GENERAL GAME PLAYING WITH INCOMPLETE INFORMATION USING ZERO-SUPPRESSED DECISION DIAGRAMS

Steve Losh

Thesis of 60 ECTS credits
Master of Science (M.Sc.) in Computer Science

June 2017
General Game Playing with Incomplete Information using Zero-Suppressed Decision Diagrams

by

Steve Losh

Thesis of 60 ECTS credits submitted to the School of Computer Science at Reykjavík University in partial fulfillment of the requirements for the degree of Master of Science (M.Sc.) in Computer Science

June 2017

Supervisor:
Stephan Schiffel, Supervisor
Professor, Reykjavík University, Iceland

Examiner:
Henning Arnór Úlfarsson, Examiner
Professor, Reykjavík University, Iceland

Stefan Edelkamp, Examiner
Professor, University of Bremen, Germany
Abstract

The field of General Game Playing is concerned with creating AI agents capable of playing any game given only its rules. Traditionally General Game Playing has been mostly concerned with complete-information games. Recently work has been done to extend the field to games with incomplete information, but playing these games is much more challenging because the amount of information to keep track of can grow quickly. We describe and implement a new technique for representing the information set of incomplete information games using Zero Suppressed Decision diagrams to reduce the amount of memory needed, as well as a technique for reasoning directly on these data structures. The results are promising, but require further refinement to become practical for playing most games.
General Game Playing with Incomplete Information using Zero-Suppressed Decision Diagrams

Steve Losh

Thesis of 60 ECTS credits submitted to the School of Computer Science at Reykjavík University in partial fulfillment of the requirements for the degree of Master of Science (M.Sc.) in Computer Science

June 2017

Student:

Steve Losh

Supervisor:

Stephan Schiffel

Examiner:

Henning Arnór Úlfarsson

Stefan Edelkamp
The undersigned hereby grants permission to the Reykjavík University Library to reproduce single copies of this Thesis entitled **General Game Playing with Incomplete Information using Zero-Suppressed Decision Diagrams** and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the Thesis, and except as herein before provided, neither the Thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author’s prior written permission.

---

Steve Losh
Master of Science
## Contents

### Contents

List of Figures  xiii

List of Tables  xv

1 Introduction & Background  1
   1.1 General Game Playing  ........................................ 1
      1.1.1 GDL  .............................................. 1
      1.1.2 GDL-II  ............................................ 4
   1.2 A Naive, Brute-Force GDL-II Player  ..................... 7
   1.3 Zero-Suppressed Decision Diagrams  ....................... 7
      1.3.1 Data Structure  ...................................... 7
      1.3.2 Operations  .......................................... 8

2 Related Work  11
   2.1 ZDDs  .......................................................... 11
   2.2 General Game Playing  ....................................... 11

3 Methods  13
   3.1 Code & Libraries  ............................................ 13
   3.2 GDL Game Flow  ............................................. 13
      3.2.1 Initial Computation  .................................. 13
      3.2.2 Playing Turns  ......................................... 15
   3.3 Grounding GDL-II  ........................................... 16
   3.4 Term Layers  ................................................ 16
   3.5 Stratifying Layers  .......................................... 17
   3.6 Ordering Terms  .............................................. 18
   3.7 Representing Information Sets as ZDDs  .................. 19
   3.8 Logical Reasoning on ZDDs  ................................ 19
      3.8.1 Representing GDL Rules as Rule Trees  .............. 20
      3.8.2 Splitting Rule Trees  .................................. 22
      3.8.3 Applying Single Rule Trees  ........................ 23
      3.8.4 Rule Tree Application Example  .................... 24
      3.8.5 Applying Multiple Rule Trees  ....................... 27
      3.8.6 Handling Cycles & Negation  ......................... 28
   3.9 Sprouting Joint Moves  ..................................... 29
   3.10 Filtering on Percepts  ..................................... 30
   3.11 Successor Computation  .................................... 31
4 Results
   4.1 Information Set Sizes ........................................... 33
   4.2 Rule Tree Sizes .................................................. 37
   4.3 Runtimes ......................................................... 38
   4.4 Impact of Variable Ordering ................................. 39

5 Conclusion & Future Work ........................................ 41

Bibliography ....................................................... 43
List of Figures

1.1 A ZDD for the family $\{a, b, c\}, \{b, d\}, \{a, c\}$. ..................... 8

3.1 Flow of a GDL-II game using our method. ......................... 14
3.2 The rule tree for one rule in Tic Tac Toe. .................. 21
3.3 The rule tree for a disjunctive rule in Tic Tac Toe. .......... 21
3.4 The rule tree for a rule involving negation from Monty Hall. .... 22
3.5 A simple rule tree and information set. .................... 24
3.6 The rule tree and information set after recurring high. ....... 25
3.7 The rule tree and information set after recurring high, then high. 25
3.8 The rule tree and information set after recurring high, then low. 25
3.9 The rule tree and information set after recurring high, then low, then high. 26
3.10 The rule tree and information set after recurring high, then low, then low. 26
3.11 The rule tree and information set after recurring low. ........... 26
3.12 The rule tree and information set after recurring low, then high. ... 27
3.13 The rule tree and information set after recurring low, then low. .... 27
3.14 A ZDD of joint moves for two roles. .............................. 30

4.1 Information set storage requirements for mastermind448.gdl........ 34
4.2 Information set storage requirements for transit.gdl......... 34
4.3 Information set storage requirements for latenttictactoe.gdl... 35
4.4 Impact of random variable orderings on rule tree sizes. .......... 39
List of Tables

3.1 Term layers and their contents. .................................................. 16

4.1 Information set sizes for one play of montyhall.gdl. .................... 33
4.2 Information set sizes for one play of mastermind448.gdl. .............. 35
4.3 Information set sizes for one play of transit.gdl. ......................... 36
4.4 Information set sizes for one play of latenttictactoe.gdl. .............. 36
4.5 Rule tree sizes for various games (with default settings). .............. 37
4.6 Effects of splitting threshold on meier.gdl. ............................. 37
4.7 Effects of splitting threshold on mastermind448.gdl. .................... 38
4.8 Run times for a single game of mastermind448.gdl (times are in seconds). 38
Chapter 1

Introduction & Background

Research in the field of General Game Playing has traditionally centered around a particular class of games, specifically those that are: discrete, finite, playable, winnable, synchronous, deterministic, and complete information.

While this class of games is large and encompasses many interesting games (e.g. Chess, Checkers, and Go), it excludes many others (e.g. Poker). Predictably the field has begun to push the boundaries of these restrictions to try to broaden the class of games and make General Game Playing even more general.

One current area of research is in allowing games of incomplete information. The main challenge for these games is that the players no longer know the exact state of the game, but instead must keep track of a set of possible states the game might currently be in, usually called the “information set”. Keeping track of and reasoning about all the states in the information set for non-trivial games can be quite challenging, because it can grow extremely large as play progresses.

In this paper we present a technique for using Zero-Suppressed Decision Diagrams to represent the information set, as well as a technique for performing logical reasoning on these information sets without resorting to reasoning on each state individually.

1.1 General Game Playing

To make AIs capable of playing arbitrary games (and to pit them against each other), we first need a programmatic way to represent the rules of a game. Game Description Language (GDL) was created [1] for this purpose. We will give a brief introduction to the relevant parts of GDL (and its successor GDL-II) here.

1.1.1 GDL

GDL is a small logic programming language, modeled after Datalog but with some extra restrictions and extensions to make it suitable for defining games.

All GDL games can be thought of as a tree\(^1\) of discrete states, where the leaves are terminal states and all other nodes are nonterminal. A single game state in GDL is represented as a set of facts of the form \((\text{true } <\text{fact}>)\). Listing 1.1 shows an example state in a

\(^1\)The “game tree” should probably be called a “game directed acyclic graph”, but we will use the more conventional term.
game of Tic Tac Toe\(^2\) (also known as Naughts and Crosses) where player X has chosen to mark the center square on her first turn.

The initial state and roles of the game are defined by the \((\text{init } \text{<fact>})\) and \((\text{role } \text{<role>})\) predicates, as seen in Listing 1.2. When a game begins, a logic programming-based AI’s reasoning system will typically load the rules into its logic database and then compute the initial state by querying the database for all results of \((\text{init } ?\text{what})\), replacing \text{init} with \text{true} in each item, and asserting the resulting facts into the database.

GDL supports logical rules of the form \((\langle= \text{head body...} \rangle)\). Arbitrary rules can be defined, but there are four special predicates that must be defined to describe the basic flow of the game:

1. \((\text{terminal})\) This predicate must be true when the game has ended and false otherwise.
2. \((\text{goal } \text{<role> } \text{<value>})\) For every terminal state, this predicate must describe a single goal value for every role. Scores range from 0 to 100 inclusive, and higher scores are better.
3. \((\text{legal } \text{<role> } \text{<move>})\) For every nonterminal state, this predicate must describe one or more valid moves that each role can perform.
4. \((\text{next } \text{<fact>})\) For every nonterminal state, the next state of the game must be defined by this predicate.

Once all players have decided on a move, the successor state can be computed by the following process:

\(^2\)This example was previously published in [2].
1.1. GENERAL GAME PLAYING

1. Assert facts of the form (does <role> <move>) into the database.

2. Query the database for all results of (next ?what).

3. Retract all true and does facts from the database.

4. For each result in step 2, assert (true ?what) into the database.

GDL allows the use of negation through the not and distinct predicates with a restriction: a cycle of recursive predicates cannot involve negation. GDL also has a few additional restrictions, but these are not important for our purposes. Interested readers should consult the full GDL spec [1] for a complete description.

