random initial allocation, using the classical planner FastDownward, improved the makespan in many instances.

6.6   Conclusion

From our experiments, we conclude that classical planning provides a strong alternative to temporal planning or SMT solving for solving the optimal circuit-layout synthesis problem. Moreover, the model using Local dependencies is superior to the one using Global levels.

Appendix

A. PDDL specification for Adder Circuit – Local Compact

We provide the detailed domain file and problem file for the adder example in Fig 6.1. We map it to the Tenerife platform by using the Local Compact encoding. These files can be generated using our tool Q-Synth[4], using the following command, where -a1 switches on the use of ancillary bits, and -b1 makes the coupling map bidirectional.

```
q-synth.py -m local -p tenerife -a1 -b1 Benchmarks/adder.qasm
```

Listing 6.13: Adder circuit – Local Compact; domain file

```
1     (define (domain Quantum)
2     (:requirements :strips :typing
3                    :negative-preconditions)
4     (:types lqubit pqubit gateid - object)
5     (:constants g4 g9 g10 g11 g12 g13
6                 g14 g19 g20 g22  - gateid
7                 10 11 12 13 - lqubit)
8     (:predicates
9        (occupied ?p - pqubit)
```

_____

[4]https://github.com/irfansha/Q-Synth/releases/tag/Q-Synth-v1.0-ICCAD23

```
10        (mapped ?l - lqubit ?p - pqubit)
11        (connected ?p1 ?p2 - pqubit)
12        (done ?g - gateid))
13     (:action swap
14      :parameters (?l1 ?l2 - lqubit
15                   ?p1 ?p2 - pqubit)
16      :precondition (and (connected ?p1 ?p2)
17         (mapped ?l1 ?p1) (mapped ?l2 ?p2))
18      :effect (and
19         (not (mapped ?l1 ?p1)) (mapped ?l1 ?p2)
20         (not (mapped ?l2 ?p2)) (mapped ?l2 ?p1)))
21     (:action swap-ancillary1
22      :parameters (?l1 - lqubit
23                   ?p1 ?p2 - pqubit)
24      :precondition (and (connected ?p1 ?p2)
25         (mapped ?l1 ?p1) (not (occupied ?p2)))
26      :effect (and
27         (not (mapped ?l1 ?p1)) (mapped ?l1 ?p2)
28         (not (occupied ?p1)) (occupied ?p2)))
29     (:action swap-ancillary2
30      :parameters (?l2 - lqubit ?p1 ?p2 - pqubit)
31      :precondition (and (connected ?p1 ?p2)
32         (mapped ?l2 ?p2) (not (occupied ?p1)))
33      :effect (and
34         (not (mapped ?l2 ?p2)) (mapped ?l2 ?p1)
35         (not (occupied ?p2)) (occupied ?p1)))
36     (:action apply_cnot_g4
37      :parameters (?p1 ?p2 - pqubit)
38      :precondition (and
39         (not (done g4)) (connected ?p1 ?p2)
40         (not(occupied ?p1)) (not(occupied ?p2)))
41      :effect (and (done g4)
42         (mapped 12 ?p1) (occupied ?p1)
43         (mapped 13 ?p2) (occupied ?p2)))
44     (:action apply_cnot_g9
45      :parameters (?p1 ?p2 - pqubit)
46      :precondition (and
47         (not (done g9)) (connected ?p1 ?p2)
48         (not (occupied ?p1)) (not(occupied ?p2)))
49      :effect (and (done g9)
50         (mapped 10 ?p1) (occupied ?p1)
51         (mapped 11 ?p2) (occupied ?p2)))
52     (:action apply_cnot_g10
53      :parameters (?p1 ?p2 - pqubit)
54      :precondition (and
55         (not (done g10)) (connected ?p1 ?p2)
56         (done g4) (mapped 12 ?p1)
57         (done g4) (mapped 13 ?p2))
58      :effect (and (done g10)))
```

```
59      (:action apply_cnot_g11
60       :parameters (?p1 ?p2 - pqubit)
61       :precondition (and
62          (not (done g11)) (connected ?p1 ?p2)
63          (done g9) (mapped 11 ?p1)
64          (done g10) (mapped 12 ?p2))
65       :effect (and (done g11)))
66      (:action apply_cnot_g12
67       :parameters (?p1 ?p2 - pqubit)
68       :precondition (and
69          (not (done g12)) (connected ?p1 ?p2)
70          (done g10) (mapped 13 ?p1)
71          (done g9) (mapped 10 ?p2))
72       :effect (and (done g12)))
73      (:action apply_cnot_g13
74       :parameters (?p1 ?p2 - pqubit)
75       :precondition (and
76          (not (done g13)) (connected ?p1 ?p2)
77          (done g12) (mapped 10 ?p1)
78          (done g11) (mapped 11 ?p2))
79       :effect (and (done g13)))
80      (:action apply_cnot_g14
81       :parameters (?p1 ?p2 - pqubit)
82       :precondition (and
83          (not (done g14)) (connected ?p1 ?p2)
84          (done g11) (mapped 12 ?p1)
85          (done g12) (mapped 13 ?p2))
86       :effect (and (done g14)))
87      (:action apply_cnot_g19
88       :parameters (?p1 ?p2 - pqubit)
89       :precondition (and
90          (not (done g19)) (connected ?p1 ?p2)
91          (done g13) (mapped 10 ?p1)
92          (done g13) (mapped 11 ?p2))
93       :effect (and (done g19)))
94      (:action apply_cnot_g20
95       :parameters (?p1 ?p2 - pqubit)
96       :precondition (and
97          (not (done g20)) (connected ?p1 ?p2)
98          (done g14) (mapped 12 ?p1)
99          (done g14) (mapped 13 ?p2))
100      :effect (and (done g20)))
101     (:action apply_cnot_g22
102      :parameters (?p1 ?p2 - pqubit)
103      :precondition (and
104         (not (done g22)) (connected ?p1 ?p2)
105         (done g20) (mapped 13 ?p1)
106         (done g19) (mapped 10 ?p2))
107      :effect (and (done g22))))
```

Listing 6.14: Adder circuit – Local Compact; problem file

```
1  (define (problem circuit)
2  (:domain Quantum)
3  (:objects p0 p1 p2 p3 p4 - pqubit)
4  (:init
5    (connected p1 p0)
6    (connected p0 p1)
7    (connected p2 p0)
8    (connected p0 p2)
9    (connected p2 p1)
10   (connected p1 p2)
11   (connected p3 p2)
12   (connected p2 p3)
13   (connected p3 p4)
14   (connected p4 p3)
15   (connected p4 p2)
16   (connected p2 p4)
17  )
18  (:goal (and
19    (done g4)
20    (done g9)
21    (done g10)
22    (done g11)
23    (done g12)
24    (done g13)
25    (done g14)
26    (done g19)
27    (done g20)
28    (done g22))))
```

An optimal plan, generated by FastDownward with BJOLP looks as follows:

```
(apply_cnot_g9 p0 p1)
(apply_cnot_g4 p2 p3)
(apply_cnot_g10 p2 p3)
(apply_cnot_g11 p1 p2)
(swap l2 l3 p2 p3)
(apply_cnot_g12 p2 p0)
(apply_cnot_g13 p0 p1)
(apply_cnot_g19 p0 p1)
(apply_cnot_g14 p3 p2)
(apply_cnot_g20 p3 p2)
(apply_cnot_g22 p2 p0)
```

The optimal plan needs 1 swap (cf Fig. 6.5). The Global encoding can be obtained in a similar manner by specifying `-m global` and the lifted Local Initial encoding by using `-m lifted_initial`.

B. Complete SWAP Actions for Global and Local Initial

Listing 6.15: Complete SWAP actions for Global and Local Initial models

```
1  (:action swap
2   :parameters (?l1 ?l2 - lbit ?p1 ?p2 - pbit)
3   :precondition (and
4       (connected ?p1 ?p2)
5       (mapped ?l1 ?p1) (mapped ?l2 ?p2))
6   :effect (and
7       (mapped ?l1 ?p2) (not (mapped ?l1 ?p1))
8       (mapped ?l2 ?p1) (not (mapped ?l2 ?p2))))
9  (:action swap-ancillary1
10  :parameters (?l1 - lbit ?p1 ?p2 - pbit)
11  :precondition (and
12      (connected ?p1 ?p2)
13      (mapped ?l1 ?p1)
14      (not (occupied ?p2)))
15  :effect (and
16      (not(mapped ?l1 ?p1)) (not(occupied ?p1))
17      (mapped ?l1 ?p2) (occupied ?p2)))
18 (:action swap-ancillary2
19  :parameters (?l2 - lbit ?p1 ?p2 - pbit)
20  :precondition (and
21      (connected ?p1 ?p2)
22      (mapped ?l2 ?p2)
23      (not (occupied ?p1)))
24  :effect (and
25      (not(mapped ?l2 ?p2)) (not(occupied ?p2))
26      (mapped ?l2 ?p1) (occupied ?p1)))
```

Chapter 7

Implicit State and Goals in QBF Encodings for Positional Games

Abstract

We address two bottlenecks for concise QBF encodings of maker-breaker positional games, like Hex and Tic-Tac-Toe. Our baseline is a QBF encoding with explicit variables for board positions and an explicit representation of winning configurations. The first improvement represents the winning configurations implicitly, exploiting their structure. The second improvement is inspired by lifted planning and avoids variables for explicit board positions, introducing a universal quantifier representing a symbolic board state. The paper evaluates the size of several encodings, depending on board size and game depth. It also reports the performance of QBF solvers on these encodings. We evaluate the techniques on Hex instances and also apply them to Harary's Tic-Tac-Toe. In particular, we study scalability to 19×19 boards, played in human Hex tournaments.

7.1 Introduction

This paper presents new encodings of positional games in Quantified Boolean Logic (QBF). In these games, two players claim empty positions on a fixed board in alternating turns. Examples include Hex, Harary's Tic-Tac-Toe (HTTT), and Gomoku. In the maker-breaker variant, the first player wins if he manages to occupy some winning configuration, while the second player wins if she can avoid that.

Quantifier alternations in QBF can naturally express the existence of a winning strategy of bounded depth. This allows solving these PSPACE-complete games using the sophisticated search techniques of generic QBF solvers. The quality (size and structure) of the encoding has a great influence on performance, but precise knowledge of what constitutes a good encoding is quite limited.

We address two bottlenecks in current QBF encodings for positional games [84]: First, explicit representation of winning configurations. In some positional games, the

winning configurations consist of a fixed number of shapes (Tic-Tac-Toe, Gomoku). However, in Hex, winning configurations are defined in terms of paths that connect two borders of the board. So an explicit representation of the goal is exponential in the board size. We study implicit goal representations, by expressing winning path conditions using neighbor relations. It appears that implicit goal constraints not only yield more concise encodings but also boost the performance of current QBF solvers by an order of magnitude. The fastest result was obtained by an implicit encoding of the winning condition in the *transversal* game (the second player cannot win).

The other bottleneck leading to large encodings is duplication of variables and clauses by instantiation to board positions and unrolling to bounded depth. Extending techniques from planning [111], we represent symbolic positions by universal variables. This lifted encoding is concise, at the expense of an extra quantifier alternation. We also study stateless encodings, which avoid position variables by expressing all constraints in terms of the chosen moves.

In this paper, we propose the following 6 QBF encodings, combining all ideas mentioned above.[1] We study the size of these encodings depending on board size and game depth. The encoded games can be solved with existing QBF solvers. We present an experimental evaluation, measuring the performance of a QBF solver for all our encodings, applied to a benchmark of Piet Hein's Hex puzzles (on 3×3 - 7×7 boards), to a set of human-played Hex championship plays (19×19 board), and to Harary's Tic-Tac-Toe (5×5).

Goal: / Board:	Explicit	Lifted	Stateless
All minimal paths	7.3.1 (EA)	-	-
Neighbor-based	7.3.2 (EN)	7.4.1 (LN)	7.4.3 (SN)
Transversal game	7.3.2 (ET)	7.4.2 (LT)	-

7.2   Preliminaries

7.2.1   Maker-Breaker Positional Games and QBF

A positional game is played on a board, where 2 players, called Black and White, occupy empty positions in turns. The initial board can be either empty or some of the positions are already occupied. In the maker-breaker variant, a game is won by the first player (Black) if and only if the final set of black positions contains a winning set; otherwise, it is won by the second player (White), so there is no draw in these games. We define these games formally as follows.

Definition 14. *Given a set of positions* $\mathscr{P}$*, define* $\eta = |\mathscr{P}|$*. A maker-breaker positional game* Π *is a tuple* $\langle I, \mathscr{W} \rangle$*, with*

- *Initial state* $I = (I_B, I_W)$ *s.t.* $I_B, I_W \subseteq \mathscr{P}$ *and* $I_B \cap I_W = \emptyset$,

- *and Goal condition* $\mathscr{W} \subseteq 2^{\mathscr{P}}$*.*

---

[1]The full version and technical appendix also consider the LA and SA encodings.

We assume, without loss of generality, that $I_B = I_W = \emptyset$, possibly after some preprocessing (cf. §7.3.2).

A single play is a sequence of moves (occupying positions) chosen by players in turns. We assume Black plays first and in a maximal play, the game ends with Black's turn.

Definition 15. *Given* Π, *a single play* ϕ *is a sequence of k positions* $\langle \phi_1, \ldots, \phi_k \rangle$ *chosen by each player alternatively. The black moves* $\phi_B = \{\phi_i \mid 1 \le i \le k, i \text{ odd}\}$ *and the white moves* $\phi_W = \{\phi_i \mid 1 \le i \le k, i \text{ even}\}$. *A play is valid when* $\{\phi_B \cup \phi_W\} \cap \{I_B \cup I_W\} = \emptyset$ *and* $\phi_i \ne \phi_j$ *for all* $i \ne j$.

Definition 16. *Given* Π *and a valid play* ϕ, *we say the play* ϕ *is won by Black if and only if there exists a set of positions* $\text{win} \in \mathscr{W}$ *such that* $\text{win} \subseteq I_B \cup \phi_B$.

The goal of all encodings in this paper is to decide if Black has a winning strategy, i.e., Black will win all valid games where it plays according to this strategy. We consider only strategies for a bounded number of moves up to depth d.

7.2.2 Hex and Generalized Hex

Hex is a well-known positional game played on an $n \times n$-board of hexagons, such that each (non-border) position has six neighbors [55]. The game is won by Black if there is a black path connecting Black's two opposite borders. On a completely filled board, Black has a winning connection if and only if White's opposite borders are not connected with a path of white stones. This is known as the *Hex Theorem* [45].

Because of its simplicity and rich mathematical structure, Hex has provided a source of inspiration for the design and implementation of specialized solvers [4], as well as for theoretical considerations on computational complexity [16, 17, 99].

Definition 17. Generalized Hex *is a 2-player game between Short and Cut, where an instance is a 3-tuple* $\langle G, s, e \rangle$, *with* $G = \langle \mathscr{P} \cup \{s, e\}, \mathscr{E} \rangle$ *an undirected graph, and s and e two distinguished nodes. The two players take turns claiming nodes from* $\mathscr{P}$ *and Short wins if s to e are connected with a path of nodes claimed by Short. Cut aims at preventing it by claiming a set of nodes that constitutes a cut from s to e.*

To simplify notation in our encodings, we introduce $\Gamma_s = \{v \mid (s, v) \in \mathscr{E}\}$ and $\Gamma_e = \{v \mid (v, e) \in \mathscr{E}\}$. Short's goal is to create a path from any node of Γ_s to any node of Γ_e.

7.2.3 Quantified Boolean Formulas

We consider closed QBF formulas in prenex normal form, i.e., $Q_1 x_1 \cdots Q_n x_n(\Phi)$, where Φ is a propositional formula with Boolean variables in $\{x_1, \ldots, x_n\}$ and each $Q_i \in \{\forall, \exists\}$. Every such formula evaluates to true or false. QBF evaluation is a standard PSPACE-complete problem. It is well known that the complexity increases with the number of quantifier alternations.

Several QBF solvers exist, which operate on QBF in either QDIMACS format, where Φ is essentially a set of CNF (conjunctive normal form) clauses, or in QCIR format [67], where Φ is provided as a circuit with and- and or-gates and negation. In general, QDIMACS is more low-level and allows efficient operations, while QCIR contains more structure and can be more readable. We use both QDIMACS, using Bule [68] for concise specifications, and QCIR, which can be transformed to QDIMACS using the Tseitin transformation [125], introducing one existential Boolean variable per gate.

7.3   Maker-Breaker Explicit Board Encodings

All our encodings provide a QBF formula that evaluates to True if and only if the corresponding game is won by Black from the empty board, using d moves starting and ending with Black (so d is odd). Using QBF, we can naturally capture the moves of the players in a maker-breaker positional game by d alternating existential and universal variables. In this section, we improve the corrective encoding COR by [84] and in later subsections we adapt to allow implicit goal constraints. The focus here is to provide concise specifications in QDIMACS with handcrafted CNF clauses.

The main idea is to unroll the transition relation d times. In COR, board positions were maintained after each move. We save many frame conditions by maintaining board positions after black moves only. These are needed to test the validity of the moves and the winning condition. All moves are encoded logarithmically (in COR only the White moves).

We identify the set of positions $\mathscr{P}$ with the integers $\{0, 1, \ldots, \eta - 1\}$. For any $v \in \mathscr{P}$, we consider the binary representation of v over $\lceil \lg \eta \rceil$ bits. We write $\bar{v}$ for the set of bits assigned 1 in v and $\underline{v}$ for the set of bits assigned 0 in v. For instance, if $v = 5 = b00\ldots101$, we have $\bar{v} = \{0, 2\}$ and $\underline{v} = \{1, 3, 4, \ldots, \lceil \lg \eta \rceil - 1\}$.

We introduce alternating variables M^t for the chosen move at step t and P^t to represent the board at odd time steps t (after each black move).

$$\exists \mathsf{M}^1 \mathsf{P}^1 \forall \mathsf{M}^2 \exists \mathsf{M}^3 \mathsf{P}^3 \ldots \forall \mathsf{M}^{t-1} \exists \mathsf{M}^t \mathsf{P}^t \ldots \exists \mathsf{M}^d \mathsf{P}^d \tag{7.1}$$

Here variables $\mathsf{M}^t = \{\mathsf{m}_i^t \mid 0 \le i < \lceil \lg \eta \rceil\}$ and variables $\mathsf{P}^t = \{\mathsf{b}_v^t, \mathsf{w}_v^t \mid v \in \mathscr{P}\}$.

For each vertex $v \in \mathscr{P}$, odd time step t, and bit index j we get the following clauses, which specify the correct color of each position at odd time steps t, depending on the previous position and the last moves chosen by Black and White.

$$\mathsf{w}_v^{t-2} \to \mathsf{w}_v^t \tag{7.2}$$

$$\bigwedge_{i \in \bar{v}} \mathsf{m}_i^{t-1} \wedge \bigwedge_{i \in \underline{v}} \neg \mathsf{m}_i^{t-1} \wedge \neg \mathsf{b}_v^{t-2} \to \mathsf{w}_v^t \tag{7.3}$$

$$\mathsf{w}_v^t \to \neg \mathsf{b}_v^t \tag{7.4}$$

$$\mathsf{m}_j^t \wedge \neg \mathsf{b}_v^{t-2} \to \neg \mathsf{b}_v^t \qquad \text{if } j \in \underline{v} \tag{7.5}$$

$$\neg \mathsf{m}_j^t \wedge \neg \mathsf{b}_v^{t-2} \to \neg \mathsf{b}_v^t \qquad \text{if } j \in \bar{v} \tag{7.6}$$

When the above clauses refer to variables unquantified in the prefix, e.g., b_v^{-1}, they are substituted with $\perp$. Note that when b_v^t is not forced to False by the clauses above, it becomes True by the existential quantification over P^t.

7.3.1 Explicit Goal (EA)

Similar to COR, we encode the goal conditions using explicit winning configurations. We introduce variables h_h in the innermost existential block, indicating that Black won by configuration $h \in \mathscr{W}$.

$$\exists\{h_h \mid h \in \mathscr{W}\} \tag{7.7}$$

The following clauses indicate that all positions in some winning configuration are black in the final state, i.e., after d moves:

$$\bigvee_{h \in \mathscr{W}} h_h \tag{7.8}$$

$$h_h \rightarrow b_v^d \qquad \text{for } h \in \mathscr{W}, \text{ and } v \in h \tag{7.9}$$

The size of the encoding grows linearly with the number of explicit winning configurations. Instead of all winning configurations which can blow-up even for small boards in games like Hex, we consider minimal winning configurations for our encoding. Although EA only considers minimal winning paths, it still grows exponentially with board size in games like Hex (with a smaller exponent than COR).

7.3.2 Implicit Goal Representation

For Gomoku and Tic-Tac-Toe, the explicit goal representation is good enough, since the number of winning configurations is polynomial in the board size. However, in Hex, the winning condition is based on paths, whose number grows exponentially in the board size. In this section, we represent path conditions compactly, using the structure of the board. We consider board games as graph games, where the edges indicate the neighbor relation. This paves the way for compact encodings of paths (and other structures). We illustrate implicit goal encodings for the Maker-Breaker positional game Hex, but it is also applicable to other positional games, like HTTT (Sec. 7.6). §7.3.2 discusses preprocessing steps from Hex to Generalized Hex. The subsequent subsections describe compact goal encodings with explicit board constraints. The last subsection considers the transversal game, avoiding White's connecting paths instead.

Transformations to Generalized Hex

Consider the Hex puzzle in Figure 7.1a due to Hein [55]. Black has a winning strategy of depth 7 starting with c_3. For the sake of a running example, we will assume we search for a strategy of depth $d = 5$. Let us call B the existence of a 5-move win for Black on Figure 7.1a.

(a) Original puzzle

(b) Filled-in Short Hex instance for $d = 5$

(c) Contracted Generalized Hex instance

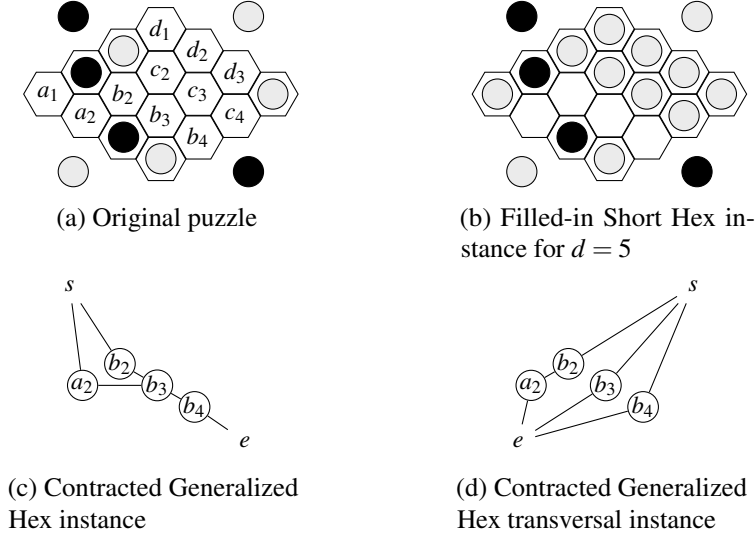(d) Contracted Generalized Hex transversal instance

Figure 7.1: Hein puzzle 9 and its reductions to Generalized Hex when $d = 5$. Black wins in 5 in (a) iff Black wins in 5 in (b) iff Short wins in 5 in (c) iff Cut wins in 5 in (d).

