University of Wrocław
Faculty of Mathematics and Computer Science

Institute of Computer Science

Doctoral Thesis

General Game Description Languages

Jakub Kowalski

Supervisor:
prof. Andrzej Kisielewicz

Wrocław 2016

Abstract

General Game Playing (GGP) is a field of Artificial Intelligence whose aim is to develop a system that can play a variety of games with previously unknown rules and without any human intervention. Using games as a testbed, GGP tries to construct universal algorithms that perform well in various situations and environments in order to achieve this aim. The core of GGP is the formalism describing the class of games that should be playable by the programs. The language used to encode game rules should be not only broad enough to provide an appropriate level of challenge, but also easily machine-processable and, if possible, human-readable.

In this thesis, we advance the state-of-the-art research in the GGP area by investigating a few specific problems concerning general game description languages.

We design new efficient algorithms for the task of learning the rules of chess-like games by observing play. Our experiments conducted on these algorithms show an increase in the quality of results compared to the existing approaches and a significant speedup of the learning process.

Our contribution to the field of Procedural Content Generation is a formal method to evaluate the strategic properties of a given game. Based on that evaluation, we use an evolutionary approach to generate complete rules for some novel, non-trivial, and human-playable chess-like games.

Further, we propose an extension of Stanford's Game Description Language (GDL) that supports games with asymmetric player moves and time-dependent events. We present the semantics of this new extension and argue that it is the broadest general game description language described so far.

We also develop an efficient GDL compiler, which enables faster computation of game states than other known approaches. Thus, it can be used to increase the quality of the GGP agents based on the Monte-Carlo Tree Search algorithm, which is the most effective search method applied in game playing in recent years.

Streszczenie

General Game Playing (GGP) jest dziedziną sztucznej inteligencji, której celem jest stworzenie uniwersalnego gracza – systemu zdolnego do autonomicznej gry w dowolną grę, której zasady poznaje bezpośrednio przed rozgrywką. Wykorzystując gry jako środowisko testowe, ogólnym zadaniem GGP jest stymulowanie rozwoju algorytmów, które umiałyby sobie radzić w różnorodnych środowiskach i w nieprzewidzianych sytuacjach. Niezwykle ważnym elementem GGP jest formalizm opisujący klasę gier akceptowanych przez programy grające. Język powinien opisywać klasę gier dostatecznie obszerną, tak żeby wyzwanie miało odpowiedni stopień trudności, a równocześnie powinien być wygodny do przetwarzania maszynowego oraz w miarę czytelny dla człowieka.

Rozprawa ta prezentuje nasz wkład w rozwój dziedziny GGP złożony z rezultatów badań nad kilkoma konkretnymi problemami dotyczącymi ogólnych języków opisu gier.

Zaczynamy od przedstawienia nowego efektywnego algorytmu uczenia się zasad gier szachopodobnych na podstawie obserwacji przebiegu rozgrywek. Przeprowadzone eksperymenty wykazały poprawę rezultatów oraz znaczne przyspieszenie czasu obliczeń w porównaniu do istniejących rozwiązań.

Dalej prezentujemy nasz wkład do dziedziny proceduralnego generowania zawartości gry poprzez zdefiniowanie formalnej metody oceny strategicznych własności gier. Stworzony system umożliwił wygenerowanie nowych, nietrywialnych gier szachopodobnych, które nadają się również do grania przez ludzi.

Kolejny przedstawiony w rozprawie rezultat to rozszerzenie języka Stanford's Game Description Language (GDL) umożliwiające definiowanie gier zawierających asymetryczne ruchy graczy oraz wydarzenia oparte o upływ czasu. Zdefiniowana została semantyka języka, a także pokazano, że jest to najbardziej ogólny z dotychczas opublikowanych języków opisujących ogólne klasy gier.

Zaprojektowaliśmy również efektywny kompilator języka GDL, pozwalający na szybsze obliczanie kolejnych stanów gry niż w innych dotychczasowych podejściach. Dzięki temu może on być wykorzystany do podniesienia jakości programów-graczy GGP bazujących na algorytmie Monte Carlo Tree Search, który jest najbardziej efektywną metodą przeszukiwania wykorzystywaną w programach grających w ostatnim czasie.

Acknowledgments

Primarily, I would like to express my gratitude to my supervisor prof. Andrzej Kisielewicz for his unwavering support whenever needed. This thesis is the result of his guidance, patience, and encouragement I received during the writing process and my research in general.

I would also like to thank prof. Krzysztof Loryś, the director of the PhD studies and my Master Thesis supervisor, for his kind support and constantly watchful eye.

I thank Marek Szykuła for fruitful collaboration and the time we spent together on numerous discussions.

The research in this thesis was supported by the National Science Centre, Poland, under the project number 2014/13/N/ST6/01817.

Contents

1 Introduction

This thesis contributes to the domain of General Game Playing (GGP), which is a field of Artificial Intelligence focusing on autonomous agents with the ability to play various games, in contrast with specialized game players who are able to play only one specific game. Games have always been a perfect testbed and motivation for the advancement of AI algorithms. Even before the era of computers, the idea of a machine being able to understand the complex rules of a strategic game such as Chess and competitively play against a human opponent fascinated mankind, for example the famous 18th century Mechanical Turk [109] – a machine believed to automatically play chess, but in reality housed a human chess player hidden inside.

For a long time, the ability to play complex strategic games was equated to having human-level intelligence, and due to this, AI research naturally focused on the challenge of beating human players in popular games like Chess, Checkers, or Go. The assumption was that by achieving this goal, the means for crafting true AI would be discovered. However, the history of human-machine challenges [170] suggests that in fact, despite developing algorithms that can beat human players, we have not been able to craft true AI, and humans have the unalterable tendency to make things work using the minimum effort.

In the case of Chess, which was the first grand AI challenge, it seemed that the simplest way to achieve expert level within the playing program was to incorporate the knowledge of human experts. However, this turned out to be a difficult and inefficient method. Instead, after a longer time than expected, the IBM Deep-Blue computer, combining highly specialized hardware, hand-crafted heuristic functions and a large database of opening and ending positions, won against Chess world champion Garry Kasparov in 1997 [22].

The success sparked mixed reviews. On the one hand, major progress had been made, and efficient algorithms like min-max with alpha-beta pruning and other enhancements had been developed. On the other hand, the program clearly had no intelligence, and we found ourselves as far from creating true AI as we had been before. Nowadays, given the computation power of the modern computer, any skilled programmer can write a Chess program for expert level.

In addition to developing AI within Chess, other game-related, spectacular projects have been undertaken. The game of Checkers has been solved and results in a draw when both sides play perfectly [169], and the supercomputer Watson has won against the top human competitors in the quiz show „Jeopardy" [41]. Various other games have also been introduced as separate AI challenges, including boardgames (Arimaa [211]) and video games (Starcraft [143]).

The next global AI milestone is to beat human players at Go, a game with 5,500 years of tradition and far more complex then Chess. After a number of approaches [17, 30, 60], this year the Google DeepMind AlphaGo program defeated professional 9-dan Go player Lee Sedol [193], and currently is unofficially counted in the top three Go players in the world. AlphaGo combines Monte Carlo Tree Search and Deep Neural Networks with reinforcement learning and distributed computations on a cluster containing CPUs and GPUs.

An argument against focusing on specialized single game programs is that a truly intelligent program should understand game rules on its own, and be able to play it even if it has never seen this game before. Such behavior is beyond the capabilities of programs like Deep Blue, which can only play Chess and can not solve simple mathematical problems, play Tic-tac-toe, or any other game even significantly simpler. Thus, the General Game Playing has been introduced as a new area of research, and an alternative Grand Challenge of Artificial Intelligence.

1.1 The General Game Playing Problem

The aim of GGP is to develop a system that can play a variety of games with previously unknown rules and without any human intervention. Using games as a testbed, GGP tries to construct universal algorithms that perform well in various situations and environments in order to achieve this aim.

A requirement of GGP is to ensure that human intelligence is not simply transferred into the program, which will then play blindly according to pre-programmed knowledge. A further reason why human intelligence is avoided is that there is no game-specific information that the program can be fed with. Given games may differ in nearly every aspect. They can be single-player (puzzles), multiplayer for any number of players, zero-sum, adversary, cooperative, turn-taking, with simultaneous moves, etc. Thus, it is expected that a successful GGP program will contain higher level intelligence than game playing agents, whose goal is to play only one game.

It is not within the scope of the thesis, but let us mention that at present it is still controversial to call even the top GGP programs intelligent. Although a lot of effort has been made and progress in the area is undoubtable, no spark of intelligence can be seen underneath the millions of Monte Carlo simulations supported by various heuristics. Top GGP players, supported by several strategies to reduce the size of the traversed part of the game tree, still mainly depend on the computational capabilities of modern computers.

Nevertheless, GGP research tackles very complex tasks that require novel approaches, provide many challenges, and combine multiple subfields of AI. The domains that are proved to play the major role in GGP research are knowledge representation (knowledge extraction, reasoning and knowledge transfer), game-tree search (knowledge-based, simulation-based), planning (opponent modeling, heuristic development), and reinforcement and machine learning.

The core of GGP is the formalism describing the class of games that should be playable by the programs. The language used to encode game rules should be broad enough to provide an appropriate level of challenge, while being easily machine-processable and desirably human-readable, to allow games to be created by human developers.

Multiple GGP standards have emerged since the first approach in 1968 [156], each using its own semantics and communication protocol, and describing a different range of games. Initially, the task of GGP was focused towards generalizing chess-like boardgames. In 2005, Stanford's Logic Group proposed its Game Description Language (GDL) describing all finite, turn-based, deterministic games with full information, and announced the International GGP Competition [65]. Since that time, GGP has become a well-known and popular research area, with multiple branches and ongoing competitions, that makes important contributions towards overall AI research.

1.2 Contributions

In this thesis, we advance the state-of-the-art research in the GGP area by investigating various aspects of general game description languages. Our work concerns the tasks of learning and generating the rules of the games, and compiling, translating, and extending game description languages.

The main contributions of this thesis can be summarized as follows.

Algorithm for learning the rules of boardgames via observation of plays: We formally define the problem of learning the optimal automaton for chess-like pieces described by the partially observable regular languages. We point out problems with the existing algorithms and provide our own contribution. Our algorithm uses heuristic optimizations to reduce the number of considered hypotheses, and allows users to define a degree of aggressiveness in attempts to minimize the output automata. We have tested our approach on both human-made and procedurally generated games, and our tests show a significant speedup of the learning process with, for some cases, increase in the quality of results.

Generating complete rules of chess-like games: Generating complete game rules is the most advanced application of procedural content generation in games. We have formalized a general method to evaluate an n-player game by measuring the performance of AI agents and comparing this with the model performance obtained on the set of known, well-behaving games. We argue that this approach preserves the core strategic properties of the games from the model set. We have applied our evaluation to the domain of chess-like boardgames and run a number of evolutionary experiments to obtain novel, non-trivial, and human-playable games.

Introducing Real-time Game Description Language: We define a new game description language, extending Stanford's GDL. Our contribution removes the GDL restriction that the games have to be turn-based. The language makes it possible to describe a large variety of games involving a real-time factor and taking into account the response time of the message arrival, which requires novel approaches from the GGP agents. We provide the language syntax, a detailed description of its semantics, and discuss the execution model. We also propose a way of combining our extension with another known extension, adding to GDL randomness and imperfect information, which creates the GGP language describing the broadest class of games so far.

Fast GDL compiler: We have developed an algorithm forming an optimized plan of computations for the game rules given in Stanford's GDL. Such a plan can be further translated into an efficient code in some programming languages, e.g. C++. To speed up the computations, we have used specialized data structures dependent on expected types of queries, and a rules-rewriting system supporting early cut-offs when possible. The compiler has been embedded into the GGP player Dumalion, which was one of the participants in the International General Game Playing Competition, and is the fastest GDL compiler described in the literature.

Part of the work described in this thesis has been published in the following publications:

1. Kowalski, J., Szykuła, M., *Evolving Chesslike Games Using Relative Algorithm Performance Profiles*, Applications of Evolutionary Computation, LNCS 9597, pages 574–589, 2016.

2. Kowalski, J., Kisielewicz, A., *Towards a Real-time Game Description Language*, Proceedings of International Conference on Agents and Artificial Intelligence (2),pages 494–499, 2016.

3. Kowalski, J., Kisielewicz, A., *Game Description Language for Real-time Games*, IJCAI Workshop on General Intelligence in Game-Playing Agents, pages 23–30, 2015.

4. Kowalski, J., Kisielewicz, A., *Testing General Game Players Against a Simplified Boardgames Player Using Temporal-difference Learning*, Proceedings of IEEE Congress on Evolutionary Computation, pages 1466–1473, 2015.