Listing 1.3 shows a complete GDL definition for the game of Tic Tac Toe.

Listing 1.3: GDL rules for Tic Tac Toe.

```gdl

;;; Roles
(role x)
(role o)

;;; Initial State
(init (control x))

(init (cell 1 1 blank))
(init (cell 1 2 blank))
(init (cell 1 3 blank))
(init (cell 2 1 blank))
(init (cell 2 2 blank))
(init (cell 2 3 blank))
(init (cell 3 1 blank))
(init (cell 3 2 blank))
(init (cell 3 3 blank))

;;; Useful Predicates
(<= (row ?n ?mark)
    (true (cell ?n 1 ?mark))
    (true (cell ?n 2 ?mark)))
(<= (column ?n ?mark)
    (true (cell 1 ?n ?mark))
    (true (cell 2 ?n ?mark))
    (true (cell 3 ?n ?mark)))
(<= (diagonal ?mark)
    (true (cell 1 1 ?mark))
    (true (cell 2 2 ?mark))
    (true (cell 3 3 ?mark)))
(<= (diagonal ?mark)
    (true (cell 1 3 ?mark))
    (true (cell 2 2 ?mark))
    (true (cell 3 1 ?mark)))
(<= (line ?mark) (row ?n ?mark))
(<= (line ?mark) (column ?n ?mark))
(<= (line ?mark) (diagonal ?mark))
```

CHAPTER 1. INTRODUCTION & BACKGROUND

(<= open
  (true (cell ?r ?c blank)))

;;; Terminal
(<= terminal (line x))
(<= terminal (line o))
(<= terminal (not open))

;;; Goal Values
(<= (goal ?player 100)
  (line ?player))
(<= (goal ?player 0)
  (line ?other)
  (distinct ?player ?other))
(<= (goal ?player 50)
  (not (line x))
  (not (line o))
  (not open))

;;; Legal Moves
(<= (legal ?player (mark ?row ?col))
  (true (cell ?row ?col blank))
  (true (control ?player)))
(<= (legal ?player noop)
  (true (control ?other))
  (distinct ?player ?other))

;;; State Transitions
(<= (next (control x)) (true (control o)))
(<= (next (control o)) (true (control x)))
(<= (next (cell ?row ?col ?player))
  (true (cell ?row ?col ?player))
  (distinct ?player blank))
(<= (next (cell ?row ?col ?player))
  (true (cell ?row ?col blank))
  (does ?player (mark ?row ?col)))
(<= (next (cell ?row ?col blank))
  (true (cell ?row ?col blank))
  (does ?player (mark ?x ?y))
  (or (distinct ?row ?x)
    (distinct ?col ?y)))

1.1.2 GDL-II

GDL only allows the definition of deterministic games with complete information. GDL-II is an extension of GDL that relaxes this restriction and supports games that involve incomplete information and randomness [3].
Randomness is added by defining a special role called random. This role is defined to select a random move (uniformly distributed) from its legal moves at each term. Randomness is independent to the addition of incomplete information and does not affect our technique, so we will not consider it further.

To support games of incomplete information two changes are made. First: the game master no longer sends the list of moves made by other players to each AI every turn. Instead it sends a lists of “percepts” to each player, which are defined by the (sees <role> <percept>) predicate. These percepts are computed by the game master after the does facts have been asserted and sent to the appropriate players. Players are expected to use these percepts to narrow down their location in the game tree.

Play of a GDL-II game proceeds much like a vanilla GDL game, with one important change. In normal GDL games each player will know the exact state of the game at the beginning of each turn. For GDL-II games, the players will not know the exact state, but will instead have a set of possible states that the game could be in. As the game progresses this set of states (hereafter called the “information set”) will expand when other players make moves and shrink as information is revealed through the percepts.

Listing 1.4 shows a simple implementation of the Monty Hall game. The random player hides the car behind a random door, and opens one of the two non-selected doors at random. The candidate player sees which door was opened because of the sees rule defined on lines 38-39, but does not see where the car was initially placed.

Listing 1.4: The Monty Hall game in GDL-II.

```gdl
(role candidate)
(role random)

(init {closed 1})
(init {closed 2})
(init {closed 3})
(init {step 1})

(<= (legal random (hide_car ?d))
    (true {step 1})
    (true {closed ?d}))

(<= (legal random (open_door ?d))
    (true {step 2})
    (true {closed ?d})
    (not (true (car ?d)))
    (not (true (chosen ?d))))

(<= (legal random noop)
    (true {step 3}))

(<= (legal candidate (choose ?d))
    (true {step 1})
    (true {closed ?d}))

(<= (legal candidate noop)
    (true {step 2}))

(<= (legal candidate switch)
    (true {step 3}))

(<= (legal candidate noop)
```
(true (step 3)))

(<= (sees candidate (does candidate ?m))
     (does candidate ?m))

(<= (sees candidate (open_door ?d))
     (does random (open_door ?d)))

(<= (next (car ?d))
     (true (step 2)))

(<= (next (car ?d))
     (true (car ?d)))

(<= (next (closed ?d))
     (true (closed ?d))
     (not (does random (open_door ?d))))

(<= (next (chosen ?d))
     (next_chosen ?d))

(<= (next_chosen ?d)
     (true (chosen ?d))
     (not (does candidate switch)))

(<= (next_chosen ?d)
     (does candidate switch)
     (true (closed ?d))
     (not (true (chosen ?d))))

(<= (next (step 2))
     (true (step 1)))

(<= (next (step 3))
     (true (step 2)))

(<= (next (step 4))
     (true (step 3)))

(<= (sees candidate (car ?d))
     (true (step 3))
     (true (car ?d))
     (next_chosen ?d))

(<= terminal (true (step 4)))

(goal random 100)

(<= (goal candidate 100)
     (true (chosen ?d))
     (true (car ?d)))

(<= (goal candidate 0)
     (true (chosen ?d))
     (not (true (car ?d))))
1.2 A Naive, Brute-Force GDL-II Player

Before describing our method it is useful to look at a simple brute-force GDL-II player for comparison. We will assume a Prolog-like logic system is available (our player uses our own TEMPERANCE logic programming library [5] internally). The brute-force process of playing a GDL-II game is as follows.

First compute the initial state of the game as defined by the rules. The initial information set is a set containing only this single initial state.

To compute the legal moves for the player, choose an arbitrary state in the information set and compute the legal moves as you would for a vanilla GDL game. If the information set has been correctly tracked, all states in it must have the same legal moves for the player’s role\(^3\). This is also the case for the terminality check and the goal values, so those can be handled similarly.

To compute the successor of the current information set, perform the following on every state in it:

1. Find all possible combinations of moves made by yourself and the other players\(^4\).
2. For each of these moves, add the corresponding \((\text{does} \text{ <role> <move>})\) facts to the state.
3. Compute the \((\text{sees} \text{ <role> <fact>})\) for the player’s role and check whether the results exactly match the percepts received from the server. If they do not match, discard this state/move combination.
4. If the percepts \emph{do} match, compute the successor state by finding all results of \((\text{next} \text{ <fact>})\) and add it to successor information state.

This process is very time consuming, as it involves computing \emph{every possible successor state} one-by-one for every current state and discarding the ones that don’t fit the percepts, and it is impractical for many non-trivial games. For example: in a two-player poker game where each player is dealt a hand of five cards at the beginning of the game, the information set could immediately expand to contain \((\binom{52}{47})\) states (1,533,939), one for each possible hand the opponent could have.

1.3 Zero-Suppressed Decision Diagrams

Zero-Suppressed Decision Diagrams (ZDDs) are an elegant data structure used to represent sets of sets efficiently. To avoid confusion the term \emph{family} of sets is often used to mean a set of sets. Each individual ZDD represents a single family of sets. We present a brief overview of ZDDs here, but interested readers should consult the original introduction [7] or Knuth [8] for a full treatment.

1.3.1 Data Structure

ZDDs are implemented as a directed acyclic graph of nodes. There are three possible types of nodes in the graph: two unique “sink” nodes used for the leaves, and internal nodes used

---

\(^3\)This property is required for a game to be considered valid GDL-II [6].

\(^4\)A combination of moves, one per player, is often called a “joint move” in General Game Playing.
to represent element membership (or lack thereof) in the sets. These nodes are defined as follows:

- The “empty” sink ($\perp$) represents the empty family: $\{\}$.  
- The “unit” sink ($\top$) represents a family containing only the empty set: $\{\emptyset\}$.  
- Internal (nonterminal) nodes consist of a term (called the “label”) and two pointers to child nodes, usually called “high” and “low”. We denote a node with label $L$ and children $Z_{high}$ and $Z_{low}$ as $(L, Z_{high}, Z_{low})$, and the family of sets it represents is defined to be $\{\{L\} \cup s \mid s \in Z_{high}\} \cup Z_{low}$.

Terms must be totally ordered, and nodes labeled with smaller terms must come before larger terms. In addition, ZDDs must satisfy the following rules:

1. There must be exactly one root node.
2. The high child of an internal node cannot be $\perp$.
3. Every combination of $(label, high, low)$ must be unique within the ZDD.

Rule 3 ensures that branches of the ZDD share structure with each other when possible. Although this does not reduce the worst-case memory usage ($2^n$), in practice many ZDDs end up sharing a significant amount of structure, which reduces the memory needed and improves traversal times.

