UNIVERSITEIT LEIDEN

MASTER THESIS

Solving Othello using BDDs

Author:
Stef VAN DIJK

First supervisor:
Dr. Alfons LAARMAN
Second supervisor:
Dr. Vedran DUNJKO

A thesis submitted in fulfillment of the requirements for the degree of MSc

in the

Leiden Institute of Advanced Computer Science

July 12, 2019

Universiteit Leiden

UNIVERSITEIT LEIDEN

Abstract

De Faculteit der Wiskunde en Natuurwetenschappen
Leiden Institute of Advanced Computer Science

MSc

Solving Othello using BDDs

by Stef VAN DIJK

Combinatorial games are deemed *solved* whenever the winner can be determined, according to perfect play. A reliable method to achieve this, is by applying retrograde analysis. In this thesis we focus on solving the combinatorial game Othello, using techniques originated in model checking. Symbolic model checking techniques, using Binary Decision Diagrams (BDDs), pushed the state of the art enormously. To determine if the field of combinatorial games also benefits from this technique, we present multiple ways to implement retrograde analysis using BDDs. Our best method turns out to reduce the peak of the BDD size by 99, 31%, compared to the naive algorithm, making this technique less resource-consuming. Using this technique, we are able to solve a small instance of Othello, up to a size of 4×5. Thus, for the first time, we realize retrograde analysis for a divergent game, which has been deemed to be infeasible. For 4×5 Othello, we can represent the set of immediately winning states with BDDs, using 1.13 bytes per state. This number decreases when increasing the size of the game, up to 0.0175 bytes per state for 5×6 Othello. This is less than the memory needed for perfect hashing (a current technique for solving combinatorial games), for which a lower bound comes down to one bit per state. By extrapolation, we estimate an order of $10^{-}5$, to be an upper bound for the number of bytes required, to store a BDD representing the set of immediately winning states.

Contents

Chapter 1

Introduction

Solving games has always been a hot topic in computer science, since there have always been harder games to solve when the simple ones had already been solved. Maybe, some games, like chess, will never be solved. Research in games is done in two different ways. In the field of Aritificial Intelligence, researchers try to make game-playing computers which use heuristics to beat their opponent. These heuristic methods are not exact and come with a trade-off between speed and accuracy. Mainly the goal here is to beat human players or other computers playing the game. An important example here is the neural network *AlphaGo* [24], which plays *Go*.
In this thesis, the focus lies on the other way of research in games, where the games are solved exactly. This means that the winner of the game is determined, from the perspective of the starting player, according to perfect play. This can only be achieved in *combinatorial* games, which have perfect information, i.e. there exist no chance elements (like dice) and no hidden information (like cards).

1.1 Retrograde Analysis

We try to solve the game *Othello*, using retrograde analysis. Retrograde analysis is a technique for solving games, where a position of the game is called a *state*. By making moves, the game traverses from one state to another, such that the *state space* is explored. Retrograde analysis starts at a set of winning (or goal) states and reasons backward.
When games are being solved using state space exploration, it is related to the field of *Model Checking*. Model checking is an automatic method used to decide whether a program meets its specification. It exhaustively checks every state that could occur in the model (the state space) for possible deviations from prescribed behavior. This makes model checking a time and space consuming process, particularly when during exploration, the state space gets too big and there must be dealt with *state space explosion*. Since this is the case in many real-world problems, model checking techniques exist which reduce this explosion. An example of such a technique is introduced in Section 1.3.

1.2 State of the Art

In 2002, an overview of the solved games was given by van den Herik et al. [11], where Othello is said to be not solved yet for the default version on an 8×8 board. However, smaller instances, such as the 6×6 board, have been solved by Takeshita et al. [26].
Heule et al. [12] state that the solvability of a game depends on whether procedures could be applied to reduce the size of the state space to a reasonable complexity.

They say that if there is no such procedure, the game can only be solved if the state space is below a reasonable complexity, such that it can be solved by simple techniques. Nowadays, the state space complexity is reasonable whenever it is somewhere below 10^{20} states. Van den Herik et al. [11] show that Othello has a state space of 10^{28} states. This would mean that without such a procedure to reduce the size of the state space, Othello cannot be solved using modern computers. Van den Herik et al. also state that Othello is immune to many retrograde analysis techniques. Othello would be immune to these techniques since it is a divergent game. In divergent games the state space grows during the game, since there exists a big set of end states, where the game is immediately won by one player. Therefore, knowledge-based techniques should be used. However, they also state that Othello belongs to category 2 in Figure 1.1, which seems rather contradictory. This indicates that Othello is hard to solve. Still, they expected Othello to be solved in 2010, which has not been the case.

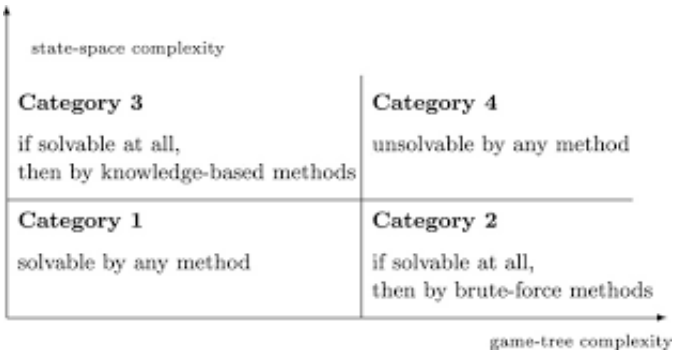

FIGURE 1.1: Double dichotomy of the game space, by van den Herik et al. [11]

1.3 Our approach

In this thesis, we try the existing data structure *Binary Decision Diagram* (BDD) as such a procedure that reduces the size of the state space. This data structure is being used nowadays in the field of model checking and has led to celebrated results [6, 5], where in the latter billions of states are represented by using only a few megabytes of memory. A BDD is data structure that can efficiently represent and manipulate *Boolean functions*. Since program states as well as program transitions can be encoded as Boolean functions, the data structure has been successfully used for program verification. In particular in model checking, where the entire reachable state space of a program can be computed in BDDs, sometimes leading to exponential reduction in the state space.
Before this technique was used in model checking, the state space was mostly represented explicitly, where each state is represented individually, which led to state space explosion in many cases when applied to big real-world problems. Using BDDs the state space can be represented symbolically. Symbolic model checking manipulates whole sets of states at once, by representing them in a BDD. Using basic set operations like intersection and union these sets can be easily modified by the BDD representing them.
In this thesis, we combine retrograde analysis and symbolic model checking to try to solve the board game Othello using BDDs and explain why it is so hard. We encode

the move relation of Othello in a BDD and provide various methods to compute retrograde analysis with BDDs for solving the game.
To check the effectiveness of this approach and to compare the different methods, we perform various experiments. These experiments show differences in the sizes of the BDDs (and thus speed of the algorithms used). This gives results in what variable ordering works for Othello and what algorithm works best for solving Othello. These results show evidence that we can solve a divergent game (4×5 Othello) using retrograde analysis, while van den Herik et al. [11] state that divergent games are deemed infeasible to solve using retrograde analysis. Also, we find big differences in terms of BDD size for different search strategies.

1.4 Contributions

One of the main contributions this thesis has to offer, is combining the fields of retrograde analysis and symbolic model checking. We give a thorough description on both retrograde analysis for solving games and binary decision diagrams. Later, we combine these two in order to solve the game Othello, where we also apply different methods on top of our approach. We find that using both these methods, we can reduce the needed memory by $99,31\%$, such that we can solve Othello up to 4×5 instances. This gives insights in how a game like Othello reacts to BDDs, such that in the future 8×8 Othello can maybe be solved by techniques introduced in this thesis.

1.5 Overview

This thesis maintains the following structure. Chapter 2 gives the preliminary knowledge required for this thesis, such as an introduction to the game Othello and an explanation of how BDDs work. Next, Chapter 3 gives related work to this topic, in order to better understand the state of the art. Chapter 4 explains how we encode a transition relation for Othello. Chapter 5 then validates the correctness of this encoding, using a counting argument. This chapter is followed by Chapter 6, where the algorithms are introduced, used to solve Othello. Then, Chapter 7 shows the experiments that are performed in order to gain knowledge on this topic, together with their results. Finally, the conclusions are given in Chapter 8.

Chapter 2

Background

This chapter gives background information needed to understand this thesis. In case the reader is highly familiar with the subjects described here, this chapter could be skipped.

2.1 Othello

Othello is a *combinatorial* [23] board game played by two players, *black* and *white* who alternately make moves. The game is played on a 8×8 board, comparable to a chess board. See Figure 2.1 for the initial board. All fields have unique identifiers, given by a letter $A - H$ followed by a number $1 - 8$. The row number is given by the letter, while the column number is given by the number. For example: the left-most black stone in the initial board is on $D4$ and the right-most black stone is on $E5$.

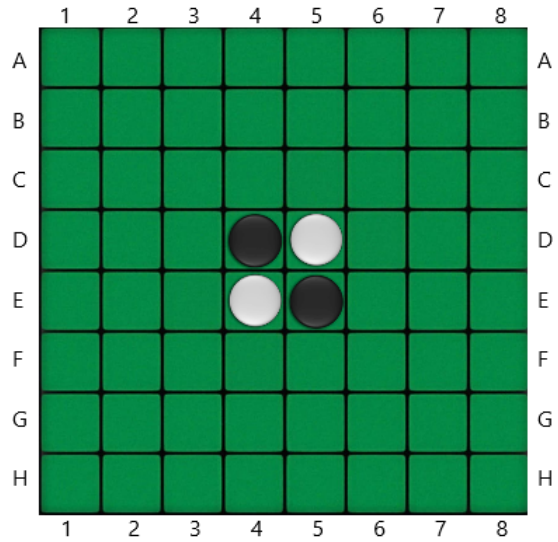

FIGURE 2.1: The initial position of a game of Othello, where black has turn.

Player black owns the black stones, while player white owns the white ones. The goal of the game is to possess more stones than the opponent when the game ends. This is the case when both players are not able to make a move.
In a move the player must add exactly one stone to the board on an empty field. This may only be one of the fields such that together with another stone of the same color, this stone encloses one or more stones from another color, in any (may be more than one) of the eight wind regions; *North*, *North-east* etc. We refer to these wind regions as *directions*. All these stones that are enclosed, change color in all directions where

enclosure is introduced by the move.
Player black must always make the first move. In the initial board of Figure 2.1, there are 4 possible moves for black which enclose (and turn) white stones in any direction, see Figure 2.2.

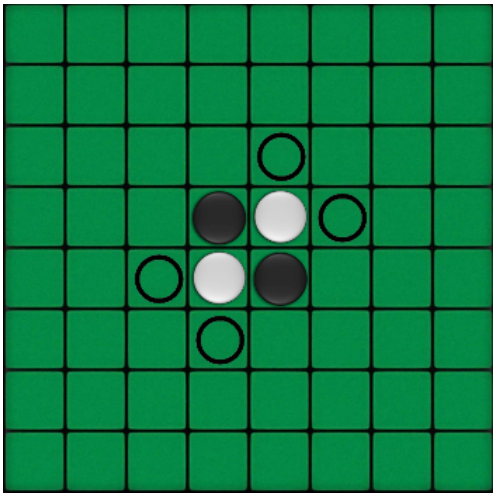

FIGURE 2.2: The 4 possible moves at the initial board are given by the black circles.

Then after placing the black stone on the east-most field, the board looks as in Figure 2.3

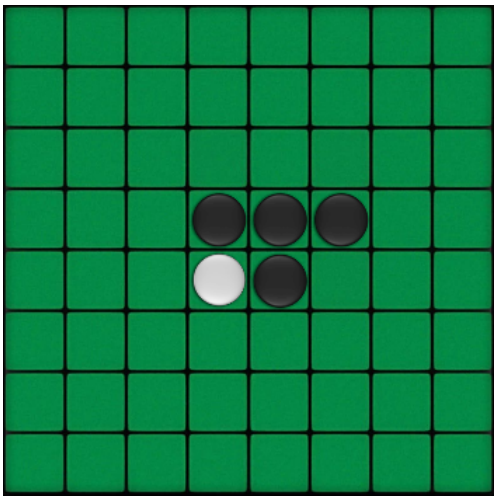

FIGURE 2.3: The board after black played on *D*6 in its first move.

Whenever a player cannot make any legal move, the turn goes to the other player without the board being changed. Lets call this move a *null-move*. If the other player also cannot make a move, then the game ends. This can only happen whenever:

- All $m \cdot n$ fields of the board are filled with stones.
- All the stones on the board have the same color.
- None of the above, but neither of the players can place a stone that encloses stones of the opponent.

Let us introduce the term *state* of an Othello game, where all information of a specific position is stored.

Definition 2.1. *A state A in $m \times n$ Othello is a tuple $(V^{m \cdot n}, p)$. Here, $V^{m \cdot n}$ is an $m \times n$ matrix, whose entries V_{ij} represent fields, with $V_{ij} \in \{B, W, E\}$. Here, B, W and E represent a black, a white or no stone on V_{ij} respectively. Besides this matrix, a state admits one extra Boolean for the turn variable $p \in \{0, 1\}$, specifying which player is to move. For convenience, we also refer to a field V_{ij}, by using the term A_{ij}.*

Let us give definitions for the number of stones of a color in an Othello state.

Definition 2.2. *We define the function $\#B(A)$, which maps an Othello state A to the number of black stones on A, as follows:*

$$\#B(A) \triangleq |\{(i, j) \mid A_{ij} = B\}|$$

Dually, we define the function $\#W(A)$, which maps an Othello state A to the number of white stones on A, as follows:

$$\#W(A) \triangleq |\{(i, j) \mid A_{ij} = W\}|$$

2.2 Solving Two-Player Games

In this section, we explain how to solve two-player games by the retrograde analysis method, given the transition relation (over pairs of states) and the required sets of states. We reason about two-player games, so we introduce *Player 0* and *Player 1* (P_0 and P_1) respectively.

