General Game Player for Board Games

Lukas Berger

Kongens Lyngby 2012
IMM-MSc-2012-0001

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-MSc-2012-0001

Summary

This project presents the implementation of a General Game Player capable of playing an arbitrary board game for two players. The players compete against each other by taking turns. The system uses game descriptions defined by the Game Description Language.

The implementation of the General Game Player was developed in Objective-C and is based on the modified Min-Max algorithm. The modifications adapt the algorithm to be able to find a solution to a given problem without the knowledge specific to a current game. A performance analysis is performed after introducing multithreading to check if utilising modern computer hardware can decrease the time needed to find the solution.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark and at the Tsinghua University in China in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

The thesis deals with the implementation of a system capable of playing an arbitrary board game for two players.

The thesis consists of a report and one application.

Lyngby, 20-September-2012

Lukas Berger

Acknowledgements

I would like to thank everyone who made this thesis possible.

My supervisor Jørgen Fischer Nilsson for guiding me through the whole process and giving me a lot of useful feedback. Thank you for always being there for me whenever I needed help.

I would like to thank Thomas Bolander for allowing me to use his game 'Kolibrat' for the testing purposes.

I would like to thank all my friends for the moral support that they have provided me: Ashley Keufsson, Yuxiao Wang, Emil Rydza, Chiara Cigarini, Elisa Canesci, Bevin Gee, Laurent Arribe, Yu Xin, Paw Berliot Sort Jensen and Jens Kruse Mikkelsen. Thank you for your support.

Last but not least I would like to show my gratitude to my whole family for the love and support I received and especially to my mother Krystyna Berger for always being there for me.

A lot of work done for the project was done during the exchange at the Tsinghua University in Beijing, China. I have to admit that the whole experience was amongst the most amazing things that have happened in my life and I would like to show my gratitude to all of the people I have met there.

Contents

Chapter 1

Introduction

General Game Playing (GGP) is a branch of Computer Science that tries to develop a General Game Playing System that accepts a formal description of a specified game and then plays the game effectively without any human intervention. This intervention might include any changes to the original source code or algorithms used for the system.[1]

Unlike specialised game players, such as Deep Blue (a Chess player), general game players do not rely on algorithms designed in advance for specific games - they are able to play different kinds of games.[1] The General Game Playing System, which can be simply called a Game Agent, should be flexible enough to be able to read a set of specifications describing a game to be played and then based on those rules construct a sequence of actions that will lead it to at least one of the predefined game goals.

The project described in this report is dealing with a specific subsection of General Game Playing (Board Games for two players) and is divided into two main parts. The first part deals with reading and parsing a set of specifications for a provided game and generating a representation of a game world based on those rules. The second part deals with the Game Agent taking actions (legal moves) in the game world processed in the first part. Those actions should create a sequence that will lead to one of the predefined game goals.

The solution to the first part has been mainly provided by scientists from the Stanford University. A set of rules has been formalised in a Game Description Language (GDL in short) that provides a toolset for describing games in a standardised manner. The Game Description Language has been a very important part of this MSc project and will be described in details in the second chapter of this report.

The second part has still a lot of space for improvement. The Stanford University is organising a competition on a yearly basis, where students from around the world try to compete and come up with the best solution to this problem. The results are getting better every year but there is still a long way to go. Implementing and testing this part was one of the main goals for this MSc project. Additionally, ways of improving the solution by utilising a modern computer hardware have also been tested.

General Game Playing is a very important field of Artificial Intelligence. Some real life examples like creating a schedule for a train network or a cargo distribution problem in a warehouse can also be described as a set of rules with the help of the Game Description Language. Once the description of a problem is created it can be used by a well developed Game Agent to try to find a solution to it. The situation described above is out of the scope of this project but it serves as a good example of a potential that GGP has and the project itself can serve as a solid foundation for a future work of this nature.[1]

This report is divided into the following chapters:

Game Description Language this chapter describes the foundations of the Game Description Language that was used for describing the rule's of the games to be played as a set of formalised rules.

Game Playing in this chapter a theoretical background for the algorithms and methods used for the project are described.

Implementation this chapter describes the design decisions and details used for the implementation of the project.

Discussion is a chapter where encountered problems and their solutions are described. Additionally, some results are also described in this section.

Summary summarises the work done on the project and proposes a future work that can be done to extend it.

Chapter 2

Game Description Language

Board Games for two players can be described with a certain set of rules. For each state of the game a set of legal moves can be applied and the players have a full view of the game environment at any time. Additionally, one of the properties of the board games dictates that players compete against each other in turns, meaning that if one player takes an action then the other Player has to wait.

In theory, for such an environment it should be possible to derive a directed graph. At any time of the game a unique state can be created for the game environment with outgoing arcs representing a move made by each player during the duration of his turn. Each node of the graph (state) would hold facts that are true in a current development of the game world. Since the number of players is limited to two and their percepts and actions are finite, the graph will have a finite size. This means that in theory it is possible to represent a whole game world (with all of it's developments at any time) with such a graph. A partial example of such a graph for a simple game like Tic-Tac-Toe is presented in figure 2.1 below.

However, the graph described above has it's limitations. For games like Tic-Tac-Toe where the number of states (state space) is just 26830 it is possible to generate and store the whole graph. For more complex games like Chess the state space increases dramatically to 10^{28}, which is impractical for a direct use

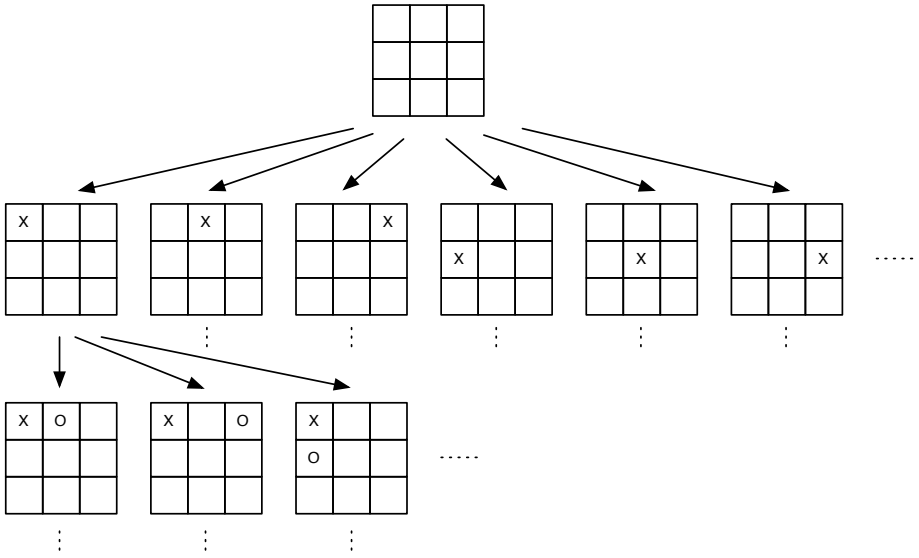

Figure 2.1: A simplified example of the Tic-Tac-Toe's game state graph. The initial state is shown on top and the successor states below. Note that this example is only a partial representation of the state graph for Tic-Tac-Toe.

in reasoning.

This limitation was one of the reasons why the Game Description Language was created. GDL allows General Game Playing reasoning to be performed efficiently providing a set of rules for a compact and modular representation of the game world and its dynamics. GDL also provides a standardised way of representing a game world for any game. This property can be used as a foundation for developing a General Game Player.[2]

2.1 Background

2.1.1 Datalog

The Game Description Language is based on the Datalog query language, which was previously used for deductive database systems. Datalog is a solid foundation for the GDL since it's main goal was to make deductions based on the rules

and facts stored in a database. The Game Description Language introduces new elements to the original language.

Datalog itself is a very complex language and not all of it's elements have been adopted in GDL. It is important to summarise and briefly describe the ones that laid the foundations for the Game Description Language. The naming conventions have been modified and adapted for the purposes of this project.

Variable Denoted by a capital letter. The scope of a variable is not limited to any specified type. Referred as Object Variable in Datalog terminology. *(e.g. X, Y, Z)*

Constant Denoted by a lowercase letter or a string starting with a lowercase letter. Referred as Constant Variable in Datalog terminology. *(e.g. a, b ,c)*

Sentence Denoted by a lowercase letter or a lowercase string. Sentence is used to bound variables and constants together. It is possible for a sentence not to have any parameters. Datalog distinguishes two types of sentences: a function (takes both variables and constants) and a relation (bounds only constants together) but for the purposes of GDL and this project those two types have been merged together into a sentence. *(e.g. father(John, Jim), distinct(n, N))*

Rule It is a logical implication of the following form: a $\Leftarrow$ b $\wedge$ ... $\wedge$ c. The implied value - 'a' is called a head of the rule and composes of only one Sentence described above. It is not allowed to put a negated sentence in the head of the Rule. The sentence 'a' is true if and only if all of the sentences - b $\wedge$ ... $\wedge$ c, called the tail of the rule, are also true. There is no limit on how many sentences there are in the Rule's tail. Referred as Datalog Rule in Datalog terminology. *(e.g. (father(John, X) <= son(X, John)))*

Disjunction Provides a mechanism of connecting multiple sentences together and will hold if at least one of the sentences in it is true. *(e.g. (son(John, Jim) <= (father(John, Jim) | father(Jim, John))))* A Disjunction can only appear in the Datalog's Rule tail as it's task is to enable two or more rules to be joint together. *(e.g. (son(John, Jim) <= (father(John, Jim))) or (son(John, Jim) <= father(Jim, John))))*

Negation Negates the truth value of the sentence. *(e.g. ˜father(Jim,*

John))

2.1.2 Knowledge Interchange Format

Knowledge Interchange Format (KIF) is a language designed for use in the interchange of knowledge among disparate computer systems (created by different programmers, at different times, in different languages, and so forth).[3]

The Game Description Language is becoming a universal language used for many projects. The latest specification of the language has adopted the Knowledge Interchange Format for it syntax and semantics.[2]

The project described in this report is dealing with an implementation of the General Game Player, the syntax and semantics of Datalog are represented in compliance with the Knowledge Interchange Format (KIF), which makes reading of the language clear for both computers and humans.

A list below presents a comparison of Datalog syntax between the original and the KIF form.

Variable KIF allows variables to be denoted by a lowercase letter or a lowercase string with a question mark in front.

```
1 DATALOG: X, Y, P
2 KIF: ?x, ?y, ?player
```

Constant KIF allows constants to be denoted by a single letter or a string.

```
1 DATALOG: a, b, white
2 KIF: a, B, White
```

Sentence In KIF, sentences are enclosed in brackets whenever they have parameters. There are no commas used to separate components of the sentence.

```
1 DATALOG: father(John, Jim)
2 KIF: (father John Jim)
```

Rule In KIF, all the sentences in the Datalog rule's head and tail have to be adopted to KIF. Additionally, the implication sign is moved in front of the whole sentence.

```
1 DATALOG: (father(John, Jim) <= son(Jim, John))
2 KIF: (<= (father John Jim)(son Jim John))
```

Disjunction In KIF the sentences that are the components of the disjunction are transformed into KIF. Disjunction uses the 'or' prefix before the sentences.

```
1 DATALOG: (father(John, Jim) | father(Jim, John))
2 KIF: (or (father John Jim)(father Jim John))
```

Negation In case of negation, again, the sentence is converted to KIF and the negation sign is transformed into 'not' prefix.

```
1 DATALOG: ~father(Jim, John)
2 KIF: not (father Jim John)
```

2.2 Specification

GDL describes the state of a game world in terms of a set of facts. The transition function between states (the rules of the game) are described using logical rules in GDL. These rules define a set of facts (that are true in the next state) in terms of the current state and the move of the player.[2]

These rules allow to deduct the next state based on the move applied in the current state. The elements of the Game Description Language the initial state, the transition functions allow a creation of a graph described in the beginning of Chapter 2. Additionally, GDL introduces a way of controlling and checking if a certain move can be performed by describing the required conditions for it. GDL language is also used to describe the terminal states, to define legal moves and to set goal conditions.[2]

2.2.1 GDL Additions

One of the main differences between Datalog and GDL is the fact that the Game Description Language allows sentences to be nested in each other, while Datalog does not. Using this property it is possible to define a certain set of control sentences that take another sentence as a parameter. For example a nested sentence like: initial(cell 1 1 blank) can be used to describe the initial state of the game.