When drawing ZDDs the convention is to draw the high edge as a solid line and the low edge as a dashed line. Figure 1.1 shows the ZDD for the family $\{\{a, b, c\}, \{b, d\}, \{a, c\}\}$.

![Figure 1.1: A ZDD for the family $\{\{a, b, c\}, \{b, d\}, \{a, c\}\}$](image)

### 1.3.2 Operations

One key advantage of ZDDs is that many common set operations like union, intersection, and so on can be implemented directly on the ZDDs themselves, without expanding them into their constituent members. As an example, an algorithm for building the union of two ZDDs $A$ and $B$ is as follows:
1. If either ZDD is ⊥, return the other ZDD.
   The union of {} and any other set is that other set.

2. If either ZDD is ⊤, return the other ZDD with the empty set added as a member (if not already present).
   The union of {} and any other set is simply that set with the empty set added as a member.

3. Otherwise both ZDDs must be nonterminal nodes \((L_A, A_{\text{high}}, A_{\text{low}})\) and \((L_B, B_{\text{high}}, B_{\text{low}})\).
   Compare \(L_A\) and \(L_B\):
   
   a) If \(L_A < L_B\), return a new ZDD node\(^5\) \((L_A, A_{\text{high}}, (A_{\text{low}} \cup Z_B))\).
      In this case some of the sets in \(A\) contain a term not found in any members of \(B\).
   
   b) If \(L_B < L_A\), return a new ZDD node \((L_B, B_{\text{high}}, (B_{\text{low}} \cup Z_A))\).
      In this case some of the sets in \(B\) contain a term not found in any members of \(A\).
   
   c) Otherwise \(L_A = L_B\), return a new ZDD node \((L_A, (A_{\text{high}} \cup B_{\text{high}}), (A_{\text{low}} \cup B_{\text{low}}))\).

The process of adding the empty set as a member of a ZDD (as needed in step 2) involves traversing the low branches of internal nodes down to a sink and replacing it with the unit sink if it happens to be the empty sink.

In [8] Knuth defines\(^6\) several operations on ZDDs. We will assume these operations are implemented, and interested readers should refer to Knuth for the full description. In particular we will be making use of the following operations:

**Meet** Given two ZDDs \(Z_1\) and \(Z_2\), \(Z_1 \cap Z_2 = \{\alpha \cap \beta \mid \alpha \in Z_1, \beta \in Z_2\}\).

**Join** Given two ZDDs \(Z_1\) and \(Z_2\), \(Z_1 \cup Z_2 = \{\alpha \cup \beta \mid \alpha \in Z_1, \beta \in Z_2\}\).

We will also define several other operations on ZDDs specific to our methods later, when needed.

\(^5\)To comply with the restrictions on ZDDs, when we say “return a new ZDD node” we will assume that if such a node already exists somewhere a reference to it is used instead of creating a fresh node. We will also assume that if the high pointer is ⊥ we will simply return the low pointer instead of creating a new node.

\(^6\)The actual implementation details are tucked away in the answers to certain exercises.
Chapter 2

Related Work

2.1 ZDDs

The main data structure our methods use is Zero-suppressed decision diagrams, which have a rich history.

The ancestors of ZDDs are Binary Decision Diagrams, which were introduced in 1959 by Lee [9] and later popularized by Akers [10] and refined by Bryant [11]. BDDs have been studied extensively and much research has been done on improving their efficiency, through variable ordering as well as other means.

ZDDs were introduced in 1993 as a variant of BDDs by Minato [7]. Whereas BDDs are often used to represent boolean functions, ZDDs are typically thought of as representing sets of sets, which lends itself nicely to representing the information sets of games. ZDDs also favor sparse information sets where only a small subset of all possible terms are present at a particular time, which may work better for General Game Playing applications.

Like BDDs, the efficiency of ZDDs is heavily dependent on variable ordering. Less research has been done on finding efficient variable orderings on ZDDs than has been done for BDDs, though some exists (e.g. [12]).

Several other variants of BDDs exist as well. For example: Algebraic Decision Diagrams (ADDs) extend BDDs to allow for non-binary variables. However, because states in General Game Playing are represented as binary facts which can only be true or false ADDs would not provide any additional advantage for our work as-is. Further processing of GDL rules to reformulate some of the terms as non-binary variables could allow for a more succinct representation of states, and ADDs could be useful to encode such information sets, but we did not investigate this possibility.

2.2 General Game Playing

Much research has been done on General Game Playing for complete information games, but far less has been published on General Game Playing with incomplete information.

HyperPlay [14] is one notable exception. HyperPlay is a General Game Player for incomplete information games, which provides a way to play games defined in GDL-II by maintaining a sample of the information set during play. This sampling reduces the amount of space needed to store the information set. If the sampled information set is found to be inconsistent with the percepts returned from the game server it can be resampled by back-tracking to a consistent state and replaying from there.
In [13] Kissman describes a technique for using Reduced, Ordered Binary Decision Diagrams to represent GDL games and search their state spaces for solutions. BDDs are used to reduce the amount of memory needed for representing the search space, and forward and backward search are both used to find paths to terminal states. Although their technique only deals with games of complete information, the use of decision diagrams to represent and search a General Game Playing state space provided valuable insight and inspiration for our technique.
Chapter 3

Methods

A brute-force player can play GDL-II games legally but is extremely inefficient. Here we present a new technique for representing information sets with ZDDs and reasoning on them directly, without enumerating each member state individually.

3.1 Code & Libraries

Our solution and player are implemented in COMMON LISP. The code is available on GitHub [15].

The player uses the CL-GGP library [2] to handle the details of the GGP network protocol, parsing GDL into Lisp forms, and so on. Support for GDL-II games was added into CL-GGP as part of this project.

After evaluating several ZDD libraries (e.g. CUDD and JDD [16]) we decided to use TRIVIALIB.BDD [17] as a starting point. It contains a very small implementation of ZDDs in pure Common Lisp. While it is likely not the fastest ZDD library in terms of raw performance, it has a relatively pleasant interface that allowed us to focus on implementing our solution, rather than on wrapping a C- or Java-based library.

Several other COMMON LISP libraries were used for various purposes, interested readers should consult the ASD file\(^1\) for the full list.

3.2 GDL Game Flow

Before looking at the individual pieces of our solution, we will look at the overall flow of a GGP game to get a sense of how everything fits together. Figure 3.1 shows an overview of the process, and we will review the individual pieces below.

3.2.1 Initial Computation

Before the first turn begins our player performs some initial computation on the rules of the game to prepare, seen above the dashed line in Figure 3.1.

The first thing our player does after receiving the GDL-II rules from the server is ground these rules to remove any logical variables in them.

Once the rules have been grounded the player must compute an acceptable ordering for every term in the rules. For speed and space reasons all terms are mapped to integers, and these integers are what is actually stored in the ZDDs and other data structures.

\(^1\)https://github.com/sjl/scully/blob/master/scully.asd
Figure 3.1: Flow of a GDL-II game using our method.
After all terms have been ordered (i.e. converted to integers) the player processes each logical rule to construct a “rule tree”. This is a data structure that will be used later for reasoning on information sets. We also create a few other ancillary data structures that will be described later.

Our player can then compute the initial information set of the game. It constructs a ZDD of a single set, containing all the \((\text{true} \ <\text{fact}>\)) terms corresponding to the \((\text{init} \ <\text{fact}>\)) terms in the rules.

### 3.2.2 Playing Turns

Below the dashed line in Figure 3.1 we see the flow of turns in a game and how the data in the information set expands and shrinks.

At the beginning of each turn, the player starts with a current information set containing one or more possible states the game could be in. Each state in this information set is a set of \((\text{true} \ <\text{fact}>\)) terms.

The player begins the turn by extending every state in the set with any additional terms that are logically derivable from the rules of the game. This includes terms for the terminal, goal, and legal predicates, as well as other intermediate terms that can be inferred from the base state. The addition of these terms is done by traversing the appropriate rule trees recursively in parallel with the information set and building the resulting (extended) information set on the way back up the stack.

Once each state has been extended to include these terms, terminality, goal values, and legal moves can be determined by simply choosing an arbitrary member of the information set and examining its contents. As long as the information set has been updated correctly over the course of the game, all of its members should have exactly the same results for each of these predicates.

After choosing a legal move and relaying it to the game server, the player will eventually receive a list of percepts and must compute the next information set. This happens in several steps.

First, each current state in the information set is turned into several new states, each of which has the \((\text{does} \ <\text{role}> \ <\text{move}>\)) terms of a joint move added into it. Unlike the brute-force approach which required enumerating every state in the information set and performing computation separately on each one, this step can be done in a single traversal of the information set’s ZDD thanks to the careful ordering of the terms done at the beginning of the game.

Once each state has been “sprouted” into many separate states (each with a single joint move added) all remaining logical rules (most importantly those for sees and next) can be processed and the results added into each state. This is done by once again traversing the rule trees and the information set’s ZDD recursively in parallel and building the new information set as the stack unwinds, and allows us to avoid performing logical computation on every individual state.

After all the logical rules have been processed, the information set is compared against the percepts received from the server to remove any states that don’t match these percepts. This too can be done with a single traversal of the ZDD.

At this point the player has a ZDD containing one member for every state that will be in the next turn’s information set. To convert it to the base information set for the next round, the ZDD is first filtered to contain only \((\text{next} \ <\text{fact}>\)) terms, and then processed to replace all of those terms with the corresponding \((\text{true} \ <\text{fact}>\)) terms. Careful
CHAPTER 3. METHODS

Listing 3.1: Grounding a GDL-II rule.