Definition 2.3. *A two-player game is a 4-tuple $G \triangleq (S, I, T, E)$ with S the set of all states, I the set of initial states, T the transition relation and E the set of immediately winning states for either one of the players. The following properties hold for G:*

1. $S \triangleq S_0 \uplus S_1$ [1] *[S consists of two disjoint sets where P_0/P_1 have turn]*
2. $I \subseteq S_0$ *[P_0 starts the game]*
3. $T \triangleq T_0 \uplus T_1$ *[T consists of two disjoint relations for P_0/P_1]*
4. $T_0 \subseteq S_0 \times S_1$ *[Transition relation from S_0 state to S_1 state]*
5. $T_1 \subseteq S_1 \times S_0$ *[Transition relation from S_1 state to S_0 state]*
6. $E \triangleq E_0 \uplus E_1$ *[E consists of two disjoint sets, immediately winning for P_0/P_1]*

Besides E_0 and E_1, there exists one more set of states that immediately ends. This is the set immediate draw*, where the game has ended and both players do not win. This set is disjoint with E, since neither player wins the game. Joined together, they form the set of all immediately ending states.*

Example 1 (Othello as a two-player game)**.** See Section 2.1 for thorough a description of Othello. Othello is a two-player game, where player 0 possesses the black stones and player 1 the white. It is played on a $m \times n$ board, with the starting position as in Figure 2.1. A player's move consists of placing a stone of the player's color on a free field, such that any number of the opponent's stones is enclosed by two

[1] $A \uplus B$ denotes the union of two disjoint sets, such that $A \cap B = \emptyset$

player's stones. After a move, all enclosed stones are turned to the other color, as in Figure 2.3. Whenever no such move is possible for the player, the other player gets turn. The game ends whenever both players cannot make a legal move. Then, the player with the most stones on the board wins. We can model 8×8 Othello according to the definition of a two-player game as follows:

- $S_i \triangleq \{A = (V^{m \cdot n}, i)\}$, where V_{ij} is a field $\in \{B, W, E\}$ and $p \in \{0, 1\}$,
- $I \triangleq \{A = (V^{m \cdot n}, p)\}$, with $A_{ij} = \begin{cases} W, \text{ if } (i = 4 \wedge j = 5) \vee (i = 5 \wedge j = 4) \\ B, \text{ if } (i = 4 \wedge j = 4) \vee (i = 5 \wedge j = 5) \\ E, \text{ else} \end{cases}$
  and $p = 0$, since $I \subseteq S_0$,
- $T \triangleq S_0 \times S_1 \cup S_1 \times S_0$ such that $(s, s') \in T_0$ if we take for instance the board from Figure 2.2 as s and the board from Figure 2.3 as s'. A more detailed logic encoding of T is described in Section 4.3,
- $E_i \triangleq \{A = (V^{m \cdot n}, p) \mid A \in S_i, p \in \{0, 1\}, \nexists A'' \in S_i : (\exists A' \in S_{\bar{i}} : (A'' \neq A \wedge (A, A') \in T \wedge (A', A'') \in T)), \begin{cases} \#B(A) > \#W(A), \text{ if } i = 0 \\ \#W(A) > \#B(A), \text{ if } i = 1 \end{cases}\}$, with $i \in \{0, 1\}$.

A game is solved whenever it's determined which player can force a win from the initial state(s), according to perfect play [23]. There exist three levels of game solving [1]:

Ultra weakly solved For solving a game ultra weakly, there only needs to be determined to which of the three outcome classes the initial position belongs:

- $\mathscr{N}$ First player (the Next player) can force a win.
- $\mathscr{P}$ Second player (the Previous player) can force a win.
- $\mathscr{D}$ Both players cannot force a win, thus the game will end in an *immediate draw*.

Every state in a two-player game is in either one of these three classes. We refer to states in $\mathscr{N}$ and $\mathscr{P}$ as winning (depending on the perspective of which player), and to states in $\mathscr{D}$ as *draw*.

Weakly solved For solving a game weakly, also a strategy must be given by which the player can guarantee that outcome class. However, it does not always guarantee the best move possible for any position. For example, in a *draw* position, the strategy will never lose. However, when the opponent does not play perfectly, the position might be changed to a winning one. Here, it could be the case that the strategy still plays to a *draw*.

Strongly solved In order to solve a game strongly, for any position (reachable when the player with perspective plays optimal) the best move must be determined. Solving the game strongly comes down to finding the partition $\langle W_{P_0}, W_{P_1}, draw \rangle$ of S, where W_{P_0} is the set of states where P_0 can force a win, W_{P_0} is the set of states where P_1 can force a win and *draw* the set of states where both players cannot force a win, given that P_0 is the starting player. When this partition is known, we only have to check in which subset of the partition the initial state I is included. That subset determines which player wins the game in perfect play.

2.2.1 Reachability

Before we start focusing on retrograde analysis, let us first describe how we can navigate through the states of a game, by treating a game as a graph. The relation T is defined on the domain of $S \times S$. This means that we can now treat the state space as a directed acyclic graph (DAG) (S, T), where states are represented by nodes and transitions between states by edges. The initial states I are the root nodes. There exist also leaf nodes which have no successors in T. The states represented by those nodes are called *deadlocks*. Thus, deadlocks are those states in a game where the player cannot make any legal move. Often, deadlocks are immediately winning for either one of the players (if not *immediate draw*). Our definition of two-player games does not require deadlocks to be an immediately ending state. In this way, the encoding given in Chapter 4, can remain as simple as possible. See Figure 2.4 for an example of such a DAG, where the nodes represent states and the edges represent transitions.

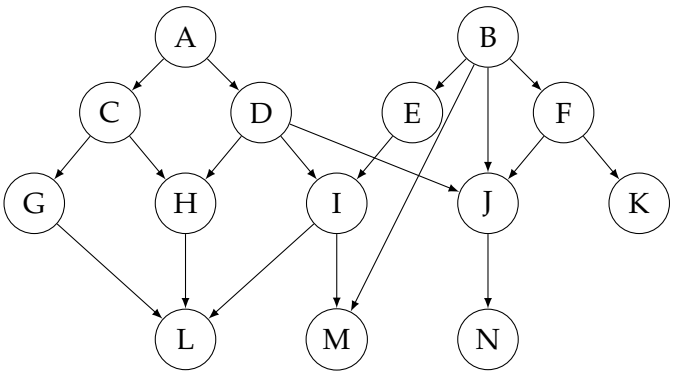

FIGURE 2.4: An example of a DAG representing a part of a game with two initial states; A and B, both $\in S_p$, with $p \in \{0,1\}$. The nodes represent states and the edges represent transitions between states. There are four leaf nodes in this example; L, M, N and K. The graph is directed and contains no cycles, therefore it satisfies the requirements of a DAG.

In Figure 2.4 we see that single states can have multiple successors according to T. We can also find the successors of a set of states, by the image of that set.

Definition 2.4. *Given a set of states $A \subseteq S$, the image of this set on a relation T, denoted by $Image_T(A)$, is the set of successor states B such that*

$$Image_T(A) \triangleq \{x' \in S \mid \exists x \in A : (x, x') \in T\}$$

This way, by calculating successors of states, the *reachability* of a set of states can be determined through the *Reflexive Transitive Closure* of T. Applying the reflexive transitive closure of T on I, denoted by T^*, gives the set $X \subseteq S$ of all states reachable from the initial states, such that

$$X = \{x \in S \mid \exists i \in I : (i, x) \in T^*\}$$

Also, using the inverse of the relation, the set of predecessors (the preimage) of a set of states can be determined.

Definition 2.5. *Given a set of states $B \subseteq S$, the preimage of this set on a relation T, denoted by $Preimage_T(B)$, is the set of predecessor states A such that*

$$Preimage_T(B) \triangleq \{x \in S \mid \exists x' \in B : (x, x') \in T\}$$

Note that $Preimage_T(B)$ will find all the states from which the player with turn can end up in X. Using the preimage the state space can be explored backward, which is called backward reachability.

2.2.2 Retrograde Analysis

For finding the sequence of moves that leads up to the best outcome for any game state, *Retrograde Analysis* can be used. In Retrograde analysis, we try to find the set of states W_{P_p} with $p \in \{0, 1\}$ that is winning for P_p. In order to find this set, we start with the *goal states* E_p, those immediately winning for P_p, and expand this set until convergence by reasoning backward.
As an example, see Figure 2.5 and start with the set E_p, where p defines the player and $\neg p$ the opponent. We call X the set that we are expanding, so X starts as E_p and ends as W_{P_p}. At any moment X represents the states where P_p can force a win. Then predecessor states of X are found wherein the player can force a play to X. In the first iteration, for a and b, P_p can directly play to X, so they are added to X. States in which the opponent has turn are only added to X if all successors states are in X. Thus, c is added to X, but d is not, since the opponent has a choice to move outside X.

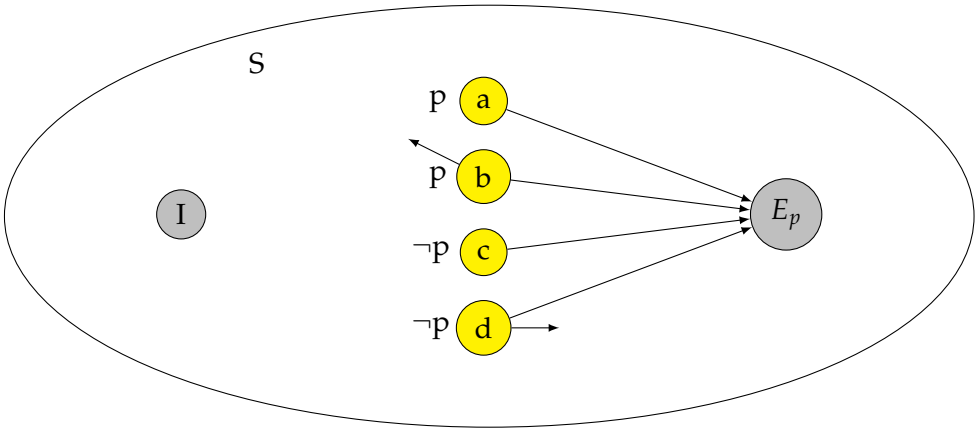

FIGURE 2.5: Part of the state space S of any game in a Venn-diagram. States are shown in yellow and sets of states are shown in gray. Arrows represent legal moves, such that an arrow to a set of states represents a move to any arbitrary state in that set.

This is an iterative process, which eventually converges when no new states can be added. Then the set X represents all states where the player can force a win. This comes down to the following two properties for all states added by this retrograde analysis technique:

1. $\forall x \in X$ where the player has turn: $\exists y : (y \in Image_T(x) \wedge y \in X)$.
2. $\forall x \in X$ where the opponent has turn: $\forall y : (y \in Image_T(x) \implies y \in X)$.

Here, states that obey to the first property can be found using the preimage function, where for the second property we introduce another function, $\forall\exists preimage_T$. For better understanding, let us first introduce $\forall preimage_T$:

Definition 2.6. *Given a set of states $B \subseteq S$ the $\forall$preimage of this set on a relation T, denoted by $\forall preimage_T(B)$ is the set of states A such that*

$$\forall preimage_T(B) \triangleq \{x \in S \mid \forall x' \in Image_T(x) : x' \in B\}$$

Then we can introduce $\forall\exists preimage_T$, where in contrast to $\forall preimage_T$, deadlock states are not found:

Definition 2.7. *Given a set of states $B \subseteq S$ the $\forall\exists$preimage of this set on a relation T, denoted by $\forall\exists preimage_T(B)$ is the set of states A such that*

$$\forall\exists preimage_T(B) \triangleq \{x \in S \mid \forall x' \in Image_T(x) : x' \in B, x \in Preimage(B)\}$$

Note that $\forall\exists preimage_T(B)$ will find all states such that no matter what move the player with turn makes, after moving, the game ends up in a state in B.
Finding W_{P_0} by expanding X is an iterative process where preimage and $\forall\exists$preimage are applied alternately (due to Definition 2.3, where two disjoint relations are defined). This process stops when no new states can be found for which the properties 1 and 2 hold, then we speak of *convergence*. After this convergence the set X defines a set of states where the player can force a win, which is the set W_{P_0}, needed to strongly solve the game.
Now, identifying W_{P_0} comes down to fixpoint computation [27]. A fixpoint y of a function f is defined to be an element within the functions domain, such that $y = f(y)$. In set computation we speak of a fixpoint of a function f whenever the image on f of set of states is equal to the the set itself.

Definition 2.8. *For any function on sets of states $f : 2^S \to 2^S$, a fixpoint is a set of states $X \subseteq S$ such that $f(X) = X$, with S all states in the space.*

In order to find W_{P_p}, with $p \in \{0, 1\}$, we need to find the least (smallest) fixpoint X on f, containing E_p, such that $f(X) = X$, where $f(X)$ is defined as

$$f(X) \triangleq E_p \cup Preimage_{T_p}(X) \cup \forall\exists preimage_{T_{\neg p}}(X) \tag{2.1}$$

By the Knaster-Tarski Lemma [27], we know that this least fixpoint exists, since f is a monotone function and $\langle 2^S, \subseteq \rangle$ is a complete lattice [2].
Note that, in retrograde analysis, $f(X)$ returns exactly the set wherein all S_p nodes can move to X and all $S_{\neg p}$ nodes can only move to X, including the immediately winning states for P_p. Then, a least fixpoint X of this function means that

- X contains E_p,
- X contains all states that are recursively found by either $\forall a \in S_p : \exists b(b \in Image(a) \wedge b \in X) \implies a \in X$ or $\forall a \in S_{\neg p} : \forall b(b \in Image(a) \implies b \in X) \implies a \in X$, and
- X contains no other states.

It follows from the application of Knaster-Tarski [13, p. 241], that for any set S with n elements and any function on the power set of S $f : 2^S \to 2^S$, this least fixpoint of f can be found by $f^n(\emptyset)$, where f^n means applying f consecutively n times.
Since the least fixpoint we need to find must contain E_p, we use E_p instead of $\emptyset$ here.

See Algorithm 1 for an outline on how to compute this fixpoint for $p = 0$.

Algorithm 1: Finding the set W_{P_0} of winning states for P_0 by fixpoint computation.

Result: W_{P_0}
1 $W_{P_0} \leftarrow E_0$
2 $old \leftarrow \emptyset$
3 **while** $W_{P_0} \neq old$ **do**
4 $\quad old \leftarrow W_{P_0}$
5 $\quad W_{P_0} \leftarrow W_{P_0} \cup Preimage_{T_0}(W_{P_0})$
6 $\quad W_{P_0} \leftarrow W_{P_0} \cup \forall\exists preimage_{T_1}(W_{P_0})$
7 **end**

See Figure 2.6 for a visual representation of Algorithm 1 in a Venn-diagram. We initialize W_{P_0} with E_0. In iteration 1, first the S_0 states are added which have a successor in W_{P_0}. Next, the S_1 states are added which have a successor in W_{P_0}. Note that W_{P_0} is already expanded within the first iteration. The second iteration the same happens. This proccess keeps iterating until no new states are found, then the least fixpoint is found.