GDL is able to define a set of keywords that when used as a name of the Sentence can be transformed into a set of rules of a game. The next subsections will define and explain all the GDL keywords and give an example for all of them based on a simple game of Tic-tac-Toe. For a full GDL description of Tic-Tac-Toe refer to Appendix A.1.

2.2.1.1 Role Keyword

The game description defines the players of the game through the 'role' keyword. In Tic-Tac-Toe, there are two players that mark the game board with 'x' and 'o'. This fact can be easily described with GDL in the following manner:

```
1 (role x)
2 (role o)
```

2.2.1.2 Init Keyword

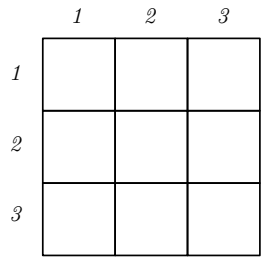

Figure 2.2: The Initial state of the Tic-Tac-Toe game board.

The game description states which facts are true in the initial state of the game by using the 'init' keyword. All the parameters used in the init sentence have to

be constants as the initial state of the game is known. In case of Tic-Tac-Toe, assuming that the first move belongs to player 'x', the initial state shown in the figure 2.2 can be described in the following way:

```
1  (init (cell 1 1 blank))
2  (init (cell 1 2 blank))
3  (init (cell 1 3 blank))
4  (init (cell 2 1 blank))
5  (init (cell 2 2 blank))
6  (init (cell 2 3 blank))
7  (init (cell 3 1 blank))
8  (init (cell 3 2 blank))
9  (init (cell 3 3 blank))
10 (init (control x))
```

2.2.1.3 True Keyword

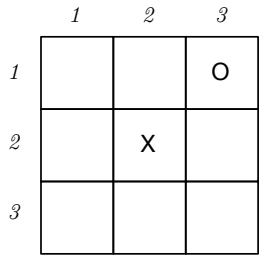

Figure 2.3: The state of the Tic-Tac-Toe game board after both players have made their first move.

The 'true' keyword is very similar to the 'init' keyword. Instead of describing a fact that holds in the initial state of the game, they are used to describe which facts hold in the current game state (for example after a players move). Assuming that the player 'o' has made his move in the last turn the GDL description of the game board shown in figure 2.3 looks as follows:

```
1 (true (cell 1 1 blank))
2 (true (cell 1 2 blank))
3 (true (cell 1 3 o))
4 (true (cell 2 1 blank))
5 (true (cell 2 2 x))
```

```
(true (cell 2 3 blank))
(true (cell 3 1 blank))
(true (cell 3 2 blank))
(true (cell 3 3 blank))
(true (control x))
```

2.2.1.4 Next Keyword

The 'next' keyword is usually used in a head sentence of a Rule and it is used to describe which facts will hold in the next state of the game. For Tic-Tac-Toe this relation can be used to describe alternating moves between the game players:

```
(<= (next (control x))
    (true (control o)))
```

2.2.1.5 Legal Keyword

The 'legal' keyword allows to specify which moves are allowed for a specific player in each of the game states. It usually appears in the Rule's head sentence and has to meet the conditions in the Rule's tail to hold. In Tic-Tac-Toe this relation can be used to describe the action of marking an empty cell:

```
(<= (legal ?player (mark ?x ?y))
    (true (cell ?x ?y blank))
    (true (control ?player)))
```

Given the move belongs to the player 'x' and the game state is described in the figure 2.3 then if the player wants to mark a cell (1,3), the rule described above would take the following form:

```
(<= (legal x (mark 1 3))
    (true (cell 1 3 blank))
    (true (control x)))
```

As the condition *true (cell 1 3 blank)* is not satisfied in the game state from the figure 2.3 it follows that the move is invalid for a current game state.

2.2.1.6 Does Keyword

The 'does' keyword allows to infer which actions were actually taken by a player for each of the game states. For Tic-Tac-Toe this relation to describe which move was made by players in any given state. For example:

```
(does x (mark 2 1))
(does o noop)
```

2.2.1.7 Goal Keyword

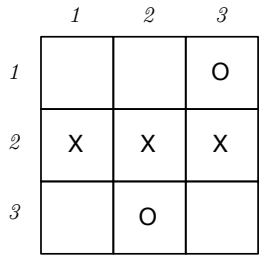

Figure 2.4: One of the possible Tic-Tac-Toe winning states for player 'x'.

The 'goal' keyword is used to assign a certain score for a specified player that once met will terminate the game making the player a winner. The minimum value is 0 and the winning value is 100 (any score from 0 to 100 is allowed). In Tic-Tac-Toe a winning condition for a player would be to have a line made of his marks, like example shown in figure 2.4:

```
(<= (goal x 100)
    (true (cell 1 2 x))
    (true (cell 2 2 x))
    (true (cell 2 3 x)))
```

2.2.1.8 Terminal Keyword

The 'terminal' keyword is used to describe a state of a game where no more moves are possible and the game should terminate. In Tic-Tac-Toe such a state

would be when either the board is full or one of the players has successfully marked a line:

```
(<= terminal
    (role ?player)
    (line ?player))
(<= terminal
    (not open))
```

2.2.2 GDL Syntax Summary

Name	Example	Description
Variable	?x, ?player	Denoted by a question mark followed by a lowercase letter or a string.
Constant	red, player	Denoted by a lowercase letter or a string.
Sentence	(cell 1 1 blank)	GDL Sentence starts with a name represented by a lowercase string, followed by constants and variables (parameters). GDL allows a sentence to take another sentence as a parameter in some cases. It is also allowed to have no parameters in the sentence.
Rule	(<= (legal x noop) (true (control o)))	Consists of a head (a GDL sentence immediately after the <= sign) and a tail (all the remaining GDL sentences after the head sentence). The head is a GDL Sentence that will hold once all the sentences in the tail are satisfied.
Disjunction	(or (p ?x)(q ?x))	Holds only if at least one of the sentences is true.
Negation	(not (p ?x))	Inverts the truth value for a specified sentence.

Table 2.1: GDL's Datalog Foundation syntax and semantics. The Usage example is based on a GDL game description written in KIF.

Additionally, the Game Description Language defines a set of keywords that are required for describing a game. Those relations provide tools needed for a description of a game environment at any time of the game. Most of the additions are based on the GDL sentence and inherit the usage from it.

Name	**Example**	**Description**
Role	(role white)	Uses the word 'role' as a GDL sentence's name and always takes one constant as a parameter. Used to initialise game players by assigning their names from the parameter.
Init	(init (control x))	Init sentence is used to define which sentences are true in the initial state of the game. Basically, a GDL sentence with a predefined name 'init' that takes a sentence as a parameter. The sentence in the parameter must be true in the initial state of the game.
True	(true (control x))	Uses the same syntax as the init sentence but a word 'true' for the name. Used to define which sentences hold for a current development of the game world (state).
Next	(next (control x))	A GDL sentence named 'next' with one argument - another sentence that will hold in the next game update. The 'next' sentence is used as a head of the Datalog rule and needs to be satisfied by all the sentences in the Datalog rule's tail to hold.
Legal	(legal x noop)	A GDL sentence named 'legal' takes one argument - another sentence that defines a possible move in the current state. The 'legal' sentence is used as a head of the Datalog rule and needs to be satisfied by all the sentences in the Datalog rule's tail to hold.

Name	Example	Description
Does	(does ?p (mark ?m ?n))	A GDL sentence named 'does' with two parameters - the first one is a variable or constant representing a player and a second one is a sentence that holds for a player in a current move. Indicates the moves actually made by players in a particular state.
Terminal	terminal	Uses a GDL sentence named 'terminal' with no parameters. Used as a head in a Datalog rule that is satisfied only if all the GDL sentences in the tail hold. Used to specify the conditions that cause the game to terminate.
Goal	(goal ?player 100)	Indicates the goal state for the players. Assigned a value from 0 to 100 where 100 is the winning condition - the final state of the game. The 'goal' relation is a GDL sentence with a first parameter being a variable or a constant representing a player and a second one is a maximum value of a goal for the specified player. Based on the Datalog rule - placed in the Datalog rule's head and all the components in the tail have to hold in order for the goal to be reached.
Distinct	(distinct ?x ?m)	GDL adds a special sentence that allows to check if two variables or constants are not the same.

Table 2.2: GDL additions syntax and semantics. The Usage example is based on a GDL game description written in KIF.

Chapter 3

Game Playing

This chapter will describe the theory and background that laid foundations for writing this MSc thesis and it is divided into three parts.

The first part will introduce a fictional board game called 'Kolibrat'. This game will be introduced due to the fact that it's difficulty and complexity lays between Chess and Tic-Tac-Toe, making it a very good choice for the testing purposes described in Chapter 5. 'Kolibrat' will also be used as a base example through the remaining chapters of this report. The second part will be dealing with the theory on how to create a directed graph of game states, connected via transition functions (moves) using the game trees. The final part will discuss possible ways of optimising a game tree traversal by using optimisation algorithms and heuristics.

3.1 Kolibrat

3.1.1 Overview

'Kolibrat' is a board game developed by an Associate Professor of the Technical University of Denmark - Tomas Bolander. It's complexity goes beyond

Tic-Tac-Toe while still being less complex then the Chess. The board game of Kolibrat consists of a 3 by 4 grid with two homelines as shown in the figure 3.5 below. There are two players of the game represented with two colours - Red and Black. The players can add pieces to the board by placing them on their homelines.

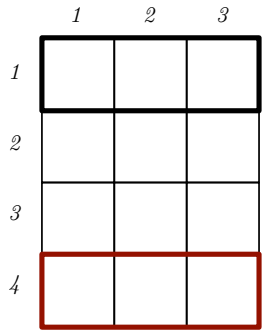

Figure 3.1: A board game representation for Kolibrat. The cells (1,1), (1,2) and (1,3) are a part of the Black player's homeline and corresponding cells on the bottom of the board (3,1), (3,2) and (3,3) being the homeline of the Red player.

The initial state of the game board can be described using the GDL language in the following way:

```
(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))
(init (cell 4 1 blank))
(init (cell 4 2 blank))
(init (cell 4 3 blank))
(homeline 1 1 black)
(homeline 1 2 black)
(homeline 1 3 black)
(homeline 4 1 red)
(homeline 4 2 red)
```

```
18 (homeline 4 3 red)
```

For a full GDL description of 'Kolibrat' refer to Appendix B.1.

3.1.2 Legal Moves

'Kolibrat' is a typical example of a turn based board game. This means that the moves made by players alternate in turns. Once one of the players had made his move the right to make the next one goes to his opponent. A list of legal moves for each payer during his turn is presented in the next subsections below.

3.1.2.1 Insert a piece

During a player's turn he is allowed to add a new piece of his colour to his homeline. This move is only valid if the player's homeline contains a cell that is not obstructed by any pieces. This move terminates the players turn.

```
1  ;; Inserting a piece
2  ;; Conditions
3  ;; 1. Cell is blank
4  ;; 2. Player is in control
5  ;; 3. Cell is players homeline
6
7  (<= (legal ?player (insert ?x ?y))
8      (true (cell ?x ?y blank))
9      (true (control ?player))
10     (homeline ?x ?y ?player))
11
12 ;; Update a cell with the players piece
13
14 (<= (next (cell ?x ?y ?player))
15     (does ?player (insert ?x ?y)))
```

Figure 3.2: A KIF description of the conditions and updates for inserting a piece in 'Kolibrat'.