;;; Original Rule
1 (<= (goal candidate 100) (true (chosen ?d)) (true (car ?d)))

;;; After Grounding
2 (<= (goal candidate 100) (true (car 1)) (true (chosen 1)))
3 (<= (goal candidate 100) (true (car 2)) (true (chosen 2)))
4 (<= (goal candidate 100) (true (car 3)) (true (chosen 3)))

ordering of the \texttt{(true <fact>)} terms done at the beginning of the game allows these steps to be done with two more traversals of the ZDD.

Once the player has finished it is left with the base information set for the next round, and the process can begin again.

3.3 Grounding GDL-II

Our technique requires the GDL-II rules contain no logical variables. The process of taking a set of logical rules and expanding them into all possibilities to remove the variables is called “grounding” and is a well-studied problem.

Rather than implement the grounding process ourselves we chose to use a standalone grounder, the one used in Fluxplayer [18], to perform the grounding. Fluxplayer uses the Potassco collection of tools under the hood, specifically the unfortunately-named Gringo grounder [19].

Our player simply shells out to Fluxplayer and parses the resulting output back into a set of GDL-II rules. The parsing is tedious but straightforward text parsing and so we will not cover it here—the code\footnote{https://github.com/sjl/scully/blob/master/src/grounders/fluxplayer.lisp} is available for those interested.

Listing 3.1 shows one of the rules from the Monty Hall game presented earlier and the result of grounding it to remove its variable.

3.4 Term Layers

Once the rules have been grounded we partition all of the resulting terms into four layers, summarized in Table 3.1. Each of these layers will be used at a different point in the playing of a game.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>init role true</td>
</tr>
<tr>
<td>possible</td>
<td>terminal goal legal</td>
</tr>
<tr>
<td></td>
<td>does</td>
</tr>
<tr>
<td>happens</td>
<td>sees next</td>
</tr>
</tbody>
</table>

Table 3.1: Term layers and their contents.
3.5. **STRATIFYING LAYERS**

- The base layer of terms is the core game state. It includes all ground terms for the true, role, and init predicates.

- The possible layer consists of all terms necessary to determine terminality of states, goal values of terminal states, and legal moves in nonterminal states. It contains all terms that can be computed directly from the base layer. This includes predicates like legal, goal, and terminal, as well as any other predicates that do not depend on does, sees, or next.

- The does layer contains all terms for the does predicate.

- The happens layer contains all other terms, most importantly those for the sees and next predicates.

To partition the rules into these layers we do the following:

1. Extract all terms in the base layer by finding those with the appropriate predicates.
2. Extract all terms in the does layer similarly.
3. Extract any sees and next terms and put them in the happens layer.
4. Extract all terms that rely only on the base and possible layers into the possible layer. This process is repeated until no further terms can be extracted.
5. Place all remaining terms into the happens layer.

All steps except the extraction of the possible layer are straightforward. We extract the possible layer by creating a directed graph of the dependencies between rules and extracting all leaves of the graph repeatedly until no more remain.

During the partitioning of these layers our player also does some additional sanity checking and adds any missing terms that might be needed. For example: a particular move might appear in a legal term but the corresponding does term might never appear in the grounded rules.

### 3.5 Stratifying Layers

In order to support negation we need to split the possible and happens rules into strata based on their negation\(^3\) dependencies. Any rule containing a negation in one of its bodies must appear in a later stratum than the term it negates [1].

Listing 3.2 shows an example of several simple rules and the resulting stratification. Rules foo and bar have no negation dependencies, so they are included in the lowest stratum. Rule baz depends on the negation of foo, and so must appear in a later stratum. Finally rule quux depends on the negation of baz and foo, and so must appear in a later stratum than both.

---

\(^3\)Both not and distinct are considered negations, though all occurrences of distinct will be optimized away by the grounder.
CHAPTER 3. METHODS

Listing 3.2: Example of layer stratification.

To stratify a layer of rules we first create two directed graphs of all rule heads in the layer:

- The first graph contains an edge from head\(_1\) to head\(_2\) if any body of head\(_1\) includes a negation of head\(_2\) — we call this the “negation dependency graph”.

- The second graph contains an edge from head\(_1\) to head\(_2\) if any body of head\(_2\) includes head\(_1\) at all (negation or otherwise) — we call this the “complete dependency graph”. Note that the edges are in the opposite order of the negation dependency graph.

We then extract a single stratum of heads at a time with the following process:

1. Find all non-leaf terms in the negation dependency graph. These terms are ineligible to be in the current stratum because they directly depend on a negation.

2. Find all nodes in the complete dependency graph that are reachable from any of the terms found in the previous step. These terms are ineligible because they depend on an ineligible term in some way.

3. All remaining terms are eligible to be in the current stratum. Remove them from both graphs and return them.

If we are ever unable to find an eligible head to include in a stratum, but the graphs are not empty, it means a cycle involving negations is present in the rules, and thus the game is not valid GDL.

3.6 Ordering Terms

As noted in the background section on ZDDs, all variables stored in a ZDD must be ordered, and so to store GDL terms we must produce an ordering for them. In our case the ordering is partially constrained by the structure of the game and rules. To produce a viable ordering we first sort the terms in each layer individually, then combine the layers linearly such that base < possible < does < happens.

The terms in the does layer are all of the form (does <role> <move>) and must be sorted according to their role. This makes the construction of ZDDs for joint moves simpler and more efficient. The relative ordering of terms for any particular role is arbitrary.

The order of terms in the possible and happens layers is determined by the stratification of their rules. We compute the ordering in two steps:
1. Any terms in the layer that are not the head of a rule come first. The order within this group is arbitrary.

2. All other terms are rule heads, and are ordered according to the strata they appear in. Ordering within each individual stratum is arbitrary.

Finally, all \((true\ <\ fact>\) terms in the base later must be sorted such that the relative order between them matches the ordering of the corresponding \((next\ <\ fact>\) terms in the \texttt{happens} layer. This allows us to convert a ZDD of \((next\ <\ fact>\) terms into a new information set of \((true\ <\ fact>\) terms in a single traversal.

The end result of this process is a flat, ordered list of all terms in the GDL rules. We assign each term an integer (its index in the sequence) and create two hash maps to allow easy conversion of terms to and from their integers.

As mentioned above, certain parts of the ordering are arbitrary and do not affect the correctness of our method, but the ordering of variables can have a large impact on the size of the resulting ZDDs\(^4\), which impacts both space and runtime requirements.

\section*{3.7 Representing Information Sets as ZDDs}

With a mapping of terms to integers in hand we can easily represent information sets with ZDDs. Each member of an information set is a set of GDL terms, and we can create and manipulate them with traditional ZDD operations.

To find the initial information set of a game we find all \((init\ <\ fact>\) terms in the rules, convert them to \((true\ <\ fact>\) versions, and map those to the appropriate integers. We then create a ZDD containing a single member with these terms — this can be done with fundamental ZDD operations as described in [7] or by more efficient methods in a ZDD library.

\section*{3.8 Logical Reasoning on ZDDs}

Once we have a ZDD representing an information set we would like to be able to apply the rules of a game to it without expanding the entire tree and computing the results for each state one-by-one.

For our purposes in the next few sections we will consider a “rule” to be the disjunction of all bodies for a particular predicate. For example: Listing 3.3 defines a single rule with head \((goal\ candidate\ 0\) and three bodies. Any one of the bodies being true is sufficient to satisfy the rule.

\footnote{As well as the size of rule trees, which will be described shortly.}
CHAPTER 3. METHODS

Listing 3.3: A single rule from the Monty Hall Game.

1 (<= (goal candidate 0))
2  (true (chosen 1))
3  (not (true (car 1))))
4
5 (<= (goal candidate 0))
6  (true (chosen 2))
7  (not (true (car 2))))
8
9 (<= (goal candidate 0))
10 (true (chosen 3))
11 (not (true (car 3))))

By “applying a rule to an information set” we mean that given an information set \(I\) and rule \(R\) we would like to produce a new information set \(I'\) such that any members of \(I\) whose terms satisfy the body of \(R\) have the head of \(R\) added into them.

For example: consider the information set \[\{\{a, b\}, \{b, c\}, \{c, d\}\}\] and the rule \((<= x (or a d))\). In this information set the members \(\{a, b\}\) and \(\{c, d\}\) satisfy the body of rule and so should have the head \((x)\) added into them, while \(\{b, c\}\) does not and should remain unchanged. Thus the result of applying this rule to the information set should be \[\{\{a, b, x\}, \{b, c\}, \{c, d, x\}\}\].

More formally, we define the application of a rule with head \(R_{head}\) and body \(R_{body}\) to an information set \(I\) to mean:

\[
\{\{R_{head}\} \cup S \mid S \in I \text{ and } S \text{ satisfies } R_{body}\} \cup \{S \mid S \in I \text{ and } S \text{ does not satisfy } R_{body}\}
\]

3.8.1 Representing GDL Rules as Rule Trees

We start by representing a GDL rule as a tree-like data structure containing three types of nodes:

- A negative sink is a leaf node, and contains no extra data.
- A positive sink is a leaf node, and contains the head of the rule.
- Internal nodes contain a single body term \(R_{term}\) and two pointers to child nodes, called “low” and “high”. We denote an internal node as \((R_{term}, R_{high}, R_{low})\). These pointers must point to a sink or to an internal node with a higher body term.

Like ZDDs we ensure that the terms in the tree are ordered. For simplicity our implementation does not require the trees to share structure, but this could be added as a performance optimization if desired.