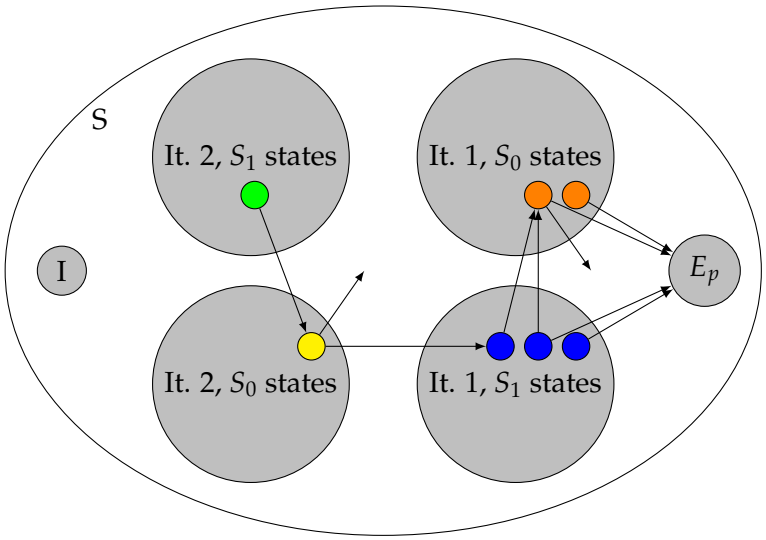

FIGURE 2.6: Venn-diagram in which Algorithm 1 is visualized, where in every iteration, the colored states are added to W_{P_0}. Before the first iteration, E_p is added to W_{P_0}. States are colored and sets of states are shown in gray. Arrows represent legal moves, such that an arrow to a set of states represents a move to any arbitrary state in that set.

.

The computation for finding the set of winning states of the other player (W_{P_1}) works similarly and the set of states where both players cannot force a win (*draw*) is equal to $S \setminus W_{P_0} \setminus W_{P_1}$. Using this we can find the partition $\langle W_{P_0}, W_{P_1}, draw \rangle$.

2.3 Binary Decision Diagram

Binary Decision Diagrams (BDDs) were introduced by Bryant [3] as an efficient way of representing Boolean functions. Briefly, the BDD is a tree-shaped data structure

that always has a maximum of 2 distinct leaf nodes, labeled *true* and *false*, or 0 and 1. All other nodes represent a variable that can be either true or false, which is specified in the tree by the two outgoing edges every node has: the high edge when the variable is true and the low edge when the variable is false. Starting from the root node, this tree can have multiple paths ending in the true leaf. All these paths are called *satisfying assignments*, where every variable has a value, determined by its outgoing edge in the path. All satisfying assignments make the function represented by the BDD evaluate to true, where all other paths make the function evaluate to false.
The following subsections give formal definitions of Boolean functions and the graphical representation of BDDs. For a more thorough introduction to BDDs, see the work of Somenzi [25].

2.3.1 Boolean function

Since BDDs represent Boolean functions $f : \mathbb{B}^n \to \mathbb{B}$, we assume the domain of such a function to be the set of n variables $x_1, \ldots, x_n$. In Ordered BDDs (OBDDs), this set of variables is ordered from x_1 to x_n. In this thesis, we only speak about OBBDs and refer to them as BDDs. Boolean functions evaluate to 0 or 1 for any given configuration of the variables. When assigning one of the variables x_i in the domain of the function f a value b (0 or 1, since the function is Boolean), the function gets restricted. We then speak of the *cofactor* of f, written by: $f_{x_i=b}$. Thus, we have either $f_{x_i=0}$ or $f_{x_i=1}$ which are defined as follows.

$$f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \tag{2.2}$$

and

$$f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) \tag{2.3}$$

Since f is a function, it can be seen as the set of configurations of its variables, for which f evaluates to 1. We call this set of configurations the set of satisfying assignments s_f, with

$$s_f \triangleq \{(x_1, \ldots, x_n) | f(x_1, \ldots, x_n) = 1\} \tag{2.4}$$

We interpret BDDs as sets and formulas dually. Therefore, we do not use the convention s_f, but instead write the A and B to denote the set of satisfying assignments for the BDDs A and B, where A and B are BDDs representing f. Therefore, we write $A \cup B$ of $A \vee B$ to denote the union of the set of satisfying assignments of BDDs A and B, depending on the context.

2.3.2 Graphical representation

This subsection focuses on how Boolean functions are represented by the graphical structure of a BDD. A BDD can be reduced (RBDD), ordered (OBDD) or both (ROBDD). The concept of reducing BDDs will be explained later.

Definition 2.9. *An* Ordered Binary Decision Diagram *is a directed acyclic graph* $G \triangleq (V, E)$*, with nodes* V *and edges* E *over an ordered set of variables* $x_n = \{x_1, x_2, \ldots, x_n\}$*. The following properties hold for* G*.*

- *G has a single root node r.*
- *Root node r, with $l(r) = x_1$, is called the first* level *of the tree, where its children are level* 2*, etc.*

- *There exist two terminal nodes;* True *and* False. *All other nodes v are non-terminal and represent a specific variable $x \in x_n$, specified by the label function $l(v)$.*
- *All nodes of the same level represent the same variable x.*
- *For every node v, holds that every child of v is either a terminal node, or a node u, such that $l(v) < l(u)$ in the ordered set of variables.*
- *Every node v has exactly two outgoing edges;* high(v) *represents the path where v is assigned true, where* low(v) *represents the path where v is assigned false.*
- *Every path in the OBDD which leads to the terminal-node true, represents exactly one satisfying assignment of the variables for the function. All other paths lead to false.*

See Figure 2.7 for an example of a OBDD. The low edges are usually given by a dotted line while the high edges are given by a solid line.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

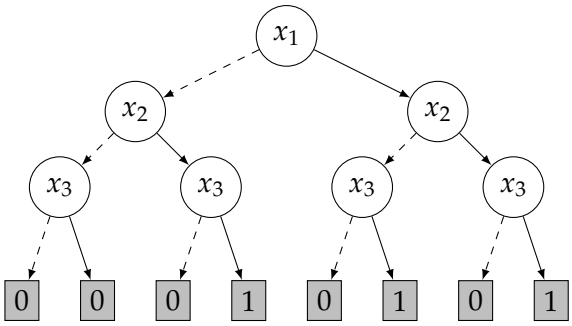

FIGURE 2.7: An example of a OBDD representing the function f, together with its corresponding truth table.

OBDDs can be reduced in terms of number of nodes, pretty easily, while still keeping the exact same information. An OBDD is called a reduced OBDD whenever $low(x_i) \neq high(x_i)$ holds for all nodes with label x_i, for all i and all nodes are unique (there exists no pair of nodes a and b such that $l(a) = l(b)$, $low(a) = low(b)$ and $high(a) = high(b)$). These two requirements are called *non-redundancy* and *uniqueness* respectively. In order to force non-redundancy, nodes are *deleted*, while in order to force uniqueness, nodes are collapsed. For deleted nodes, the OBDD acts as if the variable can have both values. See Figure 2.8 for the reduced variant of the OBDD of Figure 2.7, by deleting and collapsing nodes.

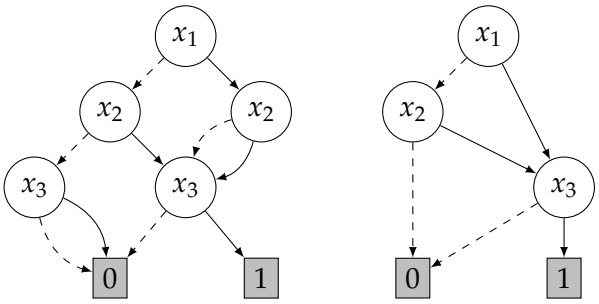

FIGURE 2.8: Reducing the OBDD of Figure 2.7, by collapsing nodes (left) and subsequently deleting nodes (right).

In the left OBDD of Figure 2.8, the terminal nodes and three x_3 nodes are collapsed. In the right part, one x_2 node and one x_3 node are deleted, yielding the

reduced OBDD. For the remainder of this thesis, we refer to a ROBDD with the term BDD.

2.3.3 Variable ordering

The order in which the variables appear in a BDD, can make difference in terms of their sizes. Note that this only holds for reduced BDDs, since unreduced BDDs always contain $2^{d+1} - 1$ nodes, with d the depth of the BDD, which is equal to the number of variables.
Take for example the BDD of Figure 2.8. If we change the variable ordering of this BDD to x_1-x_3-x_2, the BDD would contain 4 nodes instead of 3 to represent the same formula. See Figure 2.9 for that BDD.

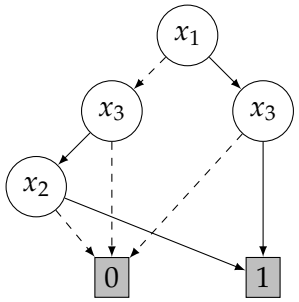

FIGURE 2.9: The BDD of Figure 2.8 with another (worse) variable ordering

2.3.4 Manipulating BDDs

BDDs alone merely provide the ability to efficiently represent boolean functions. The true power of BDDs comes from their ability to also manipulate these boolean functions in polynomial time in the number of BDD nodes. When manipulating BDDs, new BDDs are created by applying connectives, such as *disjunction*, *conjunction* and *negation*, on the operand BDDs. The given connectives are the most frequently used in this thesis. The disjunction of two BDDs A and B can be seen as the union $A \cup B$ (or $A \vee B$, by the dual interpretation of sets and formulas, see equation 2.4) on the sets of satisfying assignments represented by those BDDs. Dually, conjunction of two BDDs A and B can be seen as the intersection $A \cap B$ (or $A \wedge B$) on the sets of satisfying assignments represented by those BDDs. The connective negation ($\neg A$), has only an arity of one, which means that it operates on a single BDD. The result of negation comes down to interchanging the set of satisfying assignments with the set of non-satisfying assignments. Bryant [3] found that these three BDD operations can be executed in a time linear to the size of the operand BDDs, by using dynamic programming.
Besides these three operations, there exist two more ways to manipulate BDDs that are relevant to mention here: *existential quantification* and *universal quantification* [25]. Existential quantification $\exists_{x_i}$ on f, over $\vec{x} = \{x_0, \ldots, x_i, \ldots, x_n\}$, removes all nodes with variable label x_i and makes all paths skip this variable, such that the value of this variables doesn't matter anymore while the rest of the function represented is still preserved.

Definition 2.10. *$\exists_{x_i}$ on f, over $\vec{x} = \{x_0, \ldots, x_i, \ldots, x_n\}$, equals the smallest function independent of x_i, containing f:*

$$\exists_{x_i} f \triangleq f_{x_i=0} \cup f_{x_i=1}$$

Dually, we let's introduce the universal quantification $\forall_{x_i} f$, which gives the BDD where all nodes with variable label x_i are removed such that for all values of x_i, f is true.

Definition 2.11. *$\forall_{x_i}$ on f, over $\vec{x} = \{x_0, \ldots, x_i, \ldots, x_n\}$, equals the largest function independent of x_i, contained in f:*

$$\forall_{x_i} f \triangleq f_{x_i=0} \cap f_{x_i=1}$$

In practice, BDD manipulation is used for the following two purposes.

- The encoding of behavior of systems, such as the relation that is built in Chapter 4.
- Determining the reachability of systems, where the BDDs represent sets, as done in Chapter 7.

Chapter 3

Related Work

3.1 Chess

Kristensen [17] showed that BDDs are a useful data structure to represent a big set of states of a specific board game; *Chess*. He used BDDs to represent all 2-4 piece endgames using only 30 MB memory, which is close to the state of the art from Nalimov et al. [21], but the BDD approach has a significant advantage in the speed for requesting this memory.

3.2 Connect Four

Edelkamp et al. [9], focused on finding the reachable states for the game *Connect Four*, using BDDs. They found a variable ordering (see Subsection 2.3.3) for which the BDDs representing all reachable states were only polynomial in the board size. Their work also shows that overestimating the set of satisfying assignments can lead to smaller BDDs. When representing all reachable states, they compared stopping the search at immediately winning (terminal) states to ignoring terminal states and found that ignoring them (and thus representing more states), leads to smaller BDD sizes. See their corresponding results in Figure 3.1, where in every layer l, the BDD represents all states with l stones, i.e. after l moves in the game. For any layer $l \geq 30$, the number of states when ignoring terminal states is higher than when not ignoring them, but the number of nodes representing those states is smaller.

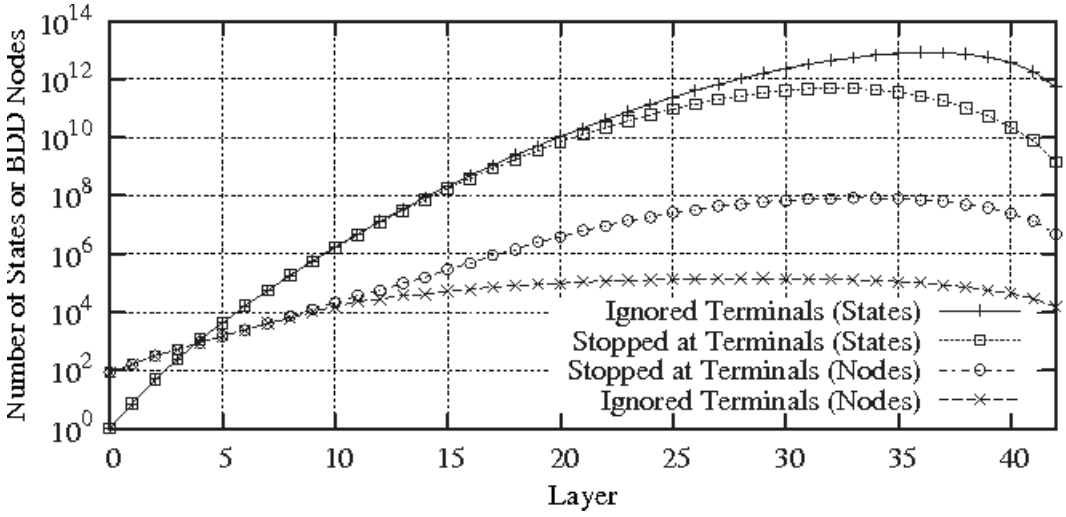

FIGURE 3.1: Results from edelkamp et al. for representing all connect four states using BDDs.

In 2008, Edelkamp et al. found the exact number of reachable Connect Four states within 15 hours, while later the exact same number was found using explicit search (without BDDs) in 10.000 hours. This shows that BDDs can be useful for representing states of a game.