5. Kowalski, J., *Embedding a Card Game Language into a General Game Playing Language*, Proceedings of the 7th European Starting AI Researcher Symposium, pages 161–170, 2014.

6. Kowalski, J., Szykuła, M., *Game Description Language Compiler Construction*, AI 2013: Advances in Artificial Intelligence, LNCS 8272, pages 234–245, 2013.

1.3 Overview of the Thesis

The thesis is structured as follows. In Chapter 2, we introduce the necessary background for the work. We explain GGP in more detail, including a brief history of the domain, and present the general game description languages that we refer to in further chapters. We also describe existing GGP competitions including the methods used by successful playing agents.

Chapter 3 describes our algorithm for learning the rules of boardgames by observing. We formulate the problem in terms of Regular Language Inference, analyze the performance and drawbacks of the existing algorithms, and present a heuristic algorithm that quickly and efficiently performs on a number of test games.

In Chapter 4, we introduce the domain of Procedural Content Generation and describe our two approaches for generating chess-like boardgames. We formalize a simulation-based approach to evaluate the strategic properties of the created game, and present examples of games obtained by combining our evaluation functions with an evolutionary algorithm search.

In Chapter 5 we describe our extension of Stanford's GDL that allows games with asymmetric player moves and time-dependent events. We present the semantics of the extension and illustrate its expressiveness with a number of examples.

In Chapter 6 we tackle the subject of relations between various GGP description languages. Firstly, we present a translation from the Card Game Description Language into an extended version of Stanford's GDL. Secondly, we show the results of the experiments to estimate the difference in players' skills depending on the generality of the class, and thus assess the progress achieved in Stanford's GGP domain.

Chapter 7 presents our method of compiling GDL rules into an optimized code in a low-level programming language. We test the efficiency of the compiled code on a set of games, comparing the results with a standard Prolog-based reasoner and other approaches described in the literature.

Finally, in Chapter 8, we summarize the presented work, draw out conclusions, and suggest future work.

2
Preliminaries

This chapter introduces to the area of General Game Playing (GGP). We present the existing game description languages, and provide extended descriptions of the most important and recent contributions. We briefly mention constructions less important from this thesis' point of view. Also, we point out the AI techniques used to play general games, and provide an overview of the existing GGP competitions.

The first section of this introduction focuses on languages describing chess-like games, which is one of the most common domain for the GGP approach. Here we describe two languages. A classical *Metagame* from 1992, and a *Simplified Boardgames* class introduced in 2012, which can be seen as the modern, generalized, and more concise descendant of the former. Section 2.2 describes the most prominent family of GGP languages, developed by the Stanford University Logic Group for the purpose of the International General Game Playing Competition in 2005. The section that follows introduce the Video Game Description Language, which is a recent contribution rapidly gaining popularity since the first competition held in 2014. Finally, in Section 2.4, we present other General Game Playing domains and the languages associated, which we will referring to throughout the thesis.

2.1 Generalized Chess-like Games

The domain of chess-like boardgames (defined in various ways) is undoubtedly the most exploited game family in the General Game Playing research. Especially in the early years of the GGP, the entire research has been focused on the generalization of Chess, and development of programs that can play its variants. It is understandable research direction as Chess is one of the longest studied game in the human history. Many human-made Chess variants (called *fairy chess*) already exist [35], and there exist effective algorithms to play such games [110].

The first work classified as belonging to the General Game Playing domain is 1968 contribution by Pitrat [156], concerning the class of arbitrary chess-like board games. Pitrat has proposed an algorithmic board-games playing framework and formulated first ideas about a language to describe generalized rules of the games [157].

Subsequent attempts to crate multi-game playing programs emerged in the early 1990s. *SAL* [67] (Search And Learning) has used alpha-beta search and a neural network state evaluation function combined with temporal difference learning to play „two-person, deterministic, zero-sum board games of perfect information”. Presented experiments show system performance using Tic-tac-toe, Connect Four and Chess.

An extension of the *Morph* chess playing program turned into a general game playing program has been presented in [108]. This approach uses adaptive predictive search and conceptual graphs to create domain-independent player, which has been tested using a wide range of problems including Othello, Tetris, 8-puzzle, Tic-tac-toe, and Pente.

The *Hoyle* system [40] focuses on the task of learning game strategies via playing again an expert opponent. Its advanced learning mechanism uses a set of algorithms for the discovery of useful knowledge, which is expected to be relevant and may be correct. Presented results demonstrate Hoyle's high playing skills in a variety of two-person, perfect information board games.

Metagame [147], presented by Pell in 1992 and considered as the predecessor of the modern General Game Playing, is described in details in the next section. Browne's *Ludi* system ([19]) from 2008, focusing more on general not necessarily chess-like boardgames, is briefly introduced in Section 4.1.2. A more extended survey on Multi-Game Playing history and achievements is available in [123].

2.1.1 Metagame

Metagame [147, 148, 149] developed by Barney Pell is a complex system consisting of the game description language, the general game player, and the game generator. The introduced class of *symmetric chess-like games* describes the large portion of two-player, rectangular board, perfect information games, where players move and capture pieces along specified directions. The system supports many concepts existing in fairy-chess games like promotions, mandatory captures, vertical-cylinder boards, and piece possession changes.

Symmetric chess-like games

The games have to be symmetric in every aspect, which means all defined entities are subject to the global inversion symmetry. It is enough to define pieces initial position, goal conditions, movement rules, and promotion rules for the white (game starting) player. An example of *American Checkers* codified as a symmetric chess-like game is presented in Listing 2.1.

The most complex part of the description is the definitions of pieces. Pieces are defined by the powers of *moving*, *capturing*, and *promoting*, which can be optionally constrained. Movement rules are defined based on the chess piece type division theory [35]. We distinguish three types of movements. The *leap* takes the piece from one square directly to the other (given the *direction vector*), without regard for intervening squares. For example the Chess *knight* movement rules consist of $\langle 1,2\rangle$*-leap*, $\langle 2,1\rangle$*-leap*, $\langle -1,2\rangle$*-leap*, and so on. A move that allows a piece to continue for some number of identical leaps, as long as visited squares are empty, is called a *ride*. Considering only its move forward, the Chess *rook* is a $\langle 0,1\rangle$*-rider*. We can modify the *ride* condition regarding visited squares and restrict that some of these squares must be occupied by pieces to construct a *hopper*. An example of such piece is the Draughts *man* or the Chinese Chess *cannon* which jumps over the piece (own or opponent's) without capturing it.

A piece movement is defined as a union of movement patterns described above. Moreover, the system takes advantage of the symmetrical nature of the fairy-chess games, and provides the set of symmetry operators: *forward*, *side*, or *rotate*. After applying to a particular movement, *forward* operator reflects the move over the x-axis, *side* reflects it over the y-axis, and *rotate* swaps x and y fields of the move's direction-vectors. Thus, the Chess *knight* is defined by $\langle 2, 1\rangle$ direction vector combined with all three symmetry operators.

Capturing moves are define similarly to movement rules. Additionally, they require determining capturing method. It is e.g. possible to capture by *clobbering* via finishing move on another piece, or by *hopping* over a piece. Allowed targets can be restricted, so there can be a piece that can capture only opponent's rooks. Lastly, the pieces can differ in capture effects. Standard *remove* effect just removes a piece from the board. A *player possesses* or *opponent possesses* effect gives the captured piece to a player (move-maker or opponent respectively), who can later put in on some empty square instead of a standard movement. A number of additional movement options and restrictions are possible e.g. setting minimal/maximal number of leaps for a rider movement, or compulsory/continuous captures requirement.

Promotions are handled by defining a *promotion territory* (shared between all pieces) and a set of possible promotions for every piece (which can be empty). When a piece finishes its move within the promotion territory it has to be replaced by one of the possible choices. The game rules can state that the promotion choice has to be made by the opponent.

There are three types of goal conditions. A *stalemate goal* is achieved in a position in which a specified player (player or opponent) has no legal moves. Family of *eradicate goals* consist of elimination of certain type(s) of pieces. Combined statements are allowed, requiring e.g. elimination of all opponents knight and all own rooks. An *arrival goal* condition is fulfilled when a specified piece (player's or opponent's) reaches a certain board square.

Expressiveness and generation

Metagame language (the class of symmetric chess-like games) has been deliberately designed to be a generalization based on the real game examples. It has been constructed accordingly to fairy-chess domain Dawson's *Theory of Movements* [35], hence the division of movement types to leap, ride, and hop. This allows to easily handle many chess variants, especially given many options to handle specific game types (after-capture possession, change of ownership, vertical-cylinder board). Yet the language lacks in flexibility to express more idiosyncratic rules.

By using various representational tricks it is possible to implement most of the rules of many games including *Chess*, *Giveaway Chess*, *Replacement Chess*, *Chinese Chess*, *Shogi*, *Checkers*, *Lose Checkers*, *Tic-tac-toe*, and *Gomoku*. However, some common chess-based concepts as the initial double pawn move, en passant, and castling, cannot be expressed properly. It is impossible to prohibit placing a pawn on a file already containing a pawn, which is a rule in Shogi. The Metagame encoding of Chess ([149]) allows to leave a king in check, so straightforward king capture is required to win the game. Also the notion of the stalemate, accordingly to the symmetric chess-like games definition, do not properly reflects the rules of the proper chess.

Metagame system provides also a game generator, which is able to produce novel, playable, chess-like games. More details on this part of Metagame is presented after the introduction to the Procedural Content Generation domain, in Section 4.1.2.

Declarative game description language

Metagame's game definition language, provided by a formal grammar, describes symmetric chess-like games in the human-friendly and generator-friendly way. A straightforward general game player implementation should follow this formalization. However, an approach taken by Pell introduces a separate intermediate language to provide efficient general game playing mechanism. Because of that, it is possible to consider the class of symmetric chess-like games as a meta-game

Listing 2.1: Definition of *American Checkers* as a symmetric chess-like game, as presented in [149].

```
GAME         american_checkers
GOALS        stalemate opponent
BOARD SIZE   8 BY 8
BOARD TYPE   planar
PROMOTE RANK 8
SETUP man AT {(1,1)(3,1)(5,1)(7,1)(2,2)(4,2)
              (6,2)(8,2)(1,3)(3,3)(5,3)(7,3)}
CONSTRAINTS must_capture

DEFINE man                          DEFINE king
  MOVING                              MOVING
    MOVEMENT                            MOVEMENT
      LEAP                                LEAP
      <1,1> SYMMETRY {side}               <1,1> SYMMETRY {forward side}
        END MOVEMENT                        END MOVEMENT
  END MOVING                          END MOVING
  CAPTURING                           CAPTURING
    CAPTURE                             CAPTURE
      BY {hop}                            BY {hop}
      TYPE [{opponent} any piece]         TYPE [{opponent} any piece]
      EFFECT remove                       EFFECT remove
      MOVEMENT                            MOVEMENT
        HOP BEFORE [X = 0]                  HOP BEFORE [X = 0]
            OVER [X = 1]                        OVER [X = 1]
            AFTER [X = 0]                       AFTER [X = 0]
        HOP_OVER [{opponent} any piece]     HOP_OVER [{opponent} any piece]
        <1,1> SYMMETRY {side}               <1,1> SYMMETRY {forward side}
      END MOVEMENT                        END MOVEMENT
    END CAPTURE                         END CAPTURE
  END CAPTURING                       END CAPTURING
  PROMOTING                           CONSTRAINTS continue captures
    PROMOTE_TO king                 END DEFINE
  END PROMOTING
  CONSTRAINTS continue captures
END DEFINE
                               END GAME.
```

representing any legal move in any game in any position. Declarative nature of that language brings out high-level relationships between games and positions in a particular game.

This language, called the *game description language*, is a direct predecessor of the *Stanford's Game Description Language* from 2005 [65] (Section 2.2.1). The syntax and semantics of the language is very similar to Prolog. Additionally, it has embedded state variables *current state*, *current player*, *current game* and the following constructs:

- `true(P)`: a state-dependent property `P` is true in the current game state;
- `add(P)`: adds state-dependent property `P` to the current state;

- `del(P)`: removes state-dependent property `P` from the current state;
- `control(Player)`: is true if `Player` is a current player;
- `game:Pred`: a current player considers a game-dependent property `Pred` true in the current state;
- `transfer_control`: transfers control from the current player to the opponent.

Rewriting rules from the *symmetric chess-like games* grammar and a complete formalization of the *game description language* (gdl) are presented in [149]. An interesting consequence of the intermediate gdl-based approach is that theoretically a class of valid games for Metagame player (described in the next section) is far more general then the boardgames. However, a player is restricted by its construction to understand just the subset of the gdl.