3.1.2.2 Move a piece

Players are allowed to move their pieces on the game board during their turn. The move has to be done forward and diagonally like shown in Figure 3.3. The cell to which the player wants to move his piece to has to be empty. Moving a piece terminates the turn for a current player.

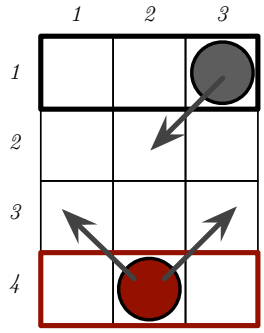

Figure 3.3: An overview of the moves for both Players in Kolibrat. The arrows represent the possible locations for each of the pieces on the game board.

```
1  ;; Moving forward
2  ;; Conditions
3  ;; 1. Cell to move to is blank
4  ;; 2. Cell to move from has a piece
5  ;; 3. Cell [m,n] is a valid move location
6
7  (<= (legal ?player (move ?x ?y ?m ?n))
8      (true (cell ?x ?y ?player))
9      (true (cell ?m ?n blank))
10     (movable ?player ?x ?y ?m ?n))
11
12 ;; Update the game board after the move
13 (<= (next (cell ?x ?y blank))
14     (does ?player (move ?x ?y ?m ?n)))
15 (<= (next (cell ?m ?n ?player))
16     (does ?player (move ?x ?y ?m ?n)))
```

Figure 3.4: A KIF description of the conditions and updates for moving a piece in 'Kolibrat'.

3.1.2.3 Attack a piece

Each player is allowed to take down the opponent's piece if it is neighbouring diagonally with any of the active players pieces. The move can only be taken forward and the opponents piece has to be removed from the board and replaced with the piece that attacked it. This move ends the current player's turn.

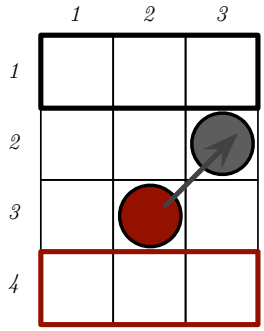

Figure 3.5: A possible Kolibrat game development when the Red player can attack the Black player's piece.

```
1  ;; Attacking the opponent
2  ;; Conditions
3  ;; 1. Player is next to opponent
4  ;; 2. & 3. Opponent is next to player
5  ;; 4. Player attack is a valid move
6
7  (<= (legal ?player (attack ?x ?y ?m ?n))
8          (true (cell ?x ?y ?player))
9          (true (cell ?m ?n ?player))
10     (opponent ?player1 ?player)
11         (attackable ?player ?x ?y ?m ?n))
```

Figure 3.6: A KIF description of Attacking a piece.

3.1.2.4 Jump over multiple pieces

Players are allowed to jump over opponent's pieces. In order for this move to be valid opponents piece has to be directly in front of the player's piece and the

jump can only be done in a straight line. It is possible to jump over more then one piece. There has to be an empty space right behind the opponents pieces for the move to be valid (like shown in Figure 3.7). The move terminates the current player's turn.

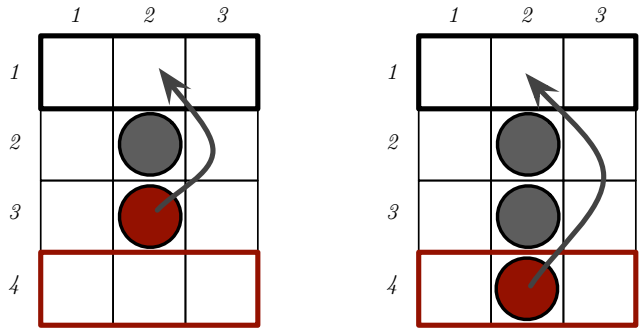

Figure 3.7: Possible configurations for jumping over opponent's pieces in Kolibrat.

```
;; Jumping over one piece
;; 1. Player is on [x,y]
;; 2. Opponent is in front
;; 3. Destination Cell is [u, v] is blank

(<= (legal ?player (jump ?x ?y ?m ?n ?u ?v))
    (true (cell ?x ?y ?player))
    (true (cell ?m ?n ?player1))
    (true (cell ?u ?v blank))
    (true (control ?player))
    (opponent ?player ?player1)
    (jumpable ?player ?x ?y ?m ?n ?u ?v))

(<= (legal ?player (double_jump ?x ?y ?m ?n ?u ?v ?r ?t))
    (true (cell ?x ?y ?player))
    (true (cell ?m ?n ?player1))
    (true (cell ?u ?v ?player1))
    (true (cell ?r ?t blank))
    (true (control ?player))
    (opponent ?player1 ?player)
    (double_jumpable ?player ?x ?y ?m ?n ?u ?v ?r ?t))
```

Figure 3.8: A KIF description of jumping over a piece in 'Kolibrat'.

3.1.2.5 Skip a turn

For some configurations of the game board, like for example shown in figure 3.9, it is possible that player will be unable to apply any of the legal moves available for the game. In that case he has to wait until a legal move can be taken.

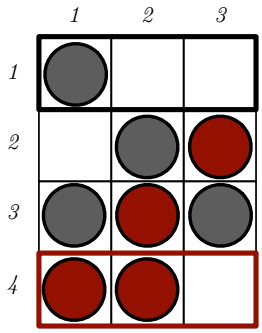

Figure 3.9: A Possible game board configuration with no legal moves for the Black player.

```
1 (<= (next (control black))
2     (true (control red))
3     (has_legal_move black))
4
5 (<= (next (control red))
6     (true (control black))
7     (has_legal_move red))
```

Figure 3.10: A KIF description of Skipping a turn in 'Kolibrat'.

3.1.2.6 Removing own piece

Once a player's piece is successfully moved to the opponent's homeline it can be removed for a point.

```
;; Removing piece from the board
;; Conditions
;; 1. Player has a piece
;; 2. Piece is on the opponents homeline

(<= (legal ?player (remove ?x ?y))
    (true (cell ?x ?y ?player))
    (true (control ?player))
    (removable ?player ?x ?y))

;; Update a players score
(<= (next (score ?player ?n))
    (true (score ?player ?m))
    (does ?player (remove ?x ?y))
    (add ?m ?n))
```

Figure 3.11: A KIF description of the conditions and updates of removing a piece in 'Kolibrat'.

3.1.3 Game goals

One of the game goals for 'Kolibrat' is for a player to collect five points by removing their pieces from the opponent's homeline. If any of the players manages to collect five points the game will terminate.

```
(<= (goal ?player 100)
    (true (score ?player 100)))

(<= terminal
    (true (score ?player 100)))
```

Figure 3.12: A KIF description of the game goal and a terminal state for 'Kolibrat'.

3.2 Game Trees

In order to create a game tree a well defined game description is needed. For a game description to be well defined the following components are required[4]:

Initial state is used to describe a set of facts that hold for the initial state of the game world. In case of for example 'Kolibrat' this information would include the description of the state of the game board before any of the players have made any moves and the information on which of the players should start the game.

Actions should describe a set of legal moves available to each player. Each of the actions should also have a successor function assigned to it that would allow to determine what changes to the game world have been done by taking the move. A good example of a legal move for 'Kolibrat' would be the ability of a player to insert a piece onto a game board and the successor function would make sure that the game board has been updated to include the new piece and that the control has been passed to another player.

Goal Test Once an action is made and the game world is updated the need to check the state of the world is required. Each game has a goal (or a set of goals) assigned to it that enable game to end with a victory of one of the players. For Kolibrat simply checking the amount of points collected by players would be a good example of a goal test.

Path Cost Each action made by player should have a cost assigned to it. The total cost of making a move would be a sum of all the step costs that lead the player to reach a current state. The ways of calculating and estimating the cost and a step cost will be described in details in the next section of this chapter called Heuristics.

Now that the problem has been well defined it needs to be solved. The components described above can be used to create a Game Tree. A Game Tree is a directed graph that is generated by the initial state and the successor function. Basically, by applying a transition function (a game move) to the initial state, a set of new game states is generated. Each of the newly generated states can be expanded even further until a terminal or goal state is reached. An example of a game tree for the Kolibrat game is shown in figure 3.13.

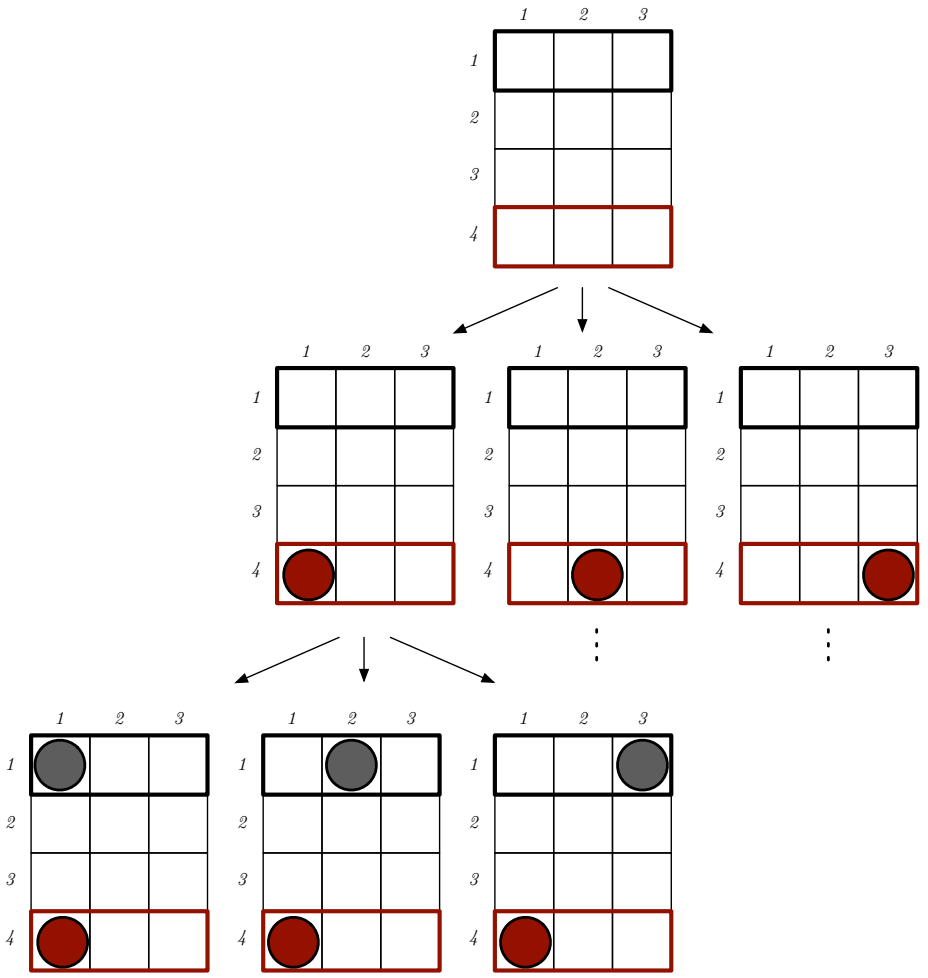

Figure 3.13: A partial Game Tree for Kolibrat.

Figure 3.13 shows some of the expansions in the 'Kolibrat' Game Tree for the first two moves made by players in Kolibrat. The root of the game tree is a node corresponding to the initial state of the game board. The first step is to test whether this is a goal state. Since the initial state is rarely a goal state, we need to consider some other states. This is done by expanding the root node - that is, applying the successor function to the node, thereby generating a new set of states. In this case, we get three new states all modifying the initial game board configuration with the piece of the Red player inserted into his homeline. All the expanded states with their transitions create a space state. Ideally, from this point expansion of nodes that will lead to a game goal would be preferable[4].