Any winning strategy of d-moves can involve $\ell + 1 = \frac{d+1}{2}$ black moves at most. As such, black cannot win through any winning configuration of size $> \ell + 1$, so these configurations can be removed from the game without altering whether a d-move winning strategy exists. Similarly, we could remove from $\mathcal{W}$ any winning configuration that is not minimal for the subset relation. Removing these configurations from $\mathcal{W}$ may result in many positions not occurring in any winning configuration at all. In a Maker-Breaker game, such positions may as well be claimed by White (Breaker). As such, our initial query, B, is equivalent to whether Black has a 5-move win in Fig. 7.1b.

In any Maker-Breaker game, positions already claimed by either player can be preprocessed. Any winning configuration containing a white-claimed position is removed from $\mathcal{W}$ and black-claimed positions can be removed from any winning configuration. Following the contraction of [17, proof of Theorem 2] for Generalized Hex, this corresponds to removing any Cut-claimed vertex and its incident edges and removing any Short-claimed vertex and turning its neighborhood into a clique. Applying this contraction to our running example gives Fig. 7.1c where Short has a 5-move win if and only if B.

The Hex Theorem provides another approach to solving Hex, which involves the transversal game. To verify if Black has won after d moves, one can fill the remaining empty cells for White and ensure that even then White has no connecting paths. For this approach, we can similarly fill unnecessary cells for White (Fig. 7.1b) and apply the contraction process to Generalized Hex with the two players swapped as in Fig. 7.1d. This position admits a 5-move win for Cut iff B.

Explicit Board Neighbor-based (EN)

We first apply the symbolic neighbor-based goal encoding to the explicit board constraints. To this end, we keep the quantification and transition constraints (Eq. 7.1-7.6). We replace the explicit goal constraints (7.7-7.9) by the following constraints. Recall that $\ell + 1 = \frac{d+1}{2}$ is the maximum length of a winning path. We introduce Boolean variables:

$$\exists\{\mathsf{p}_v^i \mid v \in \mathscr{P}, 0 \leq i \leq \ell\} \tag{7.10}$$

Here p_v^i codes that position v is the i-th position in a winning witness path. We get the following clauses, which specify that all positions on the path are black on the final board; the initial path position is on the start border Γ_s, the path is connected in the graph; and the last path position is on the end border, i.e., it cannot be outside Γ_e.

$$\mathsf{p}_v^i \to \mathsf{b}_v^d \qquad\qquad\qquad \text{for } v \in \mathscr{P}, 0 \leq i \leq \ell \tag{7.11}$$

$$\bigvee_{v \in \Gamma_s} \mathsf{p}_v^0 \tag{7.12}$$

$$\mathsf{p}_v^i \to \bigvee_{(v,w) \in \mathscr{E}} \mathsf{p}_w^{i+1} \qquad\qquad v \in \mathscr{P} \setminus \Gamma_e, 0 \leq i < \ell \tag{7.13}$$

$$\neg\mathsf{p}_v^\ell \qquad\qquad\qquad\qquad v \in \mathscr{P} \setminus \Gamma_e \tag{7.14}$$

There is a slight complication: the winning paths can be of different lengths, while our encoding assumes that all paths have the same length $\ell + 1$. This is solved by allowing some "stutter steps" at the end of the witness path. Rather than changing the QBF encoding, we implemented this by adding the reflexive closure for Γ_e nodes to $\mathscr{E}$ in the input graph.

Explicit-Board Transversal-based Goal (ET)

In this subsection, we specify the winning condition in the transversal game. This can be applied to any positional game, but it is effective for Hex because specifying that White has no connecting path is easier than specifying that Black has a connecting path.

Below, Γ_s and Γ_e denote White's starting and ending border. We introduce variables r_v, which hold for all positions that are connected to Γ_s through positions that are white or empty only (i.e., $\neg\mathsf{b}_w^d$). Then we simply check that no Γ_e position is connected. The following equations for ET are combined with the quantification and transition constraints (Eq. 7.1-7.6), but replace the goal-detection clauses in EA or EN.

$$\exists\{\mathsf{r}_v \mid v \in \mathscr{P}\} \tag{7.15}$$

We get the following clauses

$$\neg b_v^d \rightarrow r_v \qquad\qquad\qquad \text{for } v \in \Gamma_s \qquad\qquad (7.16)$$

$$r_v \wedge \neg b_w^d \rightarrow r_w \qquad\qquad \text{for } (v,w) \in \mathscr{E} \qquad\qquad (7.17)$$

$$\neg r_v \qquad\qquad\qquad\qquad \text{for } v \in \Gamma_e \qquad\qquad (7.18)$$

7.4   Implicit Board Encodings

The Explicit board encodings generate similar variables and constraints for each position after a black move. Note that the validity of all moves can be checked independently for each individual position.

Firstly, we use this structure to generate a lifted encoding in §7.4.1, representing position constraints symbolically on a universal position variable. This follows the lifted encoding for planning in [111]. Following [111], we first generate our lifted encodings in QCIR format preserving the structure of constraints. This opens up comparison with QBF solvers that operate on QCIR format directly, and nicely complements our carefully handcrafted QDIMACS instances of E*-encodings. We consider both a Neighbor-based implicit goal encoding in §7.4.1 and a Transversal-based goal encoding in §7.4.2.

Secondly, we present an encoding that avoids all intermediate states entirely in §7.4.3. The idea is similar to Causal Planning [72]. We consider the Neighbor-based implicit goal specification for the stateless encoding.

It is possible to generate implicit board encodings with explicit goal conditions, to handle arbitrary positional games. We provide lifted and stateless versions of such encodings in Appendix C.

7.4.1   Lifted Neighbor-Based (LN)

We introduce variables $M = \{M^1, \ldots, M^d\}$, where M^t is a sequence of $\lceil \lg(\eta) \rceil$ boolean variables representing one move at time step t. Given a single play, we check if all moves are valid by using a single symbolic position, represented by P, a sequence of $\lceil \lg(\eta) \rceil$ universal variables. When expanded, the universal branches of P correspond to all board positions. We use 2 variables for each time step to represent the state of the symbolic position. Let $S^t = \{o^t, w^t\}$, representing if the symbolic state is *occupied* and/or *white* at time step t. We introduce witness variables $W = \{W^0, \ldots, W^\ell\}$, i.e., a sequence of positions that should form a winning configuration for Black. Here $\ell = \frac{d-1}{2}$, so $\ell + 1$ is the maximum size of a witness set in $\mathscr{W}$. Each W^i is a sequence of $\lceil \lg(\eta) \rceil$ Boolean variables. Finally, we introduce neighbor variables N, a sequence of $\lceil \lg(\eta) \rceil$ variables representing any neighbor of symbolic position P.

The corresponding Lifted Encoding for finding a winning strategy for Black has

this shape:

$$\exists M^1 \forall M^2 \ldots \exists M^d$$
$$\exists W^0, \ldots, W^\ell$$
$$\forall P$$
$$\exists S^1, \ldots, S^{d+1} \quad \exists N$$
$$\neg o^1 \wedge \bigwedge_{t=1,3,\ldots,d} T_B^t(S^t, S^{t+1}, P, M^t) \wedge BR(M) \wedge \bigwedge_{t=2,4,\ldots,d-1} T_W^t(S^t, S^{t+1}, P, M^t) \wedge$$
$$SRC(W) \wedge TRG(W) \wedge G_{\mathscr{E}}(N, W, P) \wedge G_{sb}(S^{d+1}, W, P)$$

The initial board is empty, hence $\neg o^1$. This forces *every* node to be unoccupied, since it is under the universal quantifier $\forall P$.

We separate the transition constraints for Black and White. For Black, the position in the universal branch of the chosen move must be empty at time step t and black at time step $t+1$. In all other branches, we just propagate the position. For White, the position is only updated if it is empty in the universal branch of the chosen move. Otherwise, the position is propagated. We use "=" to denote the bit-wise conjunction of bi-implications.

Definition 18. *Black transitions:* $T_B^t(S^t, S^{t+1}, P, M^t) =$

$$\big((P = M^t) \implies (\neg o^t \wedge \neg w^{t+1} \wedge o^{t+1})\big) \wedge \big((P \neq M^t) \implies (w^t = w^{t+1} \wedge o^t = o^{t+1})\big)$$

Bound Restriction for black: $BR(M) = \bigwedge_{t=1,3,\ldots,d}(M^t < \eta)$

Definition 19. *White transitions:* $T_W^t(S^t, S^{t+1}, P, M^t) =$

$$\big((P = M^t \wedge \neg o^t) \implies (w^{t+1} \wedge o^{t+1})\big) \wedge \big((P \neq M^t \vee o^t) \implies (w^t = w^{t+1} \wedge o^t = o^{t+1})\big)$$

Finally, the restriction on black moves (BR, Def. 18) limits Black's legal moves to the board, in case the number of positions is not a power of 2. We use the standard less-than ($<$) on bit vectors. This prunes the search space. A similar constraint for White reduced the performance of the QBF solver. It is safe to drop it since White can never win by playing outside the board (it corresponds to giving up a move).

For the Goal condition, we use the neighbor relation from the Generalized Hex input. We adapt the Neighbor-based goal constraints from EN to the lifted setting. First, we specify that the first and last witness positions lie on the proper borders, $SRC(W) \wedge TRG(W)$. We use "bin" to express the logarithmic encoding of positions on the board by $\lceil \lg(\eta) \rceil$ bits. That is, $bin(x_{\lceil \lg(\eta) \rceil - 1} \ldots x_1 x_0, v) = \bigwedge_{i \in \bar{v}} x_i \wedge \bigwedge_{i \in \underline{v}} \neg x_i$ (cf. notation in §7.3).

Definition 20. *Source* $SRC(W) = \bigvee_{v \in \Gamma_s} bin(W^0, v)$; *Target* $TRG(W) = \bigvee_{v \in \Gamma_e} bin(W^\ell, v)$

The witness positions must be black (G_{sb}), i.e., the position at time step d in the corresponding universal branch must be occupied and not white. We must also specify that adjacent positions in the witness path are connected in the graph ($G_{\mathscr{E}}$). We first specify that the binary decodings of the symbolic position P and its symbolic neighbour N are related in the graph (first implication in $G_{\mathscr{E}}$). Finally, we specify that if some witness position matches P in the current branch, the next witness position matches its symbolic neighbor N (second implication in $G_{\mathscr{E}}$).

Definition 21. *Goal constraints:*

$$G_{sb}(S^{d+1}, W, P) = \Big(\bigvee_{i=0}^{\ell} (P = W^i) \Big) \implies (\neg w^{d+1} \wedge o^{d+1})$$

$$G_{\mathscr{E}}(N, W, P) = \bigwedge_{v=0}^{\eta-1} \big(\mathrm{bin}(P, v) \implies \bigvee_{(v,w) \in \mathscr{E}} \mathrm{bin}(N, w) \big) \wedge \bigwedge_{i=0}^{\ell-1} \big((W^i = P) \implies (W^{i+1} = N) \big)$$

7.4.2   Lifted Transversal-Based (LT)

We use the same move, state and symbolic position variables and the same constraints as LN, except for the variables and constraints to specify the goal condition. Similar to ET, we introduce reachability variables $\exists r_0, \ldots, r_{\eta-1}$ (one for each position) instead of the witness variables. Instead of checking for a black witness, we check that White can disconnect Black's borders.

The corresponding Lifted Encoding for finding a winning strategy for Black with Transversal goal constraints has this shape:

$$\exists M^1 \forall M^2 \ldots \exists M^d$$
$$\exists r_0, \ldots, r_{\eta-1}$$
$$\forall P$$
$$\exists S^1, \ldots, S^{d+1}$$
$$\neg o^1 \wedge \bigwedge_{t=1,3,\ldots,d} T_B^t(S^t, S^{t+1}, P, M^t) \wedge BR(M) \wedge \bigwedge_{t=2,4,\ldots,d-1} T_W^t(S^t, S^{t+1}, P, M^t) \wedge$$

We now provide a Lifted variant of the goal constraints from the ET encoding (Eq. 7.16-7.18). There are three main constraints: (1) Non-black nodes on the start boarder Γ_s are reachable; (2) For each neighbor pair, if a neighbor is reachable then current non-black position is also reachable; (3) Nodes on the end boarder Γ_e are not reachable.

$$\bigwedge_{v \in \Gamma_s} \mathrm{bin}(P,v) \implies \big((\mathrm{w}^{d+1} \vee \neg \mathrm{o}^{d+1}) \iff r_v\big) \wedge$$

$$\bigwedge_{(v,w) \in \mathscr{E}} (\mathrm{bin}(P,v) \wedge r_w) \implies \big((\mathrm{w}^{d+1} \vee \neg \mathrm{o}^{d+1}) \iff r_v\big) \wedge$$

$$\bigwedge_{v \in \Gamma_e} \neg r_v$$

7.4.3 Stateless Neighbor-Based (SN)

In positional games, Black's moves are valid if and only if they are different from all previous moves. Also, the winning condition can be expressed completely in terms of Black's moves. In the maker-breaker case, we don't even need to check the validity of White's moves. We just ignore White's moves that are played outside the board or on occupied positions. This is correct because in positional games White cannot win by giving up a move.

The move and witness variables are the same as in §7.4.1. We can now drop all other variables for the LN encoding. There are three main constraints: (1) Black's moves cannot overwrite previous moves. (2) Witness positions must be played by Black. (3) Adjacent position pairs in W are neighbors in $\mathscr{E}$, and form a winning path between Black's borders.

$$\exists M^1 \forall M^2 \dots \exists M^d$$
$$\exists W^0, \dots, W^\ell$$

$$\mathrm{BR}(M) \wedge \big(\bigwedge_{t=1,3,\dots,d} \bigwedge_{i<t} M^t \neq M^i \big) \wedge \big(\bigwedge_{i=0}^{\ell} \bigvee_{t=1,3,\dots,d} W^i = M^t \big) \wedge$$

$$\mathrm{SRC}(W) \wedge \mathrm{TRG}(W) \wedge \bigwedge_{i=0}^{\ell-1} \bigwedge_{v=0}^{\eta-1} \big(\mathrm{bin}(W^i,v) \implies \bigvee_{(v,w) \in \mathscr{E}} \mathrm{bin}(W^{i+1},w) \big)$$

Instead of the symbolic black constraint, here we use the black moves to constrain the witness to be black. First, we provide constraints for first and last witness positions, as in the LN encoding (Def. 20). Then, since the stateless encoding has no notion of a symbolic node, we need to provide the neighbor constraints for each pair of adjacent witness nodes.

7.5 Implementation and Evaluation

We provide two new tools to generate explicit goal encodings for all maker-breaker games and implicit goal encodings for Hex and HTTT. The first tool[2] is an extension of the COR encoding by [84], which allows an implicit goal specification and generates

---

[2]Available at `https://github.com/vale1410/positional-games-qbf-encoding`

Table 7.1: Alternation depth and size of explicit board encodings (QDIMACS) and implicit board encodings (QCIR).

Enc.	Alt.	# Variables	# Clauses	Enc.	Alt.	# Variables	# Gates
EA		$d\eta+|\mathscr{W}|$	$\frac{1}{2}d\eta\lg\eta+d|\mathscr{W}|$	SN	d	$\frac{3}{2}d\lg\eta$	$d^2\lceil\lg\eta\rceil+\frac{1}{2}d\eta$
EN	d	$\frac{3}{2}d\eta$	$\frac{1}{2}d\eta\lg\eta$	LN	$d+1$	$\frac{3}{2}d\lg\eta$	$4d\lg\eta+4\eta$
ET		$d\eta$	$\frac{1}{2}d\eta\lg\eta+|\mathscr{E}|$	LT		$d\lg\eta+\eta$	$2d\lg\eta+6\eta$

Table 7.2: Size of implicit goal encodings on an empty 19×19 Hex board in QDI-MACS

Enc.	Generated QBF Encodings (variables/clauses)			
	d=45	d=91	d=181	d=361
EN	25k/122k	50k/246k	100k/488k	199k/972k
ET	17k/100k	34k/200k	67k/395k	134k/785k
LN	5k/22k	9k/33k	17k/54k	32k/97k
LT	5k/18k	8k/25k	12k/37k	22k/63k
SN	53k/261k	167k/720k	560k/2182k	2027k/7342k

the EA, EN, and ET encodings in QDIMACS format. The second tool[3] generates the implicit board encodings SN, LN, and LT (and also LA and SA, Appendix C.) in QCIR format. These can be translated to the QDIMACS format using the Tseitin transformation [125].

7.5.1 Size of the QBF Encodings

Table 7.1 shows the alternation depth, number of variables, and number of clauses/gates for the explicit and implicit board encodings of Hex,[4] depending on the number of positions (η), the depth of the game (d), the size of the winning set $\mathscr{W}$, and the number of edges $\mathscr{E}$. Obviously, the explicit-goal encoding (EA) depends on $|\mathscr{W}|$, which is unfeasible for 19×19 Hex boards. The implicit board versions (L*) are more concise than the explicit board versions (E*), at the expense of an extra quantifier alternation. The stateless encoding saves this quantifier, but its size grows quadratically in the depth of the game.

In Table 7.2 we measured the actual number of variables and clauses generated by our tools for increasingly deeper games on an empty 19×19 board. These can only be generated for the implicit goal encodings (*N,*T). Here we translated the QCIR benchmarks to QDIMACS for a fair comparison. For SN, the quadratic growth in the game depth is clearly visible.

Unfortunately, existing QBF solvers cannot solve any of these games on a 19×19 board. To study the effect of the encodings on the performance of QBF solvers,

---

[3]Available at `https://github.com/irfansha/Q-sage`

[4]We display here an asymptotically equivalent function. Appendix D. shows the exact size.

we experiment with the Piet Hein benchmark of Hex puzzles on small boards. We also experiment with "shallow" end games on a 19×19 board, obtained from human championships. In the latter, after removing useless locations, the number of open positions is quite small, giving a fair chance to all encodings.

7.5.2 Solver Performance for Various Encodings

Experiment Design

We perform three experiments to evaluate the performance of QBF solvers on our encodings. The first one is on a benchmark of 30 instances of Piet Hein's Hex puzzles of board sizes 3x3 to 5x5 from [84], which we call *Hein-base*. The second one is on the extended benchmark of 10 Piet Hein's Hex puzzles with board sizes 6x6 and 7x7, which we call *Hein-hard*. The third one runs on games from the best human players in the recent 19×19 Hex championship.[5] These games are resigned at quite an early point. This yields 15 UNSAT instances (for depths 11-17). Then we finish the game using the Hex-specific solver Wolve [4] and roll back some moves. This provides 8 SAT instances (for depths 9-17). After pruning useless positions (cf. Sec. 7.3.2), the number of open positions in the first and second set of benchmarks is at most 40. The 19×19 boards have 10-146 open positions. Due to the filled positions and the moderate game depth, many open positions become useless.

We use two state-of the-art QDIMACS solvers CAQE (CQ) [98] and DepQBF (DQ) [80], and two QCIR solvers CQESTO (CT) [65] and Quabs (QU) [56]. We also use the QBF preprocessors Bloqqer (B) [11], HQSpre (H) [132], QRATpre+ (Q) [81], or none (N). We preprocess the QCIR instances by first translating them to QDIMACS, applying a preprocessor, and then trying to reconstruct the QCIR structure after preprocessing. The reconstruction uses scripts from GhostQ solver[6].

For the first experiment, we run all combinations of QBF preprocessors and solvers on all encodings of the 30 Hein-base instances. Each run is limited to 1 hour and 8 GB memory. For each encoding, we select the two best solvers, each with their best preprocessor. For the second and third experiment, we apply the two selected combinations to each encoding. We run all experiments on a cluster,[7] limited by 3 hours on a single core and 32GB memory.

Experimental Results

Tables 7.3 and 7.4 show the number of solved instances and their average solving time for all solver-preprocessor combinations on Hein-base benchmarks. The bold text shows the best preprocessor for each solver and encoding. Table 7.5 shows the

---

[5]Available at `https://www.littlegolem.net`. Appendix A. describes the benchmark selection in detail.

[6]QDIMACS to QCIR convertor available at https://www.wklieber.com/ghostq/2017.html

[7]Huawei FusionServer Pro V1288H V5; 384 GB memory; 3.0 GHz Intel Xeon Gold 6248R processor.
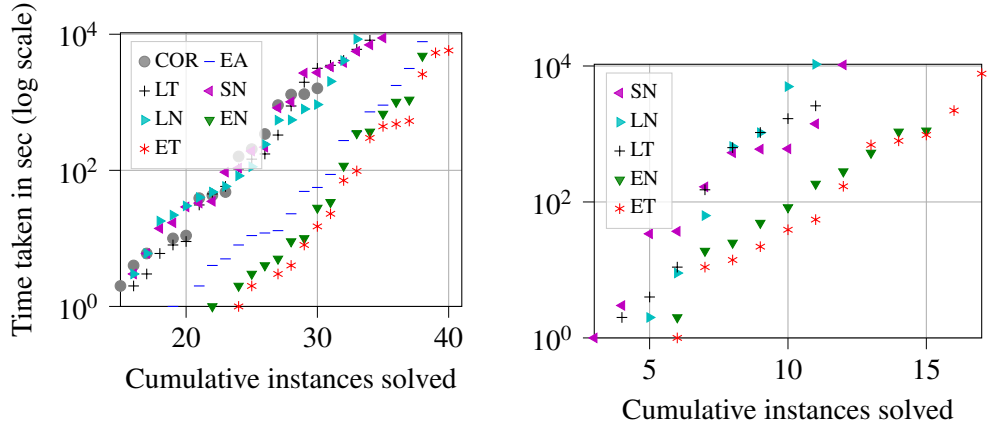
(a) Hein-base + Hein-hard instances solved by our 6 encodings and the original COR encoding. 3 hour timelimit and 8GB memory limit (log scale).

(b) Championship instances solved by the 5 implicit goal encodings. 3 hour timelimit and 32GB memory limit (log scale)

Figure 7.2: Experimental Results with best solver-preprocessor combination for each encoding

number of (uniquely) solved instances and their average solving time for the Hein-hard and Championship benchmarks, using the two best solvers selected after the first experiment. Finally, we show cactus plots illustrating the cumulative number of solved instances over time, using the best solver/preprocessor for each encoding. Figure 7.2a shows the results for all Hein instances (base and hard) on all our six encodings and the original COR encoding. Figure 7.2b shows the results for the

Table 7.3: Hein-base benchmarks with all QDIMACS solver-preprocessor combinations.
Number of solved instances / Average solving time over solved instances.