Metagamer

Metagamer is the part of Metagame system responsible for the playing algorithm [150]. The search engine of the player is mainly based on the standard chess-like game search techniques: minimax algorithm, alpha-beta pruning, iterative deepening, and principal continuation heuristic [110]. It requires a heuristic function to evaluate arbitrary game positions. In the case of metagaming, it has to do this for an arbitrary game. Thus, meta-level evaluation functions are used, which are generalizations of some standard evaluation functions used for chess-like games.

Instead of dynamically learning feature weights, the Metagamer version presented in [150] encodes *feature evaluation function* (FEF), which assigns feature weights for the given game. The function takes a feature name, and evaluate this feature with respect to the more low-level functions called *subfeatures*. So the subfeature mechanism serves to measure the importance of features considering usefulness of its aspects. For example, the subfeatures for the standard *material* feature consider e.g. general piece mobility, a number of squares it can reach in the limited number of turns, types of pieces it can capture, usefulness to achieve game goal, promotion potential, etc.

The game-independent knowledge sources (features and subfeatures), following *Hoyle*, are called *advisors*. However, Metagame restricted itself to use only constructive advisors, i.e. those that are describing behaviors positive for the player. The set of considered advisors is broad, and they can be categorized into the four groups:

- game-independent features (e.g. mobility, threats);
- game-independent feature generators (e.g. piece type features, positional bonuses);
- feature generator subfeatures (e.g. average piece mobility, number of reachable squares from a given square);
- weights for the features and subfeatures.

An experiment on the static material analysis compared weights determined by the FEF with the game-expert knowledge. The predetermined set of advisors was chosen, each with weight one assigned. The obtained results were very close to the expectations, with the tendency for undervaluing pieces that can be promoted (*pawns*, *men*). During the real gameplay Metagamer uses a dynamically-computed *promote-distance* advisor.

Barney Pell also introduces the Metagame Tournament ([149]), where different versions of the Metagamer played against each other on a number of generated games. The proposed format of the competition has been later incorporated in large part by the Stanford's International General Game Playing Competition (Section 2.2.2).

2.1.2 Simplified Boardgames

Simplified Boardgames is the class of fairy-chess-like games introduced by Björnsson in 2012 [12]. The language describes turn-based, two player, zero-sum games on a rectangular board with piece movements being a subset of a regular language and independent on the move history.

The class of Simplified Boardgames was initially developed for the purpose of learning the game rules through the observation of play (which is described in detail in Section 3.2.2). Later, it has been used for procedural content generation (Chapter 4) and as a testground for evaluating the Stanford's GGP based players (Section 6.2).

Syntax and semantics

Here we follow our definition from [98] to introduce the language describing Simplified Boardgames rules. The formal grammar in EBNF is presented in Appendix A. A Simplified Boardgames codification of the game *Gardner*[1] is presented partially, as an example, in Listing 2.2.

The game is played between the two players, *black* and *white*, on a rectangular board. White player is always the first to move. The board size is given by the two numbers in the `<BOARD>` section, which represent the *width* and the *height*, respectively. Subsequently, the initial position is given: empty squares are represented by dots, white pieces as the uppercase letters, and black pieces as the lowercase letters. To be considered as valid, there must be exactly *height* rows and *width* columns. Although it may be asymmetric, the initial position is given from the perspective of the white player, i.e. forward means "up" for white, and "down" for black.

During a single turn a player has to make a move using one of his pieces. Making a move is done by choosing the piece and change its position according to the specified movement rule for this piece. At any time at most one piece can occupy a square, so finishing a move on a square containing a piece (regardless of the owner) results in removing it (capturing). Note that in the situation when the destination square is the starting one the whole board remains unchanged. No piece addition is possible. After performing a move, the player gives control to the opponent.

The movement rules of available game pieces are declared in the `<PIECES>` section. One piece can have at most one movement rule, which consists of the letter of the piece and a regular expression. A piece without the movement rule is allowed but cannot be moved. For a given piece, the set of legal moves is the set of words described by a regular expression over an alphabet Σ containing triplets $(\Delta x, \Delta y, \mathit{on})$, where Δx and Δy are relative column/row distances, and $\mathit{on} \in \{e, p, w\}$ describes the content of the destination square: e indicates an empty square, p a square occupied by an opponent piece, and w a square occupied by an own piece. It is assumed that $x \in \{-\mathit{width}+1, \ldots, \mathit{width}-1\}$, and $y \in \{-\mathit{height}+1, \ldots, \mathit{height}-1\}$, and so Σ is finite.

While the piece's owner is defined by the case (upper or lower), its letter encode the piece's type. Pieces with the same type have the same language of legal moves, thus declaration is made for the white pieces only. A positive Δy means forward, which is a subjective direction, and differs in meaning depending on the player.

Consider a piece and a word $w \in \Sigma^*$ that belongs to the language described by the regular expression in the movement rule for this piece. Let $w = a_1 a_2 \ldots a_k$, where each $a_i = (\Delta x_i, \Delta y_i, \mathit{on}_i)$, and suppose that the piece stands on a square $\langle x, y\rangle$. Then, w describes a move of the piece, which is applicable in the current board position if and only if, for every i such that $1 \le i \le k$, the content condition on_i is fulfilled by the content of the square $\langle x + \sum_{j=1}^{i} \Delta x_j, y + \sum_{j=1}^{i} \Delta y_j\rangle$. The

[1]Gardner is 5×5 Chess variant proposed by Martin Gardner in 1969 and weakly solved in 2013 [128] – the game-theoretic value has been proved to be a draw. The starting position looks as in the orthodox Chess with removed columns f, g, h, and rows 3, 4, 5. The rules are those of classical Chess without the two squares move for pawns, en passant moves and castling. Additionally, as a countermeasure for not supporting promotions, our implementation provides additional winning condition by reaching the opponent's backrank with a pawn.

Listing 2.2: The partial description of Gardner game in Simplified Boardgames language, as presented in [98].

```
<<Simplified Gardner>>
<BOARD>
5 5
|rnbqk|
|ppppp|
|.....|
|PPPPP|
|RNBQK|
<PIECES>        // P - pawn, R - rook, N - knight, B - bishop, Q - queen, K - king
P (0,1,e) + (-1,1,p) + (1,1,p) &
R (0,1,e)(0,1,e)^* + (0,1,e)^*(0,1,p) + (0,-1,e)(0,-1,e)^* + (0,-1,e)^*(0,-1,p) +
  (1,0,e)(1,0,e)^* + (1,0,e)^*(1,0,p) + (-1,0,e)(-1,0,e)^* + (-1,0,e)^*(-1,0,p) &
N (2,1,e) + (2,-1,e) + ... + (-1,-2,p) &
B (1,1,e) + (1,1,p) + (1,1,e)^2 + (1,1,e)(1,1,p) + (1,1,e)^3 + (1,1,e)^2(1,1,p) +
  (1,1,e)^4 + (1,1,e)^3(1,1,p) + ... + (-1,-1,e)^4 + (-1,-1,e)^3(-1,-1,p) &
Q (0,1,e)(0,1,e)^* + (0,1,e)^*(0,1,p) + (0,-1,e)(0,-1,e)^* + (0,-1,e)^*(0,-1,p) +
  (1,0,e)(1,0,e)^* + (1,0,e)^*(1,0,p) + (-1,0,e)(-1,0,e)^* + (-1,0,e)^*(-1,0,p) +
  (1,1,e)(1,1,e)^* + (1,1,e)^*(1,1,p) + (1,-1,e)(1,-1,e)^* + (1,-1,e)^*(1,-1,p) +
  (1,-1,e)(1,-1,e)^*+(1,-1,e)^*(1,-1,p)+(-1,-1,e)(-1,-1,e)^*+(-1,-1,e)^*(-1,-1,p) &
K (0,1,e) + (0,1,p) + (0,-1,e) + (0,-1,p) + ...  + (-1,-1,e) + (-1,-1,p) &
<GOALS>
100 &
@P 0 4, 1 4, 2 4, 3 4, 4 4 &
@p 0 0, 1 0, 2 0, 3 0, 4 0 &
#K 0 &
#k 0 &
```

move of w changes the position of the piece piece from $\langle x, y\rangle$ to $\langle x + \sum_{i=1}^{k} \Delta x_k, y + \sum_{k=1}^{k} \Delta y_i\rangle$. An example of how move rules work is shown in Figure 2.1.1.

Lastly, the `<GOALS>` section provides the game terminal conditions. The first value is the *turnlimit*, whose exceedance automatically causes a draw if no other terminal condition is fulfilled. The turnlimit is given in the so-called „half-moves" in Chess, i.e. the value t means $\lceil \frac{t}{2} \rceil$ moves of the first player and $\lfloor \frac{t}{2} \rfloor$ moves of the second player. A player automatically loses the game when he has no legal moves at the beginning of his turn (e.g. because he has no pieces left).

A player can win by moving a certain piece to a fixed set of squares, which are defined by the entries denoted by the `@` symbol. The values are given in absolute coordinates and $(0, 0)$ square is located in the lower left corner of the board. Alternatively, a player can lose if the number of his pieces of a certain type reaches a given amount, defined by the entries denoted by the `#` symbol. The terminal conditions can be asymmetric.

Expresiveness

The language can describe many of the fairy chess variants in a concise and fairly human-readable way. Comparing to Pell's *Metagame*, Simplified Boardgames include games with asymmetry and

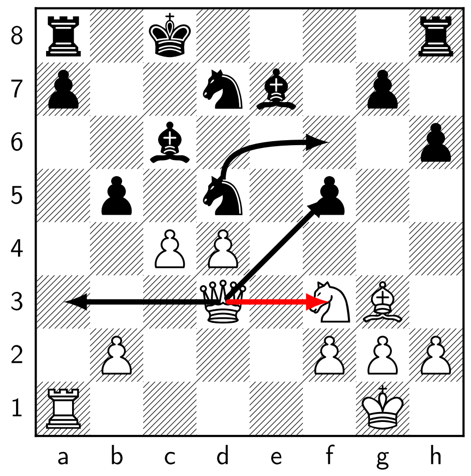


Figure 2.1.1: A Chess example. Two legal moves for the queen on $d4$ are shown. The capture to $f5$ is codified by the word $(1, 1, e)(1, 1, p)$, while move to $a3$ is encoded by $(-1, 0, e)(-1, 0, e)(-1, 0, e)$. The move to $f3$ is illegal, as in the language of queen's moves, no move can end on a square containg own's piece. The $d5 - f6$ knight move is a direct jump codified by the one-letter word $(2, 1, e)$.

position-dependent moves (e.g. the Chess initial double pawn move). Also, as the rules support any regular expression as pieces' movements, they are no longer restricted just to hop, leap and ride types.

The concise method of move encoding in Simplified Boardgames, unified and without any exceptions, is also its weakness. It makes impossible to express promotions, non-direct capturing, and interactions between concrete pieces (e.g. special knight that can capture only rooks). Also, the concept of keeping captured pieces and putting them on the board later, introduced in Metagame via special keywords, cannot be stated in the Simplified Boardgames language. Some limitations concerning castling, en passant, and all the moves depending on the game history, are common for both languages.

The usage of finite automata for describing pieces' rules, and thus to move generation, allows fast and efficient computation of all legal moves given a board setup. The regularity of the description makes it also convenient for procedural content generation and learning game rules by observing others play.

An extension of Simplified Boardgames has been recently presented in [69]. The language has been modified to support piece addition and deletion. Unfortunately, these additional capabilities are added a little bit artificially, which results in complications concerning game semantics. For example, after a creation of a piece the control of the move pattern shifts to created piece without possibility to take control back. Also, the type of piece created is not encoded in a move pattern directly.

2.2 Stanford's General Game Playing

Although the term *General Game Playing* is a common keyword for every multi-game playing approach, it is mostly associated with the standard introduced in 2005 by the Stanford University Logic Group [65] and paired with the annual International General Game Playing Competition (IGGPC). As such, Stanford's GGP was identified as a new Grand Challenge of Artificial Intelligence and quickly evolved into a well established research area.

Stanford's GGP is focused on developing programs that can play, without human intervention, in any previously unknown game described by the Game Description Language (GDL). The task requires techniques from a broad range of classic AI topics, including knowledge representation, search, planning and learning [219]. These aspects makes GGP a valuable tool for AI education, e.g. regular GGP courses are conducted by the Stanford University and shared, as a massive open online courses, on the Coursera platform [62].

In this section, we introduce two languages associated with the Stanford's General Game Playing, GDL and GDL-II, and briefly describe so-far achievements, playing techniques, and history of the IGGPC. For a detailed survey of the entire discipline we recommend [66], and [210] for the recent advances.