3.2.1 Min-Max Algorithm

As mentioned in the introduction expanding a whole tree for a complex game like Chess might be impossible due to an enormous amount of states. That is why it is important to derive a good search strategy. A Min-Max algorithm is a recursive algorithm, providing a good strategy for finding a next move for the player.

The Min-Max algorithm is based on a basic principle. First travel from the given node by entering the node's children. Afterwards continue the process recursively by entering each of the entered node's child until a certain (specified) depth is reached. For each of the reached nodes use an evaluation function that allows to calculate a certain value (Min-Max value) that indicates how good it would be for a player to reach the specified node:

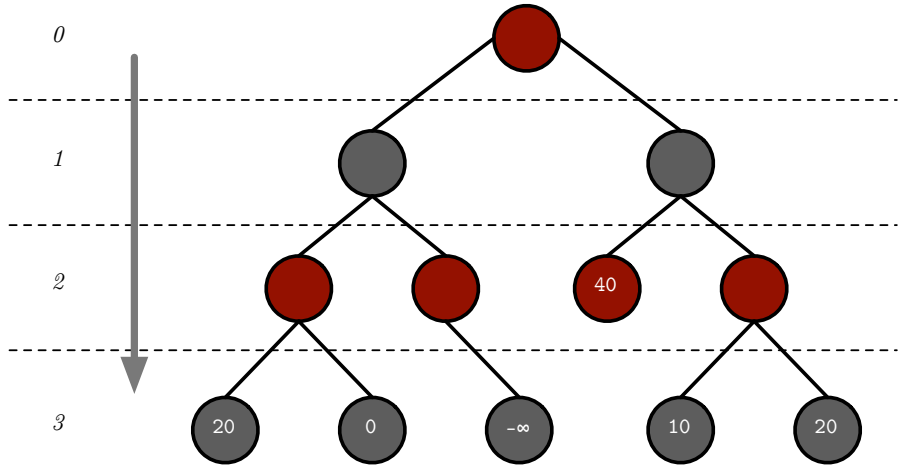

Figure 3.14: An evaluation of a Min-Max value for a game tree nodes at a depth of 3. Note that some terminal nodes at the depth of 2 have also been evaluated.

Nodes that satisfy a goal for a current player are valued with a positive infinity. The nodes for which the opponent wins hold a value of a negative infinity. The figure 3.14 above shows a node expansion for both players. The Red and the Black player are marked by respectively coloured circles.

The nodes reached in the Min-Max tree are then traversed upwards. For each node at a level closer to the original node it's Min-Max value is evaluated by comparing the values in the node's children. The process is presented in the

figure below.

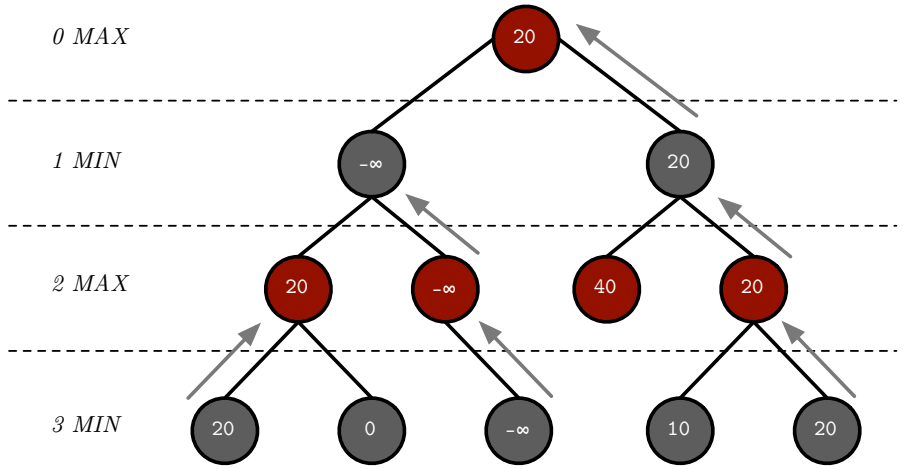

Figure 3.15: An example of a Min-Max tree expansion.

Depending on the player the selected value for each node is either the Minimal or Maximal. Since the move in the figure above belongs to the Red player than all the nodes where the move belongs to him will always take the maximum value from it's children. This process will be opposite for the opponent as the node value for him will be evaluated by taking the minimal value from the node's children. Once the tree has been traversed upwards and the values have been passed to the initial node, the move for a player is determined by choosing a child with a maximum Min-Max value.

3.3 Heuristics[4]

While performing a search on for example a Game Tree a heuristic function tries to estimate the heuristic value of the nodes that need to be checked in order to find a solution for a given problem. Based on the heuristic value the search algorithm can make a decision on which branch to follow during the search.

The heuristic value tries to estimate which of the expanded nodes, or simply moves, are the most likely to lead the search to find a goal state for a given player (without looking through irrelevant nodes). This approach can make the time needed to find a best move for a player significantly lower as there is no need to expand the nodes that will not lead to the solution. The expansion of

the nodes especially in case of the General Game Playing is a very important issue.

The heuristic value proves to be very hard to be estimated in case of General Game Playing as usually it is calculated based on facts that are very specific for a domain of a specific game. Heuristic function also needs to be always admissible, meaning that it should never overestimate the cost of reaching the goal. A good example on how to create a heuristic function can be shown on a 8-puzzle slide game:

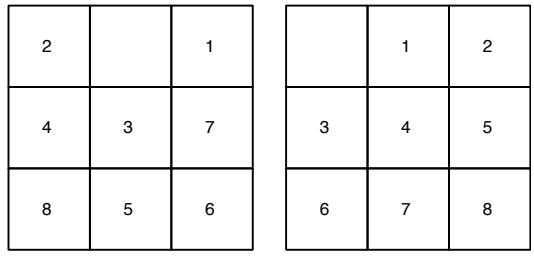

Figure 3.16: Initial and goal state for the 8-puzzle sliding game.

One example of a heuristic function for the game shown in the figure above could be simply a measure on how many misplaced tiles are there in the current game state. This estimation method is called the Hamming Distance. This estimation is always admissible because in order to put a tile into it's proper slot it will need to be moved at least once.

As presented in the example shown above finding a good heuristic estimation is a very domain specific problem that is very difficult to be achieved in case of General Game Playing.

Chapter 4

Implementation

4.1 Overview

This chapter contains a detailed description of the implementation of the General Game Player and is divided into three parts.

The first part deals with the design decisions and the implementation of the data structures used in this project. The second part describes the implementation of the algorithms used for the Game Description Language parser. Finally, the design decisions and the implementation of the actual Game Player are described.

The project has been implemented in Objective-C. The choice of the right programming language for the project was a very important task and is motivated with the following key points:

Flexibility Objective-C is a modern programming language that offers a lot of flexibility by providing dynamic typing. Dynamic typic, if executed properly, gives programmers more freedom and allows a cleaner, more readable code to be created. For example, by using dynamic typing, objects of different type can be stored in one data structure instead of having to create a separate container for every

data type or cast the data back and forth. This feature proved to be useful for the Game Player's implementation as GDL Sentences and GDL Rules could be stored in one array simultaneously.

Memory Management Objective-C, just like Java, is a high level programming language allowing complex programs to be written with ease. One of the differences between those languages is that Java is a standalone programming language, while Objective-C is a superset of C allowing a robust memory management to be used by the programmer. In case of a General Game Player that sometimes needs to store huge data structures in the memory an efficient way of allocating and freeing the memory is very important.

High performance Computing One of the goals for this projects was to check how does utilising modern computer hardware can improve the computation time for a General Game Player to make a move. Objective-C provides a rich toolset for multicore programming that fits this purpose.

One of the downsides of using Objective-C is the fact that the language has not been used for scientific purposes for a very long time. This fact limits the number of available libraries and solutions, that were implemented for similar projects and that could be potentially used for this project. That was the reason why the whole implementation of this project has been done manually.

The project has been implemented on a MacBook Pro with a quad core Intel Core i7 processor (with 8 threads) and 8 GB of 1333MHz Ram memory.

4.2 Data Structures

4.2.1 Design Decision

One of the most important decision to make, while designing a General Game Playing Agent, was to carefully select the data structures used. There are many factors that need to be taken into account. As mentioned in the introduction, a game tree for a complex game, like for example chess, can have as many as 10^{28} states. Each state (a game tree node) needs to store information about a current development of the game environment. As a result the size of the game tree can easily exceed a memory capacity of even the newest computer systems. This problem was one of the main considerations when choosing data structures used in this project.

Objective-C is an Object Oriented programming language that provides a very efficient way of storing data by introducing classes. Additionally, the Foundation Framework (provided by Apple) enables those classes to be stored and modified in data structures, like for example arrays, that allow easy data manipulation.

4.2.2 String representation

Game Description Language is a combination of a syntax that is understandable for computers and readable for humans. It often uses strings to describe variables, relations etc. The downside of this fact is that strings are not the most efficient in case of the processing power. Comparing two strings can involve comparing all the characters in both of them, which could be considerably more expensive than just comparing two integers. Additionally, strings are one of the least efficient data structures in case of memory usage. Even with efficient data structures for the game tree nodes, strings can significantly increase the memory usage and the computation time.

The solution to this problem was to read all the strings from the GDL file and to store each unique string once in a vector. The index of each string in that vector would be a kind of a 'pointer' that can be efficiently stored in the game tree node (instead of the whole string). This way in order to check if two constants are equal a very quick integer comparison can be done instead of comparing the whole strings. If the index of two items that are being compared is equal then the strings in the string vector also have to be equal. The index to the strings containing variables can be saved as a negative value. This way the variables can be distinguished from the constants. Additionally, the comparison between a constant and variable can be done easily as if one of the integers compared is negative then the comparison will always assume that the compared values are equal.

In case a user requests the data to be displayed it can be easily restored from the vector by using the index as a pointer to the original string. The illustration below presents the idea on how does the concept work in practice.

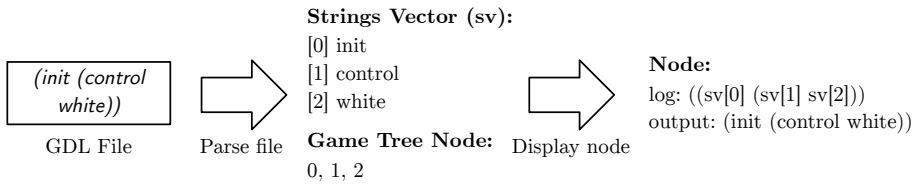

Figure 4.1: A simplified illustration of the string representation mechanism.

4.2.3 Description

4.2.3.1 GDLSentence

```
1  @interface GDLSentence : NSObject
2
3  @property NSInteger prefix;
4  @property NSInteger player;
5  @property (strong, nonatomic) NSArray *conditions;
6
7  + (GDLSentence *)sentenceWithString:(NSString *)string;
8  + (GDLSentence *)regularSentenceForString:(NSString
       *)string;
9
10 @end
```

GDLSentence class is a direct representation of the GDL Sentence defined by the Game Description Language. It is the most basic data structure used in the scope of this project and it serves as a foundation for more complex data structures described in the following subsections.

There are three properties defined in the *GDLSentence* class - an integer named *'prefix'* for storing the information about the type of the sentence (e.g. *init*, *true*, etc.), a second integer called *'player'* for storing the player's information (for the sentences of for example the *legal* type) and finally an array that can store another *GDLSentence* (or sentences).

The type of the sentence stored in the *GDLSentence* object is determined by the GDL prefix of a sentence description:

GDL Example	**Type**
(homeline ?x ?y ?player)	No prefix - a regular GDL sentence.
(not (greater ?y ?n))	'Not' prefix - a negation of a regular GDL sentence.
(or (greater ?y ?n)(smaller ?y ?n)))	'Or' prefix - a disjunction of multiple GDL sentences.
(true (control r))	'True' prefix - a regular GDL Sentence that is true for a current game development.
(not (true (cell ?m ?n blank))	'Not true' prefix - a regular GDL Sentence that is not true for a current game development.
(does ?player (insert ?x ?y))	'Does' prefix - a regular GDL Sentence that is currently executed by a specified player.