3.3 Partitioned transition relation

Sometimes, in symbolic model checking, a transition relation can be too big in terms of BDD size. One way to reduce the size of the transition relation is by partitioning it into smaller BDDs. This method was introduced by Burch et al. [4], where they distinguish two kinds of transition relations: *synchronous* and *asynchronous*. Synchronous transition relations consist of various smaller relations which are applied consecutively, whereas asynchronous transition relations, simultaneously apply various smaller relations. Therefore, Burch et al. decided to partition the transition relation into smaller relations. For synchronous systems they found that the complete transition relation would remain intact when it is expressed as a conjunction of the partitioned relations, where for asynchronous systems that same holds using disjunction instead of conjunction. The benefit of this partitioning is that the conjuncted (or disjuncted) partitioned transition relation can have a smaller BDD size than the complete transition relation. Burch et al. were able to validate models using partitioned relations, where they were not able to validate the same models using the complete relation. For example, a BDD representing the complete transition relation of a synchronous system consisting of 340.000 nodes, consisted of only 2.500 nodes when using the partitioned transition relation. This is a reduction of factor 140.

3.4 BDD packages

Tools have been developed for creating and manipulating BDDs. These tools can be used to implement models using BDDs and manipulate (as described in Subsection 2.3.4, but there are more possible manipulations) those BDDs. We introduce two widely-used tools for BDD manipulation here.

3.4.1 CUDD

CUDD stands for Colorado University Decision Diagram, and is a package for the manipulation of BDDs and some other decision diagrams. This BDD package was introduced in 1997 by Somenzi [7].

3.4.2 Sylvan

In 2012, van Dijk et al. [8] introduced a multi-core BDD package for parallel symbolic model checking, called *Sylvan*. In constrast to CUDD, Sylvan performs its manipulations in parallel. For instance, sylvan is embedded in the LTSMIN model checker [16], which achieved some great results the past few years. For instance, LTSMIN participated in the RERS challenges [22] of 2012, 2013 and 2014, winning several first prizes.

Chapter 4

Encoding

For this thesis we apply retrograde analysis on Othello in order to solve the game. The algorithms used are described in Chapter 6. In this chapter, we explain how Othello states and the relations that are needed to apply retrograde analysis are encoded into BDDs.

4.1 Variables

In this section, variables are introduced that we refer to in other sections. All variables stated here, are introduced in the definition of an Othello state in Section 2.1.

m
: The length of an Othello board (i.e. the number of fields vertically stacked which is equal to the number of rows).

n
: The width of an Othello board (i.e. the number of fields horizontally stacked which is equal to the number of columns).

A_{ij}
: This variable needs two parameters $0 \leq i < m$ and $0 \leq j < n$ and represents the corresponding field on the board.

B, W and E
: The three possible values a field A_{ij} can have, black, white and empty respectively.

p
: The player to turn, with $p \in \{0, 1\}$.

4.2 Encoding a State

Recall the definition of a state from Section 2.1. A state consists of $m \cdot n$ fields, that all have one out of three possible values: *Black*, *White* or *Empty*. This means we cannot encode one field using only one Boolean variable, since this would allow us to only distinguish between two values. Therefore, one field is encoded by two variables, X_0 and X_1, yielding four possible outcome values. The first variable distinguishes between empty or not empty and the second between black or white. See Figure 4.1.

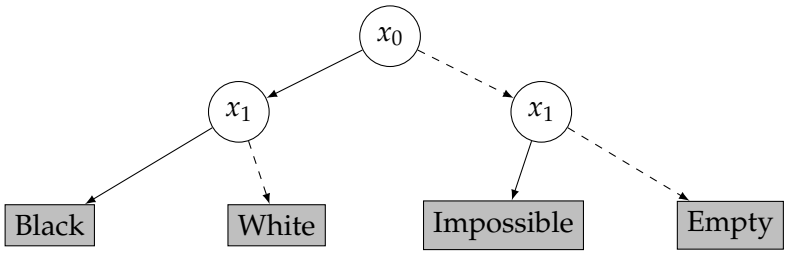

FIGURE 4.1: The encoding of the four possible values of a single field by a BDD.

Note that the value impossible is not needed here, and can easily be discarded by deleting the rightmost X_1 node and putting the terminal node with label empty at its place, as in Figure 4.2. This encoding (called *3T-encoding*) reduces the size of the BDD representing the value of a single field.

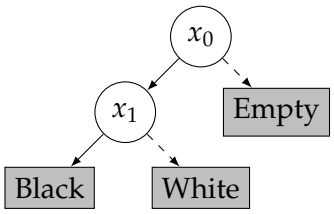

FIGURE 4.2: A different encoding (3T-encoding) of the three possible values of a single field by a BDD.

In order to represent a complete state (or set of states), for all $m \cdot n$ fields, the BDDs representing them must be connected into a bigger BDD. See Figure 4.3 for how this BDD representing a complete state is structured. This BDD uses the 3T-encoding, as described in Figure 4.2, but it is not a reduced BDD, as explained in Section 2.3.2. Reducing this BDD would remove many nodes. The first variable (p), determines which player has turn. Then two variables (x_{00a} and x_{00b}) are used to determine the value of field A_{00}. Whereas in every next level, two variables x_{ija} and x_{ijb} determine the value of the next field A_{ij}, such that x_{ija} distinguishes between E and $\neg E$ and x_{ijb} distinguishes between B and W, until the leaf nodes 0 and 1.

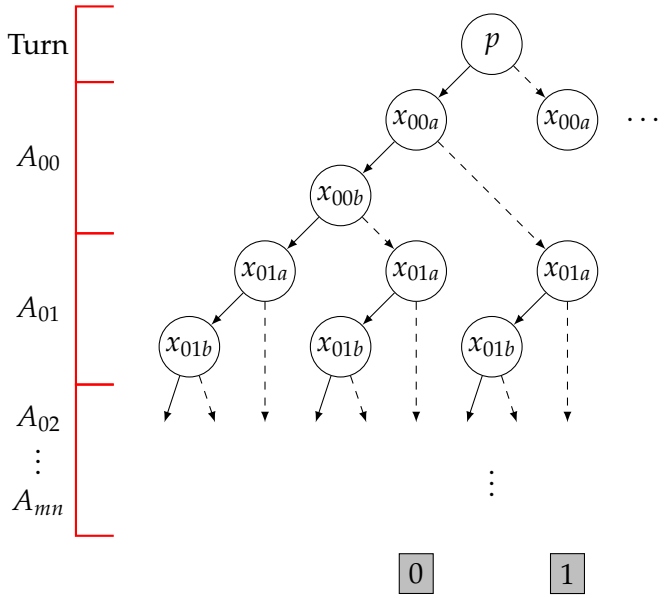

FIGURE 4.3: The structure of a BDD representing a complete (set of) state(s). The dots mean that the same pattern is repeated, for all A_{ij}, with $0 \leq i < m$ and $0 \leq j < n$. This BDD is uses the 3T-encoding and is not (yet) reduced, as it does not represent any particular set of states.

With the structure of the BDD of Figure 4.3, we can represent any set of states by evaluating the paths that represent those states to true and all other paths to false. The size of this BDD is fully dependent on the similarity of these states. For example, the set of all states S can be represented by a BDD with one node; true.
Next to the values of all $m \cdot n$ fields, a state also describes the next player to move. Since there are only two possibilities, only one Boolean variable is needed. This variable is now shown as the uppermost variable of the BDD, but in Chapter 7, we experiment with its location. This comes down to a total of $m \cdot n \cdot 2 + 1$ variables to describe a set of states.

4.3 Encoding the Transition Relation

Let x be state variables describing the Othello state, as defined in the previous section. In this section, we describe how to encode Othello moves in BDDs. Concretely, we show how to build a BDD T over variables x and x' , where x' are the primed copies of x and where all satisfying assignments of T represent tuples of states (y, z) in the move relation. Therefore, a pair of states (y, z) is called satisfied by the transition relation whenever the combination of all their variable assignments yields a satisfying assignment in the BDD.
We explain the construction of a BDD T by giving the logic formulas that represent T. Note that T is exactly the disjunction T_0 and T_1, thus we define these formulas over $p \in \{0, 1\}$, such that we construct T_p. Moreover, we use the variables introduced in Section 4.1, together with their primed versions. We use p' to indicate which player has turn in the next state (z) and A'_{ij} for referring to a field in the next state. We introduce the variable c_p which equals B when $p = 0$ and W when $p = 1$. When we say that $A_{ij} = c_p$, we mean that there is a stone of player p's color on that field. The negation, $c_{\neg p}$, means the opponents color. Besides these variables, also the term *encloser* is used for describing the encoding of the transition relation.

Definition 4.1. *For placing a stone with color c_p on an empty field A_{ij}, an encloser for that field is another field A_{kl} in any of the eight directions from A_{ij}, with color c_p and all stones between them with color $c_{\neg p}$.*

An example of a *eastern encloser* (i.e. an encloser east of A_{ij}) is given in Figure 4.4. An eastern encloser of A_{ij} is a field A_{id}, with $j + 2 \leq d \leq n - 1$, that is c_p and all fields A_{ie}, with $j < e < d$, are $c_{\neg p}$.

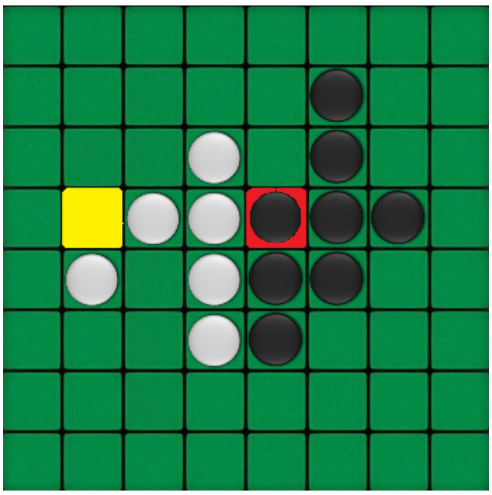

FIGURE 4.4: On this board, the red field is an eastern encloser for player black playing on the yellow field, since it encloses two white stones and turn them.

4.3.1 Partitioned transition relation

For simplicity, firstly the partitioned transition relation $T_{p_{ij}}$ is defined, which encodes a single move from a state y to a state z, playing on field A_{ij}. Playing on A_{ij} means putting a stone on that field. Thus, the partitioned relation $T_{p_{ij}}$ represents tuples (y, z) such that playing on A_{ij} is a legal move from state y to state z. When playing on A_{ij}, there is only a subset H_{ij} of fields that is relevant to the legality of the placement of a stone on A_{ij}. That is, the fields in the double cross through that A_{ij}, such that these are the fields in the eight directions from A_{ij}. Relevance here means that the value these fields matter to determine whether or not there exists an encloser for A_{ij}. All other field are only relevant in the way that they need to remain unchanged, whatever their value is. Thus, the partial relation $T_{p_{ij}}$ is defined over all variables x_{ijk}, with $k \in \{a, b\}$, while the relevant fields H_{ij} are those needed to determine the presence of an encloser. See Figure 4.5 for more information about the set H_{ij}.

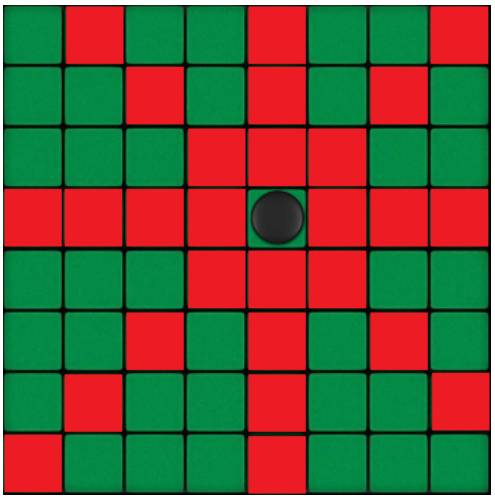

FIGURE 4.5: The double cross H_{34}, where all relevant fields for placing a (black) stone on A_{34} have been marked red. Relevant fields are those needed to determine the presence of an encloser.

Now, $T_{p_{ij}}$ can be built up step-by-step. In other words, we first focus on encoding all semi-legal moves, which we constrain later in this chapter, such that all illegal moves are removed from the encoding. Since playing on A_{ij} requires this field to be empty before the move and filled with c_p after the move and the turn must change, we start with

$$T_{p_{ij}} = p \wedge \neg p' \wedge (A_{ij} = E) \wedge (A'_{ij} = c_p) \tag{4.1}$$

What now needs to be checked is that in all of the eight directions, either there exists an encloser, or the complete direction remains unchanged. When there exists such an encloser, the relation should only satisfy the pair of states (y, z) if in z the enclosed states are turned to c_p and the encloser remains c_p. So we extend $T_{p_{ij}}$ for all eight directions as follows.

$$\begin{aligned} T_{p_{ij}} = {} & p \wedge \neg p' \wedge (A_{ij} = E) \wedge (A'_{ij} = c_p) \\ & \wedge east_{p_{ij}} \wedge south_{p_{ij}} \wedge west_{p_{ij}} \wedge north_{p_{ij}} \wedge northeast_{p_{ij}} \wedge southeast_{p_{ij}} \wedge southwest_{p_{ij}} \wedge northwest_{p_{ij}} \end{aligned} \tag{4.2}$$

where $east_{p_{ij}}$ is defined as

$$\overbrace{\bigvee_{j+2\leq d\leq n-1}\left(A_{id} = c_p \wedge A'_{id} = c_p \wedge \bigwedge_{j<e<d}(A_{ie} = c_{\neg p} \wedge A'_{ie} = c_p)\right)}^{\exists \text{ encloser}} \vee \overbrace{\bigwedge_{j+1\leq d\leq n-1}(A_{id} = A'_{id})}^{\text{unchanged}} \tag{4.3}$$

$south_{p_{ij}}$ defined as

$$\overbrace{\bigvee_{i+2\leq d\leq m-1}\left(A_{dj} = c_p \wedge A'_{dj} = c_p \wedge \bigwedge_{i<e<d}\left(A_{ej} = c_{\neg p} \wedge A'_{ej} = c_p\right)\right)}^{\exists \text{ encloser}} \vee \overbrace{\bigwedge_{i+1\leq d\leq m-1}\left(A_{dj} = A'_{dj}\right)}^{\text{unchanged}} \tag{4.4}$$