The rule tree nodes will be traversed in parallel with the information set’s ZDD, with a node’s high pointer being followed for those subsets of the information set that contain its term, and the low pointer being followed for those that don’t. Reaching the positive sink of a rule tree means that subset of the information set satisfied the rule, and reaching the negative sink means it did not. The process will be described in more detail in later sections.

---

5 Or any one of the bodies, for logical disjunctions.
6 If not already present.
7 The \(\text{or}\) predicate is a non-standard extension to GDL that many games use as a shorthand for defining multiple separate bodies.
Listing 3.4 and Figure 3.2 show a simple rule from Tic Tac Toe and its corresponding rule tree.

Listing 3.4: A single ground rule from Tic Tac Toe.

```
1 (<= (legal xplayer (mark 1 1))
2  (true (cell 1 1 b))
3  (true (control xplayer)))
```

Figure 3.2: The rule tree for one rule in Tic Tac Toe.

GDL rules support logical disjunction, so rule trees may contain more than one path to the positive sink. Listing 3.5 and Figure 3.3 show a rule from Tic Tac Toe involving disjunction and its corresponding rule tree.

Listing 3.5: A disjunctive rule from Tic Tac Toe.

```
1  (<= (next (cell 1 1 x))
2   (does xplayer (mark 1 1))
3   (true (cell 1 1 b)))
4
5  (<= (next (cell 1 1 x))
6   (true (cell 1 1 x)))
```

Figure 3.3: The rule tree for a disjunctive rule in Tic Tac Toe.

---

8The ordering of the terms in these simple rule tree examples is arbitrary.
Rule trees can also support rules that depend on the negation of terms, as shown in Listing 3.6 and Figure 3.4. Note how the path to the head follows the high or low pointers, depending on the negation in the rule definition.

Listing 3.6: A rule involving negation from Monty Hall.

```
1 (<= (legal random (open_door 1))
2  (true (closed 1))
3  (true (step 2))
4  (not (true (chosen 1)))))
5  (not (true (car 1)))))
```

![Figure 3.4: The rule tree for a rule involving negation from Monty Hall.](image)

### 3.8.2 Splitting Rule Trees

The structure of rule trees means that in the worst case a tree can require $O(2^n)$ nodes, where $n$ is the number of terms in its body, including all disjunctions. Many games contain at least a few rules with a large number of disjunctions, which presents a problem when trying to construct their rule trees.

To prevent a rule tree from exploding exponentially, rules with many disjunctions can be divided into several pieces by creating new rules, each of which is a disjunction of several of the original bodies. The original rule’s body is then replaced by a disjunction of the new rules. Listing 3.7 shows an example of a problematic rule, and Listing 3.8 shows the result of splitting it into smaller rules. Splitting rules into smaller rules allows rule trees to be made smaller, but requires more work at runtime during the rule application process.

Listing 3.7: A rule with many disjunctions.

```
1 (<= some-rule a)
2 (<= some-rule b)
3 (<= some-rule c)
4 (<= some-rule d)
5 (<= some-rule e)
6 (<= some-rule f)
7 (<= some-rule g)
8 (<= some-rule h)
```
3.8. LOGICAL REASONING ON ZDDS

Listing 3.8: Splitting a rule into smaller rules.

```plaintext
1 (<= some-rule rule-1)
2 (<= some-rule rule-2)
3
4 (<= rule-1 a)
5 (<= rule-1 b)
6 (<= rule-1 c)
7 (<= rule-1 d)
8
9 (<= rule-2 e)
10 (<= rule-2 f)
11 (<= rule-2 g)
12 (<= rule-2 h)
```

3.8.3 Applying Single Rule Trees

To apply a rule tree to an information set we traverse the tree and the ZDD recursively in parallel, constructing the result as the stack unwinds much like other ZDD operations.

The intuition is that when we look at an internal node of the rule tree, we split the traversal of the ZDD into two branches: those sets in the ZDD that contain the body term and those that do not. Eventually we will reach a sink in the rule tree, at which point we are either finished (for the negative sink) or should add the head into the remainder of the ZDD (for the positive sink).

The basic algorithm for applying a single rule tree $R$ to an information set’s ZDD $Z$ is as follows:

1. If $R$ is the negative sink, return $Z$.
   *Hitting the negative sink of the rule tree means the rule was not satisfied, and so this branch will not have the rule’s head added.*

2. If $R$ is the positive sink containing head $R_{\text{head}}$, return $Z \sqcup \{R_{\text{head}}\}$.
   *Hitting the positive sink of the rule tree means this branch satisfied the rule, and so will have the rule’s head added into it.*

3. Otherwise $R$ is an internal node $(R_{\text{term}}, R_{\text{high}}, R_{\text{low}})$. Examine $Z$:
   a) If $Z$ is $\bot$, return $Z$.
      *If the ZDD is the empty family, there are no member sets to add the rule head into.*
   b) If $Z$ is $\top$, recur on $R_{\text{low}}$ and $Z$.
      *If the ZDD is the unit family there are no remaining terms, so all subsequent terms in the rule tree are missing.*
   c) Otherwise $Z$ is an internal node $(Z_{\text{term}}, Z_{\text{high}}, Z_{\text{low}})$. Compare $R_{\text{term}}$ to $Z_{\text{term}}$:
      i. If $Z_{\text{term}} < R_{\text{term}}$:
         *This term in the information set does not matter to the rule at all.*
         A. Recur on $R$ and $Z_{\text{high}}$ to produce $Z'_{\text{high}}$.
         B. Recur on $R$ and $Z_{\text{low}}$ to produce $Z'_{\text{low}}$.

---

*We might also reach a sink in the ZDD while we’re still at an internal node of the rule tree. In this case there are no more terms in the information set, and so we can follow the low pointers in the rule tree down to a sink.*
C. Return a new ZDD node \((Z_{\text{term}}, Z'_{\text{high}}, Z'_{\text{low}})\).

ii. If \(Z_{\text{term}} > R_{\text{term}}\), recur on \(R_{\text{low}}\) and \(Z\).

The information set must not contain \(R_{\text{term}}\) at all.

iii. Otherwise \(Z_{\text{term}} = R_{\text{term}}\):

A. Recur on \(R_{\text{high}}\) and \(Z_{\text{high}}\) to produce \(Z'_{\text{high}}\).

B. Recur on \(R_{\text{low}}\) and \(Z_{\text{low}}\) to produce \(Z'_{\text{low}}\).

C. Return a new ZDD node \((Z_{\text{term}}, Z'_{\text{high}}, Z'_{\text{low}})\).

### 3.8.4 Rule Tree Application Example

Before moving on it may be helpful to look at an example of applying a simple rule tree to a small information set, to see the algorithm in action. We will consider the information set \(\{\{a, b\}, \{a, c\}, \{b, c\}\}\) and the rule shown in Listing 3.9.

Listing 3.9: A simple logical rule.

```
1 (<= h
2  a   b)
4
5 (<= h
6  c
7   (not b))
```

Figure 3.5 shows the rule tree and ZDD for this example. We begin by examining both and noting that neither is a sink, and also that their terms match, and so we need to recursively descend into both branches. The order of the recursion is unimportant, so we will look at the high branches first.

Figure 3.6 shows the next rule tree and ZDD. As we continue traversing the information set it is helpful to keep in mind what we have seen so far. We have just followed the high pointer from the node \(a\), and so are currently considering all members of the information set containing the term \(a\). Once again we must recur, and again we will arbitrarily choose the high branch.

Figure 3.7 shows the next step. The ZDD is the unit sink, and so we are considering a single member of the information set: \(\{a, b\}\). The rule tree is the positive sink containing
3.8. LOGICAL REASONING ON ZDDS

Figure 3.6: The rule tree and information set after recurring high.

```
Figure 3.7: The rule tree and information set after recurring high, then high.
```

the head of the rule, which means this member satisfies the rule and so should have the head added into it. This is indeed the case, as \{a, b\} satisfies (<= h a b).

At this point we will backtrack and examine another branch of recursion: the low branch from Figure 3.6.

```
Figure 3.8: The rule tree and information set after recurring high, then low.
```

Figure 3.8 shows this step. We are examining all members of the information set that contain a and do not contain b. Again we must recur and will choose the high branch first.

Figure 3.9 shows the branch. We are again considering one member of the information set: \{a, c\}. Once again this member satisfies the rule (specifically (<= h c (not b))) and so the ZDD will have the rule head h added into it.

Returning to Figure 3.8, we now consider the low branch.

Figure 3.10 shows the step, and is an interesting case. The information set is the empty sink, because there is no member of the original information set \{\{a, b\}, \{a, c\}, \{b, c\}\} that
contains \(a\) but does not contain \(b\) or \(c\). The rule tree is the negative sink, which means that even if the information set did contain such a member (e.g. \(\{a, d\}\)) it would never be able to satisfy either of the bodies of the rule.

At this point we are ready to return to the root nodes in Figure 3.5 and examine the low branch of recursion.

Figure 3.13 shows this branch, and reveals a new case we have not yet seen. We reached this state by recurring low twice, and so are considering members of the information set that contain neither \(a\) nor \(b\). The rule tree is an internal node which means that if such a member
3.8. LOGICAL REASONING ON ZDDS

Figure 3.12: The rule tree and information set after recurring low, then high.

Figure 3.13: The rule tree and information set after recurring low, then low.

existed, it could potentially satisfy the rule. However the information set is the empty sink, which means that no such members exist, and we can simply return.