Table 4.1: GDL Sentence prefix assignments.

The simplest type of a *GDLSentence* mentioned in the table above is called a *regular* sentence. A GDLSentence of this type is used to store sentences without any prefixes, like for example *cell 1 1 blank*. This fact is reflected in the *conditions* property of the *GDLSentence* class. Instead of storing other sentences that are nested within each other the array will hold a set of integers that are representing the name, the variables and the constants of a sentence (in accordance to the string representation mechanism described in section 4.2.2).

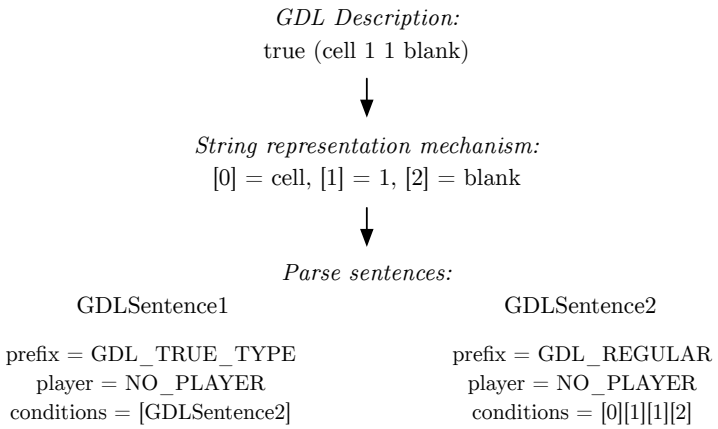

Figure 4.2: A simplified illustration of the string representation mechanism.

The process described above can be achieved thanks to the dynamic typing offered by Objective-C. The *GDLSentence* class also offers two class constructors, for creating new sentences:

sentenceWithString given a GDL description of a sentence parse all of the sentence's components and return a pointer to a new GDLSentence object initialised with the parsed components.

regularSentenceForString given a GDL description of a regular sentence create a GDLSentence object with the conditions initialised according to the string representation mechanism.

4.2.3.2 GDLRule

```
1  #import "GDLSentence.h";
2
3  @interface GDLRule : NSObject
4
5  @property (strong, nonatomic) GDLSentence *ruleHead;
6  @property (strong, nonatomic) NSArray *ruleTail;
7
8  + (GDLRule *)ruleWithString:(NSString *)string;
9
10 @end
```

The *GDLRule* object allows instances of *GDLSentences* to be joined together. The *GDLRule* class is compliant with the Game Description Language rule as it holds two properties: *ruleHead* that holds a *GDLSentence* that will be satisfied if all of the *GDLSentences* (conditions) stored in the *ruleTail* property will hold.

The *GDLRule* class provides a single constructor for creating *GDLRule* objects. Given a GDL string containing a description of a rule, extract the sentences that are nested in the description and then using constructors provided by the *GDLSentence* class initialise the properties of the rule.

4.2.4 GPDataModel

```
1  @interface GPDataModel : NSObject
2
3  @property (strong, nonatomic) NSMutableArray *roles;
4  @property (strong, nonatomic) NSMutableArray *initial;
5  @property (strong, nonatomic) NSMutableArray *legal;
6  @property (strong, nonatomic) NSMutableArray *next;
7  @property (strong, nonatomic) NSMutableArray *terminal;
8  @property (strong, nonatomic) NSMutableArray *goal;
9  @property (strong, nonatomic) NSMutableArray *condition;
10 @property (strong, nonatomic) NSMutableArray *fact;
11
12 // Data model.
13 - (void)addRolesObject:(NSInteger)object;
14 - (void)addInitialObject:(GDLSentence *)object;
15 - (void)addFactObject:(GDLSentence *)object;
16 - (void)addLegalObject:(GDLRule *)object;
17 - (void)addNextObject:(GDLRule *)object;
18 - (void)addTerminalObject:(GDLRule *)object;
19 - (void)addGoalObject:(GDLRule *)object;
20 - (void)addConditionObject:(GDLRule *)object;
21
22 + (GPDataModel *)mainDataModel;
23 + (void)resetMainDataModel;
24 - (void)setup;
25
26 - (NSArray *)conditionsForCondition:(GDLSentence
       *)condition forPlayer:(NSInteger)player;
27 - (BOOL)factCanBeSatisfied:(GDLSentence *)fact;
28 - (NSArray *)variablesForFact:(GDLSentence *)sentence;
29 - (NSArray *)sentencesFromString:(NSString *)string;
30
31 // String representation mechanism.
32 @property (strong, nonatomic) NSMutableArray *strings;
33
34 - (NSInteger)indexForString:(NSString *)string;
35 - (NSString *)stringForIndex:(NSInteger)index;
36
37 @end
```

The *GPDataModel* class is responsible for storing the data parsed from a GDL description and to manage the string representation mechanism. The Game

Description Language provides a structure to group certain types of sentences and rules by introducing prefixes, like for example *true* or *init*. This fact can be used to define class properties for storing each of the groups:

Roles defined in *@property (strong, nonatomic) NSMutableArray *roles;* is an array that stores integer values that represent the players of the game according to the string representation mechanism.

Initial defined in *@property (strong, nonatomic) NSMutableArray *initial;* is an array that stores GDLSentences. The sentences stored in this data structures contain a description of an initial state of the game board.

Legal defined in *@property (strong, nonatomic) NSMutableArray *legal;* is an array that stores GDLRules. The sentences stored in this data structures contain a description of legal moves defined in the Game Description and their conditions.

Next defined in *@property (strong, nonatomic) NSMutableArray *next;* is an array that stores GDLRules. The sentences stored in this data structures contain a description of game updates that will occur once a move has been done.

Terminal defined in *@property (strong, nonatomic) NSMutableArray *terminal;* is an array that stores GDLRules. The sentences stored in this data structures contain information on conditions that once met will terminate the game (e.g. no moves left).

Goal defined in *@property (strong, nonatomic) NSMutableArray *goal;* is an array that stores GDLRules. The sentences stored in this data structures contain information on the goal conditions that allow to derive which states are the goal states.

Condition defined in *@property (strong, nonatomic) NSMutableArray *condition;* is an array that stores GDLRules. The rules stored in this data structure can be described as conditions (GDL rules) that appear freely in the Game Description. A good example of such a condition would be a check if a piece on a board can be moved to a new cell (<= (movable red ?x ?y ?m ?n)(greater ?x ?m)).

Fact defined in *@property (strong, nonatomic) NSMutableArray *fact;* is an array that stores GDLSentences. Some games have certain fact that will hold during the whole game. Those facts are described as regular GDL sentences that appear freely in the Game description. For example *homeline 1 1 black*.

The data structures described above have been divided into groups due to the fact that in case a single data type needs to be checked (like for example a fact)

then there is no need to iterate through all the data stored in the Data Model. An efficient iteration can be conducted only through a relevant group. This data management increases the overall performance. All of the properties also have setter methods allowing easy assignment and ensure that the data passed to those structures is of a valid type.

The data model has been designed in a way that there is only one static instance of the data model accessible through the whole program. This approach increases the performance (as the data does not need to be passed back and forth) and it is presented in the figure below:

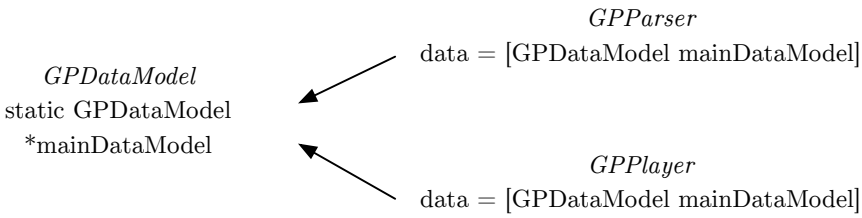

Figure 4.3: A simplified illustration of the data model design.

The GPDataModel contains a class method *+ (GPDataModel *)mainDataModel;* that returns a pointer to the main, static data model instance. This way there is no need to reallocate memory and the data model can be accessed at any moment without the need to create local copies or variables of it.

The data model also implements a set of utility methods listed below:

conditionsForCondition As mentioned in the list of data groups above, a game description can contain a certain set of conditions. A condition is a GDL rule where the sentence in the rule's head will hold only if all of the conditions in the rule's tail will also hold. This method given a condition name will return an array of the GDL sentences that are present in the condition's tail.

factCanBeSatisifed This method given a GDLSentence will determine if the sentence can be satisfied for a current game description by comparing it with the facts about the game (stored in the *fact* property).

variablesForFact Given a GDLSentence containing a fact with with unknown variables return a set of possible assignments for the variables.

sentencesFromString Given a GDL string containing a description of a multiple GDL sentences extract the sentences and return an array containing them.

The Data Model also contains the implementation for the string representation mechanism. For this purpose a property called *string* has been defined and it is used to store the unique list of the strings read from the GDL game description. There are two methods defined for accessing this property - *stringForIndex*, which given an integer returns the corresponding string stored in the *strings* property and *indexForString* that given a string returns an integer that points to that string.

4.3 GDL Parser

The Game Description parser is responsible for processing a game description read from a GDL file, parse it and then save the parsed data in a corresponding data structure held by the data model.

```
1  @property (weak) IBOutlet NSTextField *rolesLabel;
2  @property (weak) IBOutlet NSTextField *initialLabel;
3  @property (weak) IBOutlet NSTextField *movesLabel;
4  @property (weak) IBOutlet NSTextField *nextLabel;
5  @property (weak) IBOutlet NSTextField *terminalLabel;
6  @property (weak) IBOutlet NSTextField *goalsLabel;
7  @property (weak) IBOutlet NSTextField *conditionsLabel;
8  @property (weak) IBOutlet NSTextField *factsLabel;
9  @property (weak) IBOutlet NSTextField *informationLabel;
10 @property (strong) IBOutlet NSTextView *informationView;
11 @property (strong) IBOutlet NSPopover
       *informationPopover;
12
13 - (void)setup;
14 - (void)parseURL:(NSURL *)url;
15 - (void)processExpression:(NSString *)expression;
16 - (IBAction)selectFile:(id)sender;
```

The GPParser class does not store any data on it's own. It uses the static data model described in the previous section to save the parsed information. The properties defined in the class header are connected with the Graphical User Interface and will not be described in details. For more information about the User Interface and the program operation have a look into the appendix.

The class defines a set of method used for the parsing operation of a Game Description Language file:

- (void)parseURL:(NSURL *)url; The parse url method processes a raw text file containing a game description and extracts the GDL sentences and rules from it. Afterwards the extracted expressions are passed to the *processExpression* method described below.

```
READ url
 bracketsCount = 0
 termStarted = 0
 commentLine = 0
 expression = ""

 FOR character IN url
  // Processing a comment line.
  IF (character == ';')
    commentLine = 1
  // An end of a line.
  ELIF (character == '\n')
    IF (commentLine == 1) THEN commentLine = 0

  IF (commentLine == 0)
    // New expression started.
    IF (character == '(')
      bracketsCount++
      termStarted = 1
    ELIF (character == ')')
      bracketCount--
      // Expression has ended.
      IF (bracketCount == 0) AND (termStarted ==
          1)
        termStarted = 0

    // Currently processing a line with an
        expression.
    IF (termStarted == 1) AND (commentLine == 0)
      expression ADD character
    ELIF (termStarted == 0)
      PROCESS_EXPRESSION expression
```

- (void)processExpression:(NSString *)expression; This method accepts a GDL description of a sentence or a rule. The description has to be well formed which means that a single rule or a single sentence with a proper amount of brackets needs to be passed.

```
READ expression

  IF (expression.prefix == "role")
    dataModel ADD_ROLE expression
  ELIF (expression.prefix == "init")
    dataModel ADD_INITIAL expression
  ELIF (expression.prefix == "<=(legal")
    dataModel ADD_LEGAL expression
  ELIF (expression.prefix == "<=(next")
    dataModel ADD_NEXT expression
  ELIF (expression.prefix == "<=terminal")
    dataModel ADD_TERMINAL expression
  ELIF (expression.prefix == "<=(goal")
    dataModel ADD_GOAL expression
  ELIF (expression.prefix == "<=")
    dataModel ADD_CONDITION expression
  ELSE
    dataModel ADD_FACT expression
```

- **(IBAction)selectFile:(id)sender;** A utility method that provides a user with a file chooser that allows to specify a GDL file to be parsed.
- **(void)setup;** A utility method that is used to prepare a data model for accepting new data.