2.2.1 GDL

The Stanford's Game Description Language (GDL) [118] is a formal language to describe rules of any turn-based, finite, deterministic, n-player game with simultaneous moves and perfect information. By „finite" we mean that the set of possible game states, and the set of actions (moves) that players can choose in each state, should be finite. Also every match should end after a finite number of turns. Players perform actions simultaneously, which means that in each turn all players select their moves without knowing the decision of the others. Sequential games can be simulated by explicit adding a single „no-operation" move for the players that should normally wait. No game element can be random, and all players should have the full information about the game state.

GDL is a specialization of the *Knowledge Interchange Format* (KIF) [64], first-order logic based language designed to share and re-use information from knowledge-based systems. GDL is an extended variant of *Datalog* [2], allowing function constants, negation, and a restricted form of recursion. It is a high-level, strictly declarative language, using logic programming-like syntax very similar to Prolog. It makes possible to rewrite GDL game as a Prolog program without much effort.

Every game description contains declarations of player roles, the initial game state, legal moves, state transition function, terminating conditions and the goal function. GDL does not provide any predefined functions: neither arithmetic expressions nor game-domain specific structures like board or card deck. Every function and declaration must be defined explicitly from scratch, and the only keywords used to define the game are presented in Table 2.2.1. This ensures that, given the game rules, any additional background knowledge is not required for the player.

The execution model works as follows. Starting from the initial state, in every state s every player r selects one legal action. Then, the joint move of all players is applied to the state update function to obtain a new state s'. When s' is terminal, then every player has given a goal score and the game ends.

GDL rules are usually written in the KIF prefix notation. Variables have to begin with `?` symbol. Comments start with the semicolon (`;`). The following expressions are used to encode standard logic operators.

- The *rule* `(<= h b1 b2 ... bn)` is the equivalent of the $h \Leftarrow b1 \wedge b2 \wedge \ldots \wedge bn$ implication.

Table 2.2.1: GDL Keywords

`(role ?r)`	`?r` is a player
`(init ?f)`	fact `?f` is true in the initial state
`(true ?f)`	fact `?f` is true in the current state
`(legal ?r ?a)`	in the current state `?r` can perform action `?a`
`(does ?r ?a)`	`?r` performed action `?a` in the previous state
`(next ?f)`	`?f` will be true in the next state
`terminal`	current state is terminal
`(goal ?r ?n)`	player `?r` score is `?n`

The head (or consequence) h must be an atomic sentence and it is *true* if all the conditions $b1 \ldots bn$, which can be literals or disjunctions, are satisfied.

- Negation is written as `(not s)` for s being an atomic sentence.
- `(or s1 ... sn)` denotes a logical alternative operator applied to the atomic sentences $s1, \ldots, sn$.
- Inequality is expressed by the notation `(distinct p q)`, which is *true* iff p and q are syntactically not equal.

We will now present an overview of the GDL semantics using the Tic-tac-toe description in Listing 2.3 (based on the example from [66][2]). More formal description of the semantics is provided in Section 2.2.3 when introducing the GDL-II extension. To ensure the game is *valid* and a finite set of rules always has a finite model as its semantics, the game has to fulfill additional syntactic restriction, e.g. the game specification must be *stratified* and *allowed*. However, GDL itself is Turing-complete, so checking well-formedness as well as terminality and winnability is undecidable [166]. For the detailed description of the official syntax, semantics, and requirements of GDL we refer to [118].

Line 1 introduces the players using the `role` relation. In the presented description two players are defined: `white` and `black`. The `init` relation occurrences in lines 3–6 define the facts that hold in the initial state of the game. They are added to the knowledge base, and are later available using the `true` relation. A game state consists of the board content (initially all cells are blank) and the player-to-move information.

We define legal movements in lines 8–10. If the player has the control it can mark any cell that is blank. Otherwise, the only legal action is `noop`. The game advances by temporal extending the database by adding `does` fact for every `legal` move chosen by the players. The domain of `does` and `legal` relations is the same.

Then, the `next` relation (lines 12–26) determines the new game state. All variables in a rule are universally quantified and variables sharing the same name have to be equal. Thus, unification process is required to determine the equality over the terms. For example a fact that some symbol is preserved in the board (lines 18–19) is stored for every `?m`, `?n`, `?w` such that cell at (`?m`, `?n`) contains `?w` which is not equal to the blank symbol `b`.

Such a rule is required as GDL follows the closed-world assumption (*completeness*), so everything that cannot be derived as true from the current state is assumed to be false. All the facts are removed from the knowledge-base, and the facts derived by the `next` relation are the only ones which are `true` in the next game state.

[2]With the additional `(distinct ?w b)` conjunct in line 47, which fixes the error in the example rules provided in [66, 118]. Without this correction the game terminates in the first turn.

Listing 2.3: GDL rules of Tic-tac-toe.

```
(role white) (role black)

(init (cell 1 1 b)) (init (cell 1 2 b)) (init (cell 1 3 b))
(init (cell 2 1 b)) (init (cell 2 2 b)) (init (cell 2 3 b))
(init (cell 3 1 b)) (init (cell 3 2 b)) (init (cell 3 3 b))
(init (control white))

(<= (legal ?w (mark ?x ?y))  (true (cell ?x ?y b)) (true (control ?w)))
(<= (legal white noop)       (true (control black)))
(<= (legal black noop)       (true (control white)))

(<= (next (cell ?m ?n x))
    (does white (mark ?m ?n)) (true (cell ?m ?n b)))

(<= (next (cell ?m ?n o))
    (does black (mark ?m ?n)) (true (cell ?m ?n b)))

(<= (next (cell ?m ?n ?w))
    (true (cell ?m ?n ?w)) (distinct ?w b))

(<= (next (cell ?m ?n b))
    (does ?w (mark ?j ?k)) (true (cell ?m ?n b))
    (or (distinct ?m ?j) (distinct ?n ?k)))

(<= (next (control white))  (true (control black)))
(<= (next (control black))  (true (control white)))

(<= (row ?m ?x)
    (true (cell ?m 1 ?x)) (true (cell ?m 2 ?x)) (true (cell ?m 3 ?x)))
(<= (column ?n ?x)
    (true (cell 1 ?n ?x)) (true (cell 2 ?n ?x)) (true (cell 3 ?n ?x)))
(<= (diagonal ?x)
    (true (cell 1 1 ?x)) (true (cell 2 2 ?x)) (true (cell 3 3 ?x)))
(<= (diagonal ?x)
    (true (cell 1 3 ?x)) (true (cell 2 2 ?x)) (true (cell 3 1 ?x)))
(<= (line ?x)
    (or (row ?m ?x) (column ?m ?x) (diagonal ?x)))
(<= open  (true (cell ?m ?n b)))

(<= (goal white 100)  (line x) (not (line o)))
(<= (goal white 50)   (not (line x)) (not (line o)))
(<= (goal white 0)    (line o) (not (line x)))
(<= (goal black 100)  (line o) (not (line x)))
(<= (goal black 50)   (not (line x)) (not (line o)))
(<= (goal black 0)    (line x) (not (line o)))

(<= terminal  (line ?w) (distinct ?w b))
(<= terminal  (not open))
```

When the `terminal` relation holds (lines 47–48), it indicates the game is over and the current game state is the last. The `goal` relation (lines 40–45) assigns score to every player. The scores assigned in the last game state correspond to the game outcome. GDL systems assume that the valid scores are the natural numbers from range $[0, 100]$, so for the zero-sum game like Tic-tac-toe 0 means loss, 100 means win, and 50 indicates a tie.

The rules computing `goal` and `terminal` use temporary, user-defined predicates: `row`, `column`, `diagonal`, `line` and `open` (lines 28–38). The process of reasoning about the game state requires these predicates to be fully computed in order to establish holding `goal` and `terminal` facts. Such predicates are added to the knowledge-base and can be used inside the rules, yet they are not accessed via `true` keyword and they automatically vanish during the state update.

Communication protocol

Every GGP match is supervised by an application called *Game Controller* or *Game Manager* (GM), which is responsible for supervising the course of the game. A few such managers are available online and some of them allow to test a player against other online players, e.g. the Sam Shreiber's GGP.org[3], the Dresden GGP Server[4], and the Stanford's Gamemanager[5]. A standalone Game Manager application, which can be used to test a player locally, is available as the GGPServer SourceForge project[6].

During the match, players and GM communicate using HTTP protocol. All the players take role of servers, waiting for a connection initialized by the Game Manager (which is technically a client application). A match begins with the `START` message sent by GM to the players. It contains unique match identifier, player roles, game rules in GDL, and two timelimits: *startclock* and *playclock*. The players have time to process the given rules, and after *startclock* seconds they have to send confirmation message (named `READY`) to GM.

At every turn GM initializes new connection, identifying proper match using previously given identifier (so it is theoretically possible for a Stanford's GGP player to play more than one match simultaneously, although it is not used in practice). It sends the `PLAY` message containing all the moves made by the players (or `nil` value for the first move), which allows players to advance their local game state accordingly to the actual GM's state of the game. Players have then *playclock* seconds to response with a legal action they chose to make. If the timelimit is exceeded or GM receives an invalid move, a random legal move is used instead. During the competitions, repetition of such situation usually causes the player to lose by default.

If the Game Manager reaches a terminal state it sends `STOP` message, which also contains players' moves. This allows every player to compute the final scores, which are not sent directly.

Visual overview of the message exchange between GM and the players is shown in Figure 2.2.1. The detailed specification of the Game Manager implementation and communication protocol can be found in [65].

2.2.2 Methods and Results

In this section we present the progression of methods used in Stanford's GGP. Firstly we introduce the Stanford's International General Game Playing Competition, which is the leading event stimulating the research.

Then, we discuss the methods of General Game Playing used to advance in the domain, starting with so-called *knowledge-based* approach, which led to winning the two first editions of

[3]`http://tiltyard.ggp.org/`
[4]`http://ggpserver.general-game-playing.de/ggpserver/`
[5]`http://ggp.stanford.edu/applications/gamemanager.php?id=tictactoe`
[6]`https://sourceforge.net/p/ggpserver/code/HEAD/tree/trunk/`

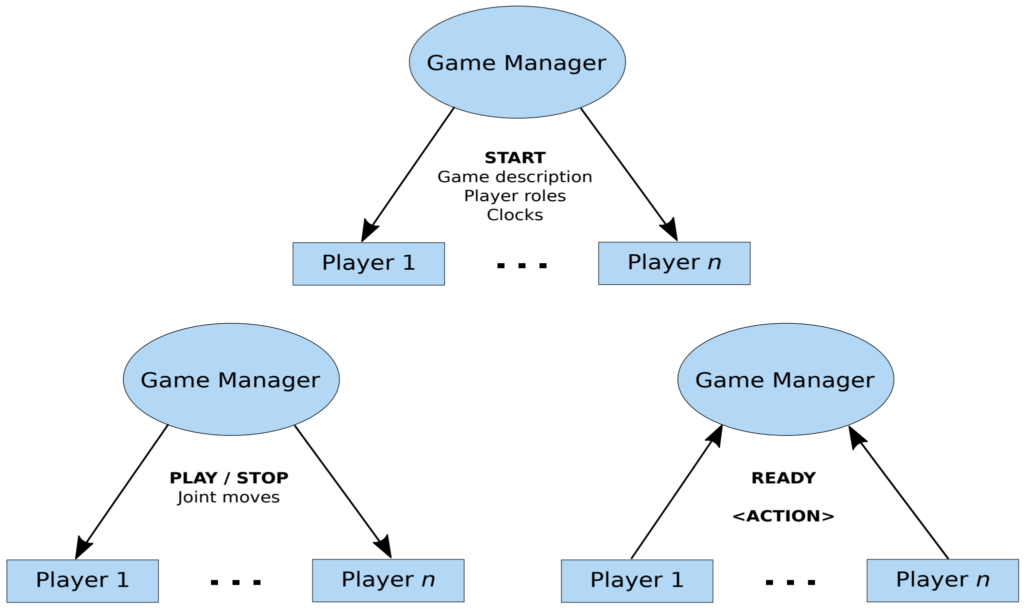


Figure 2.2.1: Communication between the Game Manager and the players.

the competition. Next, we briefly introduce the Monte Carlo Tree Search (MCTS) algorithm, which dominates GGP thereafter, and initialized an alternative branch of GGP research called *simulation-based* approach.

International General Game Playing Competition

From 2005, the annual International General Game Playing Competition (IGGPC) is taking place to foster and monitor progress in the Stanford's GGP research area [65]. University-based teams, and recently also independent developers, test their GGP players against other state-of-the-art programs. The competitions are typically associated and co-located with either the AAAI conference or IJCAI each year.