$southeast_{p_{ij}}$ defined as

$$\overbrace{\bigvee_{\substack{i+2\leq d\leq m-1\\ \wedge j+d-i\leq n-1}}\left(A_{d(j+d-i)}=c_p\wedge A'_{d(j+d-i)}=c_p\wedge\bigwedge_{i<e<d}\left(A_{e(j+e-i)}=c_{\neg p}\wedge A'_{e(j+e-i)}=c_p\right)\right)}^{\exists\text{ encloser}}$$
$$\vee\overbrace{\bigwedge_{\substack{i+1\leq d\leq m-1\\ \wedge j+d-i\leq n-1}}\left(A_{d(j+d-i)}=A'_{d(j+d-i)}\right)}^{\text{unchanged}} \tag{4.5}$$

and all other directions can be constructed similarly. In the three equations above, $\exists$ *encloser* denotes the part of the formula which enforces that there exists an encloser (and the enclosed stones are turned) in that direction and *unchanged* denotes the part of the formula which enforces that the complete direction remains unchanged.
Now, every time there exists an encloser in a certain direction, we want to enforce that the rest of that direction (beyond the encloser) remains unchanged. This means that for every direction, another clause must be added inside $\exists$ *encloser* in each formula. The formula for $east_{p_{ij}}$ now becomes

$$\bigvee_{j+2\leq d\leq n-1}\left(A_{id}=c_p\wedge A'_{id}=c_p\wedge\bigwedge_{j<e<d}\left(A_{ie}=c_{\neg p}\wedge A'_{ie}=c_p\right)\wedge\bigwedge_{d<f\leq n-1}\left(A_{if}=A'_{if}\right)\right)$$
$$\vee\bigwedge_{j+1\leq d\leq n-1}\left(A_{id}=A'_{id}\right) \tag{4.6}$$

where for $southeast_{p_{ij}}$ it becomes

$$\bigvee_{\substack{i+2\leq d\leq m-1\\ \wedge j+d-i\leq n-1}}\left(A_{d(j+d-i)}=c_p\wedge A'_{d(j+d-i)}=c_p\wedge\bigwedge_{i<e<d}\left(A_{e(j+e-i)}=c_{\neg p}\wedge A'_{e(j+e-i)}=c_p\right)\right.$$
$$\left.\wedge\bigwedge_{\substack{d<f\leq m-1\\ \wedge j+f-1\leq n-1}}\left(A_{f(j+f-i)}=A_{f(j+f-i)'}\right)\right)\vee\bigwedge_{i+1\leq d\leq m-1}\left(A_{d(j+d-i)}=A'_{d(j+d-i)}\right) \tag{4.7}$$

We still need to constrain the relation more. The current encoding always satisfies an unchanged direction. Instead, this must only be satisfied if there exists no encloser. Otherwise, the encoding of the relation satisfies pairs of states (y,z) where there exists an encloser in y, but in z nothing is changed in that direction. For example, the board in Figure 4.4 shows an eastern encloser for placement on A_{31}. Using the current encoding of the transition relation, both the board where the white stones are flipped, as the board where they remain unchanged, are satisfied. So we add the constraint

$$\wedge\neg\bigvee_{j+2\leq d\leq n-1}\left(A_{id}=c_p\wedge\bigwedge_{j<e<d}\left(A_{ie}=c_{\neg p}\right)\right) \tag{4.8}$$

to *unchanged* in the formula for $east_{p_{ij}}$. The formula for $east_{p_{ij}}$ now becomes

$$\bigvee_{j+2\leq d\leq n-1}\left(A_{id}=c_p\wedge A'_{id}=c_p\wedge\bigwedge_{j<e<d}\left(A_{ie}=c_{\neg p}\wedge A'_{ie}=c_p\right)\wedge\bigwedge_{d<f\leq n-1}\left(A_{if}=A'_{if}\right)\right)$$
$$\vee\left(\bigwedge_{j+1\leq d\leq n-1}\left(A_{id}=A'_{id}\right)\wedge\neg\bigvee_{j+2\leq d\leq n-1}\left(A_{id}=c_p\wedge\bigwedge_{j<e<d}\left(A_{ie}=c_{\neg p}\right)\right)\right) \tag{4.9}$$

Again, for the other directions a similar encoding is applied. At this point, there is only one more thing we need to enforce. Since any unchanged direction is satisfied, it is possible that all directions remain unchanged, which would always satisfy the pair of states where no field changes, no matter the values of the fields. To prevent this from happening, an extra constraint is added. In this constraint, we copy the formula for *unchanged* for every direction and index these formulas with $U_{direction}$. Then,

$$\wedge\neg\bigwedge_{\forall d\in directions}\left(U_d\right) \tag{4.10}$$

is added to the formula describing the partitioned transition relation. We now redefine the partitioned relation to T_{ij} as the union of the partitioned relations $T_{0_{ij}}$ and $T_{1_{ij}}$.
Since in the partitioned relation only the fields in H_{ij} were relevant, we must add the constraint such that all irrelevant fields remain unchanged. So the last clause we add to T_{ij} is:

$$\wedge\bigwedge_{A_{ij}\notin H_{ij}}\left(A_{ij}=A'_{ij}\right) \tag{4.11}$$

4.3.2 Encoding of complete relation using partitioned relations

With the encodings for the partitioned relations as described in the previous subsection, BDDs can be constructed (using the connectives described in Subsection 2.3.4) that represent these partitioned relations. Now, using the BDDs representing the partitioned relations T_{ij}, we can encode the complete relation by one disjunction over all these BDDs, as explained in Section 3.3 and [4].
However, there is one more possible move to make in Othello which could not be encoded in any partitioned relation. That is the null-move; whenever a player cannot make any move, it can make a move by doing nothing. Then the only thing that changes in the state is that the turn variable p swaps. Since in order to make such a move, it must be the case that playing on any other field is impossible, all fields become relevant for this move. Thus, this move cannot be encoded in the partitioned relations, since we need all partitioned relations to check if the null-move is legal.
In order to add this null-move to the formula, we first introduce a technique to find deadlock states. Deadlocks are states that have no successor in a relation. This technique works as follows: if we use $\forall preimage_{T_{ij}}(\emptyset)$ (the application of $\forall$preimage on BDDs specifically, is introduced in Equation 6.3) and conjunct this for all i and j, we get the states x where either all successors are in the empty set, or there exist no successors in x. Since it cannot be the case that a state has a successor in the empty set, we find all states that have no successors, i.e. the deadlocks states.
Now, we can conjunct the BDD representing the deadlocks to the BDD reperesenting the states where no field changes and disjunct this with the union of all partitioned

relations to obtain the complete relation T:

$$\bigvee_{0\leq i\leq m-1}\left(\bigvee_{0\leq j\leq n-1}(T_{ij})\right)\vee$$
$$\left(\bigwedge_{0\leq i\leq m-1}\left(\bigwedge_{0\leq j\leq n-1}\left(\forall preimage_{T_{ij}}(\emptyset)\right)\right)\wedge\bigwedge_{0\leq i\leq m-1}\left(\bigwedge_{0\leq j\leq n-1}\left(A_{ij}=A'_{ij}\right)\right)\wedge p\wedge\neg p'\right) \tag{4.12}$$

which can be simplified to:

$$\bigvee_{0\leq i\leq m-1}\left(\bigvee_{0\leq j\leq n-1}(T_{ij})\right)\vee\left(\bigwedge_{0\leq i\leq m-1}\left(\bigwedge_{0\leq j\leq n-1}\left(\forall preimage_{T_{ij}}(\emptyset)\wedge A_{ij}=A'_{ij}\right)\right)\wedge p\wedge\neg p'\right) \tag{4.13}$$

given that T_{ij} are the BDDs representing all partitioned relations as described in Subsection 4.3.1. Note that *deadlock relation* encodes solely the deadlock relation, which can also be seen as a partitioned relation, yielding a total of $m \cdot n + 1$ partitioned relations.

4.4 Encoding of the immediately winning states

For retrograde analysis, the transition relation as well as the immediately winning states are needed. The immediately winning states for player P_i (with $i \in \{0,1\}$) equals the set E_i, where the game has ended and player i possesses more stones than its opponent. To encode this set of states we use the transition relation T, whose encoding is defined in Section 4.3. By the existence of the null-move, a state can only be in E if the player with turn can only play the null-move and afterwards the opponent also can only play the null-move. So we are looking for the set of states where both players cannot place any stone if it would be their turn.
This set can be found by taking the BDD representing all deadlocks, as described in Equation 4.12, and applying universal quantification (see Subsection 2.3.4) on the variable p, in order to get the BDD representing E together with the set *immediate draw*. Then, within this set we have $E_p = \{A \in E \mid \begin{cases}\#B(A) > \#W(A), \text{ if } p = 0\\ \#W(A) > \#B(A), \text{ if } p = 1\end{cases}\}$, with $p \in \{0,1\}$.

Chapter 5

Validation

In Chapter 4, we described how the transition relation can be represented in a BDD or logic formula. In this chapter, we validate this representation by calculating the number of states we expect in these representations by theory and comparing it to the numbers found by testing our encoded representations. We test this by encoding the relation in a BDD and counting the number of satisfying assignments. Comparing this number to the theoretical number then tells us if the encoding is correct.

5.1 Validation of the partitioned transition relation

Here, we calculate e_{ij}, the theoretical number of pairs of states (y, z) that is satisfied by the partitioned relation T_{ij}. Later, we can compare this number to the number of satisfying assignments in the BDD representing the T_{ij}. Recall that the partitioned transition relation T_{ij} is defined as the relation for solely placing a stone on the field A_{ij}, over all variables x_{ijk}, with $k \in \{a, b\}$. First, note that some of the partitioned relations have a different value for e, since different positions A_{ij} in the grid lead to different sets of relevant fields H_{ij}. There are also different A_{ij} for which e_{ij} is equal, by the fact that the their positions in the grid are symmetrical (i.e. mirrored or rotated). Take for example the four corners fields. We expect T_{00}, T_{07}, T_{70} and T_{77} to have an equal value for e, since for any pair of boards that is satisfied by T_{00}, there exists another symmetric pair of boards that is satisfied by the other corner relations and vice versa. This way, there are more groups of fields which are symmetric with regard to the location on the grid. These groups are given in Figure 5.1, where fields with the same number have the same value of e.

1	2	3	4	4	3	2	1
2	5	6	7	7	6	5	2
3	6	8	9	9	8	6	3
4	7	9	10	10	9	7	4
4	7	9	10	10	9	7	4
3	6	8	9	9	8	6	3
2	5	6	7	7	6	5	2
1	2	3	4	4	3	2	1

FIGURE 5.1: An empty Othello board where mirrored or rotated fields have the same number.

Let's start the validation at the corner states. Recall that for validation we compute the number of pairs (y, z) that are satisfied by T_{ij}. Since we are validating the partitioned relation, any state y, has at most one state z such that (y, z) is satisfied by T_{ij} (since the relation is deterministic). Thus, in order to find the number of pairs (y, z), we only need to find the number of states y that have a successor z under T_{ij}. These are exactly the states where all of the following conditions hold.

C1 The board contains in at least one direction an encloser, such that $\exists(k, l) : A_{kl} = c_p \wedge \forall(i, j) < (r, s) < (k, l) A_{rs} = c_{\neg p}$, with p the player with turn $\in \{0, 1\}$.

C2 All relevant fields (part of H_{ij} as in Figure 4.5) can have any value $V \in \{B, W, E\}$.

C3 All irrelevant fields (not part of H_{ij} as in Figure 4.5) can have any value $V \in \{B, W, E\}$.

Now, take the relation T_{00}. Here, there exist three directions in which there can be an encloser. Any valuation of the set of fields of such a direction is called a *partial board*. See Figure 5.2 for those relevant fields.

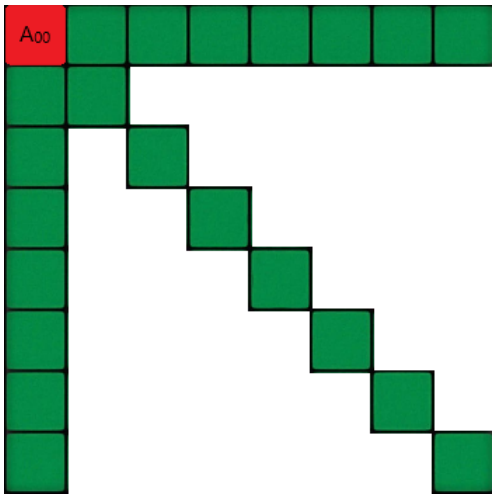

FIGURE 5.2: The relevant fields for computing e_{00}. For A_{00} there exist three directions with relevant fields.

In order to satisfy $C2$, there exist for a single direction 3^7 different partial boards. We have 3^7 different partial boards, since any of the seven fields in the direction can have one out of three possible values. This gives us $(3^7)^3$ valuations of the relevant fields (the green fields in Figure 5.2).
By $C1$, it cannot be the case that in all three directions, there is no encloser. Thus, in order to satisfy $C1$, we must subtract the number of valuations of the relevant fields, where in all three directions, this encloser is absent. For a single direction, the number of partial boards with an encloser is $\sum_{0 \leq w \leq 5} 3^w$. This is true since for any encloser (there can only be six different locations per direction), there is a single possibility for the fields until the encloser and three possibilities for all fields after the encloser. See Figure 5.3.

FIGURE 5.3: A partial board where the yellow field is the encloser for A_{00}. All fields after the yellow one are in $V \in \{B, W, E\}$. Also, on A_{00} (red) a stone must be placed with color equal to that of A_{02} (yellow) en opposite to that of A_{01}

If the encloser is on A_{02}, there are five fields after the encloser, which comes down to 3^5 possibilities.
Now we can compute the number of partial boards for one direction where there is no encloser: $3^7 - \sum_{0 \leq w \leq 5} 3^w$. Then, the number of partial boards where in all three directions there is no encloser becomes: $(3^7 - \sum_{0 \leq w \leq 5} 3^w)^3$. Note that the upper bound for w in this summation, must be equal to the number of fields in the direction (7) minus 2, since the first two positions after the encloser are always part of the enclosement and cannot be any V.
Still, C3 is not taken into account here. In order to satisfy this condition, all other 42 fields can have any value.
Also note that the state also contains a turn variable p. If we flip all stones in all computed partial boards and flip the turn variable, we get twice as many possible states y. Therefore, we multiply the formula by 2 at this point.
Thus, to reason over the number of complete states instead of the number of valuations for all fields in H_{ij}, the formula that satisfies all three conditions becomes the following.

$$e_{00} = ((3^7)^3 - (3^7 - \sum_{0 \leq w \leq 5} 3^w)^3) \cdot 3^{42} \cdot 2 = 8.803\,85 \times 10^9 \tag{5.1}$$

Now we have calculated the theoretical value of the number of pairs (i, j) in T_{ij} (e_{ij}) for the corner fields, let's calculate this number for one other group of relations: those with a 10 in Figure 5.1. These relations are never used in a normal game of Othello, since the game starts with stones on those fields en they can never be removed. However, in retrograde analysis they can be used in the search for states, such that a backward play is done on one of those fields, removing the stone. In Chapter 7, we evaluate this and test whether constraints can be made to reduce the size of the state space. For now, checking the value of e for these fields still gives us information on the correctness of our encoding for the partitioned relation. When doing the same as for calculating e_{00}, we get:

$$e_{33} = ((3^3)^5 \cdot (3^4)^3 - (3^3 - \sum_{0 \leq w \leq 1} 3^w)^5 \cdot (3^4 - \sum_{0 \leq w \leq 2} 3^w)^3) \cdot 3^{36} \cdot 2 = 1.681\,60 \times 10^{30} \tag{5.2}$$

This may look very different to the equation for e_{00}, but it is not. For calculating e_{00}, there were three relevant directions of length 7, where now there are five relevant directions of length 3 and three relevant directions of length 4. That is why both the left and the right side of the subtraction are splitted into two powers instead of one.
A general formula for validating partitioned relations can be constructed as follows:

$$e_{ij} = \left(\prod_{d \in D} \left(3^{l_d} \right) - \prod_{d \in D} \left(3^{l_d} - \sum_{0 \leq w \leq l_d - 2} (3^w) \right) \right) \cdot 3^h \cdot 2 \tag{5.3}$$

where

d a partial board for a relevant direction,

D the set of all relevant directions,

l_d the length of direction d (in number of fields),

h the number of irrelevant fields (those not in H_{ij}).