4.4 Game Player

The GPPlayer class is responsible for the initialisation of the game tree, expansion of the game tree nodes and the process of searching through the game tree in order to find a right move for a player - the solution.

```
@interface GPPlayer : NSViewController
    <PlayerViewDataSource>

@property (weak, nonatomic) GPTreeNode *selectedNode;
@property (strong) GPTreeNode *root;
@property BOOL usesMultithreading;

- (void)initializeGameTree;
- (void)expandNode:(GPTreeNode *)node;
```

```
9  - (void)setSelectedNode:(GPTreeNode *)selectedNode;
10 - (GPTreeNode *)computerMoveForNode:(GPTreeNode *)node;
11 - (void)expandComputerNode:(GPTreeNode *)node
       andDepth:(NSInteger)depth;
12
13 @end
```

The methods offered by this class are described in details in the following subsection. The GPPlayer class holds three properties:

@property GPTreeNode *selectedNode A pointer to a game tree node that has been selected by a player.

@property GPTreeNode *root A root node of a game tree.

@property BOOL usesMultithreading a boolean variable that allows to specify if multithreading should be used for the node expansion.

The concepts behind the implementation of the class methods of the GPPlayer are described in details in the following subsections.

4.4.1 Game Tree Node

As described in section 3.2, a game tree is a directed graph of game tree nodes connected via moves taken by players. A node for a game tree is defined in the GPTreeNode class and looks as follows:

```
1  #import "GDLRule.h"
2  #import "GDLSentence.h"
3  #import "GPDataModel.h"
4
5  @interface GPTreeNode : NSObject
6
7  @property (weak) GPTreeNode *parent;
8  @property (strong) NSArray *children;
9  @property (strong) NSArray *states;
10 @property (strong) GDLSentence *move;
11 @property NSInteger terminalState;
12 @property NSInteger goalState;
13
14 - (id)initWithParent:(GPTreeNode *)parent;
```

```
15
16 - (void)expand;
17 - (void)expandWithThreads;
18
19 - (NSString *)stateDescription;
20
21 @end
```

The implementation of the Game Tree Node offers all the properties needed for storing the game states and traversing the game tree:

@property (weak) GPTreeNode *parent; A pointer to a *parent* node of the current node. Allows to backtrack steps taken that lead to the current node.

@property (strong) NSArray *children; An array of game tree nodes that can be reached from the current node by making a move.

@property (strong) NSArray *states; A list of GDL sentences that define the game world description for the current node.

@property (strong) GDLSentence *move; A GDL sentence holding information about the move that was taken in order to create the current node.

@property NSInteger terminalState; An integer value that indicates if the current node is a terminal state of the game.

@property NSInteger goalState; An integer value that indicates if the current node is a goal state for the game tree.

The class implements four methods that provide mechanisms for initialisation, manipulation and obtaining a description of the game tree nodes:

- (id)initWithParent:(GPTreeNode *)parent; Given a pointer to the parent node, initialise a new Game Tree node with default values and the given parent node.

- (void)expand; A method for expanding a current node.

```
1 READ node
2
3 FOR move IN dataModel.legal
4   node EXPAND_WITH_MOVE move
```

- **(void)expandWithThreads;** Similar to *expand* but uses multithreading.

```
READ node

operationQueue = CREATE_QUEUE

FOR move IN dataModel.legal
  operationQueue ADD (node EXPAND_WITH_MOVE move)
```

- **(NSString *)stateDescription;** returns a string representation of the states of the current node.

4.4.2 Node Expansion

Hidden from the header files the Game Tree node implements additional method for expanding the game tree node with a move. This process is divided into the following steps:

Step 1: Extract Variable Mappings

All possible variable assignments for the rule are extracted from the rule's tail. This is done by checking the variable mappings for the 'true' sentences and the facts in the rule's tail:

```
READ move, node

variableMapping = []

FOR sentence IN move.tail
  IF (sentence.type == GDL_TRUE_SENTENCE)
    FOR state IN node.states
      IF (sentence.name == state.name)
        FOR i = 0 TO sentence.conditions.count
          variableMapping ADD [sentence[i], state[i]]
  ELIF (sentence.type == GDL_REGULAR_SENTENCE)
    FOR fact IN dataModel.facts
      IF (sentence.name == fact.name)
        FOR i = 0 TO sentence.conditions.count
          variableMapping ADD [sentence[i], fact[i]]
```

```
16
17 WRITE variableMapping
```

Step 2: Filter Variable Mappings

The pseudocode written above extracts all the possible variable mappings without checking their validity. The variable mappings that satisfy one of the facts might not be true for a different condition of the rule being processed. That is why it is important to filter the variable mappings that cannot be satisfied:

```
1  READ move, varaiableMapping
2
3  filteredMapping = []
4
5  FOR fact IN move.condition
6    IF (fact.type == GDL_REGULAR_SENTENCE)
7      FOR mapping IN variableMapping
8        FOR condition IN fact.conditions
9          IF (condition.type == mapping.type)
10           (condition REPLACE_WITH mapping)
11
12         // Check if the mapped fact can be satisfied.
13         IF (dataModel FACT_SATISFIED fact) THEN
               (filteredMapping ADD mapping)
```

Step 3: Genereate Variable Combinations

With the filtered variables a list of possible solutions can be generated by creating unique combinations of the variables. The computation time of this process is greatly improved by reducing the amount of variables in the second step.

Step 4: Filter solutions

Finally, for every combination derived in step 3 check if all of the variables satisfy all of the rule's tail conditions:

```
1  READ move, combination, node
2
3  FOR condition IN move.conditions
4    IF (condition.type == GDL_TRUE_TYPE)
```

```
5       condition = (condition.variables REPLACE_WITH
            combination.variables)
6     RETURN (node IS_VALID_STATE condition)
7   ELIF (condition.type == GDL_REGULAR_TYPE)
8     condition = (condition.variables REPLACE_WITH
            combination.variables)
9     RETURN (dataModel IS_VALID_FACT condition)
10  ELIF (condition.type == GDL_NOT_TYPE)
11    condition = (condition.variables REPLACE_WITH
            combination.variables)
12    RETURN NOT (dataModel IS_VALID_FACT sentence)
13  ELIF (condition.type == GDL_OR_TYPE)
14    FOR sentence IN condition
15      sentence = (sentence.variables REPLACE_WITH
              combination.variables)
16      RETURN (dataModel IS_VALID_FACT sentence)
```

4.4.3 New node generation

Once a valid set of solutions has been found it needs to be applied to generate new Game Tree nodes. This is done by replacing the variables in the move rule's head with the variables from the solution and applying the move to the child node:

```
1 READ solutions, move, node
2
3 childMove = []
4
5 FOR solution IN solutions
6   childMove = (move.ruleHead REPLACE_VARIABLES solution)
7   childNode = (NEW node WITH_MOVE childMove)
8   childNode.parent = node
9   node.children ADD childNode
```

The child nodes generated during the node expansion process need to be updated with a set of states that are valid for the current game world development.

The states generation for the new nodes is done by extracting the information from the *Next* conditions stored in the data model. Due to the fact that the

Next updates are also GDL Rules, the mechanism for finding and filtering the variable mappings and then creating the combinations of them can be reused from the previous section 4.4.2. Afterwards the validity of the combinations need to be checked:

```
1  READ next, variables, node
2
3  FOR condition IN next.tail
4    IF (condition.type == GDL_DOES_TYPE)
5      IF NOT (node.move == (condition WITH variables))
           WRITE NO
6    ELIF (condition.type == GDL_TRUE_TYPE)
7      IF NOT (node.parent.states CONTAINS (condition WITH
           variables)) WRITE NO
8    ELIF (condition.type == GDL_REGULAR_TYPE)
9      IF NOT (dataModel.facts CONTAINS (condition WITH
           variables)) WRITE NO
10   ELIF (condition.type == GDL_LEGAL_TYPE)
11     IF NOT (node.parent.children.moves CONTAINS
           (condition WITH variables)) WRITE NO
12
13 WRITE YES
```

If all the conditions in the *Next* tail are satisfied then the variables in the head's sentence are replaced with the variables from the valid combination and the state is added to the new node's state list.

4.4.4 Goal and terminal checking

Both goals and terminal conditions stored in the data model are GDL Rules. This makes the process of checking, if the current node is either the goal state or the terminal, easier as the mechanisms for checking the validity of a rule described in the section 4.4.2 can be fully reused.

4.4.5 Game Tree Traversal

With the node expansion mechanism fully implemented a game tree traversal mechanism can be described. The search function expands the nodes of the tree

on the go until a certain (predefined) depth is reached. This method is called breadth search. Once a certain depth has been reached a Min-Max value of the node (at that depth) is evaluated. The process is done recursively in the following manner:

```
READ listOfNodes, currentNode

EXPAND_NODE currentNode WITH depth

// Method definition
DEFINE (EXPAND_NODE node WITH depth):
  IF (depth > 0)
    node EXPAND
    FOR child IN node
      EXPAND_NODE child WITH (depth - 1)

  IF (depth == 0)
    node.minmaxValue = node CALCULATE_MINMAX
    listOfNodes ADD node
```

Once a list of nodes at the specified depth has been created their Min-Max values can be backed-up recursively to the initial node that the move needs to be decided for:

```
READ listOfNodes, currentNode

FOR node IN listOfNodes
  MINMAX_FOR_NODE node

// Method definition
DEFINE (MINMAX_FOR_NODE node):
  IF (node.player == MIN_PLAYER)
    node.parent.minmaxValue = node.parent.child[0]
    FOR child IN node.parent
      IF (node.parent.minmaxValue > child.minmaxValue)
        node.parent.minmaxValue = child.minmaxValue

  ELIF (node.player == MAX_PLAYER)
    node.parent.minmaxValue = node.parent.child[0]
    FOR child IN node.parent
      IF (node.parent.minmaxValue < child.minmaxValue)
        node.parent.minmaxValue = child.minmaxValue
```

```
19
20 // Finally make sure that the minmax value is eveluated
21 // for the parent node, continue until current node
       reached.
22 IF (node != currentNode)
23   MINMAX_FOR_NODE node.parent
```

In this manner the Min-Max values are passed up to the current node and the best possible move can be chosen in accordance with the Min-Max algorithm described in Chapter 4. In case that the Min-Max algorithm was unable to find the best possible option the move will be chosen randomly from the legal moves of the current node.

Chapter 5

Discussion

5.1 Heuristics and Min-Max node value

In Game Playing finding a solution involves searching through a Game Tree. The search function expands the Game Tree nodes that are most likely to lead to the goal state. This process can be achieved with a help of a carefully crafted heuristic function. This task is very difficult in case of General Game Playing. A well defined heuristic function is using a knowledge specific to a certain problem and cannot be reused for other problems. The same problem occurs for the estimation of the Min-Max value used for the Min-Max search algorithm.

This was the reason why a Min-Max value estimation and a heuristic function were joined together for this project. This combination was used for node value estimation for the Min-Max algorithm by using the following properties of the Min-Max value:

Terminal State if a state causes the game to end without anyone winning assign a value of 0 to the expanded node.