Sol.	COR	EA	EN	ET	LN	SN	LT
CQ-B	**29/227.0**	**30/7.3**	30/4.6	30/9.3	**30/175.9**	29/272.9	**29/234.0**
CQ-H	25/235.9	30/10.4	30/5.9	30/2.2	15/323.8	27/143.0	20/963.7
CQ-Q	29/362.5	30/7.8	**30/2.4**	30/1.8	13/178.3	24/401.8	13/118.3
CQ-N	29/418.1	30/8.3	30/3.0	**30/1.3**	13/177.6	19/642.5	13/188.3
DQ-B	**30/200.5**	30/55.7	30/186.4	30/62.1	**29/243.2**	26/210.7	29/359.8
DQ-H	26/181.7	**30/45.3**	30/76.8	**30/30.0**	16/478.2	25/132.9	15/144.2
DQ-Q	28/291.8	30/76.0	30/90.7	30/30.1	13/118.0	24/324.6	14/289.5
DQ-N	26/123.5	30/54.4	**30/43.9**	30/32.8	13/214.5	22/318.2	13/196.8

Table 7.4: Hein-base benchmarks with all QCIR solver-preprocessor combinations. Number of solved instances / Average solving time over solved instances.

Sol.	LN	SN	LT
CT-B	**24/154.7**	24/286.2	**25/242.6**
CT-H	10/23.9	22/187.1	11/239.6
CT-Q	10/12.2	17/50.8	12/369.2
CT-N	12/305.9	**24/134.0**	12/219.5
QU-B	**30/298.6**	24/358.8	**29/330.7**
QU-H	13/258.5	22/400.9	13/183.7
QU-Q	13/43.8	21/269.9	13/42.0
QU-N	13/63.3	23/380.1	14/322.8

Table 7.5: Hein-hard and Championship benchmarks solved by the two best solver-preprocessors. Number of solved instances / Average solving time over solved instances.

#U is the Number of unique instances solved by the encoding.

Enc.	Hein-Hard (10 instances)				Championship (23 instances)			
	#U	CQ-*	DQ-*	QU-B	#U	CQ-*	DQ-*	QU-B
COR	0	1/7003.0	0/-	-	-	-	-	-
EA	0	8/1877.1	1/695.0	-	-	-	-	-
EN	0	8/1053.2	4/5277.0	-	0	15/224.0	10/119.4	-
ET	2	10/1563.0	4/4768.2	-	2	17/746.5	11/545.2	-
LN	0	3/4268.7	-	2/1641.0	1	11/1575.8	-	8/833.5
SN	0	5/4964.4	1/1710.0	-	0	12/1147.7	10/1315.6	-
LT	0	4/4449.0	-	1/1343.0	0	11/556.9	-	9/803.4

Championship benchmarks on the five implicit goal descriptions.[8]
We now present our main observations:

(i) Compared to the original COR encoding, all implicit board encodings solve more instances. The explicit board encodings (E*) significantly outperform the COR encoding. For the Hein benchmarks, EA performs nearly as good as the implicit-goal versions EN and ET. However, note that for 19x19 boards we couldn't even generate the COR and EA encodings, since the set of winning paths cannot be practically enumerated.

(ii) Clearly, in all experiments, explicit board encodings (E*) perform an order of magnitude faster than implicit board encodings (L*, S*). So the extra quantifier alternation in L*-encodings is not compensated by their smaller size. Interestingly, LN solves one UNSAT instance uniquely in championship instances compared to all

---

[8]In Appendices E., F., we also present memory plots, and plots with SAT and UNSAT instances separated.

other encodings. The unique instance has depth 15 and 83 open positions making it the largest solved UNSAT instance.

(iii) Within the implicit board encodings, SN, LN and LT perform quite similarly, and none of them dominates the other. Among SN and LN, SN manages to solve a few more instances than LN overall. LT and LN complement each other in many cases, especially in the Championship benchmarks.

(v) ET is the only encoding that solves all 40 instances of the Hein instances (up to depth 15 for UNSAT cases). It solves most instances of the second experiment: 11 UNSAT (of which one uniquely) and 6 SAT instances (of which one uniquely). The unique SAT instance solved has depth 15 and 115 open positions, which is the largest solved instance overall. Within the explicit board encodings, ET performs somewhat better than EN, but it doesn't clearly dominate it. Apparently, checking that White blocks Black is easier than checking that Black has won, but this may be quite specific to Hex.

7.5.3   Discussion

Implicit board vs Explicit board encodings   The strength of implicit board encodings is a more concise representation, but the difference only shows for larger board sizes and game depths. Remember that LN uniquely solves an instance with depth 15 and 83 open positions in the Championship benchmarks, the largest solved UNSAT instance. In the detailed data (Appendix F., Fig. 7.4b, 7.4d), we see that implicit board encodings seem to have a lower exponent both in time and memory for Hein SAT instances. In a sense, pruning useless positions may have gained explicit board encodings (E*) more advantage than implicit board encodings (L*), especially in Championship benchmarks.

Future QBF solvers might be able to better exploit the structure present in the more concise symbolic encodings. Also, future tools might improve the naive Tseitin circuit translation to clauses, on which the symbolic encodings (L*, S*) rely, while the explicit encodings (E*) are generated with carefully handcrafted, optimal clauses.

Implicit goal vs Explicit goal representations   The EA encoding that lists all winning configurations profits from the fact that we provided only the *minimal* paths, since otherwise some instances would have had over 0.5M winning paths. The neighbor-based versions (*N) outperform this, despite the disadvantage of implicitly representing *all* winning paths. The implicit goal representation could actually be made even more compact by compressing the witness path logarithmically, at the expense of more quantifier alternations, following the iterative squaring technique in bounded model checking and compact planning [25, 69]. This would only pay off for even deeper games.

QDIMACS vs QCIR solvers   We generate the implicit board encodings first in QCIR and then convert them into QDIMACS, to enable a proper comparison between both types of QBF solvers. From Tables 7.3 and 7.4, one can conclude that *without*

preprocessing, QCIR solvers perform on par with QDIMACS solvers. CQESTO on the SN encoding performs the best with 24 instances solved out of 30. Clearly, preprocessing helps QDIMACS solvers more than QCIR solvers, which is expected. To our knowledge, there are currently no preprocessors that act on QCIR format directly, so we applied preprocessing on QDIMACS instances and reconstructed circuit encodings. Often the structure is lost in this process, which puts circuit solvers at a disadvantage.

Most current solvers proceed from the outermost quantifier layer to the innermost quantifier layers. We conjecture that in the L* encodings conflict detection is delayed, since the state information depends on the symbolic variables in the inner layers.

The effect of QBF preprocessing For the E* encodings, the preprocessing does not seem to have much effect. This is expected as the clauses are carefully hand-crafted, thus the main preprocessing techniques (variable elimination and variable expansion [58]) do not improve the solving time. On the other hand, preprocessing helps the L* and SN encodings to a great extent.

Looking at the results produced by Bloqqer, we observe that (1) it applies variable expansion in the innermost symbolic universal layer in L* encodings; (2) it tries to preserve circuit structure when applying variable elimination. It has been reported that the Tseitin transformation blurs structural information that is beneficial for solving [3]. Bloqqer tries to recover this structure in the CNF representation without blowing-up the formula.

We conclude that current solvers (both QCIR and QDIMACS) do not really take advantage of the symbolic board representation in most instances. The performance increase due to Bloqqer is essentially achieved by partially grounding the lifted instances for L*. HQSpre also applies variable expansion and variable elimination, but Bloqqer seems to be more effective overall. QRATpre+ does not apply variable expansion, and consequently it does not help QBF solving for the lifted instances.

7.6 Implicit Goals for Harary's Tic-Tac-Toe

Harary's Generalized Tic-Tac-Toe (HTTT) is a positional game on an $n \times n$ board of squares. It is won by forming (a symmetric variant of) a connected shape from a fixed set. These shapes, called polyominos, go by funny standard names. A first QBF encoding for solving the standard maker-maker version of HTTT was provided in [36]. Later, the proposed benchmarks were solved efficiently with the COR encoding [84], considering instances up to 4×4 boards with polyominos of 4 cells. Then COR was specialized for HTTT by a pairing strategy [20], solving even 7×7 boards, for a number of polyominos, including the 6-cell *Snaky*.

We extend our SN encoding from Hex to the maker-breaker variant of HTTT, by using *two* neighbor relations, *right* and *up*. The polyominos can be characterized by these neighbor relations. For example, the polyomino *Tippy* consists of 4 cells p0, p1, p2, p3, connected as right(p0, p1), up(p2, p1), right(p2, p3). Neighbors in the witness

sequence of the SN encoding are connected using the appropriate neighbor relation, depending on the polyomino. In our encoding, symmetric variants of polyominos are simply treated as different ones. We apply symmetry reduction, by restricting the position of the first move. This extension is not specific to SN, but can easily be applied to the other implicit goal encodings as well.

We performed an experiment, generating SN encodings on a 5×5 board. We consider all 8 polyominos up to 4 cells and 3 other shapes: *Z*, *L* (both with 5 cells) and *Snaky* (6 cells). For each polyomino, we generate encodings of depth 7 to 15 (in steps of 2). This gives 55 instances, encoded in a cumulative size of 2.8 MB. We try to solve these instances with the best preprocessor/solver combination Bloqqer/Caqe, limited to 3 hours and 8 GB memory. All polyominos up to 4 cells are solved within 10 seconds, except for *Fatty* (which does not have a winning strategy and times out at depth 15). The polyomino *Z* is solved at depth 11 within 8 minutes, and *L* and *Snaky* time out at depth 15. For all solved polyominos, the existence of a winning strategy of smaller depth is refuted within the time limit.

7.7   Conclusion and Future Work

We addressed two bottlenecks for the QBF encoding of maker-breaker positional games. Using *symbolic goal constraints* proved to be essential for generating QBF formulas for games with many winning configurations, such as a standard Hex 19×19 board. On smaller boards, this technique also led to a boost in performance, leading to currently the fastest way to solve Hex puzzles using a generic solver.

The other technique, using *symbolic board positions*, leads to even smaller, lifted QBF encodings, but current solvers cannot solve them as fast as the explicit board representation. Still, many game instances could be solved with implicit board representation. Some of our encodings have been submitted to the QBFeval 2022 competition [96],[9] in the hope that future QBF solvers might be able to exploit the special structure of our encodings.

On a Hex 19×19 board, the game depth that can be handled is quite limited. This is not surprising, since game depth corresponds to QBF alternation depth, which is the major factor of complexity for QBF solvers. For limited game depth, many Hex board positions can be pruned away, which brings these instances close to small puzzles. We also investigated game positions where human champions resigned, because they believe the game is lost. Still, the actual number of moves required from these lost positions is quite high. Experts use several patterns specific to Hex (like ladders and bridges) to look ahead many steps. Even specialized Hex solvers [4] have difficulty solving these "lost" positions, but the specialized solvers are of course much more efficient on Hex than generic QBF solvers on our encodings.

However, our approach is more general and is already applicable to any maker-breaker positional game, as we demonstrated for Harary's Tic-Tac-Toe. The ideas can easily be applied to maker-maker variants of positional games (for Hex, the two

---

[9]Available at `http://www.qbflib.org/qbfeval22.php`

variants are equivalent). It is interesting for future work to apply similar techniques to non-positional games, like Connect-Four and Breakthrough.

Technical Appendix – Supplementary Material

A. Selection of Championship Benchmarks

Extending the experiment design in §7.5.2, We detail the selection of the Championship Human-played HEX games. We considered all games among the top human players in the recent 19x19 Hex championship, available at `www.littlegolem.net`. These games are resigned at a quite early point. It can still be a challenge to finish those games to the end. To get an impression, the reader can try them out through a GUI, for instance by clicking on these games played by top-players:

- `game-2291874`

- `game-2291880`

We harvested games between the top players from the last championship. Since the QBF solvers can only solve these games for limited depth, we generated the encodings for depth 11-17; none of the games terminate within 17 steps, so all these instances are UNSAT. Then we pruned useless positions (see §7.3.2). After pruning, in several cases no open positions were left. We removed those games from the benchmark. As a result, we obtained 15 UNSAT instances of depth 11-17. These are part of our benchmark.

To obtain some SAT instances as well, we need to finish the game. Here we used the Hex specific solver Wolve [4]. Wolve could only solve a few of these games. From the final won position, we rolled back 9-17 moves. After rolling back n steps, we generated instances for n and $n + 2$ steps. In the end, this procedure provided us with 8 SAT instances of depth 9-17.

B. Combinations QBF Preprocessors / Solvers

We now detail the experiments on the performance of all combinations of QBF preprocessors and solvers for our encodings on a selection of the benchmark instances.

We tried all combinations of preprocessors and QBF solvers, both for QDIMACS and QCIR format on all encodings for a selection of benchmarks (excluding very small and very large ones). We now provide an overview of the tools we have experimented with.

We have considered the following preprocessors:

- Bloqqer [11]

- HQSpre [132]

- QRATpre+ [81]

We considered these QBF-solvers that operate directly on QCIR circuit format (for those benchmarks that are generated in QCIR):

- CQuesto [65]

- Qfun [64]

- Qute, Quabs [56]

We considered these QBF-solvers for QDIMACS format. We ran them both on encodings generated in QDIMACS, and on encodings generated in QCIR, after a standard Tseitin transformation to clauses.

- Caqe [98]

- DepQBF [80]

- Qute [94]

- Questo [66]

Surprisingly, the QDIMACS solvers applied after a naive Tseitin transformation seem to perform better than specialized QCIR solvers on the same encodings.

Overall, the combination of Bloqqer/Caqe performed the best, although for some encodings, another preprocessor sometimes won. We only report experiments with the best combination, Bloqqer/Caqe in §5.5.2

C. Implicit Board and Explicit Goal Encodings

In this section, we provide *explicit* goal versions of *implicit* board encodings. Further experimental comparison is available in the extended version of this paper [115].

Lifted Board Explicit Goal (LA)

We now modify the LN encoding (Sec. 7.4.1) with explicit goals. For this, we first drop the neighbour variables in the prefix and change the goal constraint. All other variables and constraints for the board remain unchanged. There are 2 goal constraints: (1) the witness must be an element of $\mathscr{W}$; (2) witness positions must be black, i.e., the position at time step d in the corresponding universal branch must be occupied and not white.

The corresponding Lifted Encoding with explicit goals for finding a winning strategy for Black has this shape:

$$\exists M^1 \forall M^2 \dots \exists M^d$$
$$\exists W^0, \dots, W^\ell$$
$$\forall P$$
$$\exists S^1, \dots, S^{d+1}$$
$$\neg o^1 \wedge \bigwedge_{t=1,3,\dots,d} T_B^t(S^t, S^{t+1}, P, M^t) \wedge BR(M) \wedge \bigwedge_{t=2,4,\dots,d-1} T_W^t(S^t, S^{t+1}, P, M^t) \wedge$$
$$\left(\left(\bigvee_{i=0}^{\ell}(P = W^i) \right) \implies (\neg w^{d+1} \wedge o^{d+1}) \right) \wedge \left(\bigvee_{\{v_0,\dots,v_{m-1}\} \in \mathscr{W}} \bigwedge_{i=0}^{m-1} bin(W^i, v_i) \right)$$

State-less Board Explicit Goal (SA)

Similar to LA, we adapt SN with explicit goal by replacing the neighbour constraints on witness positions. The variables and board constraints stay the same as SN, we encode explicit goals using two constraints: (1) the witness positions form a subset of Black's moves; (2) the witness is one of the winning sets of positions (same as in LA)

$$\exists M^1 \forall M^2 \dots \exists M^d$$
$$\exists W^0, \dots, W^\ell$$
$$BR(M) \wedge \bigwedge_{t=1,3,\dots,d} \bigwedge_{i<t} M^t \neq M^i \wedge \bigwedge_{i=0}^{\ell} \bigvee_{t=1,3,\dots,d} W^i = M^t \wedge$$
$$\left(\bigwedge_{i=0}^{\ell} \bigvee_{t=1,3,\dots,d} W^i = M^t \right) \wedge \left(\bigvee_{\{v_0,\dots,v_{m-1}\} \in \mathscr{W}} \bigwedge_{i=0}^{m-1} bin(W^i, v_i) \right)$$

D. Detailed Analysis of Encoding Sizes

§7.5.1 contains an overview of the alternation depth and size of generated encodings, in terms of the number of positions, game depth, and number of edges and winning configurations. We provided simplified, asymptotically equivalent functions. This means that we only gave the dominant terms, with the proper constants; we suppressed the smaller terms. For instance, instead of $3n^2 + 2n$ we would have reported $3n^2$.

Here, we provide the precise functions. We do this both for the EA, EN, ET encodings in DIMACS format (Table 7.6) and for the LN, SN, and LT encodings in QCIR (Table 7.7). In addition, in Table 7.7, we also provide functions for LA and SA encodings reported in §7.7 in QCIR format.

In Table 7.7, we also show two new encodings, LI and SI, which correspond to encodings with iterative squaring, as used in bounded model checking and classical planning [25, 69]. We have implemented these encodings, and we mention these

Table 7.6: Size of the proposed explicit board (QDIMACS) encodings.

Encoding		Altern.	Variables	Clauses		
Board	Goal	depth		Binary	Ternary	Long
Explicit		d	$(d+1)\eta + d\lceil\lg\eta\rceil)$	$d\eta + \eta\lceil\lg\eta\rceil$	$\frac{d-1}{2}\eta\lceil\lg\eta\rceil$	$\frac{d-1}{2}\eta$
EA	All winning		$+|\mathcal{W}|$	$+\sum_{h\in\mathcal{W}}|h|$	$+0$	$+1$
EN	Neighbors		$+\frac{d+1}{2}\eta$	$+\frac{d+1}{2}\eta$	$+0$	$+1+(\eta-|\Gamma_e|)\ell$
ET	Transversal		$+\eta$	$+|\Gamma_s|$	$+|\mathcal{E}|$	$+0$

Table 7.7: Size of the proposed implicit board (QCIR) encodings.

Encoding		Altern.	Variables	Gates
Board	Goal	depth		
Lifted		$d+1$	$(d+1)\lceil\lg\eta\rceil + 2d$	$2d\lceil\lg\eta\rceil + 9d + 1$
LA	All winning		$+\ell\lceil\lg\eta\rceil$	$+\ell(\eta+2\lceil\lg\eta\rceil+1)+2\eta+1+|\mathcal{W}|$
LN	Neighbors		$+(\ell+1)\lceil\lg\eta\rceil$	$+2\ell(2\lceil\lg\eta\rceil+1)+4\eta+|\Gamma_s|+|\Gamma_e|+4$
LT	Transversal		$+\eta$	$+6\eta+|\Gamma_s|+3$
LI	Iterative-sq.	$+\lceil\lg\ell\rceil$	$+(3\lceil\lg\ell\rceil+2)\lceil\lg\eta\rceil+\lceil\lg\ell\rceil$	-
Stateless		d	$d\lceil\lg\eta\rceil$	$\lceil\lg\eta\rceil((d-1)d+(d+1)\ell)+\frac{d^2+4d+3}{4}$
SA	All winning		$+\ell\lceil\lg\eta\rceil$	$+\ell\eta+1+|\mathcal{W}|$
SN	Neighbors		$+(\ell)\lceil\lg\eta\rceil$	$+3\eta\ell+3$
SI	Iterative-sq.	$+\lceil\lg\ell\rceil$	$+(3\lceil\lg\ell\rceil+2)\lceil\lg\eta\rceil+\lceil\lg\ell\rceil$	-

techniques in (see §5.5.2). We suppressed a detailed discussion because this technique only makes sense for deeper games (more moves) than we can handle. However, the LI and SI encoding are interesting from a theoretical perspective, because they encode the winning paths in logarithmic length, rather than linear length. This gain comes at the cost of an increase in alternation depth.

E. Memory Plots for Hein's Hex Benchmarks

In §5.5.2, we presented a cactus plot showing how many instances can be solved by the best solver-preprocessor combination for each encoding, within a certain *time* limit. Figure 7.3 shows a similar cactus plot for the number of solved instances within a certain *memory* limit.

Similar to the time-based analysis, the corresponding solver-preprocessor combination takes the least memory for the explicit board encodings with implicit goals ET and EN, followed by EA (explicit goals) Next are the symbolic encodings with implicit goal conditions LN and SN, followed by LT. Note that for COR, the best solver-preprocessor is DQ-B, DepQBF with Bloqqer. DepQBF uses less memory compared to other solvers across all encodings.
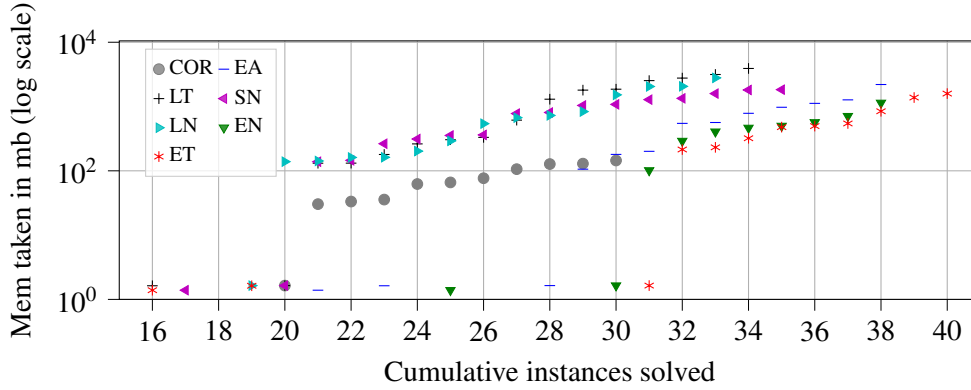
Figure 7.3: Cumulative instances of Hein's Hex (base+hard) solved by our 6 encodings and the original COR encoding within 3 hours. Dependence on memory limit (log scale).

F. Separation of SAT and UNSAT Instances

Hein's Hex Benchmarks, Separated

Here we separate the time/memory cactus plot for all Hein's Hex benchmarks in SAT instances and UNSAT instances. We investigate if different encodings would handle SAT or UNSAT instances better. This would be interesting for portfolio approaches.

Figure 7.4 shows the separation in UNSAT (left) and SAT (right) instances for Hein's benchmarks. We show the dependence on time (top) and memory (bottom). The picture shows that explicit board encodings outperform implicit board encodings. On the other hand, in case of the SAT instances, although the implicit board encodings still take more time than the explicit board encodings, it seems that around 16-18 instances, the explicit encodings (EN, EA, ET) grow faster than the implicit board encodings (LN, SN, LT), both in time (Fig. 7.4b) and memory (Fig. 7.4d).

Hex Championship Benchmarks, Separated

Here, we separate all Championship Hex benchmarks in time/memory cactus plots for the SAT instances and for the UNSAT instances. Recall that here we could only execute the 5 implicit goal encodings. Remember (Table 7.5) that ET solves 2 unique instances, the largest SAT instance in all benchmarks and another in the UNSAT instances. Also, LN solves the largest UNSAT instance in all benchmarks uniquely.

The separated results are shown in Figure 7.5. We display UNSAT instances on the left and SAT instances on the right. The top row reports instances solved within a given time limit, and the bottom row reports instances solved within a memory limit.