A wide variety of games are played: turn-based or simultaneous move, zero-sum or non-zero-sum, with complexity ranging from simple puzzles to challenging chess-like games. Competition, preceded by the qualification round, usually takes two days, with top 8 teams (usually there is 10-16 qualified participants each year) advanced to the second day's finals. Our player *Dumalion* (which core part is described in Chapter 7) joined 2014 championship (16 qualified participants) and has been classified in the top eight after eliminating previous year's champion TurboTurtle. The list of winning players is presented in Table 2.2.2.

The competition significantly contributes to improve the level of GGP programs. According to the survey in [63], over the years of the competition, general game players become more sophisticated, significantly more powerful, and undoubtedly today's players can easily beat the programs developed early on.

Knowledge-based approach

In the early years of the Stanford's GGP research the main approach was to use well-known game playing algorithms related to the min-max search. As such techniques require a state

Table 2.2.2: Winners of the Stanford's International GGP Competition

Year	**Player**	**Authors**	**University**
2005	Cluneplayer	Jim Clune	University of California, Los Angeles
2006	FluxPlayer	Stephan Schiffel Michael Thielscher	Dresden University of Technology
2007	CadiaPlayer	Yngvi Björnsson Hilmar Finnsson	Reykjavík University
2008	CadiaPlayer	Yngvi Björnsson Hilmar Finnsson Gylfi Þór Guðmundsson	Reykjavík University
2009	Ary	Jean Méhat	Paris 8 University
2010	Ary	Jean Méhat	Paris 8 University
2011	TurboTurtle	Sam Schreiber	*independent*
2012	CadiaPlayer	Hilmar Finnsson Yngvi Björnsson	Reykjavík University
2013	TurboTurtle	Sam Schreiber	*independent*
2014	Sancho	Steve Draper Andrew Rose	*independent*
2015	Galvanise	Richard Emslie	*independent*

evaluation function the key is to gather knowledge about the game rules and the state usefulness. Some innovations introduced by the knowledge-based approach dramatically improved players' performance, and are important part of most of the top-one players [74].

Cluneplayer [26], the first IGGPC winner, uses alpha-beta min-max algorithm with a paranoid assumption, i.e. every other player was considered as opponent. Based on the ideas from [103], the player assumes that the given game is a boardgame and tries to analyze predicates behavior to detect a board, pieces, turn-counters, etc. The presented algorithm is not obfuscation-safe, e.g. it assumes that the board is always stored straightforwardly as a ternary predicate. The evaluation function of the Cluneplayer is based on the features which are GDL expressions. The interpretations of every piece and its statistical properties (e.g. variance) are determined by analysis of the expression's occurrences in GDL code and random game tree exploration. Based on the abstract parameters including feature-based payoff function, terminal-state probability, and degree of player's control, the game state evaluation formula is determined.

Alternative and more sophisticated method of generating evaluation function has been embedded in the FluxPlayer [177], the winner of the second IGGPC tournament. Based on the known logical values of the predicates in the current state, the fuzzy logic is used to estimate truthfulness of the entire formulas. As every GDL game can be translated into the fully grounded formulas (without any variables) the probability of the `goal` relations can be used to estimate each player's payoff.

A real number between 0 and 1 is assigned to every GDL formula. For an atomic sentence the value is assigned based on the current state and information about the atom's behavior, e.g. any currently holding atom has a value greater then 0.5, but its value is 1 only if it is persistent (true for the rest of the game). The value of a formula $\neg\varphi$ is 1 minus the value of φ. Calculation of the value for conjunction uses some arbitrary t-norm. It is further tuned by a special formula based on the constant $t > 0.5$ to gain the desired behavior for long sequence of conjuncts. The value for disjunction is calculated from the conjunction and De Morgan's law. Finally, a formula with the value greater than t is considered as true, otherwise it is false.

An extension of the above method, presented in [132], encodes propositional logic as a neural network, which is then used for training from the past matches and improve state evaluation accuracy. A general neural networks-based state evaluation algorithm, able to correctly represent complex domain theories, has been described in [129].

The issue of detecting subgames and factoring GDL games has been addressed in [71, 240]. If the given game consists of n subgames, each with branching factor B, the process of factorization can reduce its branching factor from B^n to Bn. By analyzing the *dependency graph*, which nodes are predicates and edges connect predicates linked by the game rules, independent subgames corresponding to graph's connected components can be detected. Then, the move-searching procedure can be decomposed into combination of *subgame search*, responsible for excluding from the subgame the game tree subtrees irrelevant to the global solution, and *global game search*, which performs searching only on given subgame trees.

Another approach, whose goal is to reduce the complexity of the search space, is to exploit the game symmetries. The symmetry detection algorithm presented in [174] transforms game rules into so-called *rule graph*, and uses standard methods to compute its automorphisms. Every such automorphism describes a mapping of the game symbols (constants, functions, relations) corresponding to the symmetries of states, actions, and roles. Although the method can be fooled, because it relies on the fact that the rules with the same semantics have the same structure, it remains useful and allows to apply e.g. transposition tables to improve standard searching techniques.

A special method to gather knowledge about a given game is to formulate hypothesis and use sample matches to verify its correctness. Alternatively, a formalized method of proving arbitrary theorems concerning game rules has been described in [178, 188]. Given a propositional formula describing the game property, the static analysis of the `next` rules is performed and the Answer Set Programming [57] system is used in order to establish whether the rules entail this property. The described technique can prove/disprove theorems concerning: uniqueness of board cells content, periodic return of control features (e.g. every 4-th turn, white is the player-to-move), persistence of an atomic sentence (e.g non-blank symbol in Tic-tac-toe board stays true once it becomes true), or that the game has the zero-sum property.

The above technique is also used for solving inferential frame problem. Normally GDL requires explicit computation of all components of a successor state, even if some of them remain unchanged. The algorithm that allows to speed up the game tree search by updating the current state description only by the components that actually change has been presented in [164].

The next important topic is the knowledge transfer. During the competition it is usually assumed that the given game was not previously seen by the players. Detection of a game repetition is simple only when the game rules are not obfuscated, and some online servers use obfuscation by default. Graph-based method for identifying previously seen games has been described in [104]. It does not guarantee that every game with the equivalent semantics will be matched correctly, but it is able to improve player's performance also in a case when the given game is not particularly the same but it is a variant of already known one. More sophisticated and computationally expensive approach, using the formalized definition of *space-consistent game equivalence*, has been presented in [239]. The described system is able to detect the semantic equivalence of the games with totally different syntax, like Tic-tac-toe and Number Scrabble ([149]).

Lastly, we want to point out the research concerning the distance measurement in General Game Playing. Distance features are important aspect of evaluation functions, measuring e.g. the distance of a pawn towards the promotion rank in Chess or the distance between pac-man and the ghosts. A method to automatically construct admissible distance measures by analyzing the game rules, not limited to specific structures such as Cartesian game boards, has been described in several incremental works including [130, 131].

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [21, 25] is a simulation-based algorithm which does not require any heuristic knowledge for evaluating the game states. Also, it is not restricted to only two player, zero-sum games, like a standard min-max implementation. The random game simulations (so-called *playouts*) returning final scores are performed and the final decision is based on the collected statistical data. Starting with the success of CadiaPlayer [45] in 2007, all following IGGPC winners are the MCTS-based players.

It has been proved that the MCTS algorithm converges to min-max [16]. For some types of games, especially zero-sum two-player games with codified expert human knowledge, MCTS does not offer significant advantages over the alpha-beta min-max search. However, it has proven to be a successful algorithm in many games where the traditional searching techniques were ineffective, including Go [30, 61], Amazons [117], Arimaa [102], Chinese Checkers [199], and Settlers of Catan [212]. The capabilities of MCTS has been remarkably depicted by the recent success of the Google DeepMind AlphaGo, which become the first computer program to beat a professional human Go player on a full-sized board [193].

MCTS algorithm runs in loop, using simulations to gradually build a partial game tree and keep track of the average action scores for the in-tree states. The process of running a single simulation consists of four stages, as presented in Figure 2.2.2.

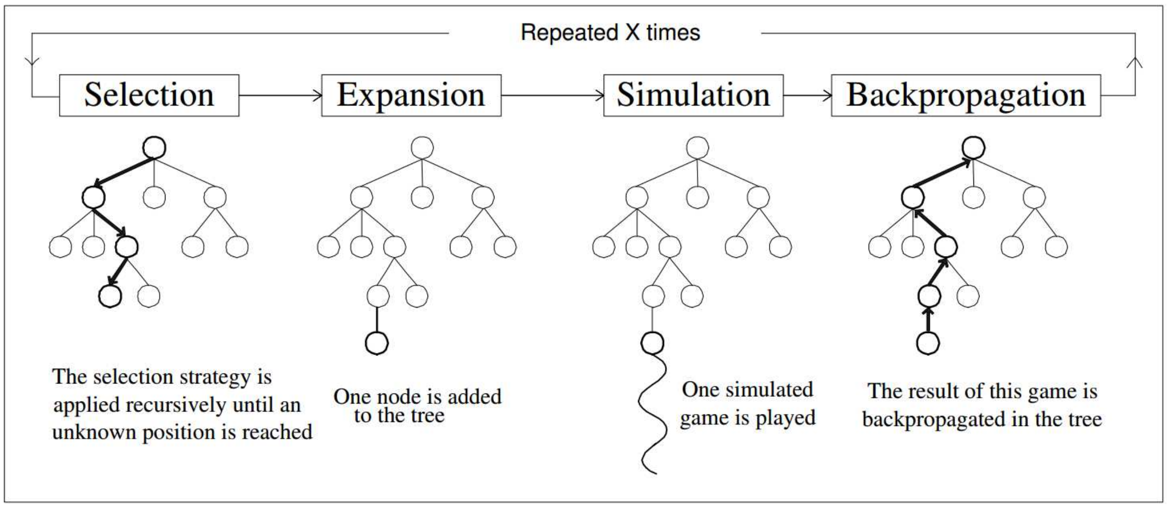

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection strategy is applied recursively until an unknown position is reached
One node is added to the tree
One simulated game is played
The result of this game is backpropagated in the tree

Figure 2.2.2: An overview of a single MCTS simulation, as presented in [25].

Selection. Starting from the root, succeeding states are selected until the leaf of the MCTS tree is reached. The right strategy selection can significantly improve the overall performance. The most popular method is the *Upper Confidence Bounds applied to Trees* (UCT) algorithm [92], which allows to set the balance between exploration and exploitation ratio. For each tree node (game state) s, the best action a^* is chosen using the formula

$$a^* = argmax_{a \in A(s)} \left\{ \frac{R(s,a)}{N(s,a)} + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\}, \tag{2.1}$$

where $A(s)$ is the set of actions legal in state s, $R(s,a)$ is the sum of rewards obtained by sampling action a in state s, $N(s)$ is the number of previous visits in s, and $N(s,a)$ is the number of times a has been sampled in s. If, for some action a, $N(s,a)$ equals 0 (action has never been sampled before) it is selected by default (with some tie-breaking strategy if there is more then one such action). The parameter C determines influence of the exploration UCT bonus to the overall score. Theoretically it should be equal to $\sqrt{2}$, yet in practice it is usually chosen empirically.

Expansion. The typical implementation of the expansion step adds to the MCTS tree exactly one new node, obtained by choosing random action from the leaf node reached in the selection phase. Although other variants exist, this one avoids too fast growth of the tree and dynamically builds the tree in the most promising areas (accordingly to the selection strategy).

Simulation. Starting from the newly expanded node, a full game simulation is performed. Thus, the result of the simulation are true game reward values. The most common are so-called *light playouts* that are generated by the random moves. The *heavy playouts*, which use more sophisticated strategies during simulation, can lead to more accurate reward estimations but are also more computationally expensive.

Backpropagation. The last stage is responsible for propagation of the knowledge obtained by the simulation. The values of $N(s)$, $N(s,a)$, and $R(s,a)$ are updated on the path from the expanded node up to the tree root, accordingly to the playout's result.

Multiple simulations are partially independent processes, which allows to parallelize some of the computational effort [127, 207]. The three main techniques are *leaf parallelization* (all child processes receive the same leaf node from the master process), *root parallelization* (all child processes maintain their own MCTS trees and send top-level statistics on master process' demand), and *tree parallelization* (only one MCTS tree is maintained and child processes receive different nodes to test).

As a single simulation is usually fast, MCTS algorithm can be terminated in any moment and it is able to return the best answer obtained so far. Usually the action with the greatest average score or the most simulations is chosen.