For those relations T_{ij} for which we calculated e_{ij}, we compared this number to the number of satisfying assignments in the BDDs that are built using our encoding. This concerns the the relations T_{00}, T_{07}, T_{33}, T_{34}, T_{43}, T_{44}, T_{70} and T_{77}. For their BDDs, the number of satisfying assignments matches the excpected number. All other encoding are constructed similarly, and we have validated the corner field and the middle fields. This gives us high confidence that the encoding is correct, based on our counting argument and the tests on our implemented encoding.

5.2 Validation of the complete transition relation

Since the complete transition relation is built up by calculating and intersecting the deadlock states for all partitioned relations, we cannot simply count the theoretical number of pairs of states (y, z) that is satisfied by the complete relation T, as done before with the partitioned relations. This counting cannot be done since an intersection of sets of states cannot be efficiently counted, given only the number of states per set. However, T can be validated as follows. For this, we look closely at how T is built up. From Subsection 4.3.2 follows that besides some minor subformulas (concerning fields to remain unchanged and swapping the turn variable), the correctness of T is solely dependent on the correctness of all partitioned transition relations T_{ij} and the function $\forall$preimage. In the previous section we validated the correctness of the encoding of all T_{ij} and we can assume that the $\forall$preimage function is defined correctly. Therefore we are convinced that the encoding of T is correct.

5.3 Validation of the immediately winning states

The set of immediately winning states can be partitioned into three categories:

1 Those states where there exist no empty fields.

2 Those states where one player has no stones on the board.

3 Those states which are non of the above, but are immediately winning for one player (see Figure 5.4).

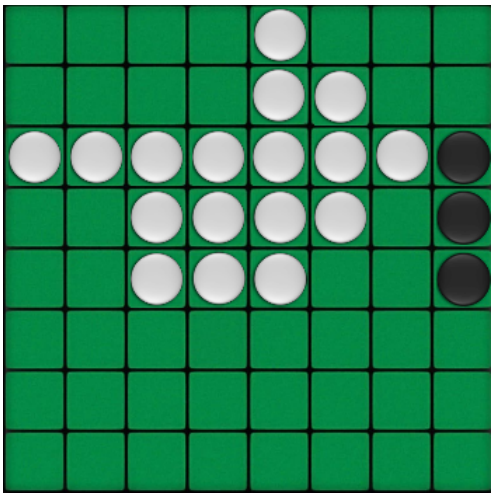

FIGURE 5.4: Immediately winning state for player white. This is an ending of a European Grand Prix game between Hassan and Verstuyft.

The first two categories of immediately winning states can be efficiently counted, but the third cannot. So for validating the winning states, we again lay our trust that ∀preimage is defined correctly and together with the validation of the partitioned transition relation this is sufficient to show the correctness of the computation of the set of immediately winning states.

Chapter 6

Algorithms for retrograde analysis

In this Chapter, we give multiple algorithms for implementing retrograde analysis using BDDs. These algorithms can reduce the needed space of the search, by using methods which store different intermediate results.
We distinguish two main approaches for searching the state space: *forward analysis* and *backward analysis*. For solving Othello, we need retrograde analysis and thus apply backward analysis. The main idea of retrograde analysis is already described in Subsection 2.2.2, where also an outline of an algorithm for finding a fixpoint is given in Algorithm 1. Although forward analysis itself cannot be used to solve a game, it can be useful to find all the reachable states, starting from the initial states. Besides knowing the size of the state space, this can be useful in the way that during retrograde analysis, we can limit ourselves to the reachable states only. Next to this, also the use of a *sweep-line* is described, where at any moment during the analysis, we only keep the states with i stones on the board, with i any monotone increasing or decreasing number such that $0 \leq i \leq m \cdot n$. The goal of these methods is to save memory, by obtaining smaller intermediate results.
As described in Subsection 2.3.1, in this chapter, we interpret BDDs as sets of states. Therefore, in the algorithms presented here, we use the operators union and intersection ($\cup$ and $\cap$) to denote operations on the sets of satisfying assignments of the operator BDDs.

6.1 Retrograde analysis with BDDs

As described in Section 2.2.2, retrograde analysis is a way of solving games by starting at the goal states and reasoning backward. Then a set of states is constructed in which a player can force the play to stay in that set. Since the set of states winning for that player (which were the goal states) are also in that set, this set represents exactly all winning states for that player.
Recall Algorithm 1, where preimage and $\forall\exists$preimage are applied alternately on the set W_{P_0}, expanding this set until convergence. This algorithm can be applied using BDDs to represent sets of states. Using BDD operations as *union*, W_{P_0} can be initialized and modified as supposed to. Since these operations only work on BDDs, we need every set or state we use, to be represented by a BDD. Therefore we consider the transition relation T as a BDD, with the encoding as described in Chapter 4. Also the partitioned relations T_{ij} are represented by BDDs T_{ij}, for all $1 \leq i, j \leq 8$.
Also, the functions image, preimage, $\forall$preimage and $\forall\exists$preimage have to be redefined as BDD operations. The image of A under T is computed by the BDD operation $Image_T(A)$:

$$Image_T(A) \triangleq (\exists \vec{x} : (A \cap T))[\vec{x}' := \vec{x}] \tag{6.1}$$

and the preimage is defined in a similar way:

$$Preimage_T(A) \triangleq \exists \vec{x}' : (A[\vec{x} := \vec{x}'] \cap T) \tag{6.2}$$

where A is a BDD on variables $\vec{x} = x_1, \ldots, x_n$, T is the transition relation and $\exists \vec{x}$ is the BDD operator existential quantification (see Section 2.3.4). The substitution and existential quantification are only applied on the relevant variables, i.e. those in both A and T. McMillan [19] shows that the quantification algorithms are $\mathcal{NP}$-complete, by proving that it belongs to $\mathcal{NP}$, and giving a reduction from the problem 3-SAT, which is also $\mathcal{NP}$-complete.
In the above equations, the function is given by three consecutive operations. However, in Sylvan, this image function is computed by one single traversal through the BDD, instead of three, which makes it more efficient, since it does not store intermediate results.
A single BDD operation for ∀preimage, is introduced by Huybers et al. [14] as follows:

$$\forall preimage_T(A) \triangleq \neg \exists \vec{x}' : (\bar{A}[\vec{x} := \vec{x}'] \cap T) \tag{6.3}$$

A BDD operation for ∀∃preimage can be obtained by using both the preimage and the ∀preimage operation:

$$\forall \exists Preimage_T(A) \triangleq \forall Preimage_T(A) \cap Preimage_T(A) \tag{6.4}$$

When using the partitioned relations T_{ij} instead of the complete relation, the operations Image, Preimage and ∀preimage are slightly different, since existential quantification is only applied on those variables that are in H_{ij}.

6.2 Limit to reachable states by forward analysis

Limiting the search to only the reachable states can reduce the BDD sizes, since fewer states have to be represented. Finding all reachable states from the initial states can be done with forward analysis by taking the initial states and consecutively add the image to the set of reachable states. This process stops when in a certain iteration, no new states are added to the set. We then speak of convergence. See Algorithm 2 for an outline for finding all reachable states R.

Algorithm 2: Finding the set of states reachable from the initial states

```
   Result: R
1  R ← I
2  old ← ∅
3  while R ≠ old do
4  |  old ← R
5  |  R ← R ∪ Image_T(R)
6  end
```

Combining Algorithm 1 and 2 gives us a retrograde analysis algorithm for finding W_{P_0}, while the search space is limited to the reachable states only, as in Algorithm 3.

Algorithm 3: Retrograde analysis limited to the reachable states

Result: W_{P_0}
$R \leftarrow I$
$old \leftarrow \emptyset$
while $R \neq old$ **do**
  $old \leftarrow R$
  $R \leftarrow R \cup Image_T(R)$
end
$W_{P_0} \leftarrow E_0 \cap R$
$old \leftarrow \emptyset$
while $W_{P_0} \neq old$ **do**
  $old \leftarrow W_{P_0}$
  $W_{P_0} \leftarrow W_{P_0} \cup Preimage_{T_0}(W_{P_0})$
  $W_{P_0} \leftarrow W_{P_0} \cup \forall\exists preimage_{T_1}(W_{P_0})$
  $W_{P_0} \leftarrow W_{P_0} \cap R$
end

In Chapter 7, we compare algorithms where the search space is constrained to those where all states are considered, in terms of BDD size.

6.3 The sweep-line method

Often, in model checking, the size of the state space shows a peak when plotted against iteration number. This means that the size of the state space is bigger somewhere during the algorithm then after the algorithm. Therefore, techniques have been developed to reduce this peak, such as the *sweep-line* method [15]. This method can reduce the memory usage by dividing all states into layers with the same amount of *progress*, where at any moment, only one layer is stored. In order to make this work, the given model needs to make progress in any way, when moving through the state space. This progress is defined by the *progress measure* as defined below, where T is the transition relation of the model and s and s' are states.

Definition 6.1. *A progress measure is a tuple* $\mathcal{P} \triangleq (O, \sqsubseteq, \psi)$ *such that* O *is a set of progress values,* $\sqsubseteq$ *is a total order on* O*, and* $\psi : S \rightarrow O$ *is a progress mapping.* $\mathcal{P}$ *is monotonic if* $\forall(s, s') \in T : \psi(s) \sqsubseteq \psi(s')$*. Otherwise,* $\mathcal{P}$ *is non-monotonic.*

Then if there can be found a monotonic progress measure for the model, then the sweep-line method can be applied, where each layer represents the states which all have the same progress value O. The sweep-line method traverses the state space in a *least-progress-first order*. Starting with the layer with the least progress value, the next layer will be explored when all states of the current layer are explored. Then, if this next layer is also explored, all the states of the current layer will be deleted from the memory. This way, all necessary states will be explored, while the memory usage can remain low.
Kristensen et al. [18] considered to use the sweep-line method on symbolic model checking, but since there is no correlation between the number of states and the size of the BDD representing it, they haven't applied it yet.
In Chapter 7, we experiment on this, by using the sweep-line method on our retrograde analysis algorithms in order to solve Othello. As a progress measure, we use a function which essentially counts the number of stones on a board. In a game of

Othello, this function is monotone, since a move does not allow to remove stones. We take $\mathcal{P} = (O, \sqsubseteq, \psi)$ as progress measure, with

O $\in \{0, \ldots, m \cdot n\}$, the number of stones on the board,

$\sqsubseteq$ the relation $\geq$ and

ψ the mapping $S \to O$, such that $\forall s \in S : \psi(s) = \#B(s) + \#W(s)$,

which is monotone. Note that we take $\geq$ instead of $\leq$, since for retrograde analysis we start with the goal states and we have to obey the least-progress-first order. Our algorithm for retrograde analysis using the sweep-line method is comparable to Algorithm 1, but in contrast to Algorithm 1, W_{P_0} is initialized to the immediately winning states with $m \cdot n$ stones on the board. Also, we iterate always exactly $m \cdot n$ times, where in each iteration only all states with the $i + 1$ stones are stored, and replaced by all boards with i stones on the board, with $i \in \{4, \ldots, m \cdot n - 1\}$. The outline can be found in Algorithm 4, where PT represents the union of all partitioned relations and DT represents solely the null-move relation, such that $PT \cup DT = T$. Also, we define $psi(X, i) = \{x \in X \mid \psi(x) = i\}$ as the BDD representing all states in X for which the progress measure equals i.

Algorithm 4: Retrograde analysis using a sweep-line on the number of stones on the board.

Result: $sweepline1$
1 $sweepline1 \leftarrow E_0 \cap psi(S, m \cdot n)$
2 **for** $i = m \cdot n - 1$ **to** 4 **do**
3 $\quad sweepline2 \leftarrow Preimage_{PT_0}(sweepline1)$
4 $\quad sweepline2 \leftarrow sweepline2 \cup \forall\exists preimage_{PT_1}(sweepline1)$
5 $\quad sweepline2 \leftarrow sweepline2 \cup Preimage_{DT}(sweepline2)$
6 $\quad sweepline1 \leftarrow sweepline2 \cup (E_0 \cap psi(S, i))$
7 **end**

In the above algorithm, in every iteration, $sweepline1$ represents the states with i stones on the board. Therefore, $sweepline1$ represents the set of winning states with four stones on the board, after termination. Then the winning player for Othello can be determined, by intersecting this set with the initial state. Note that this method solves the game ultra weakly, instead of strongly (see Section 2.2).
Every iteration, the set of winning states with i stones, are calculated and represented by $sweepline2$, while $sweepline1$ represents all winning states with $i + 1$ stones. In order to make both sweeplines always represent a set of states with a single progress value, the transition relation is partitioned into PT and DT. This way, we can force the algorithm to only find null-moves in Line 5.

Chapter 7

Experimental Evaluation

The main goal of our experiments is to come as close as possible to solving Othello. Their are multiple board sizes that can be solved, which indicate how far we are in solving the 8×8 instance of Othello. With our techniques, we are able to solve instances of Othello up to 4×5 boards. Thus, we analyze the algorithms introduced in Chapter 6 and compare the corresponding BDD sizes with each other. We also experiment on possible encodings, such as different variable orderings and the 3T-encoding of a field.
Since solving a game is bounded by the computation time and available memory, we try to keep the memory usage as low as possible. The memory usage is fully dependent on the sizes of the BDDs. The moment in an algorithm where the BDD sizes are the biggest, is called the *peak* of the algorithm. Thus, we try to keep the size of this peak low, in order to come closer to solving Othello. Also, we use these experiments to gain knowledge on our techniques, such that they can be reused for other purposes (e.g. solving other combinatorial games).