After unwinding the stack and constructing the results as specified by the algorithm we are left with a new information set \( \{ \{a, b, h\}, \{a, c, h\}, \{b, c\} \} \).

3.8.5 Applying Multiple Rule Trees

Applying rules to information sets one-by-one would result in many traversals of the information set, much of which would be needless work. Instead we would like to apply an entire set of rule trees \( R \) to a ZDD \( Z \) in a single traversal of \( Z \).

We do this with a recursive algorithm on \( R, Z, \) and a set of heads to add at the end \( H \) (initially \( \emptyset \)):

1. If \( Z = \bot \), return \( \bot \).
2. If \( Z = \top \), return a ZDD containing a single member: \( H \).
3. Otherwise \( Z \) is an internal node \((Z_{\text{term}}, Z_{\text{high}}, Z_{\text{low}})\). Process \( R \):
   a) Advance all rule trees in \( R \) up to (but not past) \( Z_{\text{term}} \) to produce \( R' \) and \( H' \).
   b) Split \( R' \) for the high branch, producing \( R'_{\text{high}} \) and \( H_{\text{high}} \).
   c) Split \( R' \) for the low branch, producing \( R'_{\text{low}} \) and \( H_{\text{low}} \).
   d) Recur on \( R'_{\text{high}}, Z_{\text{high}}, \) and \( H \cup H' \cup H_{\text{high}} \) to produce \( Z'_{\text{high}} \).
   e) Recur on \( R'_{\text{low}}, Z_{\text{low}}, \) and \( H \cup H' \cup H_{\text{low}} \) to produce \( Z'_{\text{low}} \).
f) Return a new ZDD node \((Z_{\text{term}}, Z'_{\text{high}}, Z'_{\text{low}})\).

In step 3.a we must advance the set of rule trees up to a given term. This is done by traversing the rule tree and taking the low branch for each internal node until we reach a sink or an internal node whose term is greater than or equal to the target term. If a positive sink is reached during this process, the rule was satisfied and its head will be added. If the negative sink is reached the rule cannot be satisfied and it is removed from consideration.

In steps 3.b and 3.c we have a set of rule trees, all of which are internal nodes with a term greater than or equal to the target term. For the high branch of the ZDD we must traverse down the high branch for any rule trees whose term is equal to the target term, and similarly down the trees’ low branches for the low branch of the recursion. Like before this can result in hitting a sink or another internal node, and the semantics are the same: hitting the positive sink results in adding the head, the negative sink removes the tree, and an internal node passes along the tree to the next step.

Interested readers are encouraged to peruse the code for a complete picture of all the details of this process.

3.8.6 Handling Cycles & Negation

The algorithms described so far will work for rules that do not contain cycles, but since GDL allows cycles in rules (as long as those cycles do not involve negation) we would like to be able to handle rules that contain them. We must also handle negation correctly to ensure that rules depending on the negation of a particular term are computed properly. The stratification of rule layers we described earlier allows us to handle these problems.

To account for cycles in the rules we must modify case 2 of the previous algorithm. At this point in the algorithm we have traversed the existing information set to the unit sink and have accumulated a set \(H\) of rule heads to add along the way. We may also be left with a set of remaining rule trees \(R\) that have not yet been satisfied or disproven. We treated \(R\) as a set in the previous algorithm, but in reality it is divided into one or more strata, each of which is itself a set.

Our goal now is to examine the first stratum and find any new heads we may be able to add based on the heads we accumulated along the way \((H)\). If any new heads can be added we repeat the process iteratively until no further heads can be added. We repeat this for each successive stratum. This fixed-pointing process accounts for any (positive) cyclic dependencies in the rules.

Because the rules were stratified based on negation we can assume that any negations we encounter during the process can be checked by simply looking in \(H\). A negation cannot depend on a head that might be added in a later iteration, because all negation dependencies occur in a previous stratum.

---

10The low branch is taken because all terms less than the target must be missing from this portion of the information set, otherwise they would have been seen in a previous step of the recursion.

11Rule trees whose term is greater than the current target term do not involve this term at all, and so are passed along unchanged.

12The “Logic Application” section of https://github.com/sjl/scully/blob/master/src/reasoners/zdd.lisp is the relevant portion, and totals about 200 lines.
3.9 Sprouting Joint Moves

Another operation needed to play a GDL game is the sprouting of each state in the information set into several new states, each of which has one possible joint move added into it. These new states will then be filtered to include only those which match the percepts given by the server.

We can perform this operation in a single traversal of the ZDD thanks to the careful ordering of the terms we performed earlier: the sprouting operation happens after the possible layer of rules has been computed, and the joint moves being added consist of \((\text{does} \ <\text{role}>\ <\text{move}>)\) terms, all of which are in the \text{does} layer.

To perform the sprouting we recursively traverse the information set ZDD, accumulating lists of legal moves for each role as we descend. When we reach the sinks of the ZDD we construct a child ZDD consisting of all legal joint moves in that branch, which can also be done efficiently because of the ordering of the \text{does} layer.

The algorithm is recursive on the ZDD \(Z\) and the set of legal moves accumulated so far \(L\) (initially \(\emptyset\)):

1. If \(Z\) is \(\perp\), return \(\perp\).

2. If \(Z\) is \(\top\), return a new ZDD containing all possible joint moves.

3. Otherwise \(Z\) is an internal node \((Z_{\text{term}}, Z_{\text{high}}, Z_{\text{low}})\). Examine \(Z_{\text{term}}\):
   a) If \(Z_{\text{term}}\) is of the form \((\text{legal} \ <\text{role}>\ <\text{move}>)\):
      i. Recur on \(Z_{\text{high}}\) and \(L \cup \{Z_{\text{term}}\}\) to produce \(Z'_{\text{high}}\).
      ii. Recur on \(Z_{\text{low}}\) and \(L\) to produce \(Z'_{\text{low}}\).
      iii. Return a new ZDD node \((Z_{\text{term}}, Z'_{\text{high}}, Z'_{\text{low}})\).
   b) Otherwise \(Z_{\text{term}}\) does not denote a legal move:
      i. Recur on \(Z_{\text{high}}\) and \(L\) to produce \(Z'_{\text{high}}\).
      ii. Recur on \(Z_{\text{low}}\) and \(L\) to produce \(Z'_{\text{low}}\).
      iii. Return a new ZDD node \((Z_{\text{term}}, Z'_{\text{high}}, Z'_{\text{low}})\).

To construct the ZDD of all possible joint moves in step 2 from a set of legal moves \(L\), we first group the moves by role and then work bottom up. We construct a ZDD where each role selects exactly one of the available moves.

Figure 3.14 shows an example of a ZDD of all possible joint moves for two roles, each role having three legal moves. For the highest-ordered role, the high edge for each legal move leads to the unit sink, while the low edge leads to the next possible move. Because the role must choose exactly one move, the final move’s low edge leads to the empty sink. Other roles follow the same pattern, but with the high edges leading to the next role’s first node instead of the unit sink.
3.10 Filtering on Percepts

The next operation needed to play a full GDL game is to take an information set and filter it to contain only those states that exactly match the percepts for the player’s role given by the server.

To handle this we will define a more general operation on ZDDs called “match in universe”. This operation will take a ZDD $Z$, a set of terms to match $S$, and a universe (set) of terms $U$ and return a new ZDD $Z'$ containing $\{\alpha \mid \alpha \in Z, \alpha \cap U = S\}$. We define the algorithm recursively as follows:

1. If $Z$ is $\bot$, return $\bot$. 
   *The ZDD has no members at all.*

2. If $Z$ is $\top$, examine $U$:
   *The ZDD is $\{\emptyset\}$.*
   
   (I) If $U = \emptyset$, return $\top$.
   *The only set of terms $\emptyset$ can successfully match is the empty set.*
   
   (II) Otherwise return $\bot$.

3. Otherwise $Z$ is an internal node $(Z_{\text{term}}, Z_{\text{high}}, Z_{\text{low}})$:
3.11 Successor Computation

(I) If $Z_{\text{term}} \notin U$:
   This is a term which is not in the universe and thus is irrelevant to the matching process.
   (A) Recur on $Z_{\text{high}}$, $S$ and $U$ to produce $Z'_{\text{high}}$.
   (B) Recur on $Z_{\text{low}}$, $S$ and $U$ to produce $Z'_{\text{low}}$.
   (C) Return a new ZDD node $(Z_{\text{term}}, Z'_{\text{high}}, Z'_{\text{low}})$.

(II) Otherwise $Z_{\text{term}} \in U$:
   (A) If $S = \emptyset$, recur on $Z_{\text{low}}$, $S$, and $U$.
      This term is in the universe, but must be excluded from the match, so we recur low.
   (B) Otherwise $S$ is a non-empty set with a smallest element $s$:
      (i) If $Z_{\text{term}} < s$, recur on $Z_{\text{low}}$, $S$, and $U$.
          This term is in the universe, but must be excluded from the match, so we recur low.
      (ii) If $Z_{\text{term}} > s$, return $\bot$.
          At this point the ZDD has passed the smallest term we are trying to match, and so will never include it.
      (iii) Otherwise $Z_{\text{term}} = s$:
          This term is one that we would like to match, so we recur along the high branch of the ZDD.
          (a) Recur on $Z_{\text{high}}$, $S \setminus \{s\}$ and $U$ to produce $Z'_{\text{high}}$.
          (b) Return a new ZDD node $(Z_{\text{term}}, Z'_{\text{high}}, \bot)$.