Focusing on the SAT instances, we see that the explicit board encodings (EN, ET) scale quite nicely in time (Fig. 7.5b). On the other hand, implicit board encodings (LN, LT, SN) perform quite poorly on the Championship SAT instances. So the profile

(a) UNSAT instances (time)

(b) SAT instances (time)

(c) UNSAT instances (memory)

(d) SAT instances (memory)
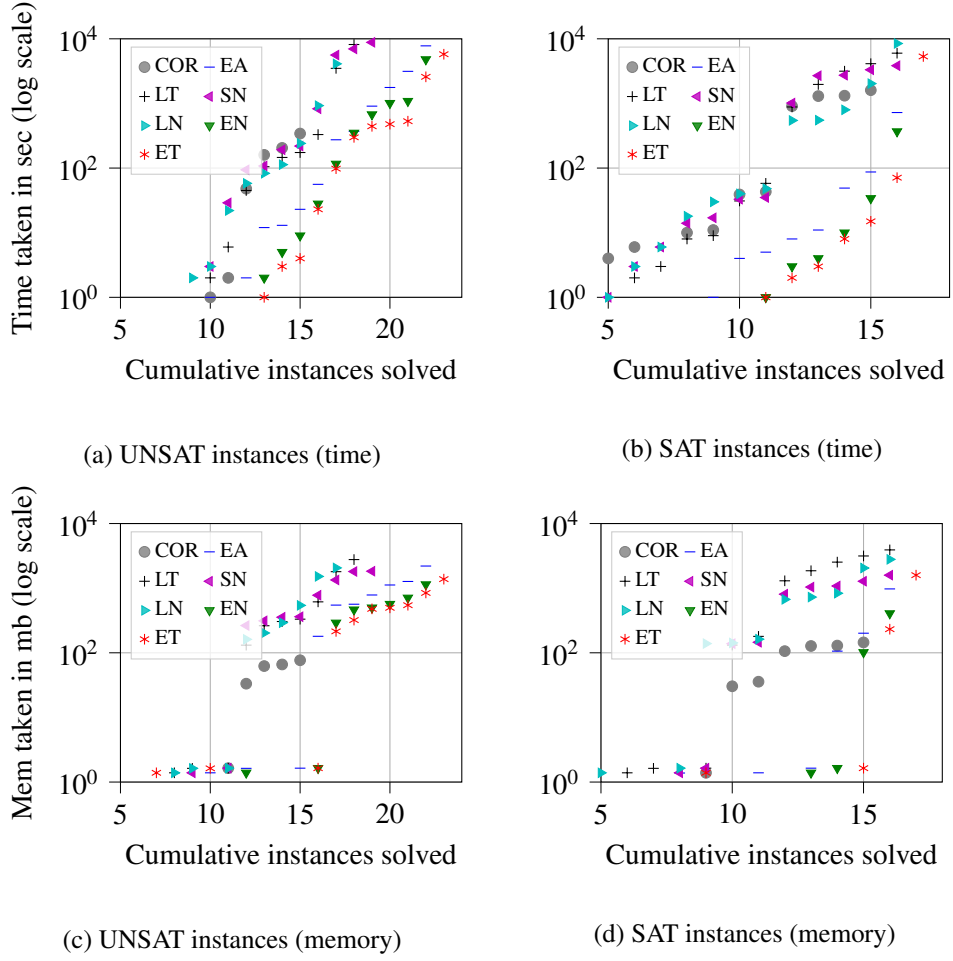
Figure 7.4: Cumulative UNSAT and SAT instances of Hein's Hex (base+hard) solved by our 6 encodings and the original COR encoding within 3 hour timelimit and 8GB memory limit (log scale).

(a) UNSAT instances (time)

(b) SAT instances (time)

(c) UNSAT instances (memory)

(d) SAT instances (memory)
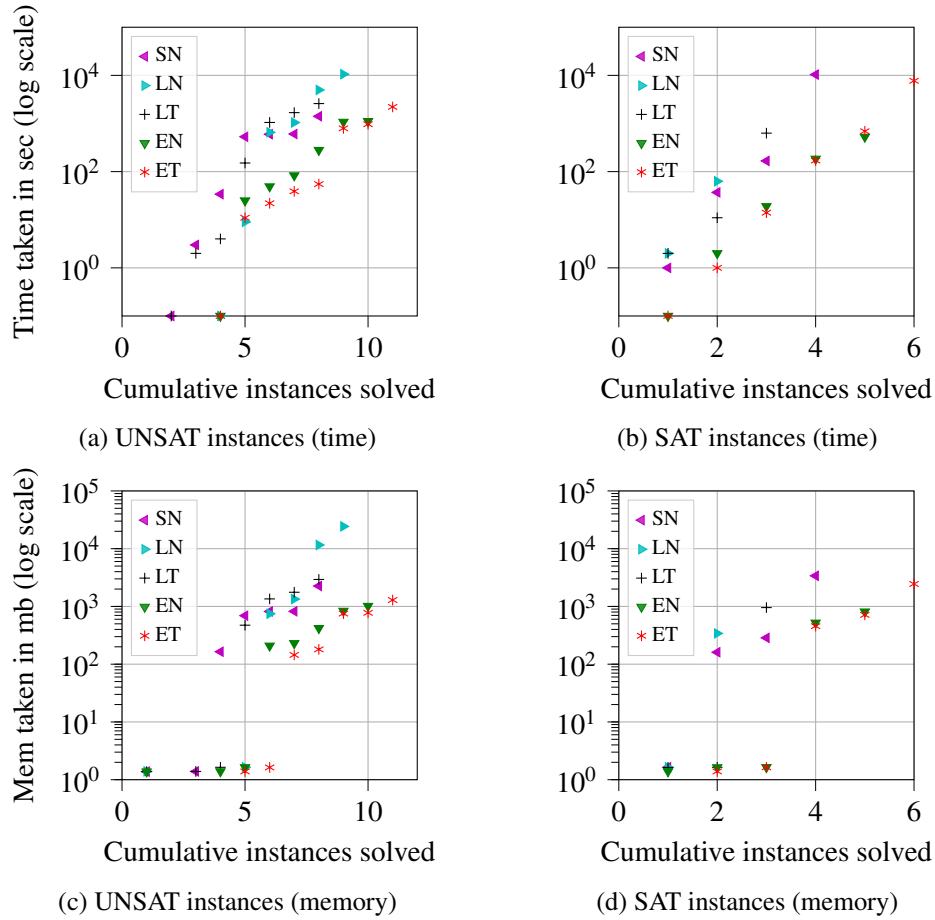
Figure 7.5: Cumulative Hex championship instances solved by 5 implicit goal encodings with 3 hour time limit and 32GB memory limit (log scale).

of the Championship benchmarks (Fig. 7.5b) seems quite different from the profile of the Hex benchmarks (Fig. 7.4b). A possible explanation is that the championship benchmarks represent endgames, in which many cells are filled. The preprocessing technique from §7.3.2 removes those cells, connecting their neighbourhood in a clique. As a consequence, the obtained graphs for the championship benchmarks are much denser than the graphs for the Hex benchmarks.

Chapter 8

Concise QBF Encodings for Games on a Grid

Abstract

Encoding 2-player games in QBF correctly and efficiently is challenging and error-prone. To enable concise specifications and uniform encodings of games played on grid boards, like Tic-Tac-Toe, Connect-4, Domineering, Pursuer-Evader and Breakthrough, we introduce BDDL – *Board-game Domain Definition Language*, inspired by the success of PDDL in the planning domain.

We provide an efficient translation from BDDL into QBF, encoding the existence of a winning strategy of bounded depth. Our lifted encoding treats board positions symbolically and allows concise definitions of conditions, effects and winning configurations, relative to symbolic board positions. The size of the encoding grows linearly in the input model and the considered depth.

To show the feasibility of such a generic approach, we use QBF solvers to compute the critical depths of winning strategies for instances of several known games. For several games, our work provides the first QBF encoding. Unlike plan validation in SAT-based planning, validating QBF-based winning strategies is difficult. We show how to validate winning strategies using QBF certificates and interactive game play.

8.1 Introduction

The existence of a bounded winning strategy for 2-player games can be encoded elegantly with Quantified Boolean Formulas (QBF) [9], where the moves of Player 1 and 2 are encoded using existentially and universally quantified variables, respectively. General QBF solvers have been applied to solve several specific games. For instance, the first QBF encoding for Connect-4 [48] was a response to a challenge posed earlier [130]. Solving the full Connect-4 game this way was not possible. It is a challenge to tune the encoding for pruning the search space. Another example is the simple chess-like game Evader-Pursuer. A first QBF encoding [1] was improved by an encoding that guides the solver to prune the search-space [3]. In recent years, there has been progress

103

in encoding positional games like Tic Tac Toe and Hex. The previous Connect-4 encoding was adapted to positional games [36]. Later, the corrective encoding was proposed [84], which improves pruning by correcting illegal white moves instead of using indicator variables. The encoding was further improved to allow a pairing strategy [20]. A concise, lifted encoding for positional games was recently introduced in [115].

The literature above focused on encoding specific games efficiently, which can be challenging and error-prone. Here we present a uniform translation that can handle a wide range of games, both positional and non-positional games. The first step is to decouple the modelling of games and their encoding in QBF. Inspired by the success of PDDL (Planning Domain Definition Language) [44] and GDL (Game Description Language) [47], we introduce BDDL (Board-game Domain Definition Language) to specify games played on grid-like boards. This allows concise specifications of games like Tic-Tac-Toe, Connect-4, Domineering, Pursuer-Evader and BreakThrough.

We propose an efficient encoding of BDDL into QBF, for the existence of a winning strategy of bounded depth. Copying explicit goal and move constraints for each position can blow up quickly, even for small boards. Instead, we provide the first lifted QBF encoding for *non-positional games*, completely avoiding grounding of concrete board positions. A lifted encoding for classical planning was presented in [111], and an extension to *positional games* was provided in [115]. Here universal symbolic variables were used to specify conditions on moves and goals just once, for a single, isolated symbolic position.

The problem for *non-positional games* is that the conditions and effects for a move on one position can depend on the state of other positions, for instance when moving or taking pieces. We extend the lifted encoding to this case, by using the structure of the grid to encode conditions on the neighborhood of a position symbolically. We use adder, subtractor and comparator circuits to efficiently handle out-of-bounds constraints and illegal moves. We use existential indicator variables with nested constraints, to avoid unnecessary search space over illegal white moves. Finally, we use a single copy of goal configurations, and use universal variables to check the goal at every time step.

To show the feasibility of such a generic approach, we use QBF solvers to compute optimal winning strategies for small instances of several known games. We also compare the results to some existing QBF game encodings. Our work provides the first QBF encoding for several games such as BreakThrough, KnightThrough and Domineering.

Validating results when using QBF is nontrivial. Errors can occur both during encoding and solving. In §8.5.1, we present a framework for validating winning strategies generated by our encoding using QBF certificates and interactive game play. We visualize the state of the board directly from the certificate, which can help with detecting errors.
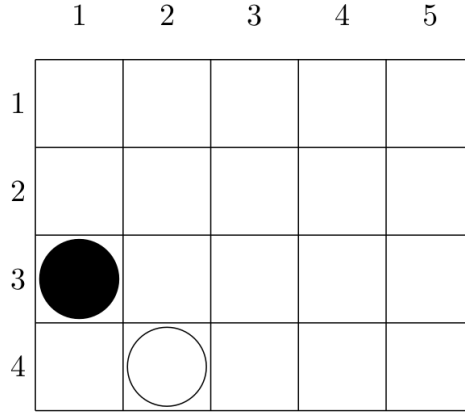
Figure 8.1: HTTT Tic, partially filled 5x4 board

8.2 Preliminaries

In *2-Player, turn-based Games*, players Black (first player) and White try to reach some goal condition. In Maker-Maker games, both players try to achieve their own goal; in Maker-Breaker games, the second player wins by stopping the first player. In this paper, we consider a subset of games that can be played on a *grid*, such as Hex, HTTT, Connect-c, Breakthrough. For example, Fig. 8.1 shows an instance of HTTT Tic, a positional game, where both players try to form a vertical or horizontal line of 3 positions on the grid. We only consider games with one type of piece, but our work could easily be extended to more complex games with different pieces, like Chess.

The *Planning Domain Definition Language* (PDDL) [85] is a standard domain specification language for classical planning problems used in International Planning Competitions (IPC). A domain file specifies predicates and actions whereas a problem file specifies objects, initial state and goal condition. PDDL is an action-centered language; the actions essentially specify how the world changes by preconditions and effects. Actions conditions and effects are represented similar to First-Order-Logic, using parameters to achieve compact descriptions.

Quantified Boolean Formulas (QBF) [9] extend propositional logic with Boolean quantifiers. We consider closed QBF formulas in prenex normal form, i.e., $Q_1 x_1 \cdots Q_n x_n(\Phi)$, where Φ is a propositional formula with Boolean variables in $\{x_1, \ldots, x_n\}$ and each $Q_i \in \{\forall, \exists\}$. Every such formula evaluates to true or false. QBF evaluation is a standard PSPACE-complete problem. It is well known that the complexity increases with the number of quantifier alternations. Several QBF solvers exist, which operate on QBF in either QDIMACS format, where Φ is essentially a set of CNF clauses, or in QCIR format [67], where Φ is provided as a circuit with and- and or-gates and negation. In §8.4, we present our encoding of grid-games in QCIR, which can be transformed to QDIMACS using the Tseitin transformation [125], introducing one existential Boolean variable per gate.

QBF Certificates, proposed by [7], are resolution-proofs or Skolem/Herbrand functions which are used for validating the QBF solver result. A Skolem function for a True instance is essentially a function mapping from universal to existential variables. These certificates can be used to extract winning strategies for 2-player games. Existing tools for QBF certificate generation include sKizzo, using symbolic solving [7], QBFcert, based on resolution proofs [89], and FERPModels, which is expansion based [13]. In §8.5.1, we extract and validate winning strategies using certificate extraction with QBFcert and interactive game play. Our main purpose is to validate our QBF encoding, rather than the correctness of the QBF solver.

8.3   Board-game Domain Definition Language (BDDL)

8.3.1   Syntax of BDDL

Listings 8.1, 8.2, 8.3, specify the grammar for BDDL conditions, domain files, and problem files.

Listing 8.1: Grammar for BDDL conditions relative to board position $(?x, ?y)$. We will use `nl` for newline, `int` for integers and `str` for ASCII strings of letters, digits and underscores. We use | for choice, * for 0 or more repetitions. All other symbols are non-terminals or literals.

```
1       condition ::= (sub-cond*) nl
2       sub-cond  ::= pred(e1,e2) | NOT(pred(e1,e2))
3       pred      ::= open | white | black
4       e1        ::= ?x + int | ?x - int | ?x | int | xmin | xmax
5       e2        ::= ?y + int | ?y - int | ?y | int | ymin | ymax
```

We follow the separation of predicates and actions from PDDL. For our purpose, the predicates are fixed to {black, white, open}. Contrary to PDDL, we allow structured expressions in position conditions (see (`e1,e2`) in Listing 8.1). This allows us to specify multiple positions in the grid, relative to an arbitrary board position $(?x, ?y)$.

Definition 22. *A condition is a sequence of sub-conditions $p(e_1, e_2)$ or $\neg p(e_1, e_2)$, where the predicate p and relative coordinates (e_1, e_2) are defined according to the grammar in Listing 8.1. We write $\mathscr{C}$ for the set of all conditions. A condition will be interpreted as a conjunction.*

Listing 8.2: Grammar for BDDL domain files.

```
1       domain ::= #blackactions nl action*
2              #whiteactions nl action*
3       action ::= :action str nl
4              :parameters (?x,?y) nl
5              :precondition condition
6              :effect condition
```

The domain file (Listing 8.2) specifies the set of actions that black and white players can play. Each action is defined uniformly over the board positions by using fixed parameters $(?x, ?y)$ as coordinates. Each action is specified by its precondition

and effect, both consisting of a conjunction of positive or negative single position conditions.

In the domain file, absolute int-indices as in black$(1,2)$ *are not allowed* (since the board size is unknown), but we allow references to the minimal and maximal (x,y)-positions. Listing 8.4 shows an example domain for positional games. §8.3.2 will illustrate more games.

Definition 23. *Each domain file according to the grammar in Listing 8.2 defines a* Game Domain $(A_b, A_w, \text{pre}, \text{eff})$, *where*

- A_b *and* A_w *are the set of black, resp. white, action symbols specified*

- $\text{pre} : A_b \cup A_w \to \mathscr{C}$ *specifies the precondition of each action*

- $\text{eff} : A_b \cup A_w \to \mathscr{C}$ *specifies the effect of each action*

For each action $a \in A_b \cup A_w$, *we write* $\text{all}(a) := \text{pre}(a) \cup \text{eff}(a)$ *for all its conditions.*

Listing 8.3: Grammar for BDDL problem files.

```
1      problem ::= size init depth goals
2      size  ::= #boardsize nl int int nl
3      init  ::= #init nl (pred(int,int)*) nl
4      depth ::= #depth nl int nl
5      goals ::= #blackgoals nl condition*
6             #whitegoals nl condition*
```

In the problem file, we specify the rectangular board size $m \times n$, the initial state, the winning conditions for the black and white player, and we also specify the considered depth d of the game.[1] The initial state is specified by a list of absolute black and white positions. This list describes a single state, i.e., each position on the board is either black, or white, or open (in case it is not listed in #init). Black and White can have multiple alternative winning conditions, each of which is described by condition on positions. Here we allow both absolute positions (for specific board positions) and relative positions (to describe winning patterns uniformly over all board positions).

Definition 24. *Each problem file according to the grammar in Listing 8.3 defines a* Game Instance (m, n, I, G_b, G_w, d), *where*

- (m, n) *denotes the board size.*

- $I \subseteq \{p(i,j) \mid p \in \{\text{black}, \text{white}\}$ *and* $i, j \in \mathbb{N}\} \subseteq \mathscr{C}$ *specifies the initial state.*

---

[1]We make d part of the problem file for convenience, since we will only consider bounded plays consisting of d moves, but one could study unbounded plays as well; in both cases, the state space will be finite.

- G$_b$ ⊆ 𝒞 *and* G$_w$ ⊆ 𝒞 *specify the winning conditions of the black, respectively, white player.*

- *d is an odd number, specifying the considered depth of the play.*

For example, Listing 8.5 specifies a 5×4 board, which also determines xmin = ymin = 1, xmax = 5, ymax = 4. It specifies two initial board positions: $(1,3)$ is black and $(2,4)$ is white, whereas all other positions are open. This problem file corresponds to the Tic instance in Figure 8.1. The problem file also specifies that we consider a winning strategy of depth 5.

Finally, the same file specifies two possible winning configuration patterns for black, and two for white. For instance, white wins as soon as there is a horizontal white line of length 3 *that fits entirely on the board*. This is specified as `white(?x,?y)`, `white(?x+1,?y)`, `white(?x+2,?y)`. Note that the implicit boundary conditions on ?*x* and ?*y* will be inferred automatically, for ease of specification.

8.3.2   Examples: Modelling Some Classical Games

Positional Games   In a positional game, a player can only occupy an open position. In the domain file, we list black and white actions i.e., a single *occupy* (Listing 8.4). We can define goal conditions for games like HTTT and Gomoku implicitly, where we list shapes as disjunction of conjunctions in reference to some existential position. For example, Listing 8.5 is a problem input for the HTTT Tic (this corresponds to Fig. 8.1). Here the board size is 5x4 and the goal is to form a line of 3 positions either horizontally or vertically. We can also encode complex goal conditions for games like Hex by listing explicit winning sets of indices. To encode maker-breaker versions of the games, one can simply drop white goal configurations.

Listing 8.4: Positional games' domain

```
1    #blackactions
2    :action occupy
3    :parameters (?x,?y)
4    :precondition (open(?x,?y))
5    :effect (black(?x,?y))
6    #whiteactions
7    :action occupy
8    :parameters (?x,?y)
9    :precondition (open(?x,?y))
10   :effect (white(?x,?y))
```

Listing 8.5: HTTT Tic problem instance

```
1    #boardsize 5 4
2    #init
3    (black(1,3)white(2,4))
4    #depth      5
5    #blackgoals
```

```
6        (black(?x,?y)black(?x,?y+1)black(?x,?y+2))
7        (black(?x,?y)black(?x+1,?y)black(?x+2,?y))
8        #whitegoals
9        (white(?x,?y)white(?x,?y+1)white(?x,?y+2))
10       (white(?x,?y)white(?x+1,?y)white(?x+2,?y))
```

Listing 8.6: Connect4: Black

```
1        :action occupyOnTop
2        :parameters (?x,?y)
3        :precondition (open(?x,?y) NOT(open(?x,?y+1)))
4        :effect (black(?x,?y))
5        :action occupyBottom
6        :parameters (?x,?y)
7        :precondition (open(?x,ymax))
8        :effect (black(?x,ymax))
```

Connect-*c* is similar to a positional game, except one can only occupy an open position if the position below it is already occupied. We can encode this using two actions (Listing 8.6), *occupyOnTop* and *occupyBottom*. The last one handles the special case when choosing a position on the bottom row. Here *ymax* represents the 'bottom' row. The actions for white would be similar. The goal conditions are similar to the Tic problem (Listing 8.5): in Connect-*c* we would specify a line of *c* positions by its 4 symmetric variants (vertical, horizontal and diagonals). Connect4 on a 6×7 board is a popular instance.

Listing 8.7: Breakthrough (snippets)

```
1        #blackactions
2        :action north-east
3        :parameters (?x,?y)
4        :precondition (black(?x,?y) NOT(black(?x+1,?y-1)))
5        :effect (open(?x,?y) black(?x+1,?y-1))
6        #blackgoals
7        (black(?x,ymin))
8        #whitegoals
9        (white(?x,ymax))
```

Breakthrough is a non-positional, chess-like game, played with pawns only. In the Initial board, Black starts with two bottom rows of pawns on ymax, ymax -1, whereas White starts with two top rows of pawns ymin, ymin $+1$. A player can move one step forward or diagonally. On the diagonal steps, it can capture the pieces of the opponent. The players can only move towards the opposite side of the board. A player wins if any of its pawns reaches that side. Model 8.7 shows one of the black actions (going north-east with or without capturing), and the goal conditions.

Evader and Pursuer   In Evader and Pursuer, each player has a single piece starting at different positions of the board. Black player tries to evade (cannot capture) white

player to reach a specific position whereas white player tries to capture black player. In our model 8.8, Black player can move up to 2 steps vertically or horizontally but only 1 step diagonally. White player can only move 1 step vertically or horizontally. In goal conditions (see 8.9), we simply specify the target position. Black player wins if it reaches the target position whereas white player wins if it stops black player or reaches the goal first.

Listing 8.8: Evader and Pursuer snippet: one of the actions

```
1    #blackactions
2    :action down-two
3    :parameters (?x,?y)
4    :precondition (black(?x,?y) open(?x,?y+1) open(?x,?y+2))
5    :effect (open(?x,?y)black(?x,?y+2))
```

Listing 8.9: Goal spec for EP

```
1    #blackgoals
2    (black(xmin,ymin))
3    #whitegoals
4    (white(xmin,ymin))
```

Listing 8.10: Domineering (Black only)

```
1    #blackactions
2    :action vertical
3    :parameters (?x, ?y)
4    :precondition (open(?x,?y) open(?x,?y+1))
5    :effect (black(?x,?y) black(?x,?y+1))
6    #blackgoals
7    #whitegoals
```

In Domineering, the initial board is empty and players take turns to place dominoes. Black places a domino vertically, covering 2 open positions (illustrated in Listing 8.10, top), whereas White places dominoes horizontally. A player wins if the opponent cannot make a move in its turn. This is the default, so we can just omit the goal conditions (cf. Listing 8.10, bottom). Note that the empty disjunction corresponds to False.