7.1 Experiments

In order to perform the experiments in this chapter, we implemented a tool in *c++* which operates on the BDD package Sylvan (Subsection 3.4.2. This tool applies the algorithms of Chapter 6 on the encoding of Othello, described in Chapter 4. The source code of this tool can be found on Github: `https://github.com/Stefvandijk/Othello_BDD.git`. The encoding in Chapter 4, is used to construct BDDs for:

- the initial state,
- all $m \cdot n + 1$ partitioned transition relations,
- the complete transition relation,
- the set of winning states.

Together with the implemented algorithms described in Chapter 6, these BDDs can be used to perform retrograde analysis on Othello. Now, we introduce and evaluate the experiments that might gain knowledge on how to solve Othello (and maybe other similar games). The experiments we perform in this chapter, compare BDD sizes and are only applied to Othello on 4×4 and 4×5 grids, since bigger grids require more resources than we have available. For all experiments performed in this chapter on 4×5 a board, we fixed the location of the four non-empty fields in the initial state as in Figure 7.1.

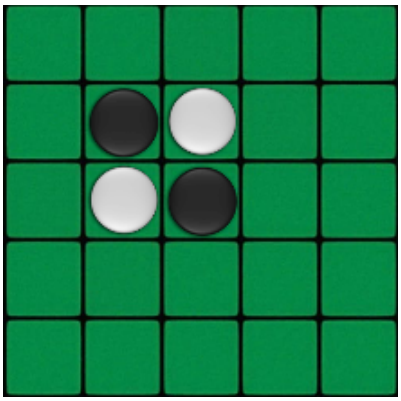

FIGURE 7.1: The initial state in 4×5 Othello, where we fixed the location of the non-empty fields this way.

7.1.1 Using the 3T-encoding for an empty field

In Section 4.2, we introduced a different encoding for BDDs representing states. In this 3T-encoding, an empty field is encoded by one variable instead of two, as in the standard encoding. In this section, we validate if this 3T-encoding results in smaller BDDs, compared to the standard encoding. In this experiment, we compare the BDD sizes for both encodings on three (arbitrary) BDDs, which represent: the initial state, the complete relation and the set of immediately winning states for one player. See table 7.1 for the results for 4×4 Othello.

BDD	#nodes using standard encoding	#nodes using 3T-encoding	compression
Initial	34	22	35.29%
Relation	215801	192422	10.83%
Winning	45478	32636	28.24%

TABLE 7.1: Comparison between the standard encoding and the 3T-encoding, for three (arbitrary) BDDs.

These results show that using a single variable for encoding an empty field results in smaller BDDs. It also shows that the percentage of saved space is fully dependent on the BDD itself, since it is dependent on the number of empty fields in the represented BDDs.

7.1.2 Comparing Algorithms

We hypothesize that the algorithms in Chapter 6 behave very differently when applied on Othello. However, we have no a priori criteria to decide which works best for solving the game. Therefore, we compare the performance of these methods by comparing the sizes of the peak BDDs. The performance is mostly limited by the size of the peak BDD, since the memory usage is particularly dependent on the biggest BDD size throughout the algorithm. Thus, this also concerns intermediate BDDs, instead of only the result BDD.
In Chapter 6, we gave several modifications to the standard retrograde analysis algorithm introduced in Subsection 2.2.2. We introduce the following algorithms and refer to these algorithms with Alg.

Alg 1	Finding the reachable states by forward analysis	(Algorithm 2)
Alg 2	Retrograde analysis	(Algorithm 1)
Alg 3	Retrograde analysis limited by the reachable states	(Algorithm 3)
Alg 4	Retrograde analysis with using the sweep-line method	(Algorithm 4)
Alg 5	Retrograde analysis limited by the reachable states using the sweep-line method	Not yet defined

Most of the algorithms introduced here are introduced before, but are now renumbered. *Alg 5* is a combination of *Alg 3* and *Alg 4*, where both the sweep-line method as well as the limitation to the reachable states are applied.
In order to gain knowledge on what algorithm works best for solving Othello, we compare the *Alg 2* to *Alg 5* with each other in terms of BDD size. Since *Alg 3* and *Alg 5* are bounded by finding the reachable states, we take the result of algorithm 1 also into account. Note that *Alg 1* is not able to solve Othello, but *Alg 3* and *Alg 5* are limited by this forward search. We keep track of the BDD size per iteration of the algorithm and plot the results for these five algorithms in Figure 7.2.

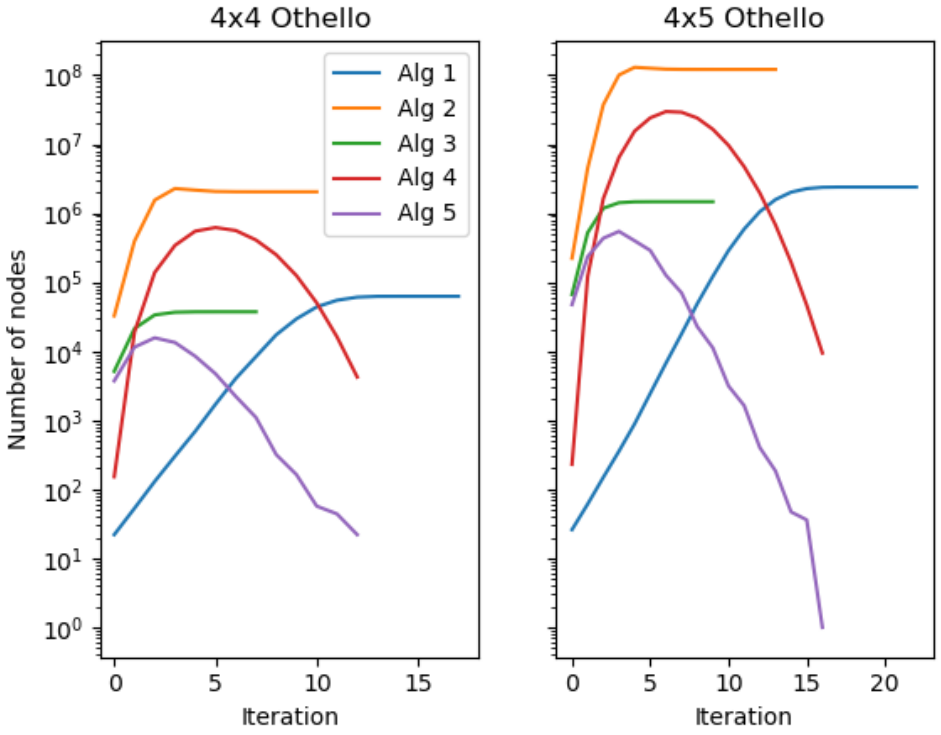

FIGURE 7.2: BDD sizes per iteration for the five algorithms, on a log-scaled y-axis. The left figure gives the sizes on a 4×4 board and the right figure gives the sizes on a 4×5 board. Later, we explain why the different algorithms terminate at different points.

These figures are plotted on a log-scaled y-axis, since the differences are too big for a linear y-axis. Comparing these figures directly shows their similarity. All algorithms behave the same in both 4×4 and 4×5 Othello. Therefore, we expect that scaling up to bigger variants of Othello (for instance 8×8 Othello) would yield similar trends, with of course larger BDD sizes in general.

What stands out in each figure, is the differences in the length between the lines of a figure. The length of a line represent the number of iterations performed in the algorithm, which equals the longest sequence of moves from one of the states that are in the initial set of the algorithm. These differences exist because the algorithms have different termination criteria. *Alg 4* and *Alg 5*, use the sweep-line method. Therefore, they perform $m \cdot n - 4$ iterations, which results in 12 and 16 iterations respectively, as visible in the graphs. The blue lines, representing *Alg 1*, terminate at convergence. This algorithm performs forward search. Then, the occurrence of each null-move in a sequence of moves, lengthen the number of iterations, since most sequences terminate whenever the board is full. Thus, by the fact that not every move adds a stone to the board, the forward search can have more than $m \cdot n - 4$ iterations. For both the orange as the green lines, representing *Alg 2* and *Alg 3*, in one iteration, a move is applied for both players. Both algorithms terminate at convergence, after 20 and 14 moves respectively, for 4×4 Othello. What also stands out in both figures, is that the four algorithms for retrograde analysis show big differences in their BDD sizes. These figures tell us that both limiting the search to the reachable states as well as the sweep-line method reduce the BDD size during every iteration of the algorithm, where combining these two methods delivers an even better reduction. See Table 7.2 for the exact numbers of the reductions for 4×4 Othello.

Algorithm	Peak BDD size	Reduction compared to algorithm 2
Alg 2	2279148	N/A
Alg 3	37298	98.36%
Alg 4	620067	72.79%
Alg 5	15656	99.31%

TABLE 7.2: Comparison of the four retrograde analysis algorithms, with regard to their peak BDD size, for 4×4 Othello.

7.1.3 Variable ordering

As described in Subsection 2.3.3 and 3.2, an optimal variable ordering can reduce the BDD sizes. We therefore try to find this optimal ordering in order to solve bigger instances of Othello. The standard variable ordering used so far (called VO_a), starts at the upper left field (A_{00} and is followed by the field right of it (A_{01}), row by row, as shown in Figure 7.3 for a 4×5 board.

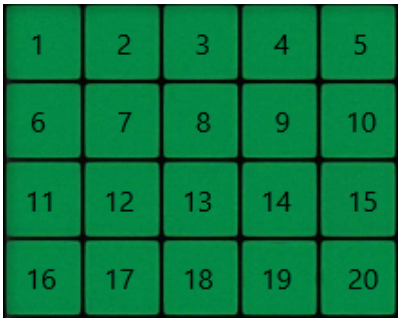

FIGURE 7.3: The standard variable ordering used so far (VO_a), given on a 4×5 board.

We now introduce two different variable orderings, for which the BDDs might be smaller. We think that it could be the case that the fields in the center of the board

have a higher dependence on each other, as well as for the fields located more to the edges of the board. We came up with two variable orderings: one where the fields that are occupied earlier in the game, are lower in the BDD and one the other way around. For these two variable orderings, called and VO_b and VO_c respectively, see Figures 7.4 and 7.5.

FIGURE 7.4: Variable Ordering VO_b, where the grid is split up in rings, such that the corners of each ring are higher in the BDD than the rest.

FIGURE 7.5: Variable Ordering VO_c, where the grid is split up in rings, such that the corners of each ring are lower in the BDD than the rest. This ordering is the opposite of VO_b.

We experiment with these variables orderings on a 4×5 board, and compare the BDD sizes when performing the best algorihm we found so far: retrograde analysis using the sweep-line method and limiting the search to the reachable states (*Alg 5*). The results of this experiment are in Figure 7.6.

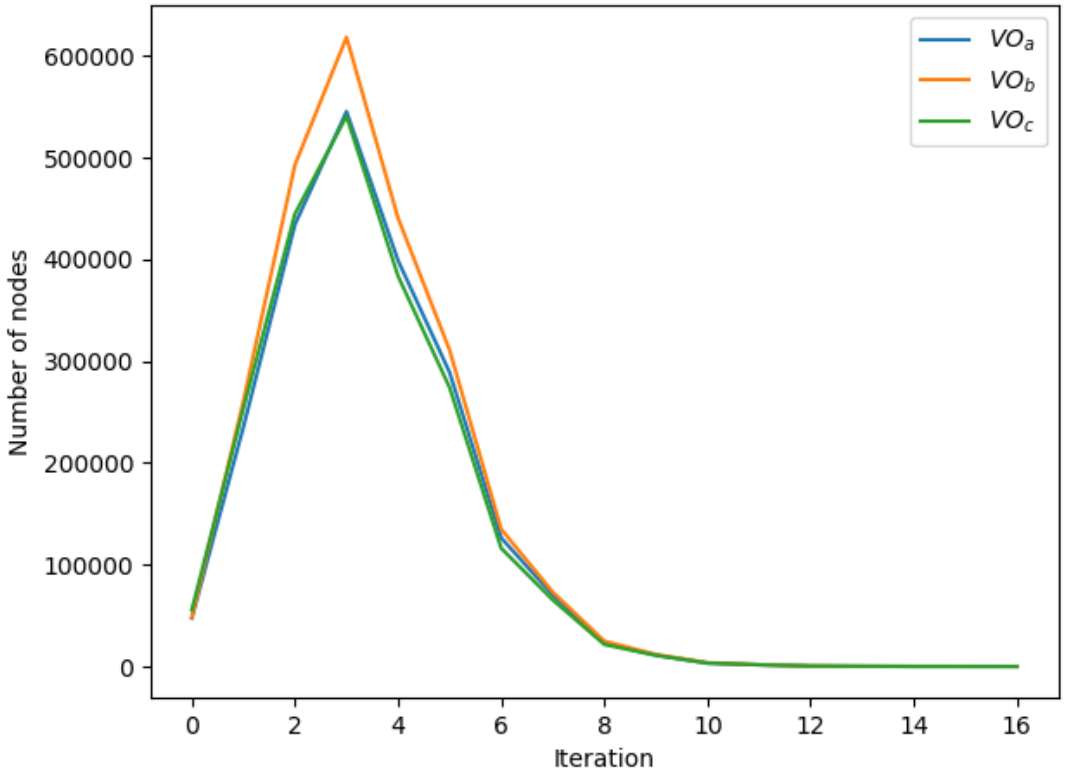

FIGURE 7.6: The results of comparing the three different variable orderings.

Figure 7.6 shows that VO_a and VO_c perform equally good, while VO_b constantly performs worse (VO_b is factored by 1.13 compared to VO_a at their top). Although we did not find a better variable ordering than the standard one, we now know that different orderings can have effect on the BDD sizes throughout the algorithms. We know that there exist one or more optimal variable orderings, but we still do not know what they are.
In these three variable orderings, there is one variable that we haven't taken into account: the turn variable, at the top of each BDD representing states. Next, we experiment on putting this turn variable at the bottom of the BDD instead of at the top. Again, we experiment on the best algorithm for 4×5 Othello. The result is in Figure 7.7.