Once this operation is defined, filtering an information set based on percepts is straightforward. When the game begins we extract all possible percepts for the player’s role from the grounded rules — this is the role’s universe of percepts for the game. The percept filtering at each turn is then simply an application of the match in universe operation to the information set, the percepts received from the server, and the role’s universe of percepts.

3.11 Successor Computation

Once all states not fitting the percepts have been removed the only remaining step is to translate each state in the information set into its successor. We do this in two steps: first we filter every state in the information set to contain only $(\text{next } \text{<fact>})$ terms, then we convert each of these terms to the corresponding $(\text{true } \text{<fact>})$ term.

To assist with the filtering step we precompute an additional ZDD $Z_{\text{next}}$ at the beginning of the game. This ZDD contains a single member: the set of all $(\text{next } \text{<fact>})$ terms in the grounded GDL. Filtering an information set $Z$ to contain only next terms then simply uses the standard ZDD meet operation: $Z \cap Z_{\text{next}}$.

Converting next terms to the corresponding true terms can be done in a single traversal of the resulting ZDD, because we ensured these terms have the same relative order when we ordered terms earlier. The algorithm is recursive on the ZDD $Z$: 
1. If $Z$ is ⊥, return ⊥.

2. If $Z$ is ⊤, return ⊤.

3. Otherwise $Z$ is an internal node $(Z_{\text{term}}, Z_{\text{high}}, Z_{\text{low}})$, with $Z_{\text{term}}$ being of the form (next <fact>).

   a) Look up the corresponding (true <fact>) term for $Z_{\text{term}}$ to produce $Z'_{\text{term}}$.
   b) Recur on $Z_{\text{high}}$ to produce $Z'_{\text{high}}$.
   c) Recur on $Z_{\text{low}}$ to produce $Z'_{\text{low}}$.
   d) Return a new ZDD node $(Z'_{\text{term}}, Z'_{\text{high}}, Z'_{\text{low}})$. 
Chapter 4

Results

4.1 Information Set Sizes

One of the goals of using ZDDs to represent information sets is to take advantage of their structural sharing to reduce the amount of information that must be stored.

Table 4.1 shows the results of one playout of `montyhall.gdl` as a simple example to start. The turn and number of states in the information set are shown, along with the number of ZDD nodes required to represent it. The final column (vanilla size) is the sum of the number of terms for each state in the information set. This gives an idea of how many linked list nodes would be required to store the information set in a more traditional manner.

<table>
<thead>
<tr>
<th>Turn</th>
<th>States</th>
<th>ZDD Nodes</th>
<th>Vanilla Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 4.1: Information set sizes for one play of `montyhall.gdl`.

On the first turn the information set contains a single initial state with four terms. There are six ZDD nodes required: one for each term plus the two sinks.

On the second turn Monty has chosen where to place the car, and the information set now contains three members (one for each possible location). The ZDD is now more compact than a vanilla linked list would be, requiring only ten nodes to the vanilla list’s 18.

On turn three the player has chosen a door and Monty has revealed one of the others, which the player sees as a percept. This narrows the information set down to two members, and once again the ZDD is (slightly) more compact.

On the final turn the game is over, and the player has seen the result.

Next we will look at a game that produces information sets with more states: `mastermind448.gdl`. Figure 4.1 compares the number of ZDD nodes with the vanilla size for several playouts. As the size of the information set increases the number of ZDD nodes and vanilla objects required to store it both increase, but the ZDDs increase more slowly.

Figures 4.2 and 4.3 show the same information for several playouts of two more games.

Unfortunately the amount of information required to store the basic information set does not tell the entire story. Table 4.2 returns to `mastermind448.gdl`, showing the result of one playout. The additional column shows the maximum number of ZDD nodes required...
CHAPTER 4. RESULTS

Figure 4.1: Information set storage requirements for mastermind448.gdl.

Figure 4.2: Information set storage requirements for transit.gdl.
4.1. INFORMATION SET SIZES

Figure 4.3: Information set storage requirements for latenttictactoe.gdl.

Once again the ZDD is more compact at representing the information sets when they contain multiple members. The difference can become quite pronounced — on turn five the information set contains 256 states which the ZDD represents with 97 nodes, while a linked list would require 1280 nodes.

<table>
<thead>
<tr>
<th>Turn</th>
<th>States</th>
<th>ZDD Nodes</th>
<th>Vanilla Size</th>
<th>Maximum ZDD Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>n/a</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>38</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>18</td>
<td>64</td>
<td>167</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>42</td>
<td>320</td>
<td>746</td>
</tr>
<tr>
<td>5</td>
<td>256</td>
<td>97</td>
<td>1280</td>
<td>2994</td>
</tr>
<tr>
<td>6</td>
<td>81</td>
<td>52</td>
<td>486</td>
<td>70404</td>
</tr>
<tr>
<td>7</td>
<td>24</td>
<td>36</td>
<td>144</td>
<td>22784</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>22</td>
<td>48</td>
<td>6770</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>13</td>
<td>24</td>
<td>2262</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>9</td>
<td>12</td>
<td>1133</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>9</td>
<td>12</td>
<td>567</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>8</td>
<td>6</td>
<td>569</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>8</td>
<td>6</td>
<td>289</td>
</tr>
</tbody>
</table>

Table 4.2: Information set sizes for one play of mastermind448.gdl.

The main limitation of our method can be seen in the last column: adding all the logical terms by applying the rule trees can cause the ZDDs to become extremely large before being filtered back down to the base layers. This could be improved by intertwining the application
CHAPTER 4. RESULTS

<table>
<thead>
<tr>
<th>Turn</th>
<th>States</th>
<th>ZDD Nodes</th>
<th>Vanilla Size</th>
<th>Maximum ZDD Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>n/a</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>55</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>97</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>13</td>
<td>32</td>
<td>164</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>13</td>
<td>32</td>
<td>165</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>17</td>
<td>48</td>
<td>303</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>17</td>
<td>48</td>
<td>209</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>21</td>
<td>64</td>
<td>441</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>21</td>
<td>64</td>
<td>277</td>
</tr>
<tr>
<td>10</td>
<td>18</td>
<td>23</td>
<td>72</td>
<td>583</td>
</tr>
<tr>
<td>11</td>
<td>18</td>
<td>23</td>
<td>72</td>
<td>365</td>
</tr>
<tr>
<td>12</td>
<td>21</td>
<td>27</td>
<td>84</td>
<td>657</td>
</tr>
<tr>
<td>13</td>
<td>20</td>
<td>26</td>
<td>80</td>
<td>425</td>
</tr>
<tr>
<td>14</td>
<td>25</td>
<td>32</td>
<td>100</td>
<td>731</td>
</tr>
<tr>
<td>15</td>
<td>24</td>
<td>31</td>
<td>96</td>
<td>585</td>
</tr>
<tr>
<td>16</td>
<td>28</td>
<td>35</td>
<td>112</td>
<td>871</td>
</tr>
<tr>
<td>17</td>
<td>28</td>
<td>35</td>
<td>112</td>
<td>567</td>
</tr>
<tr>
<td>18</td>
<td>29</td>
<td>36</td>
<td>116</td>
<td>995</td>
</tr>
<tr>
<td>19</td>
<td>29</td>
<td>36</td>
<td>116</td>
<td>587</td>
</tr>
<tr>
<td>20</td>
<td>29</td>
<td>36</td>
<td>116</td>
<td>1022</td>
</tr>
<tr>
<td>21</td>
<td>28</td>
<td>35</td>
<td>112</td>
<td>581</td>
</tr>
<tr>
<td>22</td>
<td>29</td>
<td>36</td>
<td>116</td>
<td>992</td>
</tr>
<tr>
<td>23</td>
<td>28</td>
<td>35</td>
<td>112</td>
<td>681</td>
</tr>
</tbody>
</table>

Table 4.3: Information set sizes for one play of transit.gdl.

<table>
<thead>
<tr>
<th>Turn</th>
<th>States</th>
<th>ZDD Nodes</th>
<th>Vanilla Size</th>
<th>Maximum ZDD Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>30</td>
<td>28</td>
<td>n/a</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>30</td>
<td>28</td>
<td>105</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>177</td>
<td>224</td>
<td>569</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>141</td>
<td>196</td>
<td>728</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>385</td>
<td>588</td>
<td>2638</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>120</td>
<td>168</td>
<td>1730</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>96</td>
<td>140</td>
<td>494</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>175</td>
<td>280</td>
<td>1499</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>102</td>
<td>168</td>
<td>758</td>
</tr>
<tr>
<td>10</td>
<td>24</td>
<td>324</td>
<td>672</td>
<td>1395</td>
</tr>
<tr>
<td>11</td>
<td>36</td>
<td>471</td>
<td>1008</td>
<td>3840</td>
</tr>
<tr>
<td>12</td>
<td>24</td>
<td>306</td>
<td>672</td>
<td>4397</td>
</tr>
</tbody>
</table>

Table 4.4: Information set sizes for one play of latenttictactoe.gdl.

of the happens layer and filtering on percepts, to avoid generating parts of the ZDD that cannot match the percepts.

Tables 4.3 and 4.4 show results for transit.gdl and latenttictactoe.gdl, respectively. The same size patterns can be seen, with the ZDDs requiring fewer nodes to store the basic information state but more nodes when computing the rules (though these games do not explode quite as badly during that phase as mastermind448.gdl).
Several other games were also attempted (vis_pacman3p.gdl and stratego.gdl) but unfortunately turned out to require too much time for computation of the happens layer to be practical.