While being considerably less general than the Game Description Language (GDL) [47], one can model many other games in BDDL, e.g., Hex, KnightThrough, CrossPurpose and Konane, etc. We have chosen a limited setting, in order to provide a uniform encoding in QBF. To this end, we will now first provide the formal semantics of BDDL.

8.3.3 Semantics

From now on, we assume a fixed Game Domain $(A_b, A_w, \text{pre}, \text{eff})$ and Game Instance (m, n, I, G_b, G_w, d), as introduced in Definitions 23 and 24.

We first define the set of indices Ind and the implicit bounds induced by using relative positions in the conditions. For example, the bounds for the second winning

condition of white in Listing 8.1 is $[1,3] \times [1,4]$: If $?x \in [1,3]$ and $?y \in [1,4]$, then the positions mentioned in the condition, $(?x,?y)$, $(?x+1,?y)$, and $(?x+2,?y)$, refer to actual positions on a 5x4 board. We assume that all absolute positions in the problem file refer to positions in Ind.

Definition 25. *The set of valid indices is defined by* $\mathrm{Ind} = [1..m] \times [1..n]$. *Given a sub-condition* $c = p(e_1, e_2)$ *or its negation, the* implicit bounds $\mathrm{BD}(c)$ *are defined as the sub-interval* $[1+\ell_x, m-u_x] \times [1+\ell_y, 1-u_y] \subseteq \mathrm{Ind}$, *where* $\ell_x = a$ *if* $e_1 = ?x - a$ *and* $\ell_x = 0$, *otherwise. Similarly,* $u_x = a$ *if* $e_1 = ?x + a$, $\ell_y = a$ *if* $e_2 = ?y - a$, $u_y = a$ *if* $e_2 = ?y + a$; *otherwise these quantities are 0.*

Given a condition C, its implicit bounds $\mathrm{BD}(C)$ *are defined as the intersection* $\bigcap_{c \in C} \mathrm{BD}(c)$.

We now define the 2-player game graph induced by a domain and problem file in BDDL. The nodes of such a graph represent the state space of the game, i.e., a game position corresponds to a possible placement of black and white pieces on the board:

Definition 26. *We define the set of states* $S = \mathrm{Ind} \to \{\text{black}, \text{white}, \text{open}\}$. *We assume that each* (i, j) *occurs at most once in the init-specification I. Then the initial state* s_0 *is defined such that for all* $(i, j) \in \mathrm{Ind}$, $s_0(i, j) = \text{black}$ *if* $\text{black}(i, j) \in I$, $s_0(i, j) = \text{white}$ *if* $\text{white}(i, j) \in I$, $s_0(i, j) = \text{open}$, *otherwise.*

We can now define when a condition holds in a given game position. For instance, $\text{open}(?x, ?y)\,\text{black}(?x-1, y)\,\text{white}(?x, y+1)$ holds at tile (2,3) in the state depicted by Fig. 8.1, because $\text{open}(2,3)$, $\text{black}(1,3)$ and $\text{white}(2,4)$ hold (fill in (2,3) for $(?x, ?y)$).

Definition 27. *Given a state* $s \in S$, *a sub-condition* $c = p(e_1, e_2)$, *and a concrete position* $(i, j) \in \mathrm{BD}(c)$, *we write* $e_1[i] \in [1, m]$ *for the value of* e_1 *after substituting* $?x/i$, $\text{xmin}/1$ *and* xmax/m. *Similar for* $e_2[j] \in [1, n]$. *Now we define* $s, (i, j) \vDash c$ *if* $s(e_1[i], e_2[j]) = p$. *For a condition C, and* $(i, j) \in \mathrm{BD}(C)$, *we define* $s, (i, j) \vDash C$ *if* $s, (i, j) \vDash c$ *for all* $c \in C$. *That is, the conjunction of sub-conditions hold relative to a single fixed position* (i, j).

We now define the transitions in the game graph. A player can move from state s_1 to s_2 if it has an action whose preconditions hold in s_1 at some valid position (i, j) and whose effects hold in s_2 at the same position.

Definition 28. *Given the set of states S and set of actions* A, *define for each* $a \in A$ *the interval* $I_a = \mathrm{BD}(\text{pre}(a)) \cap \mathrm{BD}(\text{eff}(a))$. *We define the transition function* $T(A) \subseteq S \times S$ *by* $(s_1, s_2) \in T(A)$ *iff there exists* $a \in A$ *and* $(i, j) \in I_a$ *such that* $s_1, (i, j) \vDash \text{pre}(a)$ *and* $s_2, (i, j) \vDash \text{eff}(a)$. *Black's moves are* $T(A_b)$ *and White's moves are* $T(A_w)$.

A state is won by a player, if one of its goal conditions holds at some index (i, j).

Definition 29. *Given a state $s \in S$ and a set of conditions D, we write $s \vDash D$ if there exists a condition $C \in D$, and a position $(i, j) \in BD(C)$, such that $s, (i, j) \vDash C$. We define $s \in win_B$ if $s \vDash G_b$ (won by black) and $s \in win_W$ if $s \vDash G_w$.*

In order to simplify the definition of winning strategy (and make our QBF encoding a bit more efficient), we will assume the following sanity conditions on the game. These conditions hold naturally for all the games that we considered. The sanity conditions ensure that the winning configuration of a player can only become satisfied by playing one of its own moves.

Definition 30. *The 2-player game graph $(S, s_0, T_b, T_w, win_b, win_w)$ satisfies the* Sanity Conditions *if:*

- $s_0 \notin win_b$ *and* $s_0 \notin win_w$ *(the initial state is not winning for any of the players)*

- *For all $s_1, s_2 \in S$, if $T_b(s_1, s_2)$ and $s_2 \in win_w$, then $s_1 \in win_w$.*

- *For all $s_1, s_2 \in S$, if $T_w(s_1, s_2)$ and $s_2 \in win_b$, then $s_1 \in win_b$.*

We are finally in the position to define that Black has a winning strategy of at most d moves. Here we assume that the players alternate moves, and the first and last move is by Black, so d is always odd. A player who cannot move in its turn has lost. We define the sets $\mathscr{W}_k$ of states where Black can win in at most k steps, by induction over k.

Definition 31. *Given a 2-player game graph $(S, s_0, T_b, T_w, win_b, win_w)$ that satisfies the sanity conditions, we define the sets $\mathscr{W}_k$, for odd k, as follows:*

- $s \in \mathscr{W}_1$ *if there exists a state s' such that $T_b(s, s')$ and $s' \in win_b$.*

- $s \in \mathscr{W}_{k+2}$ *if there exists a state s' with $T_b(s, s')$, such that either $s' \in win_b$, or for all states s'' with $T_w(s', s'')$, $s'' \notin win_w$ and $s'' \in \mathscr{W}_k$.*

We say that Black has a winning strategy of depth d, if the initial state $s_0 \in \mathscr{W}_d$.

8.4   QBF encoding

8.4.1 Prefix

We now present a uniform encoding in QBF of "there exists a winning strategy for Black of at most d moves", given a fixed game domain and problem instance in BDDL. Following the descriptions of variables from Table 8.1, we first introduce variables for black and white moves. Black's moves are existential whereas White's moves are universal. We represent a move by using action A and index variables X, Y at each time step. To handle games shorter than d steps, we introduce game-stop variables (gs).

Since a white move is encoded with universal variables, some encoded moves may not satisfy the required constraints. To remember if a played white move is valid, we use existential indicator variables lb and P. Here P will mean that the preconditions for White's move A on (X, Y) hold and lb will mean that A, X and Y stay within legal bounds. Note that, in each specific universal branch of a white move the indicator variables are completely determined by the chosen move and the state of the board.

$$\underbrace{\exists A^1, X^1, Y^1, gs^1}_{\text{Black move 1}} \quad \underbrace{\forall A^2, X^2, Y^2, gs^2}_{\text{White move 2}} \quad \underbrace{\exists lb^2, P^2}_{\text{White Indicators 2}} \quad \cdots \quad \underbrace{\exists A^d, X^d, Y^d, gs^d}_{\text{Black move d}}$$

Next, we introduce the variables to check for black and white goals. For Black, we need to show that there exists a position B_x, B_y on the board that satisfies one of Black's goal conditions, $G_b[B_c]$. For White, we need to show absence of the white goal condition, i.e., for every board position W_x, W_y and for each white goal condition $G_w[W_c]$, we need to show there exists a sub-condition $G_w[W_c][W_{ce}]$ that is violated (i.e., a counter example).

Finally, we introduce variables to check the validity of all the moves. Universal variables P_x, P_y represent a symbolic position on the board. These are used to check the preconditions and effects on each board position in a uniform manner. At each time step, the state variables o, w represent if the symbolic position (P_x, P_y) is open and/or white, respectively; so black corresponds to $o \wedge \neg w$. The symbolic and state variables together represent a full state of the board at any time step.

Variable	Description
A^i	$\lceil \log(|A_b|) \rceil$ / $\lceil \log(|A_w|) \rceil$ vars for B's/W's action if i is odd/even
gs^i	game-stop variable at time step i
X^i, Y^i	$\lceil \log(m+1) \rceil, \lceil \log(n+1) \rceil$ variables for the action parameters
lb^i	white legal bound indicator variable at time step i
P^i	max #W-preconditions indicator variables for White preconditions at time i
$B_x, B_y/W_x, W_y$	$\lceil \log(m+1) \rceil, \lceil \log(n+1) \rceil$ variables for Black's/White's goal position
B_c/W_c	$\lceil \log(|G_b|) \rceil / \lceil \log(|G_w|) \rceil$ variables for Black's/White's goal index
W_{ce}	$\lceil \log(\text{max \#W-goal subconditions}) \rceil$, counter-example to White's goal W_c
P_x, P_y	$\lceil \log(m+1) \rceil, \lceil \log(n+1) \rceil$ variables indicating a symbolic board position
o^i, w^i	two variables indicating the state at the symbolic board position at time i

Table 8.1: Encoding variables and descriptions

$$\underbrace{\exists B_x, B_y, B_c}_{\text{Black goal}} \quad \underbrace{\forall W_x, W_y, W_c \exists W_{ce}}_{\text{White goal}} \quad \underbrace{\forall P_x, P_y}_{\text{symbolic pos}} \quad \underbrace{\exists o^1, w^1, \ldots, o^{d+1}, w^{d+1}}_{\text{state variables}}$$

8.4.2 Matrix

The matrix of our QBF encoding consists of an initial constraint and a constraint on moves.

$$\mathscr{I} \wedge B^1$$

$\mathscr{I}$ (initial state), M_b^i, M_w^i (valid moves), $\mathscr{G}_b^i$, and $\mathscr{G}_w^i$ (goal checks) will be introduced in subsequent sections. We now recursively define the turn based player constraints as follows:

> For $i = 1, 3, \ldots, d-2$:
> $B^i := M_b^i \wedge (\text{gs}^i \implies \mathscr{G}_b^{i+1}) \wedge (\neg \text{gs}^i \implies W^{i+1})$
> For $i = 2, 4, \ldots, d-1$:
> $W^i := M_w^i \wedge ((\text{lb}^i \wedge \bigwedge_{p \in P^i} p) \implies ((\text{gs}^i \implies \mathscr{G}_w^{i+1}) \wedge (\neg \text{gs}^i \implies B^{i+1})))$
> $B^d := M_b^d \wedge \mathscr{G}_b^{d+1}$

- In Black's turn (odd i), the black move must be valid (M_b^i). If the game is stopped (gs), the black goal is enforced ($\mathscr{G}_b^{i+1}$), else we continue with White's turn W^{i+1}.

- In White's turn (even i), the white move constraints must hold (M_w^i). M_w^i also specifies the indicator variables lb^i and P^i. Note that we only care about valid moves (if White can only play invalid, Black has won). If the game is stopped, the white goal is enforced ($\mathscr{G}_w^{i+1}$), else we continue with Black's turn (B^{i+1}).

- At time step d, a valid black move should lead to the black goal condition in time $d+1$.

8.4.3 Defining Initial, Move and Goal Circuits

Auxiliary Circuits In our QBF encoding, we generate three auxiliary constraints:

- Compute relative symbolic index for a sub-condition (RI).

- Compute state constraints for a sub-condition (SC).

- Compute bound constraints for a condition (BDCIR).

The key idea of the relative symbolic index is to test if the parameters $(?x, ?y)$ that occur in some sub-condition $p(e1, e2)$ refer to the current value of the symbolic position variables (S_x, S_y). If so, the current state (o, w) will be forced to the value indicated by p. For the conditions, also bound constraints are generated to avoid out-of-bound references.

Before we can define these constraints more precisely, we first need to shortly explain how we encoded some arithmetic sub-circuits, in particular ADD, SUB, ADDSUB, LT and EQ.

In sub-conditions, we allow addition and subtraction with a constant value. Given a sequence of Boolean variables V and an integer k, we generate an adder circuit $\text{ADD}(V, k)$ with V as input gates and a sequence of output gates for the binary representation of $V + k$. For subtracting, we apply an adder circuit using 2's complement i.e., $\text{SUB}(V, k) := \text{ADD}(V, k')$, where k' is a 2's-complement of k. Given an expression $E = (e1, e2)$, we use an *Adder-Subtractor* generator $\text{ADDSUB}(V, e1)$ to generate index constraints on $?x$ and $\text{ADDSUB}(V, e2)$ to generate index constraints on $?y$. The generator $\text{ADDSUB}(V, e)$ returns the sub-circuit:

- $\text{ADD}(V, k)$ if e is $?x+k$ or $?y+k$; $\text{SUB}(V, k)$ if e is $?x-k$ or $?y-k$;

- V, if e is $?x$ or $?y$; and e, if e is an integer.

The equality generator $\text{EQ}(p, p')$ generates (1) an equality circuit with p and p' as inputs if both are sequence of variables; (2) a single And gate if either p or p' is an integer. In both cases there is only one output gate which is true iff the inputs are equal.

The less-than comparator circuit $\text{LT}(V, k)$ for integer k takes as input a sequence of Boolean variables V and has a single output gate which is true iff the binary input is less than k. Lower bounds (greater-than-or-equal) can be achieved using the negation of less-than.

We can now define the circuits for the relative index, the sub-condition constraint, and the bound constraints.

Definition 32. *Given a sub-condition $p(e1, e2)$, sequences of variables V_x, V_y representing a position on the board. We define a constraint generator,* Relative Index

$$\text{RI}(V_x, V_y, p(e1, e2)) := \text{EQ}(\text{ADDSUB}(V_x, e1), P_x) \wedge \text{EQ}(\text{ADDSUB}(V_y, e2), P_y)$$

Definition 33. *Given a sub-condition $p(e_1, e_2)$ and state variables* o, w, *we define sub-condition constraint*

$$\text{SC}(p(e_1, e_2), o, w) := \neg o \land \neg w, \textit{if } p = \text{black}; \ \neg o \land w, \textit{if } p = \text{white}; \ o, \textit{if } p = \text{open}$$

We negate the sub-condition constraint if the sub-condition $p(e1, e2)$ is negated.

Definition 34. *Given a condition C, and sequences of variables V_x, V_y representing a position. Assume* $\text{BD}(C) = [l_x, u_x] \times [l_y, u_y]$. *We define a bound generator:*

$$\text{BDCIR}(V_x, V_y, C) := \neg \text{LT}(V_x, l_x) \land \text{LT}(V_x, u_x + 1) \land \neg \text{LT}(V_y, l_y) \land \text{LT}(V_y, u_y + 1)$$

The output gate of circuit is true all upper and lower bounds hold.

For example, consider a sub-condition $\text{open}(?x, ?y + 1)$ in precondition from Evader-Pursuer (see Listing 8.8). In our encoding at time 1 we generate a corresponding constraint:

$$\text{RI}(X^1, Y^1, \text{open}(?x, ?y + 1)) \implies \text{SC}(\text{open}(?x, ?y + 1), o^1, w^1)$$

This is equivalent to generating the circuit

$$\text{EQ}(X^1, P_x) \land \text{EQ}(\text{ADD}(Y^1, 1), P_y) \implies o^1$$

This circuit is true if the open predicate in the relative symbolic branch $(?x, ?y + 1)$ is true. To give a concrete example, if the first black move is played on position $(X^1, Y^1) = (1, 1)$ then in the symbolic branch $(P_x, P_y) = (1, 2)$ the open predicate is implied.

For the same action, we enforce the bound constraints using $\text{BDCIR}(X^1, Y^1, \text{all}(\text{down-two}))$ which is equivalent to generating the circuit

$$\neg \text{LT}(X^1, \text{xmin}) \land \text{LT}(X^1, \text{xmax} + 1) \land \neg \text{LT}(Y^1, \text{ymin}) \land \text{LT}(Y^1, \text{ymax} - 1)$$

Every position where *y* is less than ymax $- 1$ is a legal position for down-two action.

Initial State $\mathscr{I}$ Given the initial specification I (as in Def. 24), we specify that the state (o^1, w^1) at time step 1 has the proper value for symbolic position (P_x, P_y).

$$
\begin{aligned}
\mathscr{I} := ((\bigvee_{\text{black}(e1,e2) \in I} \text{EQ}(e1, P_x) \land \text{EQ}(e2, P_y)) &\implies \neg o^1 \land \neg w^1) \\
\land ((\bigvee_{\text{white}(e1,e2) \in I} \text{EQ}(e1, P_x) \land \text{EQ}(e2, P_y)) &\implies \neg o^1 \land w^1) \\
\land (\bigwedge_{p(e1,e2) \notin I} \text{EQ}(e1, P_x) \land \text{EQ}(e2, P_y) &\implies o^1)
\end{aligned}
$$

Black Move M_b^i The following constraints specify that the black action $A^i(X^i, Y^i)$ at time step i is a legal move, with respect to the state transition $(o^i, w^i) \to (o^{i+1}, w^{i+1})$ of the current symbolic (P_x, P_y)-branch.

$$
\begin{aligned}
M_b^i := \ &\mathrm{LT}(A^i, |A_b|) \wedge \bigwedge_{j=1}^{|A_b|} \big(\mathrm{EQ}(A^i, j) \implies && \text{index bound on conditions} \\
&\big(\mathrm{BDCIR}(X^i, Y^i, \mathrm{all}(A_b[j])) \quad \wedge && \text{index bound subconditions} \\
&\bigwedge_{C \in \mathrm{pre}(A_b[j])} \mathrm{RI}(X^i, Y^i, C) \implies \mathrm{SC}(C, o^i, w^i) \quad \wedge\ \text{preconditions hold at } i \\
&\bigwedge_{C \in \mathrm{eff}(A_b[j])} \mathrm{RI}(X^i, Y^i, C) \implies \mathrm{SC}(C, o^{i+1}, w^{i+1}) \quad \wedge\ \text{effects hold at } i+1 \\
&\big(\neg \bigvee_{C \in \mathrm{eff}(A_b[j])} \mathrm{RI}(X^i, Y^i, C) \big) \implies && \text{unchanged positions} \\
&\quad (\mathrm{EQ}(o^i, o^{i+1}) \wedge \mathrm{EQ}(w^i, w^{i+1}))\big)\big) && \text{are propagated}
\end{aligned}
$$

White Move M_w^i For white moves at time step i, we first specify that the legal bound variable is true iff for each action the bound constraints hold. For each action, we also set the precondition flags P^i to true iff the preconditions hold[2]. Since the preconditions refer to positions in different symbolic branches, we set the flags after each move so that the values are available in all symbolic branches. The effects should hold at time step $i+1$ for each action, if the bounds are legal and preconditions are true. The positions that are not changed by the legal move are propagated.

$$
\begin{aligned}
M_w^i := \ &\big(\big(\bigwedge_{j=1}^{|A_w|} \mathrm{EQ}(A^i, j) \implies \mathrm{BDCIR}(X^i, Y^i, \mathrm{all}(A_w[j])) \wedge \mathrm{LT}(A^i, |A_w|)\big) \iff \mathrm{lb}^i \big) \wedge \\
&\big(\bigwedge_{j=1}^{|A_w|} \mathrm{EQ}(A^i, j) \implies \\
&\bigwedge_{k=1}^{|P|} \mathrm{RI}(X^i, Y^i, \mathrm{pre}(A_w[j])[k]) \implies (\mathrm{SC}(\mathrm{pre}(A_w[j])[k], o^i, w^i) \iff P^i[k]) \wedge \\
&\big(\mathrm{lb}^i \wedge \bigwedge_{p \in P^i} p \implies \bigwedge_{C \in \mathrm{eff}(A_w[j])} \mathrm{RI}(X^i, Y^i, C) \implies \mathrm{SC}(C, o^{i+1}, w^{i+1}) \quad \wedge \\
&\big(\neg \bigvee_{C \in \mathrm{eff}(A_w[j])} \mathrm{RI}(X^i, Y^i, C) \big) \implies (o^i \iff o^{i+1}) \wedge (w^i \iff w^{i+1})\big)\big)
\end{aligned}
$$

Black Goal $\mathscr{G}_b^i$ We check if Black meets its B_c's goal condition at position (B_x, B_y) at time i. Since goal conditions are encoded in binary, we need to restrict B_c within boundaries. To check the goal at time step i, we first propagate the state variables to

---

[2]For simplicity of the presentation, we assume that all white actions have the same number of preconditions.

last time step $d + 1$. Next, we check the goal condition at $d + 1$. This avoids copies of black goal per time step.

$$\mathcal{G}_b^i := \mathrm{LT}(\mathrm{B}_c, |\, \mathrm{G}_b\, |) \wedge \mathrm{EQ}(\mathrm{o}^i, \mathrm{o}^{d+1}) \wedge \mathrm{EQ}(\mathrm{w}^i, \mathrm{w}^{d+1}) \quad \wedge$$

$$\bigwedge_{j=1}^{|\mathrm{G}_b|} \big(\mathrm{EQ}(\mathrm{B}_c, j) \implies \mathrm{BDCIR}(\mathrm{B}_x, \mathrm{B}_y, \mathrm{G}_b[j]) \quad \wedge$$

$$\bigwedge_{C \in \mathrm{G}_b[j]} \big(\mathrm{RI}(\mathrm{B}_x, \mathrm{B}_y, C) \implies \mathrm{SC}(C, \mathrm{o}^{d+1}, \mathrm{w}^{d+1}) \big) \big)$$

White Goal $\mathcal{G}_w^i$ For White's goal, we check that W_{ce} is a counter-example to White's goal condition W_c at position $(\mathrm{W}_x, \mathrm{W}_y)$ at time step i. As for the black goal, we first propagate the state variables to the last time step $d + 1$. White's goal is violated if $(\mathrm{W}_x, \mathrm{W}_y)$ are out-of-bounds, or if W_{ce} is a legal index and the corresponding sub-constraint in W_c is violated.

$$\mathcal{G}_w^i := \mathrm{EQ}(\mathrm{o}^i, \mathrm{o}^{d+1}) \wedge \mathrm{EQ}(\mathrm{w}^i, \mathrm{w}^{d+1}) \quad \wedge$$

$$\bigwedge_{j=1}^{|\mathrm{G}_w|} \big(\mathrm{EQ}(\mathrm{W}_c, j) \wedge \mathrm{BDCIR}(\mathrm{W}_x, \mathrm{W}_y, \mathrm{G}_w[j]) \implies \big(\mathrm{LT}(\mathrm{W}_{ce}, |\, \mathrm{G}_w[j]\, |) \quad \wedge$$

$$\bigwedge_{k=1}^{|\mathrm{G}_w[j]|} \big(\mathrm{RI}(\mathrm{W}_x, \mathrm{W}_y, \mathrm{G}_w[j][k]) \wedge \mathrm{EQ}(\mathrm{W}_{ce}, k) \implies \neg\mathrm{SC}(\mathrm{G}_w[j][k], \mathrm{o}^{d+1}, \mathrm{w}^{d+1}) \big) \big) \big)$$

8.5   Implementation and Evaluation

We provide an open source implementation for the BDDL to QBF translation (in QCIR format, which can be translated to QDIMACS). All the BDDL models for games described, benchmarks and data are available online[3]. One can easily model other games in BDDL and generate QBF formulas to solve those games up to a depth and extract a winning strategy.