Simulation-based approach

Since CadiaPlayer [45, 48] win in 2007, simulation-based approach became leading method for General Game Playing. Because MCTS does not require specific domain knowledge it is the ideal choice for playing games that are unknown in advance. However, such generality implies that for very large games, given the competition clock restrictions, it is impossible to randomly visit part of the tree big enough to stabilize weights and provide a reliable output. (The influence of GGP game tree properties including depth and width, and progression on MCTS efficiency has been studied in [49].) Thus, most of attention is given to the tasks of guiding MCTS tree expansion into the most promising regions and increasing reliability of simulation by using the *heavy playouts*. Such techniques were already incorporated for MCTS-based Go playing, e.g. *Rapid Action Value Estimation* (RAVE) heuristic introduced in [59].

Four novel search-control mechanisms for guiding MCTS simulations have been described and empirically evaluated in [44, 46, 47]. *Move-Average Sampling Technique* (MAST) learns average score for every move regardless of the game state. Return value of every simulation is backpropagated from the simulation leaf to the tree root, updating scores for the actions kept in some global lookup table. During the selection phase the strategy is modified to be biased towards selecting moves with higher average score. This heuristic assumes that there exist „universal" moves, which are better in most of the situations, e.g. marking center cell in Tic-tac-toe.

Tree-Only MAST (TO-MAST) is a variation of MAST where the action scores are calculated only inside the MCTS tree (i.e. starting from the newly expanded node rather than from the leaf reached by the simulation). This methods prefers quality of data rather then quantity, as information gathered is more robust and with less variance, so potentially leads to decisions based on a better basis.

Predicate-Average Sampling Technique (PAST) is based on the same scheme as MAST, however it looks over the game state predicates, i.e. predicates that are true in a given state. Now, the global lookup table contains average scores for predicate-action pairs. This approach is computationally more expensive, because for each state encountered during every simulation, for each state predicate, predicate-action average values have to be recalculated. PAST is also more sophisticated technique than MAST, as it takes into account that some actions are good only in a given context.

Features-to-Action Sampling Technique (FAST), unlike previously mention algorithms is not knowledge-free. It uses the same framework as MAST, but captures the relation between actions and boardgame-type features: piece types and cells. The features are detected using knowledge-based techniques, and their relative importance is learned using Temporal Difference $TD(\lambda)$ algorithm [200]. FAST allows to use high-level properties and relations between game state and action importance. This is particularly effective for many games appearing at IGGPC, but gathering domain-dependent knowledge is troublesome, expensive, and not always reliable. Overall, the experiments presented in [44] show that the best simulation control algorithm (among presented here), against both standard MCTS and MAST, is PAST.

In [234], the authors present technique for generation evaluation functions inspired by [26]. It advances the feature construction method by, inter alia, feature generalization phase which allows to build and use compound features. Generated heuristics are applied to $MTD(f)$ [159] min-max algorithm and Guided Monte Carlo Tree Search. Although multiple guidance techniques are considered, reported results use novel approach to use evaluation function for simulations cutoff. For every node encountered during the random simulation, there is a minor chance that the simulation will be halted, and the heuristic evaluation of the state will be used as a playout's result.

Alternative method of guided search [202, 205], being a part of the MINI-Player program, applies a set of predetermined heuristics (called a *portfolio of strategies*) which are used to guide MCTS simultaneously, but with different weights assigned. The presented portfolio consists of seven strategies, including four with novel contributions.

Approximate Goal Evaluation uses and-or trees to compute degree of satisfying the goal. *Exploration* strategy uses a notion of a difference between two game states (computed as a number of newly appeared facts) to guide the search to a state that maximize the minimal difference with the last N visited states. *Statistical Symbols Counting*, inspired by [124], counts the occurrences of facts within the same relation and numbers of symbol occurrences in a specific relation's position. Lastly, the *Score* strategy detects the situations when the score depends on a counter relation (e.g. the number of pieces) and uses a degree of counter progression as a heuristic score.

The set of priorities for portfolio of strategies can be assigned statically before the game, learned during the initial start phase, or learned on the fly based on the historical average score. The priority for each strategy is a natural number indicating the proportional number of simulations guaranteed for this strategy, i.e. if one strategy has two times larger priority than the other, it will perform twice as much simulations. The adaptation of guided MCTS based on the portfolio of strategies, dedicated to single player games, has been presented in [208].

The recent proposition in [231] turns fuzzy logic state evaluation from [177] into an action heuristic. It is achieved by taking the goal condition, regressing it one step and filtering it according to an action of a player. The heuristic is incorporated into guided MCTS and tested in two roles: as the playout heuristic, and as the tree heuristic.

In [43] two extensions to MCTS are described. The *Early Cutoff* extension decides when continuing the current simulation is less profitable than cutting it and starting a new one. Such situation may take place when the simulation seems irrelevant and backpropagated information will probably be just a noise, e.g. playout has reached distant state in a long chess-game expected to draw by the turn-limit. The *Unexplored Action Urgency* extension uses gathered heuristic knowledge to bypass some unexplored actions, resulting in more narrowed search beam. For an initial study of human-machine iterative cooperation in the GGP domain, based on the MCTS+UCT search, we refer to [209]. In the experiments provided, the human player had insight into the results computed by the assisting MCTS player and could guide simulations into more interesting areas (by disabling some moves or changing their selection priorities).

Other improvements of MCTS for GGP can be found in [44, 202]. Attempts to improve the efficiency, rather than the accuracy of MCTS, has been presented in Section 7.1.

2.2.3 GDL-II

Game Description Language for Incomplete Information games (GDL-II) [217], introduced in 2010, is an extension of GDL describing even larger class of games by removing the GDL restrictions that games have to be deterministic and with full information. As its predecessor, it is a strictly declarative language using logic programming-like syntax. It has been shown that any extensive-form game [161] can be translated into GDL-II, which means that GDL-II can describe all finite, synchronous, turn-based, n-player games [220].

The extension introduces only two new keywords in addition to the GDL keywords listed in Table 2.2.1, whose semantics remains unchanged. These new keywords are listed in Table 2.2.3. Intuitively, `random` role is a special player handled by the Game Manager who always makes a random legal move, while `sees` allows to hide parts of the game state from the players in an asymmetric manner.

Table 2.2.3: The new keywords in GDL-II

`random`	random player role (Nature, casino, ...)
`(sees ?r ?p)`	player `?r` will perceive `?p` in the next state

Semantics

GDL-II specification has to fulfill the same syntactic restrictions as GDL to ensure that every game has a unique standard model with only finite number of true positive instances [118]. Given that, all deductions in Definition 2.2.1 providing GDL-II semantic are finite and decidable

Let G be a valid GDL-II game description. It contains a finite number of function symbols and constants that determines the set of possible ground terms Σ. Although Σ can be infinite, syntactic restrictions ensure that all sets needed to compute game flow (roles, legal moves, reachable states, etc.) are finite subsets of Σ [118]. Let $S = \{f_1, \ldots, f_k\}$ be a state of the game denoted as the set of predicates that are true in the current position. Then we define a *base state* as $S^{\texttt{true}} \overset{\text{def}}{=} \{(\texttt{true}\ f_1), \ldots, (\texttt{true}\ f_k)\}$. Let us also denote joint move $A^{\texttt{does}} \overset{\text{def}}{=} \{(\texttt{does}\ r_1\ a_1), \ldots, (\texttt{does}\ r_n\ a_n)\}$ if players $r_1, \ldots, r_n$ took actions $a_1, \ldots, a_n$. We can now introduce

Definition 2.2.1. *[217] The semantics of a valid GDL-II n-player game specification G with set of ground terms Σ is given by a state transition system composed as follows.*

- $R = \{r \in \Sigma : G \models \texttt{(role r)}\}$ *(player names);*
- $s_0 = \{f \in \Sigma : G \models \texttt{(init f)}\}$ *(initial state);*
- $t = \{S \in 2^\Sigma : G \cup S^{\texttt{true}} \models \texttt{terminal}\}$ *(terminal states);*
- $l = \{(r, a, S) : G \cup S^{\texttt{true}} \models \texttt{(legal r a)}\}$, *for all* $r \in R$, $a \in \Sigma$ *and* $S \in 2^\Sigma$ *(legal actions);*
- $u(A, S) = \{f : G \cup A^{\texttt{does}} \cup S^{\texttt{true}} \models \texttt{(next f)}\}$, *for all joint moves* $A : (R \mapsto \Sigma)$ *and states* $S \in 2^\Sigma$ *(state update);*
- $\mathcal{I} = \{(r, A, S, p) : G \cup A^{\texttt{does}} \cup S^{\texttt{true}} \models \texttt{(sees r p)}\}$, *for all* $r \in R \setminus \{\texttt{random}\}$, $A : (R \mapsto \Sigma)$, $S \in 2^\Sigma$ *and* $p \in \Sigma$ *(players' percepts);*
- $g = \{(r, n, S) : G \cup S^{\texttt{true}} \models \texttt{(goal r n)}\}$, *for all* $r \in R$, $n \in \{0, \ldots, 100\}$ *and* $S \in 2^\Sigma$ *(goal values);*

The execution model works as follows. Starting from the initial state s_0, in every state S every player $r \in R$ selects one legal action a such that $(r, a, S) \in l$. The `random` player chooses his moves randomly with uniform probability. Then the joint move is applied to the state update function $u(A, S)$ to obtain a new state S'. In S' every role $r \in R \setminus \{\texttt{random}\}$ perceives every p that satisfies $(r, A, S, p) \in \mathcal{I}$. If the current state is terminal, i.e. $S \in t$, then every player has given the score by relation $(r, n, S) \in g$ and the game ends.

The communication protocol differs in the arguments of the `PLAY` and `STOP` messages. Instead of the moves made by the players, only the move of the current player and the *percepts* are sent. The percepts of the player are all predicates that are visible to him according to the `sees` relation.

As an example of a GDL-II game the Monty Hall rules are presented in Listing 2.4[7]. It is a simple but famous game where a car prize is hidden behind one of the three doors and the player is given two chances to pick a right door. For more detailed language specification and game examples we refer to [180, 218].

Reasoning about the games in GDL-II

GDL-II makes the general game playing problem even harder, as any belief state the player is in can correspond to multiple real-game states. So far only two international competitions were organized: 2011 German Open in GGP (won by the Fluxii, the GDL-II version of the FluxPlayer), and 2012 Australian Open (won by GDL-II version of CadiaPlayer).

One approach to reasoning about GDL games with imperfect information is to translate them into some other well studied formalism. In [180] GDL-II is fully embedded into the specific version of the Situation Calculus [173]. The Situation Calculus axiomatisation tells the players how to reason about their own percepts, what they entail about the current position, and what the other players may know. The syntactic and semantic translation to Alternating-time Temporal Epistemic Logic, presented in [165], allows to verify even larger class of strategic and epistemic properties, and provides a proof that the model checking in this setting is decidable.

The straightforward playing algorithm propositions include the *HyperPlay* and *Norns* algorithms. The HyperPlay technique ([183]) translates imperfect-information games into a format acceptable by the standard perfect-information players. It uses sampling of the imperfect information game state space followed by the simulation plays in each of these perfect-information samples. The obtained separate results are combined afterwards to make the best choice of movement. The Norns algorithm ([56]) traces the belief state tree to simulate a potential past

[7]The game is written in an alternative infix Prolog-like notation, which is common in many papers due to its better readability in comparison with the KIF-based notation.

Listing 2.4: A GDL-II description of the Monty Hall game, as presented in [181].

```
role(candidate).
role(random).

init(closed(1)).
init(closed(2)).
init(closed(3)).
init(step(1)).

legal(random,hide_car(D))  :- true(step(1)), true(closed(D)).
legal(random,open_door(D)) :- true(step(2)), true(closed(D)),
                              not true(car(D)), not true(chosen(D)).
legal(random,noop)         :- true(step(3)).

legal(candidate,choose(D)) :- true(step(1)), true(closed(D)).
legal(candidate,noop)      :- true(step(2)).
legal(candidate,noop)      :- true(step(3)).
legal(candidate,switch)    :- true(step(3)).

sees(candidate,D) :- does(random,open_door(D)).

next(car(D))    :- does(random,hide_car(D)).
next(car(D))    :- true(car(D)).
next(closed(D)) :- true(closed(D)), not does(random,open_door(D)).
next(chosen(D)) :- does(candidate,choose(D)).
next(chosen(D)) :- true(chosen(D)), not does(candidate,switch).
next(chosen(D)) :- does(candidate,switch),
                   true(closed(D)), not true(chosen(D)).

next(step(2)) :- true(step(1)).
next(step(3)) :- true(step(2)).
next(step(4)) :- true(step(3)).

terminal :- true(step(4)).

goal(candidate,100) :- true(chosen(D)), true(car(D)).
goal(candidate,  0) :- true(chosen(D)), not true(car(D)).
goal(random,     0).
```

that leads to a present state that is consistent with the received observations. The underlying mechanism consists of Monte Carlo belief state tree search and Bayesian weight updates. It has been proved that this algorithm asymptotically converges to the optimal action.