Goal State if a goal state is reached for the current player then assign the value of the expanded node to a positive infinity and if the node is a winning node for the opponent mark it with a negative infinity.

For the remaining nodes that could not be evaluated with the classical Min-Max estimation, a certain heuristic function needs to be used. This can be achieved by using the *score* property of the Game Description Language. A well defined game in GDL will keep track of a certain score that will be present for all nodes. The value of the score can be assigned a positive integer from 0 to 100, where reaching 100 will mean that a goal state for a player has been reached. For games like 'Kolibrat' if a player removes a piece from the board his score will increase by 20. This is a good estimation of a heuristic value for a current node and can be used for the purposes of the Min-Max algorithm. It will try to find the moves that will increase the value of the score.

The approach described above works in many cases but it is still possible that during a time limit for a move no nodes with a score different than 0 will be reached. In such a case in order to render the game playable a random legal move needs to be chosen.

5.2 Min-Max Modifications

Min-Max is a search algorithm that is very often used for games that are played by two players, making it a good choice for the purposes of this project. It is quite easy to implement and with a good evaluation function it can produce good results fairly quickly.

As mentioned in the previous section in case of the General Game Playing estimating a Min-Max value of a specific node is not as straight forward as for a single game. That is why it was important to introduce some modification to the original Min-Max algorithm in order to make it work with the General Game Player.

The original Min-Max uses the depth-first node expansion method that reduces the memory usage by expanding the nodes deepest in the tree. This approach reduces the memory usage, as the most probable nodes are expanded based on the calculated Min-Max value. Unfortunately, without the accurate means of calculating this value expanding the game tree in this manner might cause the algorithm to spend time on expanding a branch that will not yield the expected results.

That is why the Min-Max node expansion method was modified in order to use the breadth-first node expansion method. This method expands the nodes recursively until a specified depth is reached. Afterwards the Min-Max values are calculated on the deepest level and then backed up normally according to

the classical Min-Max algorithm. In this way more nodes at deeper levels are expanded and a possible Min-Max value estimation is more likely to be reached. The downside of this approach is that it increases the memory usage. That is why it was very important for the data structures to be very efficient and that the memory can be freed and allocated manually as needed.

Another modification that was done to the original Min-Max algorithm was that the algorithm keeps all the expanded nodes for a current move. The reason for this approach is that in some cases after reaching a specified depth of the tree there is no guarantee that a Min-Max estimation will be found. If the time limit for the move allows the depth of the tree can be increased and the probability of finding the solution will also increase. This approach consumes considerably more memory but it reduces the computation time, making a game more playable.

5.3 The Frame Problem

To most AI researchers, the frame problem is the challenge of representing the effects of action in logic without having to represent explicitly a large number of intuitively obvious non-effects[5]. This is a very important consideration in case of the General Game Playing as passing irrelevant information to the expanded nodes might render any game unplayable.

The frame problem was one of the main considerations when the GDL Game Description of 'Kolibrat' was created. The problem was easily satisfied for the moves taken by the players as a direct action (a move) modified a certain cell on the game board. The problem appeared when there was a need to update the remaining cells.

This problem was solved by introducing a separate rule for each of the possible moves. In case a player has taken a specified move a separate game update will be chosen from the list presented below:

```
1 (<= (next (cell ?x ?y ?mark))
2     (true (cell ?x ?y ?mark))
3     (distinctCell ?x ?y ?m ?n)
4     (does ?player (insert ?m ?n)))
5
6 (<= (next (cell ?x ?y ?mark))
7     (true (cell ?x ?y ?mark))
8     (distinctCell ?x ?y ?m ?n)
```

```
9       (distinctCell ?x ?y ?u ?v)
10      (does ?player (move ?m ?n ?u ?v)))
11
12  (<= (next (cell ?x ?y ?mark))
13      (true (cell ?x ?y ?mark))
14      (distinctCell ?x ?y ?m ?n)
15      (distinctCell ?x ?y ?u ?v)
16      (does ?player (double_jump ?m ?n ?i ?j ?r ?t ?u ?v)))
17
18  (<= (next (cell ?x ?y ?mark))
19      (true (cell ?x ?y ?mark))
20      (distinctCell ?x ?y ?m ?n)
21      (does ?player (remove ?m ?n)))
22
23  (<= (next (cell ?x ?y ?mark))
24      (true (cell ?x ?y ?mark))
25      (distinctCell ?x ?y ?m ?n)
26      (distinctCell ?x ?y ?u ?v)
27      (does ?player (jump ?m ?n ?i ?j ?u ?v)))
28
29  (<= (next (score ?player ?p))
30      (true (score ?player ?p))
31      (opponent ?player ?player1)
32      (does ?player1 (remove ?x ?y)))
```

This way the unaffected cells can be filtered (from the ones affected by each move) and passed to the newly expanded node.

5.4 Multithreading

Multithreading as a programming technique allows multiple threads to exist within a single process. Those threads share the process resources, like for example the data model used in this project, but are able to run independently[6]. In this way a computation time can be decreased by for example expanding two nodes at the same time concurrently.

One of the purposes of this project was to check if utilising modern computer hardware can improve the performance of a General Game Player by using multithreading. The Game Tree expansion mechanism was designed and implemented with Multithreading in mind. This way the real speedup could be

measured. The table below shows a single node expansion time for a game of 'Kolibrat' with and without multithreading.

Multithreading	**No Multithreading**
0.037845 s	0.081293 s
0.035321 s	0.076830 s
0.035119 s	0.075838 s

Table 5.1: Comprison of a node expansion time for Kolibrat.

The obtained results give an average time of 0.036095 seconds for multithreading and an average time of 0.077987 second when no multithreading was used. This yields a speedup of almost 2.2 times when multithreading is used.

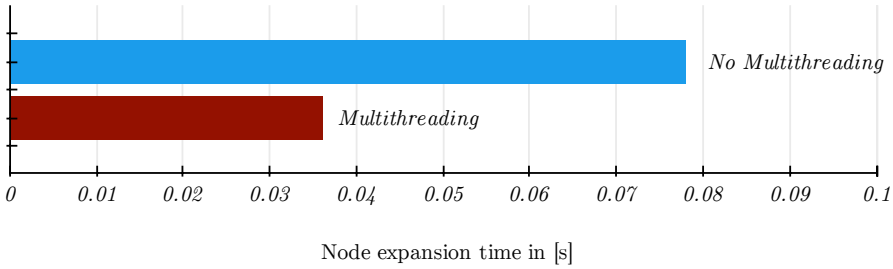

Figure 5.1: A speed comparison of a Kolibrat game tree node expansion.

Chapter 6

Summary

The main goal for the project was to design and implement a program capable of playing more then one board game for two players. The final implementation was tested and it managed to find moves for games like Tic-Tac-Toe, Kolibrat and Checkers.

One of the assumptions made for the project was to use the Game Description Language for the description of the games to be played. The game agent was written in compliance with the newest GDL specification. This approach made the project open for accepting and testing new game descriptions and providing a standardised manner for possible future updates.

The user interface of the project offers a user the ability to observe the game that is being played and to play against the computer by choosing a move from the set of legal moves. Making the project interactive was also amongst the goals of this project.

One of the assumptions made in the project proposal was to use Alpha-Beta pruning in order to optimise the search done by the Min-Max algorithm. Alpha-Beta pruning can effectively cut the number of the nodes that need to be expanded by the factor of two. The advantages of this method could not be utilised in this project. Alpha-Beta pruning depends on an accurate estimation of the Min-Max value that can not be fully achieved in the scope of this project, as

described in Chapter 5.

For the future work it would be recommended to try to expand the program so that it can play not only two player board games but a multi agent problems described in GDL. This way the project might lay foundations for a solution to the Stanford University Competition, that was an inspiration for this project.

Appendix A

Appendices

A.1 Tic-Tac-Toe game rules written in GDL

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                Tic-Tac-Toe                ;;
;;  ------------------------------------     ;;
;;  Lukas Berger, s081733, DTU               ;;
;;                2012                       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                Roles                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(role x)
(role o)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;             Initial State                 ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (cell 1 1 b))
```

```
20 (init (cell 1 2 b))
21 (init (cell 1 3 b))
22 (init (cell 2 1 b))
23 (init (cell 2 2 b))
24 (init (cell 2 3 b))
25 (init (cell 3 1 b))
26 (init (cell 3 2 b))
27 (init (cell 3 3 b))
28 (init (control x))
29
30 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
31 ;;          Legal Moves         ;;
32 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
33
34 (<= (legal ?player (mark ?x ?y))
35     (true (cell ?x ?y b))
36     (true (control ?player)))
37
38 (<= (legal x noop)
39     (true (control o)))
40
41 (<= (legal o noop)
42     (true (control x)))
43
44 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
45 ;;       Board description      ;;
46 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
47
48 (<= (row ?x ?player)
49     (true (cell ?x 1 ?player))
50     (true (cell ?x 2 ?player))
51     (true (cell ?x 3 ?player)))
52
53 (<= (column ?y ?player)
54     (true (cell 1 ?y ?player))
55     (true (cell 2 ?y ?player))
56     (true (cell 3 ?y ?player)))
57
58 (<= (diagonal ?player)
59     (true (cell 1 1 ?player))
60     (true (cell 2 2 ?player))
61     (true (cell 3 3 ?player)))
62
63 (<= (diagonal ?player)
```

```
64      (true (cell 1 3 ?player))
65      (true (cell 2 2 ?player))
66      (true (cell 3 1 ?player)))
67
68  (<= (line ?player)
69      (row ?x ?player))
70
71  (<= (line ?player)
72      (column ?y ?player))
73
74  (<= (line ?player)
75      (diagonal ?player))
76
77  (<= open
78      (true (cell ?x ?y b)))
79
80  (<= (distinctCell ?x ?y ?m ?n)
81      (distinct ?x ?m))
82
83  (<= (distinctCell ?x ?y ?m ?n)
84      (distinct ?y ?n))
85
86  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
87  ;;          Game Updates             ;;
88  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
89
90  (<= (next (cell ?x ?y ?player))
91      (does ?player (mark ?x ?y)))
92
93  (<= (next (cell ?x ?y ?mark))
94      (true (cell ?x ?y ?mark))
95      (does ?player (mark ?m ?n))
96      (distinctCell ?x ?y ?m ?n))
97
98  (<= (next (control x))
99      (true (control o)))
100
101 (<= (next (control o))
102     (true (control x)))
103
104 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
105 ;;                Goals              ;;
106 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
107
```

```
(<= (goal ?player 100)
    (line ?player))

(<= (goal ?player 50)
    (not (line x))
    (not (line o))
    (not open))

(<= (goal ?player1 0)
    (line ?player2)
    (distinct ?player1 ?player2))

(<= (goal ?player 0)
    (not (line x))
    (not (line o))
    open)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;              Terminal             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= terminal
    (line ?player))