For experimental evaluation, we consider small boards for various games described in this paper. We generate QBF instances and try to solve them with the QDIMACS solvers DepQBF [80] and CAQE [98] and with the QCIR solvers Quabs (QU) [56] and CQESTO (CT) [65]. We also applied QBF preprocessors Bloqqer (B) [58] and HQSpre (H) [132] on the QDIMACS encodings. We give a 1 hour time limit and 8GB memory limit (preprocessing+solving) for each instance. All computations for the experiments are run on a cluster.[4]

In Table 8.2, we report the critical depths in **bold** when a winning strategy is found by any solver-preprocessor combination. We report the non-existence of winning strategies in plain font if the depth-bound is known to be complete. Otherwise, we show the maximum refuted depth for the unsolved instances in *italic*.

---

[3]https://github.com/irfansha/Q-sage

[4]http://www.cscaa.dk/grendel-s, each problem uses one core on a Huawei FusionServer Pro V1288H V5 server, with 384 GB main memory and 48 cores of 3.0 GHz (Intel Xeon Gold 6248R).

(a) Harary's Tic-Tac-Toe (HTTT)

n	D	E	Ey	F	K	S	T	Tp
				Several standard shapes				
3	**3**	**5**	9	9	9	-	9	**9**
4	**3**	**5**	**7**	15	*11*	*13*	**5**	**9**

(b) Breakthrough, for first and second player

m / n	first player			second player		
	4	5	6	4	5	6
2	13	*17*	**15**	**8**	**10**	*14*
3	19	*11*	9	**12**	*12*	*10*

(c) Connect-c

nxn	C-2	C-3	C-4
2x2	**3**	-	-
3x3	**3**	9	-
4x4	**3**	**9**	15
5x5	**3**	**9**	*11*
6x6	**3**	**9**	*11*

(d) Domineering

n / m	2	3	4	5	6
2	**2**	**2**	5	**6**	**6**
3	**4**	**4**	7	8	10
4	**4**	**6**	**8**	**10**	**12**
5	6	**8**	11	13	*11*
6	**6**	**6**	**12**	*11*	*11*

(e) Evader-Pursuer (dual)

nxn	P	EP	EP-d
4x4	(1,2)	*21**	**2**
4x4	(2,3)	**3**	*10*
8x8	(2,3)	*11*	**6**
8x8	(3,4)	**7**	*10*

*We solved 21, but not 19

Table 8.2: Results for HTTT, Breakthrough, Connect-c, Domineering and Evader-Pursuer

Positional games In positional games for HTTT, we consider benchmarks from [36]. In Table 8.2(a), we solve all shapes (standard names) on a 3x3 board. On a 4x4 board, we solve all shapes except skinny and knobby. For Hex games, we use Hein's benchmarks as in [84]. For these, we first generate simplified BDDL problem files with explicit winning sets. We drop explicit winning sets with more than the number of black moves. In Table 8.3, we report the number of instances solved. Our simplified instances perform on par with the COR encoding by [84].

Breakthrough Boards up to 6x5 were solved in [109] using Proof Number Search and handcrafted race patterns. As far as we know, we provide the first encoding of BreakThrough in QBF. On an $m \times n$ board, we need $4mn - 10m + 1$ moves to show that the first player has no winning strategy. On the other hand, a second player winning strategy, i.e., giving up the first move, may be demonstrated much earlier. In our experiments, Table 8.2(b), we search for both first and second player winning

strategies. For the first player, we prove non-existence for boards 2x4 and 3x4 and existence for 2x6. For the second player, we show existence for boards 2x4, 2x5 and 3x4 (all optimal strategies). Partial endgame tablebases were used [62] for improving a 6x6 Breakthrough playing program. One could also use QBF for such shallow endgames.

Connect-c   The first QBF encoding for Connect-c [48] restricted the search to columns. The authors reported that no winning strategies exist for Connect-4 on a 4x4 board. In Table 8.2(c), we consider $c \in 2, 3, 4$ and nxn board sizes up to n=6. Connect-2 and Connect-3 have small critical depths and can be solved easily even for large board sizes. For Connect-4, we can solve the 4x4 board within 1 hour. By relaxing action definitions in BDDL, one could improve the model to allow only column search. This would probably solve larger boards.

Evader and Pursuer   We consider the instances from [3]. In an nxn grid, the Evader starts at (xmax,ymin) and the goal is to reach (xmin,ymin). Pursuer (column P) starts at different places. [1] designed the instances for early termination, so all are UNSAT. We also add SAT instances with different Pursuer positions. In the first instance in Table 8.2(e), Evader wins after one white move. Ideally, QBF solvers should be able to infer this quickly. For depths $17, 21$, DepQBF with Bloqqer indeed finds unsatisfiability within seconds. Surprisingly, for depths $15, 19$ it ends up searching unnecessary search space. Among other solvers, only CQESTO scales up to depth 17 (solves within seconds). However, it times out on deeper games. We consider a dual version of Evader-Pursuer as well: instead of search for Evader winning, we search for Pursuer winning. Here, Pursuer wins if it reaches the goal first or kills the Evader.

Domineering   In table 8.2(d), we report results on board sizes up to 6x6. Domineering on rectangular boards has been solved mainly by combinatorial theory non-optimally [74], perfect solving i.e., without any search [126] and domain specific solvers [127]. 11x11 has been solved to show first player winning (took 174 days and 15 h on a standard desktop computer) by [127]. As far as we know, we provide the first encoding of Domineering in QBF. Our results are consistent with the results in literature; we also provide critical depths, i.e., optimal winning strategies.

Comparing Multiple QBF Solvers and Preprocessors   In Table 8.3, we report results from different QBF solvers on all solved instances. Overall, DepQBF with Bloqqer performs well and solves most instances. If we look at the clauses generated by Bloqqer, it adds long clauses with indicator variables for early backtracking in case of an illegal move. On the other hand, HQSpre does not seem to fully recover this information with shorter clauses. Interestingly, there are some unique instances solved with HQSpre. The performance of CAQE is interesting: it performs well for the domains like positional games where there is a low branching factor for moves. On the contrary, for the domains like Breakthrough and Evader-Pursuer, Circuit solvers

Dom / SP	DepQBF		CAQE		CT	QU
	B	H	B	H		
HTTT	**15**	12	12	11	10	11
Hex	**26**	18	22	19	10	18
B	**6**	4	2	2	2	3
B-SP	**6**	5	2	3	4	4
C-c	**12**	**12**	**12**	8	8	8
EP	**3**	2	2	2	**3**	2
EP-dual	1	**3**	1	2	2	**3**
D	**25**	20	23	19	22	23
total	**94**	76	76	66	61	72
Avg. Cov.	**0.86**	0.74	0.61	0.58	0.6	0.67

Table 8.3: Number of instances solved by various solver-preprocessor combinations. Average Coverage indicates the fraction of solved instances by a combination, averaged per game row.

outperform CAQE. The circuit solver CQESTO uniquely solves an 8x8 instance in the Evader-Pursuer domain. On the other hand, it performs badly on the Hex domain. We suspect that this might be due to lack of early pruning in Hex, where the solver often needs to explore until the innermost quantifier to conclude satisfiability or unsatisfiability. In the Domineering domain, all solvers perform relatively well, perhaps due to very simple action and goal configurations. Since different domains have a different number of instances, we also report the average coverage per domain, In this weighted measure, circuit solvers perform on par with CAQE. These results suggest that the domains encoded using BDDL provide a rich set of benchmarks.

8.5.1 Winning Strategy Validation

In case of classical planning, one can use an external plan validator for correctness of plans. Validation of winning strategies from QBF is more complex, errors can occur in modelling, encoding or solving. Testing if a winning strategy exists for already solved game instances and testing if the winning move is correct can be used to some extent. However, this is not sufficient: The encoding can only be validated properly by looking at the values of variables.

We propose to use QBF certificates, as generated by some QBF solvers. By giving assumptions for universal variables to a SAT solver, along with the certificate, one can extract the values of existential variables, i.e., black moves and board state variables. We interactively play as a white player with the winning strategy as an opponent, and visualize the board at each time step. At the end of the play, we check if the output goal gate is true, i.e., Black won the game. Illegal white moves are identified by checking the outputs of certain intermediate gates. Fig. 8.2 shows the flow chart for our QBF-based validation framework. Appendix A. presents a demo of validating
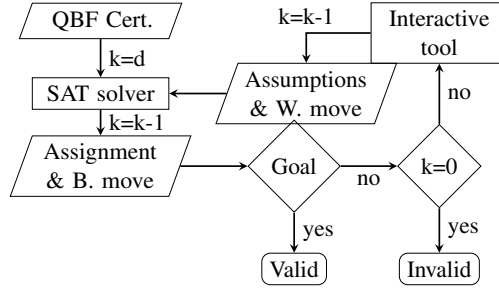
Figure 8.2: Winning strategy validation with Interactive play

a strategy for the Tic game. We visualize the board by extracting values for state variables from the certificate.

8.6 Conclusion and Future Work

We propose a compact modelling format BDDL for a subset of 2-player games played on a grid. Several classical games are modelled, such as positional games, Connect4, BreakThrough, Evader and Pursuer, and Domineering. We provide a concise QBF translation from BDDL to QBF for generating winning strategies, and provide an open source implementation for the translation. This provides the first QBF encoding for the games BreakThrough, KnightThrough and Domineering. Using existing QBF solvers and preprocessors, we solved several small instances of these games, yielding concrete winning strategies. This provides a rich set of benchmarks for QBF solvers. Hopefully, such diverse QBF instances can help the QBF solving community. We also provided a framework for validating our winning strategies by using QBF certificates.

As of now, our action and goal conditions are somewhat restricted. One research direction would be to relax these restrictions. For example, allowing static predicates can extend our models with multiple pieces. In fact, with static predicates one could already model several chess puzzles. Allowing non-static predicates can result in better domains for games like Connect4. Relaxing goal conditions can help to model more complex games implicitly. Our validation of QBF encodings is especially useful when checking alternative game encodings. QBF solvers that generate certificates are several orders slower than their counterparts and can blow up in size even for small depths. There is a need to improve certificate generation for extensive validation.

Appendix

A. A demo for winning strategy validation

We provide a script for winning strategy validation demo for HTTT Tic problem. One could play with the winning strategy to validate various game plays. Use the following command for demo in our Q-sage repository:

```
python3 general_interactive_play.py
```

In Figure 8.3, we provide a screenshot of interactive play during validation.

```
Enter your white move: occupy(3,3)
time step: 2
  1 2 3 4
1 - - - -
2 - - - -
3 - B W -
4 - - - -

Move played by black: occupy(2,2)
time step: 3
  1 2 3 4
1 - - - -
2 - B - -
3 - B W -
4 - - - -

Enter your white move: ▯
```

Figure 8.3: A snapshot of winning strategy validation

Chapter 9

Validation of QBF Encodings with Winning Strategies

Abstract

When using a QBF solver for solving application problems encoded to quantified Boolean formulas (QBFs), mainly two things can potentially go wrong: (1) the solver could be buggy and return a wrong result or (2) the encoding could be incorrect. To ensure the correctness of solvers, sophisticated fuzzing and testing techniques have been presented. To ultimately trust a solving result, solvers have to provide a proof certificate that can be independently checked. Much less attention, however, has been paid to the question how to ensure the correctness of encodings.

The validation of QBF encodings is particularly challenging because of the variable dependencies introduced by the quantifiers. In contrast to SAT, the solution of a true QBF is not simply a variable assignment, but a winning strategy. For each existential variable x, a winning strategy provides a function that defines how to set x based on the values of the universal variables that precede x in the quantifier prefix. Winning strategies for false formulas are defined dually.

In this paper, we provide a tool for validating encodings using winning strategies and interactive game play with a QBF solver. As the representation of winning strategies can get huge, we also introduce validation based on partial winning strategies. Finally, we employ winning strategies for testing if two different encodings of one problem have the same solutions.

9.1 Introduction

Quantified Boolean formulas (QBFs) extend propositional formulas with universal and existential quantifiers over the Boolean variables [9], rendering their decision problem PSPACE-complete. As many application problems from artificial intelligence and formal verification have efficient QBF representations (see [117] for a survey) and as much progress has been made in the development of QBF solving tools [95], QBFs

provide an appealing framework for solving such problems. In practice, however, obtaining correct and concise QBF encodings can be complex and error-prone as currently hardly any support for testing and debugging QBF encodings is available. The complexity of getting correct encodings comes on the one hand from the fact, that QBFs provide only a low-level language operating on the bit level and on the other hand from their interactive nature resulting from the quantifier alternations. The evaluation of a QBF can be seen as a two-player game between the existential and the universal player, where the existential player's goal is to satisfy the formula and the universal player's goal is to falsify the formula. Hence, models of true QBFs and counter-models of false QBFs are also called their *winning strategies*.

A winning strategy is a set of Boolean functions that defines how to set existential (universal) variables of a QBF to satisfy (falsify) a true (false) formula. A winning strategy provides the solution of the encoded application problem like the plan of a planning problem or the witness that no plan exists. Basically, winning strategies can be obtained in two forms: (1) statically, in terms of serialized functions that map existential variables to universal variables (or vice versa) [6], or (2) as interactive game play either with a QBF solver as opponent or by proof rewriting [50]. In contrast to SAT, where a solution is simply a variable assignment, the correctness of a winning strategy is challenging to validate, because of the dependencies between the variables. To prove that a winning strategy in serialized form is indeed a solution of a given QBF ϕ, a co-NP-hard problem has to be solved if ϕ is true, and an NP-hard problem has to be solved otherwise. This check can be automated by using a SAT solver. To show that a winning strategy is indeed a solution of an application problem, mainly remains a manual task. It is even non-trivial to find out if two variants of a problem encoding (e.g., a basic version and an optimization) share some common solutions.

We present a tool that supports the interactive testing of encodings based on serialized winning strategies in terms of Boolean functions as well as on dynamic validation, i.e., playing interactively against a QBF solver. We also propose a combination of both approaches for scalable validation based on partial winning strategies. To automate the testing process of an encoding, we implement a fuzz-testing approach for randomly exploring different parts of the search space. Fuzz testing has been successfully employed for testing solvers [21], but not for testing encodings. Finally, we present an approach to compare different variants of an encoding. While it is in general not feasible to prove that two encodings are equivalent, our tool supports testing if the two encodings share a common winning strategy. With a case study, we illustrate how our tool can be used to evaluate and understand QBF encodings.

9.2 Preliminaries

We consider closed QBF formulas in prenex normal form, i.e., of the form $Q_1 X_1 \cdots Q_n X_n . \phi$, where quantifiers $Q_i \in \{\forall, \exists\}$, $Q_i \neq Q_{i+1}$, and X_i are disjoint sets of variables. The matrix ϕ is a propositional formula over $\bigcup X_i$. A QBF $\forall X \Pi . \phi$ is true iff both $\forall X' \Pi . \phi[x/\top]$ and $\forall X' \Pi . \phi[x/\bot]$ are true where $X' = X \setminus \{x\}$ and
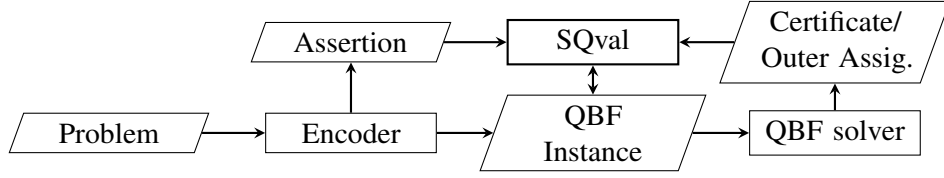
Figure 9.1: Validation workflow with SQval

$\phi[x/t]$ is obtained from ϕ by replacing x by truth constant t. A QBF $\exists X \Pi.\phi$ is true iff $\exists X' \Pi.\phi[x/\top]$ or $\exists X' \Pi.\phi[x/\bot]$ is true. Often, the semantics of a QBF is also expressed as a two-player game: In the *ith* move, the values of the variables in X_i are chosen by the existential player if $Q_i = \exists$ and otherwise by the universal player. If all variables are assigned and the formula evaluates to true (false), then the existential (universal) player wins. A QBF is true (false) iff there is a winning strategy for the existential (universal) player. Winning strategies can be represented in terms of *Skolem/Herbrand functions*. A winning strategy for a true QBF $\Pi.\phi$ over existential variables X is a set $S = \{f_x \mid x \in X\}$ of Skolem functions such that each f_x is a Boolean function over the universal variables preceding x in the prefix and $\phi[X/S]$ is valid. Dually, a winning strategy for a false QBF $\Pi.\phi$ over universal variables Y is a set $H = \{f_y \mid y \in Y\}$ of Herbrand functions such that f_y is a Boolean function over the existential variables preceding y in the prefix and $\phi[Y/H]$ is unsatisfiable.

9.3 QBF Validation With Interactive Play

With our tool SQval (Scalable QBF Validator), we support the validation of QBF encodings independent of any specific QBF application. The general workflow is shown in Figure 9.1. The tool accepts both formulas in prenex conjunctive normal form (PCNF) and formulas in prenex non-CNF format. Hence, the QDIMACS format for PCNF formulas and the QCIR format [67] for prenex non-CNF formulas are supported.

9.3.1 Playing with Skolem/Herbrand Functions

Given a true QBF $\exists X_1 \forall Y_1 \exists X_2 \forall Y_2 \ldots \exists X_n \forall Y_n.\phi$ which has the set S of Skolem functions as one solution. These functions can be used to calculate the values of the existential variables in X_i based on provided values of the universal variables Y_j with $j < i$. As the variables of X_1 occur in the outermost quantifier block, they do not depend on any universal variables. Therefore, their Skolem functions are constant and can immediately be provided. Then the user has to enter the values for variables Y_1. Alternatively, they can also be randomly selected. Based on the values of the variables in Y_1, the values of the variables in X_2 are calculated. This procedure is repeated until all variables are assigned a value and the matrix ϕ evaluates to true. Evaluation of a false QBF works dually by using Herbrand functions.

There are solvers like Caqe [98] that construct Herbrand and Skolem functions during the solving process and there are frameworks that retrospectively extract functions from proofs produced by the solvers [7, 13, 89, 93]. Such functions allow to independently certify the correctness of a solving result by using a SAT solver. Little information, however, is provided on the correctness of the problem encoding. Here, we use the functions to "execute" test cases and interpret the results in an interactive manner. In our interactive play, Skolem functions automatically decide the moves of the existential player in the case of true instances. For false instances, Herbrand functions provide the moves of the universal player.

The Herbrand and Skolem functions have to be precomputed and can be provided in the AIGER format or as propositional formula in CNF format. In addition, encoding-specific assertions can be provided to SQval as CNF formulas. These assertions are checked under the full variable assignment at the end of the play. Such an assertion could be used, for example, to check if some goal condition is met or if some invariants are not violated in the game play. An example is given in the case study presented in §9.5.

The advantage of this method is the one-time cost of generating Skolem/Herbrand functions. Their evaluation is computationally cheap, because only truth values need to be propagated. However, even for simple problems, these functions can become very large and producing such functions often considerably slows down the solvers, because powerful pre- and inprocessing techniques have to be disabled. To overcome this drawback, we present an alternative approach in the following.

9.3.2 Playing with a QBF Solver

Given a true QBF of the form $\Phi_1 = \exists X_1 \forall Y_1 \Pi.\phi_1$ or a false QBF of the form $\Phi_2 = \forall Y_2 \exists X_2 \Pi.\phi_2$, most QBF solvers are able to provide assignments σ_{X_1} and σ_{Y_2} such that Φ_1 evaluates to true under σ_{X_1} and Φ_2 evaluates to false under σ_{Y_2}. If we apply σ_{X_1} on Φ_1, we obtain the true QBF $\Phi_1' = \forall Y_1.\phi'$. As this QBF has to be true for all assignments of variables Y_1, we pick now an interesting assignment and apply it on Φ_1' to obtain the true QBF Φ_1'' which starts with an existential quantifier block (or it has become the empty formula). Now we can ask a QBF solver for a satisfying assignment of these outermost variables and proceed in this way until all variables are finally assigned. Similarly, we can interactively evaluate Φ_2.

Based on this approach, we can replace large Skolem/Herbrand functions and avoid slower QBF solvers. The disadvantages of the approach is that a linear number of QBF problems must be solved: one for each round of validation.

9.3.3 Hybrid Validation

For many instances, the two interactive play approaches presented above are either limited by the size of the Skolem/Herbrand functions or by the costs of the QBF solver calls. As a solution, we suggest combining both approaches. First, we precompute Skolem/Herbrand functions only for the variables in the first k quantifier blocks of

a QBF Φ. Based on these functions, we calculate the truth values of the variables which they define, and replace them respectively in Φ. Now we obtain a QBF Φ' with k quantifier alternations less and fewer variables. Then we proceed with evaluating Φ' by playing against a QBF solver.

The certification framework QBFcert [89] supports the generation of partial winning strategies, so we can use the respective options to obtain the functions of the variables from the first k quantifier blocks. We also provide an extractor for obtaining partial winning strategies from a full winning strategy by specifying the variables that should be considered. The partial winning strategies remain smaller than full winning strategies and we can take advantage of faster non-certifying solvers for the validation. Note that, one can generate assignments in the outer-most quantifier block with almost any non-certifying solver. In case the certifying solvers are too slow, one would use such an assignment to speed up the validation. In §9.5 we will demonstrate the memory/time tradeoff with partial strategies.

9.4 Common Winning Strategies of Two Encodings

Often, the same problem can be encoded in many ways resulting in formulas with different winning strategies. Consider for example two encodings of a two-player game for which we have a basic reference encoding that is very hard to solve, but which is most likely correct. Further, there is an optimized encoding which can be solved faster and which only has winning strategies that must also be solutions to the basic encoding. Therefore, we call the basic encoding *more relaxed* than the optimized encoding. To increase the trust in the optimized encoding, we want to validate if a found winning strategy is indeed a solution of the basic encoding. To this end, we want to take a winning strategy of one formula and enrich the other formula with this encoding. If this enriched formula has the same truth value, then we can conclude that the winning strategy of the first encoding is also a winning strategy of the second encoding. As two encodings might be defined over different variables, we need to introduce a set of common variables $\mathscr{C}$ that occur in both formulas (in practice some renaming of the variables not occurring in $\mathscr{C}$ might be necessary to avoid name clashes). This validation approach also works both for true and false formulas.

We first define a subsumption check between two true QBF formulas ϕ_1, ϕ_2. For this purpose, we use certificates, which are essentially winning strategies.