2.3 General Video Game Playing

The General Video Game AI (GVGAI) framework and competition [107, 154] is a brand new and very rapidly growing area of GGP research. Although it was inspired by Stanford's GDL, it represents a totally different philosophy. Firstly, it is restricted only to Atari-like, 2-dimensional, one-player games. Secondly, all submissions have to fit within the given framework (inherit from a certain Java class) and are tested locally by the competition organizers. Note that due to the Stanford's GGP communication protocol the players are allowed to use e.g. clusters of computers to support their programs. Thirdly, the players are not allowed to see the rules of the games. Instead, they are provided with the forward model in the form of a Java game state object. This object allows the player to query the game status (winner, score, time step), player's state (position, orientation, resources), positions of various objects in the level, and history of events during the game. Given that, the players can use search techniques to learn how the game behaves without the necessity of reimplementing the game engine. Also, the game is more real-time as every game step is required to take less than 40 ms. The GVGAI framework is open-source and can be downloaded from `www.gvgai.net`.

The GVGAI Competition is based on the Video Game Definition Language (VGDL), described briefly in the next section. It is similar to the Arcade Learning Environment (ALE) [10] used by Google DeepMind in their GGP research concerning Atari 2600 games [133]. VGDL is more open ended and provides for the players more game state related information, as the input for ALE controllers is just the raw screen capture and the score counter.

2.3.1 Video Game Definition Language

VGDL is a high-level description language for (one-player, non-deterministic) 2D video games. The prototype of the language has been presented in [39], and its formalization and detailed syntax description are available in [171]. As the game interpreter and visualization engine have been developed in Python, based on the *pygame* library, this initial version is referred to as *PyVGDL*. Currently, the GVGAI Competition uses the Java port of the PyVGDL, which is usually called just the Video Game Definition Language.

The language is still evolving and new concepts are regularly added accordingly to the requisition of the constantly expanding GVGAI Competition. The language is based on a number of predefined concepts such as map, player agent, and sprite. Interactions between sprites are defined using functions from the predefined set, e.g. kill the sprite, change its type, add some amount of resources, or modify the game score.

The VGDL codification of the game Space Invaders (named Aliens in the GVGAI game set), originally released for the Atari 2600 in 1980, has been presented in Listing 2.5. The `SpriteSet` section (lines 2–11) defines the classes of objects (sprites) with possible subclasses. For example an avatar of the type `FlakAvatar` represents player's spaceship that can move sideways and make a „shoot" action. Missiles of two types are defined, one going up (fired by the player) and the other going down (slower, fired by the alien bombers). The `LevelMapping` section (lines 13–16) defines mapping between characters used for codifying the initial game state and game sprites. An exemplary level is presented in Figure 2.3.1. By default, letter `A` represents the player avatar and `w` a wall. For every game multiple levels are defined. `TerminationSet` section (lines 18–20) defines a termination criterion for the game. The game can end in two main states: win or

Listing 2.5: VGDL description of the game Aliens (GVGAI port of Space Invaders).

```
 1 BasicGame
 2   SpriteSet
 3     base    > Immovable    color=WHITE img=base
 4     avatar  > FlakAvatar   stype=sam
 5     missile > Missile
 6         sam  > orientation=UP     color=BLUE singleton=True img=spaceship
 7         bomb > orientation=DOWN   color=RED  speed=0.5 img=bomb
 8     alien   > Bomber   stype=bomb prob=0.01 cooldown=3 speed=0.8 img=alien
 9     portal  > invisible=True hidden=True
10         portalSlow  > SpawnPoint   stype=alien cooldown=16 total=20 img=portal
11         portalFast  > SpawnPoint   stype=alien cooldown=12 total=20 img=portal
12
13   LevelMapping
14     0 > base
15     1 > portalSlow
16     2 > portalFast
17
18   TerminationSet
19     SpriteCounter        stype=avatar               limit=0 win=False
20     MultiSpriteCounter   stype1=portal stype2=alien limit=0 win=True
21
22   InteractionSet
23     avatar  wall  > stepBack
24     alien   wall  > turnAround
25     missile wall  > killSprite
26
27     base bomb > killBoth
28     base sam  > killBoth scoreChange=1
29
30     base   alien > killSprite
31     avatar alien > killSprite scoreChange=-1
32     avatar bomb  > killSprite scoreChange=-1
33     alien  sam   > killSprite scoreChange=2
```

loss. If a player wins, then its score is taken into account. In the example, the player loses if its avatar is killed, and wins if he removes all alien structures from the game. The potential events, triggered when two objects collide, are defined in the `InteractionSet` section (lines 22-33). In our example alien ships turn around after hitting the wall (line 24), player's avatar is destroyed after a contact with an alien ship or a bomb (lines 31,32), and player's missile hitting an alien ship destroys it and increases the player's score (line 33).

There are many event methods and types defined in the ontology that can describe objects behavior. There is even physics-defining option that allows to alter default grid-type physics to make an object subjected to the gravity rules. Although the number of predefined concepts is large, some behaviors are impossible to express, and adding new capabilities requires framework modification.

```
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
w                              w
w1                             w
w000                           w
w000                           w
w                              w
w                              w
w                              w
w                              w
w    000      000000     000   w
w   00000    00000000   00000  w
w   0   0    00    00   00000  w
w                A             w
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
```

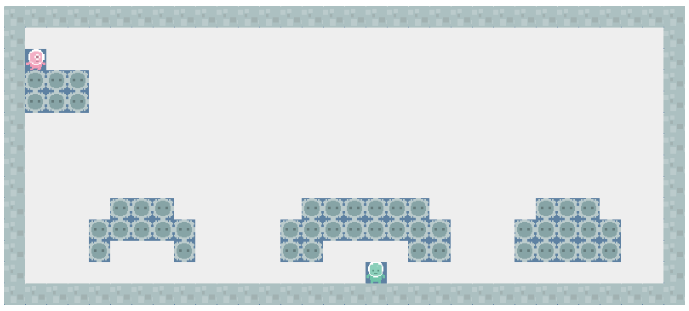


Figure 2.3.1: One of the levels for VGDL Aliens game. Plain text codification on the left. On the right visualization as seen during the game.

2.3.2 Methods and Results

In this section, we briefly present advancements in GVGAI research. We begin with introducing the official General Video Game Playing AI Competition. Then we provide a short survey of published game playing approaches, mostly focused on the MCTS-based algorithms. Research related to the procedural games and levels generation, introduced in the recent framework extension, is presented in Chapter 4.

GVGAI Competition

The first GVGAI competition was associated with IEEE CIG conference in 2014, and received 14 submissions. The summary of the competition, including detailed results, game descriptions, and winners strategies, has been presented in [153]. In 2015, three competitions co-located with ACM GECCO, IEEE CIG, and IEEE CEEC conferences were organized, each with more than 45 participants. In 2016 four competitions were held including two new tracks – level generation and two player planning, in addition to the GVGAI traditional one-player planning track.

Algorithms

Due to the model used by the GVGAI competition, the most popular approach is to use the MCTS algorithm. To increase its efficiency various modifications were proposed.

In [155] the authors introduce the Knowledge-based Fast Evolutionary MCTS for GVGAI, and show that it is significantly better than the plain MCTS. In every simulation phase the presented algorithm evaluates a single individual of the evolutionary algorithm, and provides the reward calculated at the end of the simulation as a fitness value. The subject to evolution is a vector of features' weights, and features are the euclidean distances to the closest sprites of different types. The final score function uses a concept knowledge base with two factors: curiosity and experience. The reported downside of the approach is that it does not take into account the direction of a collision, which matters in some games.

In [52] four more MCTS modifications have been tested. *MixMax backups* modifies the exploitation part of UCT by interpolating between the average score and the maximum score, which makes the good path contributes more to the average than the bad ones, and prevent too cowardly agent behavior. *Macro Actions* modifies the expansion process by extorting a fixed number of simulations using all actions before creating the new MCTS tree node. This modification increases the search depth at the cost of the precision. *Partial Expansion* is another approach (similarly as *Unexplored Action Urgency* tested for Stanford's GDL) allowing algorithm to consider further node's descendants without the necessity to explore all its children. The novel *Reversal Penalty* modification decreases the number of agent's oscillations, i.e. situations when the player goes back and forth between a few adjacent tiles. In the proposed implementation, five most recent positions are stored for every node, and UCT values of moves that lead to these positions are multiplied by 0.95 (which is a 0.05 penalty). The presented experimental results show that the combination of MixMax and UCT with Reverse Penalty behaves best for most (but not all) of the tested games.

The solutions that deals with nondeterminism in GVGAI domain using *open loop search* approach has been presented in [152]. The idea is to store in the search tree not the game states but rather the states statistics. Thus, if chosen properly, the statistics will be representative for the set of states that can be obtained after applying an action in the state. The experiments mainly tested the application of the Rolling Horizon Evolutionary Algorithm. An individual is encoded as a sequence of actions of the predetermined length, and recombination and mutation are used to generate offspring. Every individual is evaluated multiple times, building the game tree that allows to reuse statistics from the already visited nodes.

Alternatively, an approach based on the Iterated Width (IW) algorithm, previously used for playing video-games in the Arcade Learning Environment [10], has been tested in [55]. IW(i) is a classical planning algorithm that is based on the breadth-first search modified such that newly created states are pruned if they do not make at most i new state atoms true [114]. In the presented experiments, IW(1) algorithm outperforms sample controllers including plain BFS, MCTS and Open Loop MCTS for all tested response times.

Lastly, we point out the experiments based on learning. The authors in [168] implemented four versions of the Separable Natural Evolution Strategies algorithm [172] and trained them on 10 VGDL games, using 1000 trials for each game. In most of the tested examples, learning process led to the significant improvement of the player's average score. The game features extracted for every state were not only euclidean distances to the closest different type sprites, but also the speed of the avatar and the amounts of owned resources.

2.4 Other Domains

In this section, we briefly introduce other general game description languages developed during past few years and describing games from various domains. These are less known languages, mostly developed for a purpose of a single research project and not associated with any competition, nevertheless they represent interesting and novel approaches.

2.4.1 Strategy Game Description Language

Strategy Game Description Language (SGDL), introduced in [119] and further extended in [122], has been developed to uniformly describe a wide range of concepts used in video/boardgame strategies. The main difference between arbitrary strategy games and chess-like games described in Section 2.1 is in the complexity of the game state and additional data assigned to the *units* (which are equivalents of the figures in boardgames). Usually, a strategy game unit is described using several attributes like *health*, *speed*, *range*, *attack*, etc., and interactions between them are more complex (e.g. subtract half of the opponent *attack* from unit's *health*, if the unit is in the range between opponent's *range* and 2×*range*).

Main criteria for the SGDL development were that the language should be: *complete* – to model most aspects of the existing strategy games, *evolvable* – to allow searching through the space of the possible games, and *human-readable* – to make hand-made rule changes possible.

Language structure

SGDL decomposes games into three layers:

1. The *mechanics* layer, which determines the fundamental rules of the game such as how attack actions work, what is the game environment (e.g. 2D grid), and what are the winning conditions.
2. The *ontology* layer, which specifies types of elements existing in the game (e.g. roads, mountains, factories, tanks) and their properties (movement cost for mountains, speed for tanks, etc.)
3. The *instance* layer, which describes the setup of an individual match: map layout, initial unit placement, and other match specific conditions.

To fulfill the evolvability criterion, mechanics and ontology layers are encoded using a tree-based representation, as commonly used in genetic programming [160]. In SGDL an object class (e.g. unit) consists of the three components: a unique identifier, a set of attributes (with numerical, alphanumerical, or boolean values assigned), and a set of actions that maps conditions to effects, encoding rules of the „*if* ... *then* ...” form. An action tree structure, as shown in Figure 2.4.1, consists of various types of nodes:

- *Actions* (triangle shape). When action is invoked test given *conditions*, if they are true *consequences* are executed.
- *Comparators* (oval shape) combine their children's outputs and return to their parent a boolean value. They can be used to refer arguments passed into the current action invocation or access special game objects like the map.
- *Operators* (diamond shape) combine value of the right child with the operator and assign the result as the left child for the set operators (e.g. =, !), or return the numerical result to the parent for the mathematical operators ($+$, $-$, $\times$, $/$).
- *Constants* (circle shape) are leaf nodes that contain constants.
- *Non-deterministic* nodes represent the parameterizable random number generators.

Actions can create new objects on the map, e.g. units, obstacles, or temporary objects that disappear after some number of turns. Action consequences can be layered to simplify the notation for the sequential ordering of their application. Global game state and the players can also contain attributes and actions similarly as the game objects described above.