(<= terminal
    (not open))
```

Appendix B

Appendices

B.1 Kolibrat game rules written in GDL

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                Kolibrat                ;;
;;  ------------------------------------  ;;
;;   Lukas Berger, s081733, DTU           ;;
;;                2012                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                Roles                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(role black)
(role red)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;          Initial State                 ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(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))
(init (cell 4 1 blank))
(init (cell 4 2 blank))
(init (cell 4 3 blank))
(init (score black 0))
(init (score red 0))
(init (control black))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;           Legal Moves           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Moving forward
;; Conditions
;; 1. Cell to move to is blank
;; 2. Cell to move from has a piece
;; 3. Cell [m,n] is a valid move location

(<= (legal ?player (move ?x ?y ?m ?n))
        (true (cell ?x ?y ?player))
        (true (cell ?m ?n blank))
    (true (control ?player))
        (movable ?player ?x ?y ?m ?n))

;; Inserting a piece
;; Conditions
;; 1. Cell is blank
;; 2. Cell belongs to players homeline
;; 3. Player is in control

(<= (legal ?player (insert ?x ?y))
        (true (cell ?x ?y blank))
        (true (control ?player))
    (homeline ?x ?y ?player))

;; Jumping over one piece
;; Conditions
```

```
64 ;; 1. Player is on [x,y]
65 ;; 2. Opponent is in front
66 ;; 3. Destination Cell is [u, v] is blank
67 ;; 4. All the cells are in the same columns
68
69 (<= (legal ?player (jump ?x ?y ?m ?n ?u ?v))
70         (true (cell ?x ?y ?player))
71         (true (cell ?m ?n ?player1))
72         (true (cell ?u ?v blank))
73     (true (control ?player))
74     (opponent ?player ?player1)
75         (jumpable ?player ?x ?y ?m ?n ?u ?v))
76
77 (<= (legal ?player (double_jump ?x ?y ?m ?n ?u ?v ?r ?t))
78         (true (cell ?x ?y ?player))
79         (true (cell ?m ?n ?player1))
80         (true (cell ?u ?v ?player1))
81     (true (cell ?r ?t blank))
82     (true (control ?player))
83     (opponent ?player1 ?player)
84         (double_jumpable ?player ?x ?y ?m ?n ?u ?v ?r
               ?t))
85
86 ;; Attacking the opponent
87 ;; Conditions
88 ;; 1. Player is next to opponent
89 ;; 2. & 3. Opponent is next to player
90 ;; 4. Player attack is a valid move
91
92 (<= (legal ?player (attack ?x ?y ?m ?n))
93         (true (cell ?x ?y ?player))
94         (true (cell ?m ?n ?player))
95     (true (control ?player))
96     (opponent ?player1 ?player)
97         (attackable ?player ?x ?y ?m ?n))
98
99 ;; Removing piece from the board
100 ;; Conditions
101 ;; 1. Player has a piece
102 ;; 2. Piece is on the opponents homeline
103
104 (<= (legal ?player (remove ?x ?y))
105         (true (cell ?x ?y ?player))
106     (true (control ?player))
```

```
107     (removable ?player ?x ?y))
108
109 ;; Move belongs to a player in control
110
111 ;(<= (legal black noop)
112 ;       (true (control red)))
113
114 ;(<= (legal red noop)
115 ;       (true (control black)))
116
117 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
118 ;;        Board description        ;;
119 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
120
121 ;; Move from [x,y] to [m,n]
122
123 (<= (movable red ?x ?y ?m ?n)
124         (greater ?x ?m)
125         (or (greater ?y ?n)(smaller ?y ?n)))
126
127 (<= (movable black ?x ?y ?m ?n)
128         (smaller ?x ?m)
129         (or (greater ?y ?n)(smaller ?y ?n)))
130
131 (<= (attackable red ?x ?y ?m ?n)
132         (greater ?x ?m)
133         (not (greater ?y ?n))
134         (not (smaller ?y ?n)))
135
136 (<= (attackable black ?x ?y ?m ?n)
137         (smaller ?x ?m)
138         (not (greater ?y ?n))
139         (not (smaller ?y ?n)))
140
141 (<= (jumpable black ?x ?y ?m ?n ?u ?v)
142         (smaller ?x ?m)
143         (smaller ?m ?u)
144         (not (smaller ?y ?n))
145         (not (greater ?y ?n))
146         (not (smaller ?n ?v))
147         (not (greater ?n ?v)))
148
149 (<= (jumpable red ?x ?y ?m ?n ?u ?v)
150         (greater ?x ?m)
```

```
151         (greater ?m ?u)
152         (not (smaller ?y ?n))
153         (not (greater ?y ?n))
154         (not (smaller ?y ?v))
155         (not (greater ?y ?v)))
156
157 (<= (double_jumpable black ?x ?y ?m ?n ?u ?v ?r ?t)
158         (smaller ?x ?m)
159         (smaller ?m ?u)
160         (smaller ?u ?r)
161         (not (smaller ?y ?n))
162         (not (greater ?y ?n))
163         (not (smaller ?y ?v))
164         (not (greater ?y ?v))
165         (not (smaller ?y ?t))
166         (not (greater ?y ?t)))
167
168 (<= (double_jumpable red ?x ?y ?m ?n ?u ?v ?r ?t)
169         (greater ?x ?m)
170         (greater ?m ?u)
171         (greater ?u ?r)
172         (not (smaller ?y ?n))
173         (not (greater ?y ?n))
174         (not (smaller ?y ?v))
175         (not (greater ?y ?v))
176         (not (smaller ?y ?t))
177         (not (greater ?y ?t)))
178
179 (<= (removable ?player ?x ?y)
180     (opponent ?player ?player1)
181         (homeline ?x ?y ?player1))
182
183
184 (homeline 1 1 black)
185 (homeline 1 2 black)
186 (homeline 1 3 black)
187 (homeline 4 1 red)
188 (homeline 4 2 red)
189 (homeline 4 3 red)
190
191 (greater 2 1)
192 (greater 3 2)
193 (greater 4 3)
194
```

```
195 (smaller 1 2)
196 (smaller 2 3)
197 (smaller 3 4)
198
199 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
200 ;;          Game Updates             ;;
201 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
202
203 (<= (next (cell ?x ?y ?player))
204         (does ?player (insert ?x ?y)))
205
206 (<= (next (cell ?x ?y blank))
207         (does ?player (move ?x ?y ?m ?n)))
208
209 (<= (next (cell ?m ?n ?player))
210         (does ?player (move ?x ?y ?m ?n)))
211
212 (<= (next (cell ?x ?y blank))
213         (does ?player (jump ?x ?y ?m ?n ?u ?v)))
214
215 (<= (next (cell ?u ?v ?player))
216         (does ?player (jump ?x ?y ?m ?n ?u ?v)))
217
218 (<= (next (cell ?x ?y blank))
219         (does ?player (double_jump ?x ?y ?m ?n ?u ?v ?r
                ?t)))
220
221 (<= (next (cell ?r ?t ?player))
222         (does ?player (double_jump ?x ?y ?m ?n ?u ?v ?r
                ?t)))
223
224 (<= (next (cell ?x ?y blank))
225         (does ?player (remove ?x ?y)))
226
227 ;; Remaining cells
228
229 (<= (next (cell ?x ?y ?mark))
230         (true (cell ?x ?y ?mark))
231         (distinctCell ?x ?y ?m ?n)
232         (does ?player (insert ?m ?n)))
233
234 (<= (next (cell ?x ?y ?mark))
235         (true (cell ?x ?y ?mark))
236         (distinctCell ?x ?y ?m ?n)
```

```
237         (distinctCell ?x ?y ?u ?v)
238         (does ?player (move ?m ?n ?u ?v)))
239
240 (<= (next (cell ?x ?y ?mark))
241         (true (cell ?x ?y ?mark))
242         (distinctCell ?x ?y ?m ?n)
243         (distinctCell ?x ?y ?u ?v)
244     (does ?player (double_jump ?m ?n ?i ?j ?r ?t ?u ?v)))
245
246 (<= (next (cell ?x ?y ?mark))
247         (true (cell ?x ?y ?mark))
248         (distinctCell ?x ?y ?m ?n)
249         (does ?player (remove ?m ?n)))
250
251 (<= (next (cell ?x ?y ?mark))
252         (true (cell ?x ?y ?mark))
253     (distinctCell ?x ?y ?m ?n)
254         (distinctCell ?x ?y ?u ?v)
255         (does ?player (jump ?m ?n ?i ?j ?u ?v)))
256
257 (<= (next (score ?player ?p))
258     (true (score ?player ?p))
259     (opponent ?player ?player1)
260     (does ?player1 (remove ?x ?y)))
261
262 (<= (next (score ?player ?n))
263         (true (score ?player ?m))
264     (does ?player (remove ?x ?y))
265         (add ?m ?n))
266
267 (<= (next (score ?player ?p))
268     (true (score ?player ?p))
269     (does ?player (double_jump ?x ?y ?m ?n ?u ?v ?r ?t)))
270
271 (<= (next (score ?player ?p))
272     (true (score ?player ?p))
273     (opponent ?player ?player1)
274     (does ?player1 (double_jump ?x ?y ?m ?n ?u ?v ?r
            ?t)))
275
276 (<= (next (score ?player ?p))
277     (true (score ?player ?p))
278     (does ?player (jump ?x ?y ?m ?n ?u ?v)))
279
```

```
280 (<= (next (score ?player ?p))
281     (true (score ?player ?p))
282     (opponent ?player ?player1)
283     (does ?player1 (jump ?x ?y ?m ?n ?u ?v)))
284
285 (<= (next (score ?player ?p))
286     (true (score ?player ?p))
287     (opponent ?player ?player1)
288     (does ?player1 (insert ?x ?y)))
289
290 (<= (next (score ?player ?p))
291     (true (score ?player ?p))
292     (does ?player (move ?x ?y ?m ?n)))
293
294 (<= (next (score ?player ?p))
295     (true (score ?player ?p))
296     (opponent ?player ?player1)
297     (does ?player1 (move ?x ?y ?m ?n)))
298
299 (<= (next (score ?player ?p))
300     (true (score ?player ?p))
301     (does ?player (insert ?x ?y)))
302
303 (<= (has_legal_move ?player)
304         (or (legal ?player (insert ?x ?y))
305                 (legal ?player (move ?x ?y ?m ?n))
306                 (legal ?player (jump ?x ?y ?m ?n ?u ?v))
307                 (legal ?player (double_jump ?x ?y ?m ?n
                        ?u ?v ?r ?t))
308                 (legal ?player (remove ?x ?y))))
309
310 (<= (next (control black))
311         (true (control red))
312         (has_legal_move black))
313
314 (<= (next (control red))
315         (true (control black))
316         (has_legal_move red))
317
318 (<= (distinctCell ?x ?y ?m ?n)
319     (or (distinct ?x ?m)
320         (distinct ?y ?n)))
321
322 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
323 ;;                Goals                ;;
324 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
325
326 (<= (goal ?player 100)
327         (true (score ?player 100)))
328
329 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
330 ;;               Terminal               ;;
331 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
332
333 (<= terminal
334         (true (score ?player 100)))
335
336 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
337 ;;                Facts                 ;;
338 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
339
340 (opponent red black)
341 (opponent black red)
342
343 (add 0 20)
344 (add 20 40)
345 (add 40 60)
346 (add 60 80)
347 (add 80 100)
```

Appendix C

Appendices

C.1 Software Instructions

In order to successfully parse and then play a selected game the following steps need to be taken:

Step 1: After running the application a file containing a GDL game description needs to be chosen by clicking on the 'select' button in the upper right corner of the program.

Step 2: After the GDL file has been parsed the information about the rules of the game in the top part of the window should be updated. The view in the lower right corner should also reflect the initial state of the game and a set of possible moves rendered as red circles. The left side of the window allows the user to specify if multithreading should be used for the node expansion. At this point the user can specify a move that he wants to take by selecting it from a drop down list in the bottom of the window.

Step 3: After the move has been taken by a player the computer will calculate it's move. Once the move has been calculated the changes in the game world will be reflected in the view. A user will also be notified of the move taken by

the computer. From this point a user can continue playing the game until one of the terminal or goal nodes have been reached.

Figure C.1: A Graphical User Interface of the General Game Player developed for the scope of this project.

Bibliography

---

[1] Michael Genesereth and Nathaniel Love. General game playing: Overview of the aaai competition. 2005.

[2] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General game playing: Game description language specification. 2008.

[3] Stanford University. Knowledge interchange format, draft proposed american national standard (dpans).

[4] Russell Norvig and Stuart Russell. *Artificial Inteligence, A Modern Approach, 2nd Edition*. Stanford University, 2003.

[5] Stanford University. Stanford encyclopedia of philosophy.

[6] Sergey Ignatchenko. Single-threading: Back to the future?, journal.