Definition 35. *Let ϕ_1 and ϕ_2 be two true QBFs with common variables $\mathscr{C}$. We define ϕ_1 solution-subsumes ϕ_2 (written as $\phi_1 \sqsubseteq \phi_2$) iff for all winning strategies S for ϕ_1, also $\phi_2[S/\mathscr{C}]$ is true.*

We can only test $\phi_1 \sqsubseteq \phi_2$ for particular instances of S. To show the subsumption of ϕ_1 to ϕ_2, we need to show that a given strategy S of ϕ_1 is also a strategy for ϕ_2. Recall that a strategy is simply a function from universal variables to existential variables. As long as the same function works for ϕ_2, the subsumption relation is not refuted.

To show that, we first rewrite S to S' to avoid common name conflicts with ϕ_2. While rewriting S to S', we leave the common variables untouched (which are part of the winning strategy we consider). For example, in the game instances in the case study 9.5, we do not rewrite black player and white player moves. Finally, we create a new formula $S' \wedge \phi_2$. We claim that the new formula is True iff S' is a strategy for ϕ_2. Otherwise, checking this single S is sufficient to refute solution-subsumption, which is useful for bug detection. The S' formula essentially forces the existential variables given values to the universal variables. Since the QBF solver has to satisfy both S' and ϕ_2 which share common variables, the assignments to the common variables are always the same. In our game instances, the black moves for the opponent's white moves are forced by the rewritten strategy S'.

While for true formulas, the (possibly modified) winning strategy is just conjunctively added to the matrix, for false formulas it is additionally necessary to change the quantifier type of the variables defined by the winning strategy to existential. If the formula remains false, then the winning strategy of the first formula is also a winning strategy of the second formula. Since checking for common solutions has similar memory problems as the winning strategies for the interactive play presented in §9.3, we also support checking solution-subsumption with partial winning strategies. Our tool is completely agnostic to the completeness of a winning strategy. Examples are shown in the next section.

9.5   Case Study

We provide an open source implementation of validation and winning strategy equivalence. All benchmarks and data are available online.[1] To obtain a winning strategy, our tool SQval (Scalable QBF Validator) first generates a proof trace in QRP-format using the solver DepQBF [80]. Then it extracts the winning strategy using the QRPcert framework [89]. For interactive plays, it uses solver DepQBF for QDIMACS instances and solver Quabs [56] for QCIR instances. All computations for the experiments are run on a cluster.[2]

As a case study, we conducted an experiment with two QBF encodings for positional games. In particular, we compare two encodings for the game Hex: Lifted Neighbour-Based (LN) and Stateless Neighbour-based (SN) in [115]. In the Hex game, players (black, white) take turns to occupy empty positions on a NxN board with hexagonal cells. The player who connects appropriate opposite borders with a path of pegs of their own color wins the game. To demonstrate validation and equivalence checking, we first consider a small Hex instance, Hein-12 (see Figure 9.2a), due to Piet Hein [55]. We first preprocess the instance as in [115], resulting in 8 open positions. Using the preprocessed instance, we generate three QBF instances for a winning strategy of depth 7:

---

[1]`https://github.com/irfansha/SQval`

[2]`http://www.cscaa.dk/grendel-s`, each problem uses one core on a Huawei FusionServer Pro V1288H V5 server, with 384 GB main memory and 48 cores of 3.0 GHz (Intel Xeon Gold 6248R).
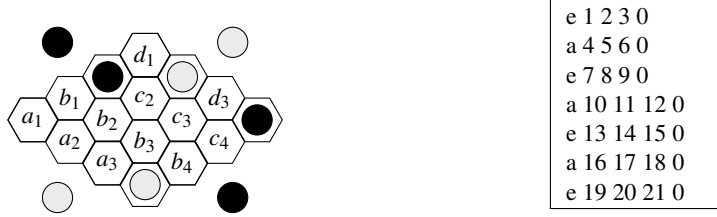
(a) Hein puzzle 12, before Hex preprocessing

```
e 1 2 3 0
a 4 5 6 0
e 7 8 9 0
a 10 11 12 0
e 13 14 15 0
a 16 17 18 0
e 19 20 21 0
```

(b) Move variables in all Hein-12 QBF instances

- LN-Hein-12 : Instance generated with LN, both players can only occupy empty positions.

- SN-Hein-12 : Instance generated with SN, black player can only occupy empty positions.

- SN-R-Hein-12 : Instance generated with SN with relaxed constraints for optimization, in which the black player is allowed to occupy previous black positions.

All three QBF instances solve the Hex instance, and their first 7 layers (cf. 9.2b) correspond to the moves taken, i.e., variables encoding 1 out of max 8 open positions (3 bits) per layer.

9.5.1 Validating Hein-12 Instances

The Hein-12 puzzle indeed has a winning strategy at depth 7, so all 3 instances are true formulas. For validation, we propose three assertions that are relevant to these encodings.

- Goal-Assertion (GA): "Goal is reached at the end of the play". For the LN instance, satisfying the goal constraint is specified by the assertion "314 0".

- Legal-Play-Assertion (LPA): "Black does not play on white positions". We generate inequality constraints between black moves and preceding white moves as a CNF.

- Legal-Black-Assertion (LBA): "Black does not play on black positions". We generate inequality constraints between different black moves as a CNF.

GA and LPA should hold for all instances, whereas LBA should only hold for the LN-Hein-12 and SN-Hein-12 instances. We use all 3 types of QBF validation for checking these 3 assertions on all 3 encodings. We run 100 iterations with a random generator with seeds ranging from 0-99. For hybrid validation, we use partial certificates up to depth 3 and validate the rest with a QBF solver. In Table 9.1, we present for each case the number of passing/failing runs. Indeed, both static, dynamic, and hybrid validation show some runs revealing the failure of assertion LBA for

Inst: / Assert:	static			dynamic			hybrid		
	GA	LPA	LBA	GA	LPA	LBA	GA	LPA	LBA
LN-Hein-12	100/0	100/0	100/0	100/0	100/0	100/0	100/0	100/0	100/0
SN-Hein-12	100/0	100/0	100/0	100/0	100/0	100/0	100/0	100/0	100/0
SN-R-Hein-12	100/0	100/0	91/9	100/0	100/0	84/16	100/0	100/0	84/16

Table 9.1: Number of valid runs/ Number of invalid runs for each instance and each assertion.

Q1: / Q2:	LN-Hein-12	SN-Hein-12	SN-R-Hein-12
LN-Hein-12	T	T	T
SN-Hein-12	T	T	T
SN-R-Hein-12	F	F	T

Table 9.2: Listing the result of the subsumption tests between instances for Q1 $\sqsubseteq$ Q2.

SN-R-hein-12, while all other tests pass. For Hein-12, all three validation techniques take a few seconds and a few MB for each iteration.

9.5.2   Equivalence Check for Hein-12

Validating GA and LPA increases the confidence in the correctness of the previous encodings. Additionally, we expect that LN-Hein-12 and SN-Hein-12 have the same winning strategies, since in both encodings, the black player can only play on open positions, and the same moves lead to equivalent states. This cannot be checked with our testing or fuzzing approach. Instead, we apply our subsumption check on all combinations of the 3 encodings. Table 9.2 shows the results from our subsumption check. Indeed, LN-Hein-12 and SN-Hein-12 appear to be equivalent (on the winning strategy returned by the solver). However, we found a strategy for SN-R that is not valid for SN and LN. Indeed, SN-R can play on already occupied black positions, which leads to invalid strategies for the LN and SN encodings. On the other hand, every move played in LN or SN is also valid in SN-R, resulting in subsumption in the other direction. These results are consistent with the intention behind the encodings.

9.5.3   Validating Larger Hex Instances With Partial Certificates

The Hein-12 instance has only 8 open positions after preprocessing, and we checked only for depth 7. On this small example, all 3 validation approaches worked equally well. Since the certificates remain small, the subsumption checks can be done within a few seconds. However, certificates grow exponentially in size with the number of variables in each layer and alternation depth. To show the difference between the validation strategies, we experiment with a harder Hex instance, Hein-09, which has 10 open positions after preprocessing, and we generate instances with a winning strategy of depth 9.

In Table 9.3, we observe that the full certificates in AAG format (Ascii And-Inverter Graph) are quite large, in the range of 532.7 MB – 7.8 GB. Note that the partial certificates are much smaller, but increasing with the level. Table 9.4 shows the resources required for generating the QRP traces for different settings, and for extracting certificates from the traces.

To show the difference between the 3 validation approaches on the harder instance Hein-09, we consider assertion GA for validation. From Table 9.5, we observe that static validation with full certificates for LN is infeasible. While we can validate the other encodings using full certificates, it takes up to 20 GB. Dynamic validation performs well on SN and SN-R, while it takes 1203 seconds for validating a single iteration of an LN instance. Note that to run 100 iterations with different seeds, we would need approximately 100*1203 seconds. Hybrid validation performs clearly the best in both time and memory, never exceeding a couple of seconds or 2 MB. Of course, generating a partial certificate via QRP trace is still the bottleneck for hybrid validation. However, we only pay a one-time cost for generating partial certificates, which can be used any number of times for validation or subsumption checks.

We will now experiment with subsumption checking, using full or partial certificates. For Hein-09 the QBF instances appended with a certificate can exceed 15 GB in CNF. Note that, we only need partial certificates with all existential black move variables, i.e., L9 in Table 9.3. These never exceed 10 MB (in AAG format), so a complete subsumption check with L9 partial certificates only is feasible. We compute subsumption checks on all combinations of 3 encodings, i.e., for each combination we append one QBF with the certificate of the other, and use a QBF solver. From Table 9.6, it is clear that subsumption with full certificates blows up, often exceeding 50 GB of memory. In fact, we could not generate the appended instance with LN certificates since the instances themselves can exceed 20 GB.

On the other hand, with L9 certificates, we could check non-subsumption for SN-R with SN and LN, taking 446 and 84 seconds, respectively. DepQBF runs out of time when trying to prove the subsumption cases, but never uses more than 2 GB for solving with L9 certificates. Intuitively, proving non-subsumption is indeed easier than proving subsumption. One could try to use preprocessors, to speed up the subsumption checks for L9 certificates, but full certificate subsumption would still be out of reach.

Enc: / Cert:	Full	L1	L3	L5	L7	L9	QRP trace
LN-Hein-09	7.8G	108	1.1K	21.8K	434K	8.2M	6.8G
SN-Hein-09	641.6M	96	1.1K	23.4K	437.3K	8.2M	344M
SN-R-Hein-09	532.7M	96	1.2K	24.2K	461.1K	8.7M	394.5M

Table 9.3: Size in Bytes of (partial) certificates in AAG format, for each encoding instance with increasing levels of partial certificates. QRP trace is the size of the trace generated by solver DepQBF.

Enc: / Cert:	Peak Memory (MB)				Time Taken (Sec)			
	QRP trace	Full	L3	L9	QRP trace	Full	L3	L9
LN-Hein-09	13470	10390	7580	9750	1165	261	59	233
SN-Hein-09	615.46	1.54	1.54	1.54	54	20	21	15
SN-R-Hein-09	535.9	1.54	1.54	1.54	54	13	4	15

Table 9.4: Time and Memory used for generating the QRP trace and extracting certificates from it.

Inst:	static		dynamic		hybrid-L3	
	PM	TT	PM	TT	PM	TT
LN-Hein-09	-	TO	64.1	1203	**1.54**	**1.6**
SN-Hein-09	20.08K	2100	1.54	9.4	**1.53**	**0.4**
SN-R-Hein-09	18.35K	1226	**1.53**	5.4	**1.53**	**0.4**

Table 9.5: Peak Memory (PM) in MB and Time Taken (TT) in seconds for assertion GA (seed 0).

Inst:	LN-Hein-09		SN-Hein-09		SN-R-Hein-09	
	Full	L9	Full	L9	Full	L9
LN-Hein-09	-	1.53	-	1.57	-	1.57
SN-Hein-09	63.7	1.53	63.7	1.6	63.7	1.6
SN-R-Hein-09	50.74	1.6	50.78	1.6	50.8	1.69

Table 9.6: Peak Memory in GB during subsumption checks between Hein-09 instances.

9.6 Conclusion and Future Work

In this paper, we proposed validation techniques with winning strategies and interactive play with a QBF solver. For scalable validation, we proposed using partial winning strategies for outer layers and interactive play for deeper layers. We extended the idea of validation to solution-equivalence of encodings that have some common winning strategy. To evaluate various techniques proposed, we conducted a case study on 2-player game encodings for the game Hex. We showed that with the use of winning strategies, one can increase the confidence of encoding correctness. In the most scalable approach, generating QRP traces remains the bottleneck, as solvers generate complete traces. While checking solver correctness requires complete traces, partial traces/certificates are sufficient for encoding validation. One future research direction would be to allow QBF solvers to generate partial traces efficiently.

Bibliography

[1] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic computational techniques for solving games. *International Journal on Software Tools for Technology Transfer*, 7:118–128, 2004. 18, 20, 21, 103, 120

[2] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 32(6):818–830, 2013. doi: 10.1109/TCAD.2013.2244643. URL `https://doi.org/10.1109/TCAD.2013.2244643`. 54

[3] Carlos Ansotegui, Carla P. Gomes, and Bart Selman. The Achilles' heel of QBF. In *20th National Conference on Artificial Intelligence*, volume 1 of *AAAI'05*, pages 275–281. AAAI Press, 2005. ISBN 1-57735-236-x. URL `http://dl.acm.org/citation.cfm?id=1619332.1619378`. 18, 20, 21, 93, 103, 120

[4] Broderick Arneson, Ryan Hayward, and Philip Henderson. MoHex wins Hex tournament. *ICGA journal*, 32(2):114, 2009. 79, 89, 94, 95

[5] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019. 53

[6] Valeriy Balabanov and Jie-Hong R. Jiang. Unified QBF certification and its applications. *Formal Methods Syst. Des.*, 41(1):45–65, 2012. doi: 10.1007/s10703-012-0152-6. URL `https://doi.org/10.1007/s10703-012-0152-6`. 126

[7] Marco Benedetti. Extracting certificates from Quantified Boolean Formulas. In *19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 47–53. Professional Book Center, 2005. URL `http://ijcai.org/Proceedings/05/Papers/0985.pdf`. 28, 106, 128

[8] Marco Benedetti and Hratch Mangassarian. QBF-based formal verification: Experience and perspectives. *Journal on Satisfiability, Boolean Modeling and Computation*, 5:133–191, 2008. 5

[9]   Olaf Beyersdorff, Mikolás Janota, Florian Lonsing, and Martina Seidl. Quantified Boolean Formulas. In *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1177–1221. IOS Press, 2021. doi: 10.3233/FAIA201015. URL `https://doi.org/10.3233/FAIA201015`. 18, 103, 105, 125

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999. doi: 10.1007/3-540-49059-0_14. URL `https://doi.org/10.1007/3-540-49059-0_14`. 4, 5

[11] Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 2011. doi: 10.1007/978-3-642-22438-6_10. 46, 89, 95

[12] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021. ISBN 978-1-64368-160-3. doi: 10.3233/FAIA336. URL `https://doi.org/10.3233/FAIA336`. 3, 4

[13] Roderick Bloem, Vedad Hadzic, Ankit Shukla, and Martina Seidl. FERP-Models: A certification framework for expansion-based QBF solving. In *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC) 2022*, September 2022. 28, 106, 128

[14] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artif. Intell.*, 90(1-2):281–300, 1997. doi: 10.1016/S0004-3702(96)00047-1. URL `https://doi.org/10.1016/S0004-3702(96)00047-1`. 10

[15] Miquel Bofill, Joan Espasa, and Mateu Villaret. The RANTANPLAN planner: system description. *Knowl. Eng. Rev.*, 31(5):452–464, 2016. doi: 10.1017/S0269888916000229. 50

[16] Édouard Bonnet, Florian Jamain, and Abdallah Saffidine. On the complexity of connection games. *Theoretical Computer Science (TCS)*, 644:2–28, 2016. 79

[17] Édouard Bonnet, Serge Gaspers, Antonin Lambilliotte, Stefan Rümmele, and Abdallah Saffidine. The parameterized complexity of positional games. In *44th International Colloquium on Automata, Languages, and Programming, (ICALP)*, pages 90:1–90:14, July 2017. doi: 10.4230/LIPIcs.ICALP.2017.90. 79, 82

[18] Kyle E. C. Booth, Minh Do, J. Christopher Beck, Eleanor Gilbert Rieffel, Davide Venturelli, and Jeremy Frank. Comparing and integrating constraint programming and temporal planning for quantum circuit compilation. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*, pages 366–374. AAAI Press, 2018. URL `https://aaai.org/ocs/index.php/ICAPS/ICAPS18/paper/view/17787`. 54, 55

[19] Adi Botea, Akihiro Kishimoto, and Radu Marinescu. On the complexity of quantum circuit compilation. In Vadim Bulitko and Sabine Storandt, editors, *Proceedings of the Eleventh International Symposium on Combinatorial Search, SOCS 2018, Stockholm, Sweden - 14-15 July 2018*, pages 138–142. AAAI Press, 2018. URL `https://aaai.org/ocs/index.php/SOCS/SOCS18/paper/view/17959`. 55

[20] Steve Boucher and Roger Villemaire. Quantified Boolean solving for achievement games. In *44th German Conference on Artificial Intelligence (KI)*, pages 30–43, 2021. 20, 21, 93, 104

[21] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Proc. of the 13th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2010)*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010. 126

[22] Lukas Burgholzer, Sarah Schneider, and Robert Wille. Limiting the search space in optimal quantum circuit mapping. In *27th Asia and South Pacific Design Automation Conference, ASP-DAC 2022, Taipei, Taiwan, January 17-20, 2022*, pages 466–471. IEEE, 2022. doi: 10.1109/ASP-DAC52403. 2022.9712555. URL `https://doi.org/10.1109/ASP-DAC52403.2022.9712555`. 16, 56, 68

[23] Sam Buss and Jakob Nordström. Proof complexity and SAT solving. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 233–350. IOS Press, 2021. doi: 10.3233/FAIA200990. URL `https://doi.org/10.3233/FAIA200990`. 27

[24] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artif. Intell.*, 69(1-2):165–204, 1994. doi: 10.1016/0004-3702(94) 90081-7. URL `https://doi.org/10.1016/0004-3702(94)90081-7`. 7

[25] Michael Cashmore, Maria Fox, and Enrico Giunchiglia. Planning as quantified boolean formula. In *20th European Conference on Artificial Intelligence (ECAI)*, volume 242, pages 217–222, August 2012. doi: 10.3233/978-1-61499-098-7-217. 92, 97

[26] Michael Cashmore, Maria Fox, and Enrico Giunchiglia. Planning as Quantified Boolean Formula. In *ECAI*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 217–222. IOS Press, 2012. doi: 10.3233/978-1-61499-098-7-217. 5, 11, 36, 49, 50, 51

[27] Michael Cashmore, Maria Fox, and Enrico Giunchiglia. Partially grounded planning as Quantified Boolean Formula. In *ICAPS*. AAAI, 2013. URL `http://www.aaai.org/ocs/index.php/ICAPS/ICAPS13/paper/view/5991`. 11, 13, 50

[28] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018. ISBN 978-3-319-10574-1. doi: 10.1007/978-3-319-10575-8. URL `https://doi.org/10.1007/978-3-319-10575-8`. 4

[29] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971. doi: 10.1145/800157.805047. URL `https://doi.org/10.1145/800157.805047`. 4

[30] Augusto B. Corrêa, Florian Pommerening, Malte Helmert, and Guillem Francès. Lifted successor generation using query optimization techniques. In *ICAPS*, pages 80–89. AAAI Press, 2020. URL `https://aaai.org/ojs/index.php/ICAPS/article/view/6648`. 9, 47

[31] Augusto B. Corrêa, Guillem Francès, Florian Pommerening, and Malte Helmert. Delete-relaxation heuristics for lifted classical planning. In Susanne Biundo, Minh Do, Robert Goldman, Michael Katz, Qiang Yang, and Hankz Hankui Zhuo, editors, *ICAPS*, pages 94–102. AAAI Press, 2021. URL `https://ojs.aaai.org/index.php/ICAPS/article/view/15951`. 9, 46

[32] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language, 2017. 54, 55

[33] Nachum Dershowitz, Ziyad Hanna, and Jacob Katz. Bounded model checking with QBF. volume 3569 of *Lecture Notes in Computer Science*, pages 408–414. Springer, 2005. doi: 10.1007/11499107_32. 5

[34] Nachum Dershowitz, Ziyad Hanna, and Jacob Katz. Bounded model checking with QBF. In *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 408–414. Springer, 2005. doi: 10.1007/11499107_32. 10, 36

[35] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in nonmonotonic logic programs. In Sam Steel and Rachid Alami,

editors, *Recent Advances in AI Planning, 4th European Conference on Planning, ECP'97, Toulouse, France, September 24-26, 1997, Proceedings*, volume 1348 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1997. doi: 10.1007/3-540-63912-8_84. URL `https://doi.org/10.1007/3-540-63912-8_84`. 10

[36] Diptarama, Ryo Yoshinaka, and Ayumi Shinohara. QBF encoding of generalized tic-tac-toe. In *4th International Workshop on Quantified Boolean Formulas (QBF)*, volume 1719 of *CEUR Workshop Proceedings*, pages 14–26. CEUR-WS.org, 2016. URL `http://ceur-ws.org/Vol-1719/paper1.pdf`. 18, 19, 93, 104, 119

[37] Minh Do, Zhihui Wang, Bryan O'Gorman, Davide Venturelli, Eleanor Gilbert Rieffel, and Jeremy Frank. Planning for compilation of a quantum algorithm for graph coloring. In *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 2338–2345. IOS Press, 2020. doi: 10.3233/FAIA200363. URL `https://doi.org/10.3233/FAIA200363`. 13, 15, 54, 55, 72

[38] Carmel Domshlak, Malte Helmert, Erez Karpas, Emil Keyder, Silvia Richter, Gabriele Röger, Jendrik Seipp, and Matthias Westphal. Bjolp: The big joint optimal landmarks planner. 2011. 56, 68

[39] Stefan Edelkamp. *General Game Playing*, pages 281–296. Springer International Publishing, Cham, 2023. ISBN 978-3-319-65596-3. doi: 10.1007/978-3-319-65596-3_15. URL `https://doi.org/10.1007/978-3-319-65596-3_15`. 4

[40] Joan Espasa, Jordi Coll, Ian Miguel, and Mateu Villaret. Towards lifted encodings for numeric planning in essence prime. In *CP 2019 Workshop on Constraint Modelling and Reformulation*, 2019. URL `https://modref.github.io/ModRef2019.html`. 50

[41] Johannes Klaus Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. The silent (r)evolution of SAT. *Commun. ACM*, 66(6):64–72, 2023. doi: 10.1145/3560469. URL `https://doi.org/10.1145/3560469`. 3, 4

[42] Bernd Finkbeiner. *Bounded Synthesis for Petri Games*, pages 223–237. Springer International Publishing, Cham, 2015. ISBN 978-3-319-23506-6. doi: 10.1007/978-3-319-23506-6_15. URL `https://doi.org/10.1007/978-3-319-23506-6_15`. 18