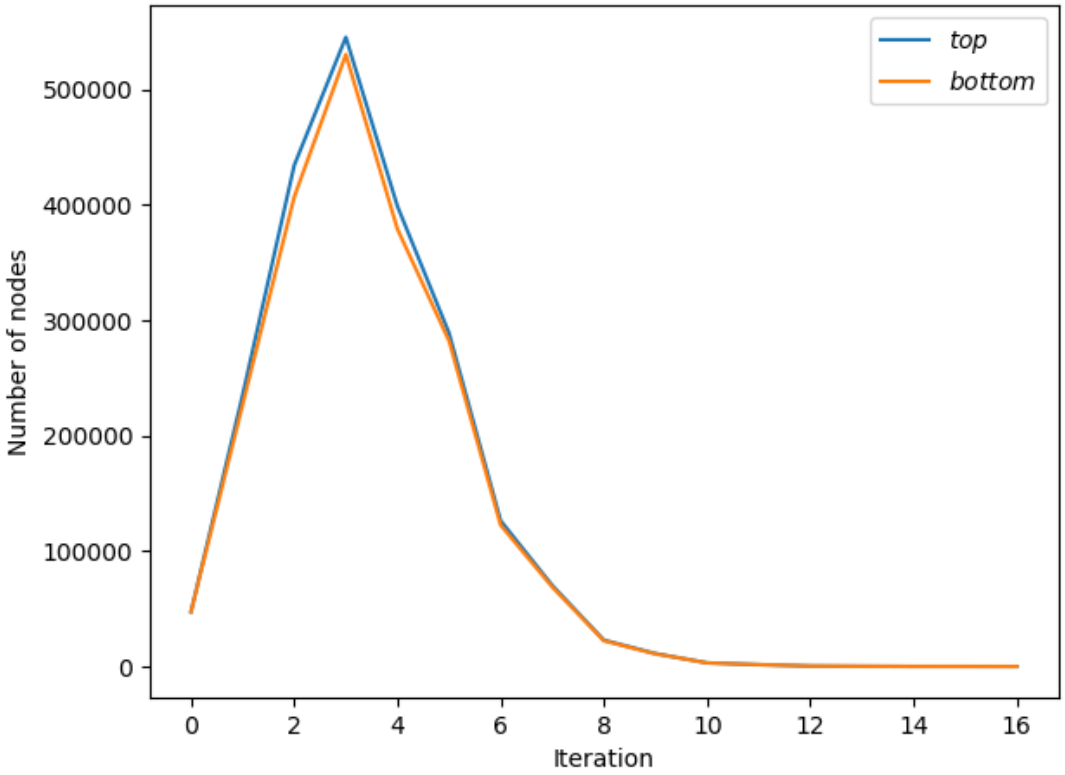

FIGURE 7.7: The results of locating the turn variable at the top of each BDD to at the bottom of each BDD.

This shows us that the location of the turn variable makes a negligible difference. Note that placing the turn variable at the bottom of the BDD takes the complete relation from 1.4 million nodes, to over two million nodes. This can force the algorithms to consume more time, since the operations on the relations are more complex. This makes the approach of placing the turn variable at the bottom less attractive.

7.2 Extrapolating the BDD sizes

In order to evaluate if we can solve Othello using BDDs and retrograde analysis, once our resources are powerful enough, we calculate the number of bytes, needed to store a single state in the set of immediately winning states. This number shows the memory efficiency of the BDDs for that specific size of Othello. We can than extrapolate this number to bigger instances, such as 8×8 Othello, to evaluate the needed memory for solving this instance of Othello. See table 7.3 for the results.

Othello size	#states	#bytes	bytes per state	factor
4×4	901,134	1,072,054	1.1397	N/A
4×5	$1.517\,22 \times 10^7$	7,012,428	0.46219	0.4055
5×5	4.6418×10^8	35,523,524	0.0765	0.1655
5×6	$1.556\,36 \times 10^{10}$	272,472,389	0.0175	0.22

TABLE 7.3: Calculation of the number of bytes per state in the BDD representing the immediately winning states, for different Othello sizes. The column factor gives the multiplication factor from an Othello instance one size smaller. This is needed for extrapolation.

First note that we cannot solve 5×5 and 5×6 Othello, but we can find the set of immediately winning states. In order to estimate the needed memory to solve the game, we should extrapolate for the peak of the algorithm, instead of for the set of immediately winning states. However, we chose to extrapolate for the immediately winning states, since we can find this number for four sizes of Othello, which gives a more precise estimation.
These results show that the number of bytes per state decreases when increasing the board size. We have some evidence that the number of bytes per state decreases faster than linear. However, we assume linear regression in our estimation, to find an upper bound. We therefore take the average factor per increase of the size (0.2637), and extrapolate with this number on the extra fields added. When taking the average factor for extrapolation, we roughly estimate (as an upper bound) the BDD representing the set of winning states for 8×8 Othello, to use $0.0175 \cdot 0.2637^5 = 2.23 \times 10^{-5}$ bytes per state. With the lower bound estimation of Takeshita et al. on the number of final positions (10^{22}), we estimate the lower bound for the number of bytes needed to solve 8×8 Othello with BDDs, to be $10^{22} \cdot 2.23 \times 10^{-5} = 2.23 \times 10^{17}$ bytes, which comes down to 2.23×10^8 gigabytes.

Chapter 8

Conclusions

In this thesis, we gave a thorough description on the topics of Othello, retrograde analysis and BDDs. We encoded Othello in a BDD and validated that this encoding is correct. In Chapter 7, we experimented with this encoding of Othello. The main results these experiments show, concern the different algorithms that can be used to solve Othello. All of these algorithms involve retrograde analysis, but we compared the use of two methods: limiting the search to the reachable states and the sweep-line method. From these experiments, we can conclude that both methods reduce the size of the search, in terms of number of BDD nodes. The biggest improvement can be gained by using both methods simultaneously. This showed a reduction of 99.31% of the biggest BDD size throughout the algorithm, compared to the standard retrograde analysis algorithm. We found the same results for using a bigger instance of Othello, namely 4×5 Othello, and believe that scaling up the size of the board would give the same results. Therefore we are convinced that both methods have a positive effect on solving Othello using retrograde analysis.
We also experimented on the variable orderings, which gave us the insight that some orderings perform better than others, although we did not find any best variable ordering. Locating the turn variable at the top or at the bottom of a BDD representing a set of Othello states, does not show any significant differences in the size of the BDD representing the winning states for one player, during retrograde analysis. However, the size of the transition relation was smaller when locating the turn variable at the top of the BDDs.
Using the encoding and algorithms introduced in this thesis, we were able to solve the instances of Othello on a 4×4 board and a 4×5 board. Bigger instances turned out to be too big to solve yet, given our resources and techniques. We believe the main reason for this is that different Othello states are hard to represent using a BDD, since one move can swap many stones. Then, the resemblance between the set of boards can get too low, making the BDD representing it too big. With state resemblance, we mean that the states in the set, are almost the same, such that they differ only a small amount of bits in their representations. Some of those structural problems yield small BDDs, for instance symmetric functions. In the extreme, the set of all boards is expressed with the True BDD leaf. Another set with high state resemblance is the set of boards with only white stones, which yields a small BDD. We are convinced that the power of the work of Edelkamp et al. [9] (see Section 3.2), lies in the fact that they increase their state resemblance by adding more states to the set. This resemblance is absent in our BDDs, and we do not see a way of overestimating our sets of states more than we do.
By being able to solve instances of Othello using our techniques, our results show some evidence, that divergent games are not immune to retrograde analysis, which contradicts to work of van den Herik et al. [11]. There can be discussed on whether we were only able to solve the instances of Othello because they are small, or that

divergent games are indeed not immune to retrograde analysis. Although BDDs do not seem to scale very well for computing fixpoints for a game like Othello, they can efficiently represent many states of a game where the state resemblance is high. Therefore, we believe that our methods might be useful for solving other combinatorial games.
By extrapolation, we estimated a lower bound on the number of gigabytes needed to solve Othello using BDDs. This number turned out to be 2.23×10^8, which is more than the current memory available. This indicates that Othello is still impossible to solve nowadays, although our techniques gave an efficient way of representing states. The extrapolation also gave an upper bound estimation for 8×8 Othello, that the BDD representing all immediately winning states uses only 2.23×10^{-5} bytes per state. This is less than the current techniques applied for solving combinatorial games, where hashing methods are used, for instance the work of Edelkamp et al. [10]. Perfect hashing requires one bit per state, which is a lower bound for techniques based on explicit methods. Probably, this lower bound does not even hold for solving Othello, since perfect hashing does not work for Othello, where many states are not reachable. Then, a hash table might be a solution, which requires $\log_2(n)$ bits per bucket, with n the number of states in the state space.

8.1 Future work

With the contributions of this thesis, future work can be performed, where BDDs together with retrograde analysis are applied to other combinatorial games. For solving Othello, more work can be done by trying new methods, such that less memory is needed. An examples of such a method is to apply *Property Directed Reachabililty* (PDR) using SAT solvers, as done by Morgenstern et al. [20].

Bibliography

[1] V Allis. "Searching for Solutions in Games and Artificial Intelligence". PhD thesis. Jan. 1994.

[2] G. Birkhoff. *Lattice Theory*. Colloquium publications. American Mathematical Society, 1948. ISBN: 9780821889534. URL: https://books.google.nl/books?id=o4bu3ex9BdkC.

[3] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819. URL: https://doi.org/10.1109/TC.1986.1676819.

[4] Jerry R. Burch, Edmund M. Clarke, and David E. Long. "Symbolic Model Checking with Partitioned Transistion Relations". In: *VLSI*. 1991, pp. 49–58.

[5] Jerry R. Burch et al. "Symbolic Model Checking: 10^20 States and Beyond". In: *Inf. Comput.* 98.2 (1992), pp. 142–170. DOI: 10.1016/0890-5401(92)90017-A. URL: https://doi.org/10.1016/0890-5401(92)90017-A.

[6] Sagar Chaki and Arie Gurfinkel. "BDD-Based Symbolic Model Checking". In: *Handbook of Model Checking*. 2018, pp. 219–245. DOI: 10.1007/978-3-319-10575-8_8. URL: https://doi.org/10.1007/978-3-319-10575-8_8.

[7] *CUDD: Colorado University Decision Diagram Package*. https://github.com/ivmai/cudd. Last accessed: 2019-07-03.

[8] Tom van Dijk, Alfons Laarman, and Jaco van de Pol. "Multi-core and/or Symbolic Model Checking". In: *ECEASST* 53 (2012). DOI: 10.14279/tuj.eceasst.53.773. URL: https://doi.org/10.14279/tuj.eceasst.53.773.

[9] Stefan Edelkamp and Peter Kissmann. "On the Complexity of BDDs for State Space Search: A Case Study in Connect Four". In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. 2011. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3690.

[10] Stefan Edelkamp, Damian Sulewski, and Cengizhan Yücel. "GPU Exploration of Two-Player Games with Perfect Hash Functions". In: *Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010, Stone Mountain, Atlanta, Georgia, USA, July 8-10, 2010*. 2010. URL: http://aaai.org/ocs/index.php/SOCS/SOCS10/paper/view/2088.

[11] H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Jack van Rijswijck. "Games solved: Now and in the future". In: *Artif. Intell.* 134.1-2 (2002), pp. 277–311. DOI: 10.1016/S0004-3702(01)00152-7. URL: https://doi.org/10.1016/S0004-3702(01)00152-7.

[12] M. J. H. Heule and Léon J. M. Rothkrantz. "Solving games: Dependence of applicable solving procedures". In: *Sci. Comput. Program.* 67.1 (2007), pp. 105–124. DOI: 10.1016/j.scico.2007.01.004. URL: https://doi.org/10.1016/j.scico.2007.01.004.

[13] Michael Huth and Mark Dermot Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.)* Cambridge University Press, 2004.

[14] Richard Huybers and Alfons Laarman. "A Parallel Relation-Based Algorithm for Symbolic Bisimulation Minimization". In: *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*. 2019, pp. 535–554. DOI: 10.1007/978-3-030-11245-5_25. URL: https://doi.org/10.1007/978-3-030-11245-5_25.

[15] Kurt Jensen, Lars Michael Kristensen, and Thomas Mailund. "The sweep-line state space exploration method". In: *Theor. Comput. Sci.* 429 (2012), pp. 169–179. DOI: 10.1016/j.tcs.2011.12.036. URL: https://doi.org/10.1016/j.tcs.2011.12.036.

[16] Gijs Kant et al. "LTSmin: High-Performance Language-Independent Model Checking". In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, pp. 692–707. DOI: 10.1007/978-3-662-46681-0_61. URL: https://doi.org/10.1007/978-3-662-46681-0_61.

[17] Jesper Torp Kristensen. "Generation and compression of endgame tables in chess with fast random access using OBDDs". Master's Thesis. U. of Aarhus, Dept. of Computer Science, 2005.

[18] Lars Michael Kristensen and Thomas Mailund. "Efficient Path Finding with the Sweep-Line Method Using External Storage". In: *Formal Methods and Software Engineering, 5th International Conference on Formal Engineering Methods, ICFEM 2003, Singapore, November 5-7, 2003, Proceedings*. 2003, pp. 319–337. DOI: 10.1007/978-3-540-39893-6_19. URL: https://doi.org/10.1007/978-3-540-39893-6_19.

[19] Kenneth Lauchlin McMillan. "Symbolic Model Checking: An Approach to the State Explosion Problem". UMI Order No. GAX92-24209. PhD thesis. Pittsburgh, PA, USA, 1992.

[20] Andreas Morgenstern, Manuel Gesell, and Klaus Schneider. "Solving Games Using Incremental Induction". In: *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*. 2013, pp. 177–191. DOI: 10.1007/978-3-642-38613-8_13. URL: https://doi.org/10.1007/978-3-642-38613-8_13.

[21] E. V. Nalimov, G. M. Haworth, and E. A. Heinz. "Space-efficient indexing of endgame tables for chess." In: *Advances in Computer Games 9*. IKAT, University of Maastricht, 2001.

[22] *RERS – Rigorous Examination of Reactive Systems*. http://rers-challenge.org/. Last accessed: 2019-07-03.

[23] A.N. Siegel. *Combinatorial Game Theory*. Graduate studies in mathematics. American Mathematical Society, 2013. ISBN: 9780821851906. URL: https://books.google.nl/books?id=VUVrAAAAQBAJ.

[24] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489. DOI: 10.1038/nature16961. URL: https://doi.org/10.1038/nature16961.

[25] Fabio Somenzi. "Binary decision diagrams". In: *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES* 173 (1999), pp. 303–368.

[26] Yuki Takeshita et al. "Perfect Play in Miniature Othello". In: *Genetic and Evolutionary Computing - Proceedings of the Ninth International Conference on Genetic and Evolutionary Computing, ICGEC 2015, August 26-28, 2015, Yangon, Myanmar - Volume II*. 2015, pp. 281–290. DOI: 10.1007/978-3-319-23207-2_28. URL: https://doi.org/10.1007/978-3-319-23207-2_28.

[27] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific J. Math.* 5.2 (1955), pp. 285–309. URL: https://projecteuclid.org:443/euclid.pjm/1103044538.