4.2 Rule Tree Sizes

The size of the rule trees affects both memory requirements (for storing the trees) and run times (for applying them to information sets). Table 4.5 shows the number and total size of rule trees for several games\(^1\) with the default maximum rule size splitting threshold of 8, along with the total number of terms in the grounded GDL.

<table>
<thead>
<tr>
<th>Game</th>
<th>Ground Terms</th>
<th>Rule Trees</th>
<th>Rule Tree Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>montyhall.gdl</td>
<td>162</td>
<td>44</td>
<td>298</td>
</tr>
<tr>
<td>meier.gdl</td>
<td>2569</td>
<td>528</td>
<td>65146</td>
</tr>
<tr>
<td>mastermind448.gdl</td>
<td>21143</td>
<td>2349</td>
<td>95335</td>
</tr>
<tr>
<td>transit.gdl</td>
<td>2930</td>
<td>307</td>
<td>20413</td>
</tr>
<tr>
<td>vis_pacman3p.gdl</td>
<td>45732</td>
<td>3829</td>
<td>197267</td>
</tr>
<tr>
<td>latenttictactoe.gdl</td>
<td>1648</td>
<td>208</td>
<td>445076</td>
</tr>
<tr>
<td>stratego.gdl</td>
<td>45967</td>
<td>4786</td>
<td>1835632</td>
</tr>
</tbody>
</table>

Table 4.5: Rule tree sizes for various games (with default settings).

The threshold of how many disjunctions to allow in a rule’s body before splitting it into separate rules (as described in Section 3.8.2) can have a large impact on the size of the rule trees. Table 4.6 shows the results of building the rule trees for meier.gdl with various splitting thresholds. As the threshold is increased the number of rule trees decreases (because trees are split less often) but the total number of nodes increases.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Number of Rule Trees</th>
<th>Total Rule Tree Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>555</td>
<td>20731</td>
</tr>
<tr>
<td>7</td>
<td>538</td>
<td>41384</td>
</tr>
<tr>
<td>8</td>
<td>528</td>
<td>65146</td>
</tr>
<tr>
<td>9</td>
<td>520</td>
<td>130722</td>
</tr>
<tr>
<td>10</td>
<td>516</td>
<td>281542</td>
</tr>
<tr>
<td>11</td>
<td>501</td>
<td>584505</td>
</tr>
<tr>
<td>12</td>
<td>498</td>
<td>785200</td>
</tr>
</tbody>
</table>

Table 4.6: Effects of splitting threshold on meier.gdl.

The relationship between threshold and sizes is not as simple for every game. Table 4.7 shows the results for mastermind448.gdl.

Generating a set of rule trees usually takes several seconds, so for games where the pre-computation period is long enough a player could potentially try several different thresholds to find one that produced smaller and/or fewer trees.

\(^1\)All games used are from http://ggpserver.general-game-playing.de/ggpserver/public/show_games.jsp
CHAPTER 4. RESULTS

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Number of Rule Trees</th>
<th>Total Rule Tree Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2457</td>
<td>90307</td>
</tr>
<tr>
<td>7</td>
<td>2424</td>
<td>101126</td>
</tr>
<tr>
<td>8</td>
<td>2349</td>
<td>95335</td>
</tr>
<tr>
<td>9</td>
<td>2343</td>
<td>108509</td>
</tr>
<tr>
<td>10</td>
<td>2321</td>
<td>112105</td>
</tr>
<tr>
<td>11</td>
<td>2302</td>
<td>128020</td>
</tr>
<tr>
<td>12</td>
<td>1918</td>
<td>108986</td>
</tr>
</tbody>
</table>

Table 4.7: Effects of splitting threshold on mastermind448.gdl.

4.3 Runtimes

As mentioned in Section 4.1, storing the information set as a ZDD is more compact for the raw information set, but explodes in size as the possible and happens layers are computed. This explosion is unfortunately mirrored in the runtimes, which makes our technique impractical for most games without addressing this issue.

Table 4.8 compares run times for a Prolog-based player and our ZDD-based one. It also includes garbage collection times, which are significantly long for the ZDD player (the Prolog player had no noticeable GC pauses). Both players were modified to select the same moves, and the game controller was seeded with the same seed so that both game playouts would contain the same information sets.

<table>
<thead>
<tr>
<th>Turn</th>
<th>States</th>
<th>Prolog Run Time</th>
<th>ZDD Run Time</th>
<th>ZDD Garbage-Collection Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0.0000</td>
<td>0.0030</td>
<td>0.0020</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>0.0000</td>
<td>0.0080</td>
<td>0.0060</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>0.0040</td>
<td>0.0260</td>
<td>0.0200</td>
</tr>
<tr>
<td>5</td>
<td>256</td>
<td>0.0160</td>
<td>0.1120</td>
<td>0.0910</td>
</tr>
<tr>
<td>6</td>
<td>27</td>
<td>0.0190</td>
<td>70.3660</td>
<td>42.8490</td>
</tr>
<tr>
<td>7</td>
<td>27</td>
<td>0.0040</td>
<td>13.1360</td>
<td>2.0790</td>
</tr>
<tr>
<td>8</td>
<td>27</td>
<td>0.0030</td>
<td>17.9520</td>
<td>2.6850</td>
</tr>
<tr>
<td>9</td>
<td>27</td>
<td>0.0020</td>
<td>13.9220</td>
<td>2.9770</td>
</tr>
<tr>
<td>10</td>
<td>27</td>
<td>0.0040</td>
<td>15.2440</td>
<td>2.9040</td>
</tr>
<tr>
<td>11</td>
<td>27</td>
<td>0.0030</td>
<td>15.6870</td>
<td>3.5830</td>
</tr>
<tr>
<td>12</td>
<td>27</td>
<td>0.0030</td>
<td>13.6030</td>
<td>2.6320</td>
</tr>
<tr>
<td>13</td>
<td>27</td>
<td>0.0030</td>
<td>9.2780</td>
<td>1.0020</td>
</tr>
</tbody>
</table>

Table 4.8: Run times for a single game of mastermind448.gdl (times are in seconds).

There are several reasons for the slow run times, some of which could be fixable with future work. The Prolog player uses an implementation of the Warren Abstract Machine as its Prolog, and the WAM is fairly performance-focused.

In contrast, our ZDD reasoner is aimed at being a proof of concept and uses an inefficient ZDD library in the name of simplicity. During informal profiling we noted that over 90% of the run time when calculating the happens layer was being spent in the new ZDD node creation function, which uses vanilla Lisp hash tables under the hood. Using a ZDD library focused on performance (e.g. CUDD) would likely improve runtimes dramatically.
4.4 Impact of Variable Ordering

While we focused on correctness in our methods and did not investigate any methods for finding optimal variable orderings, we can look at the effect of randomly shuffling the orderings (within the constraints our methods require) to get an idea of whether the orderings matter at all.

Figure 4.4 shows the total number of rule tree nodes for various games when the variable ordering is randomly shuffled, normalized to the unshuffled sizes for comparison. 50 random orderings were evaluated for each game. Boxes denote the 25th and 75th percentiles, with the whiskers marking the 5th and 95th percentiles and the internal line the median value. For example: a value of 1.1 means the rule trees with shuffled orderings have 1.1 times as many nodes as the unshuffled version.

The results vary by game. latenttictactoe.gdl’s rule trees do not vary in size much as the ordering is shuffled, but for other games shuffling the variable orderings can have a significant effect, which suggests some potential for optimization.

---

A significant portion of time is also being spent in our Lisp’s garbage collection routines. The ZDD library uses SBCL’s weak hash tables to handle the structural sharing, and these are often notoriously problematic for GC\(^2\). A more performance-oriented ZDD library would likely implement its own garbage collection, specifically tailored to ZDDs, which could potentially be much faster.

---

\(^2\)https://bugs.launchpad.net/sbcl/+bug/1241771
Chapter 5

Conclusion & Future Work

In this thesis we introduced a technique for representing the information sets of GDL-II games using ZDDs, and a method for reasoning about the game states without enumerating each state individually. The structural sharing of ZDDs allows the information sets of certain games to be represented more compactly, which improves both memory requirements and the processing time.

We did not place a high priority on efficiency in our implementation, preferring to focus on a proof-of-concept. Future work could investigate using a more efficient ZDD implementation with our technique and benchmarking the results.

Much of the efficiency of ZDDs and rule trees depends on the ordering of the variables — different orderings can result in larger or smaller ZDDs and rule trees, as noted in Section 4.4. For example: the rule tree shown in Figure 3.3 would contain one fewer node if \( \text{true (cell 1 1 x)} \) were ordered before \( \text{true (cell 1 1 b)} \). Although our techniques place constraints on the ordering of variables, there is still some freedom in the ordering of terms within individual layers and strata. Future work could investigate methods for finding variable orderings that satisfy our constraints while improving the size of the ZDDs and rule trees.

As noted in Chapter 4 ZDDs are often an efficient way of storing the basic information sets, but start to grow large and require long computations when the happens layer of terms is added. One approach to this issue could be intertwining the process of computing the happens layer and matching based on percepts, allowing computation of non-matching successor states to be cut off much earlier.

Our technique focuses on basic representation and logical reasoning on games. In reality we would also like to play games intelligently, and so another area of future study could be on integrating some variety of Monte Carlo Tree Search [20] with our technique. The simplest approach would be to simply choose arbitrary members from the information set and perform traditional MCTS on them; this would allow traditional MCTS to be adapted to GDL-II easily.
Bibliography