[43] Daniel Fiser. Lifted fact-alternating mutex groups and pruned grounding of classical planning problems. In *AAAI*, pages 9835–9842. AAAI Press,

2020. URL `https://ojs.aaai.org/index.php/AAAI/article/view/6536`. 51

[44]   Maria Fox and Derek Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.*, 20:61–124, 2003. doi: 10.1613/jair.1129. URL `https://doi.org/10.1613/jair.1129`. 37, 104

[45]   David Gale. The game of hex and the brouwer fixed-point theorem. *The American mathematical monthly*, 86(10):818–827, 1979. 79

[46]   Michael Genesereth and Michael Thielscher. *General game playing*. Springer Nature, 2022. 4

[47]   Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI Mag.*, 26(2):62–72, 2005. doi: 10.1609/aimag.v26i2.1813. URL `https://doi.org/10.1609/aimag.v26i2.1813`. 17, 104, 110

[48]   Ian P. Gent and Andrew Rowley. Encoding Connect-4 using Quantified Boolean Formulae. In *2nd Intl. Work. Modelling and Reform. CSP*, pages 78–93, 2003. 18, 19, 20, 21, 103, 120

[49]   Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004. 3, 35, 57

[50]   Alexandra Goultiaeva, Allen Van Gelder, and Fahiem Bacchus. A uniform approach for generating proofs and strategies for both true and false QBF formulas. In *Proc. of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 546–553. IJCAI/AAAI, 2011. 126

[51]   Lov K. Grover. A fast quantum mechanical algorithm for database search. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219. ACM, 1996. doi: 10.1145/237814.237866. URL `https://doi.org/10.1145/237814.237866`. 53

[52]   Andrew R Haas. The case for domain-specific frame axioms. In *The Frame Problem in Artificial Intelligence*, pages 343–348. Elsevier, 1987. URL `https://www.sciencedirect.com/science/article/pii/B9780934613323500265`. 9, 36

[53]   Immanuel Halupczok and Jan-Christoph Schlage-Puchta. Achieving snaky. *Integers*, 7(1), 2007. 20

[54]   Frank Harary. Achievement and avoidance games for graphs. In Béla Bollobás, editor, *Graph Theory*, volume 62 of *North-Holland Mathematics Studies*, pages 111–119. North-Holland, 1982. doi: https://doi.org/10.1016/S0304-0208(08)

73554-0. URL `https://www.sciencedirect.com/science/article/pii/S0304020808735540`. 18

[55] Ryan B. Hayward and Bjarne Toft. *Hex, the full story*. AK Peters/CRC Press/Taylor Francis, 2019. 19, 79, 81, 130

[56] Jesko Hecking-Harbusch and Leander Tentrup. Solving QBF by abstraction. In *9th International Symposium on Games, Automata, Logics, and Formal Verification (GandALF)*, volume 277 of *EPTCS*, pages 88–102, 2018. doi: 10.4204/EPTCS.277.7. URL `https://doi.org/10.4204/EPTCS.277.7`. 89, 96, 118, 130

[57] Malte Helmert, Gabriele Röger, and Erez Karpas. Fast downward stone soup: A baseline for building planner portfolios. In *ICAPS 2011 Workshop on Planning and Learning*, volume 2835, 2011. 8, 47, 56, 59

[58] Marijn Heule, Matti Järvisalo, Florian Lonsing, Martina Seidl, and Armin Biere. Clause elimination for SAT and QSAT. *Journal of Artificial Intelligence Research (JAIR)*, 53:127–168, 2015. 93, 118

[59] Daniel Höller and Gregor Behnke. Encoding lifted classical planning in propositional logic. In Akshat Kumar, Sylvie Thiébaux, Pradeep Varakantham, and William Yeoh, editors, *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling, ICAPS 2022, Singapore (virtual), June 13-24, 2022*, pages 134–144. AAAI Press, 2022. URL `https://ojs.aaai.org/index.php/ICAPS/article/view/19794`. 16

[60] Rostislav Horcík, Daniel Fiser, and Álvaro Torralba. Homomorphisms of lifted planning tasks: The case for delete-free relaxation heuristics. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pages 9767–9775. AAAI Press, 2022. URL `https://ojs.aaai.org/index.php/AAAI/article/view/21212`. 9

[61] Richard Howey, Derek Long, and Maria Fox. VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA*, pages 294–301. IEEE Computer Society, 2004. doi: 10.1109/ICTAI.2004.120. URL `https://doi.org/10.1109/ICTAI.2004.120`. 8

[62] Andrew Isaac and Richard Lorentz. Using partial tablebases in Breakthrough. In *Computers and Games*, pages 1–10, 2016. 120

[63] Toshinari Itoko, Rudy Raymond, Takashi Imamichi, Atsushi Matsuo, and Andrew W. Cross. Quantum circuit compilers using gate commutation rules. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASPDAC 2019, Tokyo, Japan, January 21-24, 2019*, pages 191–196. ACM, 2019. doi: 10.1145/3287624.3287701. URL `https://doi.org/10.1145/3287624.3287701`. 54

[64] Mikolás Janota. QFUN: towards machine learning in QBF. *CoRR*, abs/1710.02198, 2017. URL `http://arxiv.org/abs/1710.02198`. 96

[65] Mikolás Janota. Circuit-based search space pruning in QBF. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 10929 of *Lecture Notes in Computer Science*, pages 187–198. Springer, 2018. doi: 10.1007/978-3-319-94144-8_12. URL `https://doi.org/10.1007/978-3-319-94144-8_12`. 89, 96, 118

[66] Mikolás Janota and João Marques-Silva. Solving QBF by clause selection. In *IJCAI*, pages 325–331. AAAI Press, 2015. URL `http://ijcai.org/Abstract/15/052`. 96

[67] Charles Jordan, Will Klieber, and Martina Seidl. Non-cnf QBF solving with QCIR. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016. 80, 105, 127

[68] Jean Christoph Jung, Valentin Mayer-Eichberger, and Abdallah Saffidine. Qbf programming with the modeling language bule. In *25th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, August 2022. 80

[69] Toni Jussila and Armin Biere. Compressing BMC encodings with QBF. *Electronic Notes in Theoretical Computer Science*, 174(3):45–56, 2007. ISSN 1571-0661. doi: 10.1016/j.entcs.2006.12.022. 10, 36, 92, 97

[70] Henry A Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, volume 92, pages 359–363, 1992. URL `http://www.cs.cornell.edu/selman/papers/pdf/92.ecai.satplan.pdf`. 9, 35, 36

[71] Henry A. Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In William J. Clancey and Daniel S. Weld, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 2*, pages 1194–1201. AAAI Press / The MIT Press, 1996. URL `http://www.aaai.org/Library/AAAI/1996/aaai96-177.php`. 4

[72] Henry A. Kautz, David A. McAllester, and Bart Selman. Encoding plans in propositional logic. In *5th International Conference on Principles of Knowledge*

Representation and Reasoning (KR), pages 374–384, November 1996. 9, 10, 84

[73] Hans Kleine Büning and Uwe Bubeck. Theory of Quantified Boolean Formulas. In *Handbook of Satisfiability*, pages 735–760. 2009. doi: 10.3233/978-1-58603-929-5-735. 5

[74] Michael Lachmann, Cristopher Moore, and Ivan Rapaport. Who wins Domineering on rectangular boards? 2000. doi: 10.48550/ARXIV.MATH/0006066. URL https://arxiv.org/abs/math/0006066. 120

[75] Pascal Lauer, Álvaro Torralba, Daniel Fiser, Daniel Höller, Julia Wichlacz, and Jörg Hoffmann. Polynomial-time in PDDL input size: Making the delete relaxation feasible for lifted planning. In Zhi-Hua Zhou, editor, *IJCAI*, pages 4119–4126. ijcai.org, 2021. doi: 10.24963/ijcai.2021/567. URL https://doi.org/10.24963/ijcai.2021/567. 9, 46

[76] Leonid A. Levin. Universal sequential search problems. *Problems Inform. Transmission*, 9(3):265–266, 1973. 4

[77] H. R. Lewis. Complexity results for classes of quantificational formulas. *Journal of Computer and System Science*, 21(3), 1980. URL https://doi.org/10.1016/0022-0000(80)90027-6. 50

[78] Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for nisq-era quantum devices. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 1001–1014. ACM, 2019. doi: 10.1145/3297858.3304023. URL https://doi.org/10.1145/3297858.3304023. 12, 54, 56, 68

[79] Wan-Hsuan Lin, Jason Kimko, Bochen Tan, Nikolaj Bjørner, and Jason Cong. Scalable optimal layout synthesis for nisq quantum processors. In *Design Automation Conference (DAC)*, 2023. 16

[80] Florian Lonsing and Uwe Egly. Depqbf 6.0: A search-based QBF solver beyond traditional QCDCL. In *CADE*, volume 10395 of *Lecture Notes in Computer Science*, pages 371–384. Springer, 2017. doi: 10.1007/978-3-319-63046-5_23. 27, 89, 96, 118, 130

[81] Florian Lonsing and Uwe Egly. Qratpre+: Effective QBF preprocessing via strong redundancy properties. In *SAT*, volume 11628 of *Lecture Notes in Computer Science*, pages 203–210. Springer, 2019. doi: 10.1007/978-3-030-24258-9_14. 89, 95

[82]   Arman Masoumi, Megan Antoniazzi, and Mikhail Soutchanski. Modeling
       organic chemistry and planning organic synthesis. In *GCAI*, volume 36 of
       EPiC Series in Computing, pages 176–195. EasyChair, 2015. URL `https://easychair.org/publications/paper/cQB6`. Benchmarks available
       `https://www.cs.ryerson.ca/mes/publications/`. 46

[83]   Rami Matloob and Mikhail Soutchanski. Exploring organic synthesis with state-
       ofthe-art planning techniques. 2016. URL `https://api.semanticscholar.org/CorpusID:20528568`. 36

[84]   Valentin Mayer-Eichberger and Abdallah Saffidine. Positional games and QBF:
       The corrective encoding. In *Theory and Applications of Satisfiability Testing
       (SAT)*, pages 447–463, 2020. 18, 20, 21, 77, 80, 87, 89, 93, 104, 119

[85]   Drew V. McDermott. The 1998 AI planning systems competition. *AI Mag.*, 21
       (2):35–55, 2000. doi: 10.1609/aimag.v21i2.1506. URL `https://doi.org/10.1609/aimag.v21i2.1506`. 7, 55, 57, 105

[86]   D. Michael Miller, Robert Wille, and Zahra Sasanian. Elementary quantum gate
       realizations for multiple-control toffoli gates. In *2011 41st IEEE International
       Symposium on Multiple-Valued Logic*, pages 288–293, 2011. doi: 10.1109/ISMVL.2011.54. 54

[87]   Abtin Molavi, Amanda Xu, Martin Diges, Lauren Pick, Swamit S. Tannu,
       and Aws Albarghouthi. Qubit mapping and routing via maxsat. In *55th
       IEEE/ACM International Symposium on Microarchitecture, MICRO 2022,
       Chicago, IL, USA, October 1-5, 2022*, pages 1078–1091. IEEE, 2022. doi:
       10.1109/MICRO56248.2022.00077. URL `https://doi.org/10.1109/MICRO56248.2022.00077`. 12, 55, 70, 71

[88]   Juan Antonio Navarro Pérez and Andrei Voronkov. Planning with effectively
       propositional logic. In Andrei Voronkov and Christoph Weidenbach, edi-
       tors, *Programming Logics - Essays in Memory of Harald Ganzinger*, volume
       7797 of *Lecture Notes in Computer Science*, pages 302–316. Springer, 2013.
       doi: 10.1007/978-3-642-37651-1_13. URL `https://doi.org/10.1007/978-3-642-37651-1_13`. 50

[89]   Aina Niemetz, Mathias Preiner, Florian Lonsing, Martina Seidl, and Armin
       Biere. Resolution-based certificate extraction for QBF. In *Theory and Applica-
       tions of Satisfiability Testing (SAT)*, volume 7317 of *Lecture Notes in Computer
       Science*, pages 430–435. Springer, 2012. doi: 10.1007/978-3-642-31612-8_33.
       URL `https://doi.org/10.1007/978-3-642-31612-8_33`. 28, 106, 128,
       129, 130

[90]   Siyuan Niu, Adrien Suau, Gabriel Staffelbach, and Aida Todri-Sanial. A
       hardware-aware heuristic for the qubit mapping problem in the nisq era. *IEEE*

Transactions on Quantum Engineering, 1:1–14, 2020. doi: 10.1109/TQE.2020. 3026544. 54

[91] Charles Otwell, Anja Remshagen, and Klaus Truemper. An effective QBF solver for planning problems. In Hamid R. Arabnia, Rose Joshua, Iyad A. Ajwa, and George A. Gravvanis, editors, *Proceedings of the International Conference on Modeling, Simulation & Visualization Methods, MSV '04 & Proceedings of the International Conference on Algorithmic Mathematics & Computer Science, AMCS '04, June 21-24, 2004, Las Vegas, Nevada, USA*, pages 311–316. CSREA Press, 2004. 5

[92] Tom Peham, Lukas Burgholzer, and Robert Wille. On optimal subarchitectures for quantum circuit mapping. *ACM Transactions on Quantum Computing*, 4 (4), jul 2023. ISSN 2643-6809. doi: 10.1145/3593594. URL https://doi. org/10.1145/3593594. 55, 56, 68, 71

[93] Tomás Peitl, Friedrich Slivovsky, and Stefan Szeider. Polynomial-time validation of QCDCL certificates. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Proc. of the 21st Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2018)*, volume 10929 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2018. doi: 10.1007/978-3-319-94144-8_16. URL https://doi.org/10.1007/978-3-319-94144-8_16. 28, 128

[94] Tomás Peitl, Friedrich Slivovsky, and Stefan Szeider. Dependency learning for QBF. *J. Artif. Intell. Res.*, 65:180–208, 2019. doi: 10.1613/jair.1.11529. 96

[95] Luca Pulina and Martina Seidl. The 2016 and 2017 QBF solvers evaluations (qbfeval'16 and qbfeval'17). *Artif. Intell.*, 274:224–248, 2019. doi: 10.1016/ j.artint.2019.04.002. URL https://doi.org/10.1016/j.artint.2019. 04.002. 30, 125

[96] Luca Pulina, Martina Seidl, and Ankit Shukla. The 14th QBF solvers evaluation (QBFEVAL'22), 2022. URL http://www.qbflib.org/QBFEVAL22_PRES. pdf. 94

[97] Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023. 15, 55, 56

[98] Markus N. Rabe and Leander Tentrup. CAQE: A certifying QBF solver. In *FMCAD*, pages 136–143. IEEE, 2015. URL https://www.react. uni-saarland.de/publications/RT15.pdf. 28, 46, 89, 96, 118, 128

[99] Stefan Reisch. Hex ist PSPACE-vollständig. *Acta Informatica*, 15:167–191, 1981. doi: 10.1007/BF00288964. 79

[100] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res.*, 39:127–177, 2010. doi: 10.1613/jair.2972. URL https://doi.org/10.1613/jair.2972. 8

[101] J. Rintanen. Madagascar: Scalable planning with SAT. In *8th International Planning Competition*, 2014. URL `https://research.ics.aalto.fi/software/sat/madagascar/`. 10, 14, 46, 56, 68

[102] Jussi Rintanen. Constructing conditional plans by a theorem-prover. *J. Artif. Intell. Res.*, 10:323–352, 1999. doi: 10.1613/jair.591. 5, 51

[103] Jussi Rintanen. Partial implicit unfolding in the Davis-Putnam procedure for Quantified Boolean Formulae. In *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 2001. doi: 10.1007/3-540-45653-8_25. 5, 10, 36, 50, 51

[104] Jussi Rintanen. Asymptotically optimal encodings of conformant planning in QBF. In *AAAI*, pages 1045–1050, 2007. URL `http://www.aaai.org/Library/AAAI/2007/aaai07-166.php`. 5, 51

[105] Jussi Rintanen. Regression for classical and nondeterministic planning. In *ECAI*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 568–572. IOS Press, 2008. doi: 10.3233/978-1-58603-891-5-568. 9, 10, 36, 51

[106] Jussi Rintanen. Planning and SAT. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 483–504. IOS Press, 2009. doi: 10.3233/978-1-58603-929-5-483. URL `https://doi.org/10.3233/978-1-58603-929-5-483`. 9

[107] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.*, 170(12-13):1031–1080, 2006. doi: 10.1016/j.artint.2006.08.002. 9, 10, 36, 51

[108] Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. Sat-based parallel planning using a split representation of actions. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *ICAPS*. AAAI, 2009. URL `http://aaai.org/ocs/index.php/ICAPS/ICAPS09/paper/view/732`. 36

[109] Abdallah Saffidine, Nicolas Jouandeau, and Tristan Cazenave. Solving Breakthrough with race patterns and job-level proof number search. In *13th Advances in Computer Games (ACG)*, volume 7168 of *Lecture Notes in Computer Science*, pages 196–207, 2011. doi: 10.1007/978-3-642-31866-5_17. URL `https://doi.org/10.1007/978-3-642-31866-5_17`. 119

[110] Irfansha Shaik and Jaco van de Pol. Classical planning as QBF without grounding (extended version). *ArXiv/CoRR*, 2106.10138, 2021. URL `https://arxiv.org/abs/2106.10138`. Accepted at ICAPS-2022. 45, 49

[111] Irfansha Shaik and Jaco van de Pol. Classical planning as QBF without grounding. In *ICAPS*, pages 329–337. AAAI Press, 2022. 6, 56, 78, 84, 104

[112] Irfansha Shaik and Jaco van de Pol. Concise QBF encodings for games on a grid (extended version). *CoRR*, abs/2303.16949, 2023. doi: 10.48550/arXiv. 2303.16949. URL `https://doi.org/10.48550/arXiv.2303.16949`. 6

[113] Irfansha Shaik and Jaco van de Pol. Optimal layout synthesis for quantum circuits as classical planning. In *ICCAD'23*, San Diego, California, USA, 2023. IEEE/ACM. 6

[114] Irfansha Shaik, Maximilian Heisinger, Martina Seidl, and Jaco van de Pol. Validation of QBF Encodings with Winning Strategies. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:10, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-286-0. URL `https://drops.dagstuhl.de/opus/volltexte/2023/18486`. 6

[115] Irfansha Shaik, Valentin Mayer-Eichberger, Jaco van de Pol, and Abdallah Saffidine. Implicit state and goals in QBF encodings for positional games (extended version). *CoRR*, abs/2301.07345, 2023. doi: 10.48550/arXiv.2301. 07345. URL `https://doi.org/10.48550/arXiv.2301.07345`. 6, 32, 96, 104, 130

[116] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 124–134. IEEE Computer Society, 1994. doi: 10.1109/SFCS.1994.365700. URL `https://doi.org/10.1109/SFCS.1994.365700`. 53

[117] Ankit Shukla, Armin Biere, Luca Pulina, and Martina Seidl. A Survey on Applications of Quantified Boolean Formulas. In *Proc. of the 31st IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 78–84. IEEE, 2019. 125

[118] Silvan Sievers. Fast downward merge-and-shrink. *IPC-9 planner abstracts*, pages 85–90, 2018. 56, 68

[119] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintão Pereira. Qubit allocation. In Jens Knoop, Markus Schordan, Teresa Johnson, and Michael F. P. O'Boyle, editors, *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, pages 113–125. ACM, 2018. doi: 10.1145/3168822. URL `https://doi.org/10.1145/3168822`. 54

[120] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009. doi: 10.1007/978-3-642-02777-2_24. 47

[121] David Speck. Symbolic search for optimal planning with expressive extensions, 2022. URL `https://freidok.uni-freiburg.de/data/225448`. 9

[122] Stefan Staber and Roderick Bloem. Fault localization and correction with QBF. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*, pages 355–368. Springer, 2007. doi: 10.1007/978-3-540-72788-0_34. URL `https://doi.org/10.1007/978-3-540-72788-0_34`. 29

[123] Bochen Tan and Jason Cong. Optimal layout synthesis for quantum computing. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020*, pages 137:1–137:9. IEEE, 2020. doi: 10.1145/3400302.3415620. URL `https://doi.org/10.1145/3400302.3415620`. 12, 16, 55, 56, 67, 68

[124] Bochen Tan and Jason Cong. Optimality study of existing quantum computing layout synthesis tools. *IEEE Trans. Computers*, 70(9):1363–1373, 2021. doi: 10.1109/TC.2020.3009140. URL `https://doi.org/10.1109/TC.2020.3009140`. 54, 69

[125] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. 1983. doi: 10.1007/978-3-642-81955-1_28. 80, 88, 105

[126] Jos W. H. M. Uiterwijk. Perfectly solving Domineering boards. In *Workshop on Computer Games, (CGW)*, volume 408 of *Communications in Computer and Information Science*, pages 97–121. Springer, 2013. doi: 10.1007/978-3-319-05428-5_8. URL `https://doi.org/10.1007/978-3-319-05428-5_8`. 120

[127] Jos W. H. M. Uiterwijk. 11 \times 11 Domineering is solved: The first player wins. In *9th International Conference Computers and Games (CG)*, volume 10068 of *Lecture Notes in Computer Science*, pages 129–136. Springer, 2016. doi: 10.1007/978-3-319-50935-8_12. URL `https://doi.org/10.1007/978-3-319-50935-8_12`. 120

[128] Davide Venturelli, Minh Do, Eleanor Gilbert Rieffel, and Jeremy Frank. Temporal planning for compilation of quantum approximate optimization circuits. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August*

19-25, 2017, pages 4440–4446. ijcai.org, 2017. doi: 10.24963/ijcai.2017/620. URL `https://doi.org/10.24963/ijcai.2017/620`. 13, 15, 54, 55, 71

[129] Davide Venturelli, Minh Do, Bryan O'Gorman, Jeremy Frank, Eleanor Rieffel, KE Booth, Thanh Nguyen, Parvathi Narayan, and Sasha Nanda. Quantum circuit compilation: An emerging application for automated reasoning. In *Proceedings of the 12th International Scheduling and Planning Applications Workshop (SPARK 2019)*, pages 95–103, 2019. 54, 55

[130] Toby Walsh. Challenges for sat and qbf. *Presentation at SAT*, 2003. 103

[131] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. Mapping quantum circuits to IBM QX architectures using the minimal number of SWAP and H operations. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 142. ACM, 2019. doi: 10.1145/3316781.3317859. URL `https://doi.org/10.1145/3316781.3317859`. 12, 14, 54

[132] Ralf Wimmer, Christoph Scholl, and Bernd Becker. The (D)QBF preprocessor HQSpre - underlying theory and its implementation. *J. Satisf. Boolean Model. Comput.*, 11(1):3–52, 2019. doi: 10.3233/SAT190115. 89, 95, 118

[133] Alwin Zulehner, Alexandru Paler, and Robert Wille. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 38(7):1226–1236, 2019. doi: 10.1109/TCAD.2018.2846658. URL `https://doi.org/10.1109/TCAD.2018.2846658`. 54

[134] Álvaro Torralba, Vidal Alcázar, Peter Kissmann, and Stefan Edelkamp. Efficient symbolic search for cost-optimal planning. *Artificial Intelligence*, 242: 52–79, 2017. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2016.10.001. URL `https://www.sciencedirect.com/science/article/pii/S000437021630114X`. 9