Usage

Examples of SGDL games are not published (probably due to the rules complexity), and detailed syntax and semantic description is partially provided by the „work in progress” document[8]. SGDL-based recreation of Dune 2 [198] has been used in [121] to develop fast map generation algorithm. The language has been also used as a testbed for a balanced unit sets generation. (See Section 4.1 for the introduction into the Procedural Content Generation.) This experiment used two types of strategy games described in SGDL: *Complex Rock Paper Scissors* (CRPS) and *Rock Wars* (RW).

[8]`http://game.itu.dk/index.php/Strategy_Games_Description_Language`

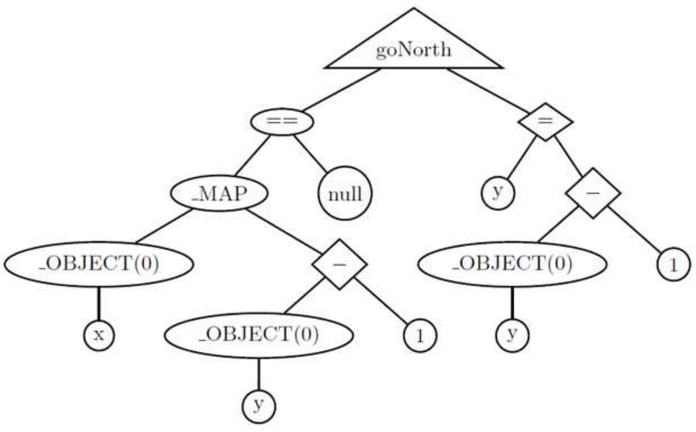


Figure 2.4.1: An example of the *goNorth* action tree. The action is possible if the output of the special *_MAP* predicate is *null*. Invocation arguments are stored in x, y and consequence of the rule is the y value subtracted by one ([119]).

In the first game, every player contains three pieces that can move into adjacent squares of the quadratic map. There are three possible types of units and they have various attributes responsible for number of damage points dealing to every opponent type. This allows to create complementary units interacting in a paper-rock-scissor manner, which is a behavior embedded in many popular strategy games.

In the latter scenario, each player contains a single factory that can produce units accordingly to their cost attribute and the number of resources left. The map is no longer plain but the squares can contain impassible *rock* objects. Also, in contrast to CRPS, a player can move all units simultaneously.

The task of balanced unit generation in CRPS was to generate attribute values for three unit types such that an army consisting of only one type of unit is weaker than the army containing all of them [119]. Later experiments use min-max and MCTS agents to evaluate generated game rules in the RW scenario by measuring balance, tension, and thrill of the playouts [122].

2.4.2 Toss

The Toss language is a game describing formalism alternative to Stanford's GDL. It is using so-called structure rewriting games ([83]), based on the first-order logic with counting. States of the game are represented by the relational structures (labeled directed hypergraphs), legal moves by the structure rewriting rules guarded by logic formulas, and the goals of the players by formulas which extend first-order logic with counting.

The associated GGP system is available online[9]. It is an open-source program implementing the game model and GUI for the players. It contains a few board games and some systems with continuous dynamics, and allows players to play against a built-in AI. Toss implements an efficient reasoning engine including CNF and DNF conversions and formula simplifications (interfacing a SAT solver, MiniSAT), and a model-checker for the first-order logic. A various game playing algorithms were developed for Toss, including UCT [85] and the heuristic-based alpha-beta agent, able to win against the state-of-the-art GGP player [86].

[9]`www.toss.sourceforge.net`

Let us briefly consider an example of Toss codification of Tic-tac-toe. The starting structure, presented in Figure 2.4.2, has 9 elements connected by binary row and column relations, R and C. Two unary relations, P and Q, are used to mark the moves of the players. The allowed move of the first player is to mark any unmarked element with P, while the second player can mark such elements with Q. Thus, the game graph (left side of Figure 2.4.2) has two locations representing which player is to-move, and two possible moves (edges) both with one rewriting rule assigned.

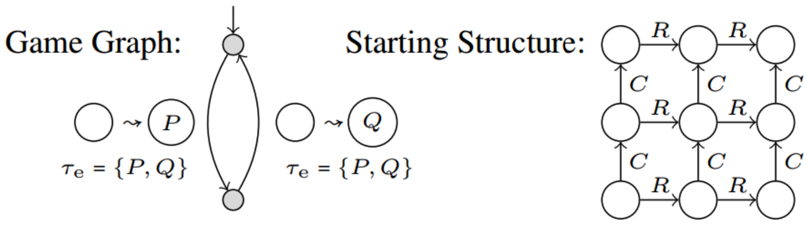


Figure 2.4.2: Tic-tac-toe as a structure rewriting game ([86]).

The complete code for Tic-tac-toe is presented in Listing 2.6. It contains codifications of logical formulas to express the game starting structure, movement rules, players' goals, and payoff function. For example, the concept of diagonals is defined as

$$D_A(x,y) = \exists z\ R(x,z) \land C(z,y) \qquad \text{and} \qquad D_B(x,y) = \exists z\ R(x,z) \land C(y,z),$$

and a line of three by

$$L(x,y,z) = (R(x,y) \land R(y,z)) \lor (C(x,y) \land C(y,z)) \lor (D_A(x,y) \land D_A(y,z)) \lor (D_B(x,y) \land D_B(y,z)).$$

Given these definitions, a goal for the first player can be defined as

$$\varphi = \exists x,y,z\ P(x) \land P(y) \land P(z) \land L(x,y,z),$$

and the payoff is defined using characteristic function of φ, i.e. assigning 1 for all assignments that satisfy φ. For the detailed description of Toss syntax, semantics, and more advanced examples we refer to [85, 86] and the official documentation[10].

Specifications in Toss are more declarative than in GDL and make certain useful game characteristics more visible and easy to capture by the players. While it is not obvious which syntax is more human-readable, the Toss format is certainly more concise.

As presented in [87], it is possible to translate GDL games into Toss. In fact, the Toss player (based on the game rules reformulation) took a part in some of the Stanford's GGP competitions. Such a solution has of course its drawbacks. Transformation between the languages requires prior rewriting of GDL code into normalized form, which is not suitable for too complicated game descriptions. Also, automatically generated Toss rules are not as well-written and easy to process as these prepared especially for this system.

2.4.3 Card Game Description Language

Card Game Description Language (CGDL) has been introduced in [50] for the task of procedural generation of card games. It is a relatively simple context-free-grammar based language capable of expressing the broad range of card games, including games with coins and bets like poker.

For the sake of readability, the detailed description of CGDL, containing examples, is provided in Section 6.1 along with our translation from CGDL to GDL-II.

[10]Available at `http://toss.sourceforge.net/docs.html`.

Listing 2.6: Toss rules of Tic-tac-toe, as presented in [86].

```
PLAYERS X, O
REL Row3(x, y, z) = R(x, y) and R(y, z)
REL Col3(x, y, z) = C(x, y) and C(y, z)
REL DgA(x, y) = ex z (R(x, z) and C(z, y))
REL DgB(x, y) = ex z (R(x, z) and C(y, z))
REL DgA3(x, y, z) = DgA(x, y) and DgA(y, z)
REL DgB3(x, y, z) = DgB(x, y) and DgB(y, z)
REL Conn3(x, y, z) = Row3(x, y, z) or Col3(x, y, z)
                     or DgA3(x, y, z) or DgB3(x, y, z)
REL WinP()= ex x,y,z(P(x) and P(y) and P(z) and Conn3(x, y, z))
REL WinQ()= ex x,y,z(Q(x) and Q(y) and Q(z) and Conn3(x, y, z))
RULE Cross:
   [a | - | - ] -> [a | P (a) | - ]
   emb P, Q pre not WinQ()
RULE Circle:
   [a | - | - ] -> [a | Q (a) | - ]
   emb P, Q pre not WinP()
LOC 0 {
   PLAYER X {PAYOFF :(WinP()) - :(WinQ())
             MOVES [Cross -> 1] }
   PLAYER O {PAYOFF :(WinQ()) - :(WinP())}
}
LOC 1 {
   PLAYER X {PAYOFF :(WinP()) - :(WinQ())}
   PLAYER O {PAYOFF :(WinQ()) - :(WinP())
             MOVES [Circle -> 0]}
}
MODEL [ | P:1 {}; Q:1 {} | ] "
. . .
. . .
. . .
"
```

3
Learning

In this chapter, we deal with the problem of learning game rules by observing. We continue the study initiated by Björnsson in [12] for the class of Simplified Boardgames (Section 2.1.2), focusing on efficient learning of the observed boardgame moves. First, we analyze applications of existing DFA learning algorithms for the task of learning piece movements from the set of game records, and restate the problem in terms of Regular Language Inference [34]. Secondly, we propose our own domain-specific algorithms to ensure better efficiency and higher chance of obtaining a correct approximation of the actual DFA, assuming incompleteness of given training data. We test all presented approaches combining various learning algorithms with different procedures checking DFA consistency in a number of games, both human-made and artificially generated. Following Björnsson's approach we consider two types of training data: GGP-like, where for every position all legal moves are listed, and human-play-like, providing more sparse data where only the move actually made by a player has been recorded.

The chapter is organized as follows. Section 3.1 provides necessary background for the class of Simplified Boardgames, learning by observing, and Regular Language Inference. In Section 3.2, we formally state the problem and analyze selected existing approaches from the theoretical point of view. In Section 3.3, we introduce a new consistency checking procedures, and in Section 3.4 we present our new algorithm learning piece's moves by observing. Section 3.5 presents the results of experiments, evaluating the performance of all the considered approaches. We conclude and give perspective of future research in Section 3.6.

3.1 Preliminaries

In this section we introduce domains relevant to our study, providing necessary algorithms and terminology.

Let Σ^* be the set of all possible words over the alphabet Σ. For DFA $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$, where Q is the set of states, q_0 is the initial state, $F \subseteq Q$ is the set of accepting states, and $\delta : Q \times \Sigma \to Q$ the transition function (which, in our study, is usually a partial function), by

$L(\mathcal{A})$ we denote the language accepted by $\mathcal{A}$.

3.1.1 Learning by Observing

The initial purpose for introduction Simplified Boardgames was to study capabilities of learning game rules, given records of previously played games [12]. As terminal conditions were deliberately kept simple, the proper task comes to learning the languages of piece movements. This extends the directions of the GGP research, beforehand focused mainly on learning how to play given game.

Two types of training data were proposed: a real-world *single move known* scenario – where the observer sees only the performed move, and a GGP-like *all legals known* scenario – where for every position all possible legal moves are visible. The latter case reflects the situation when we want to learn rules of the game given only the reasoner able to compute moves, advance game position, and detect the terminal states. Such novel, entirely simulation based approach of general playing is used for General Video Game Playing Competition.

In [12], the algorithm for learning a DFA consistent with the observed game positions has been presented. To improve learning rate, some additional assumptions were made, allowing unseen moves to be accepted as correct if certain conditions were met. In more detail, the algorithm (referred further as Björnsson's algorithm) is analyzed in Section 3.2.2. Experiments were conducted using three games: Breakthrough, Checkers Breakthrough, and Chess Breakthrough. They consist of 20 test sets containing 50 games for the all legals known scenario, and 1000 games for the single move known scenario. Results contained times required to learn the proper automata for every piece, and number of the games sufficient to produce consistent training data.

Extended approach, using slightly modified Simplified Boardgames domain, has been recently presented in [69]. The language has been modified to support basic piece addition and deletion. The process of learning was performed using the LOCM acquisition system [32, 33], which is an inductive reasoning system that learns planning domain models from action traces.

Other approaches described in the literature are mainly logic-based. In [136], Chess variant rules described as first order logic programs are learned using positive and negative examples, background knowledge, and the theory revision. The system that learns board-based games through the video demonstration using logic and descriptive complexity is presented in [84]. Alternatively, an interaction based approach to learn rules of simple boardgames from the dialogue with human is described in [90].

3.1.2 Regular Language Inference

Regular Language Inference is the problem of finite automata identification using labeled samples: given a disjoint sets of words S_+ containing words belonging to the target language L, and S_- containing words that do not belong to L, we ask what the size of the minimal DFA consistent with these sets is [34].

Gold ([68]) has proven that this problem is NP-hard, and Pitt and Warmuth ([158]) showed that given labeled samples corresponding to automata with n states, there is no polynomial algorithm that guarantees output DFA with at most $n^{(1-\epsilon)\log\log(n)}$ states for any $\epsilon > 0$.

The problem may be efficiently solved by polynomial algorithms if sets S_+, S_- fulfill additional restrictions, i.e. they are somehow representative. The first algorithm, known as TB/Gold was introduced by Trakhtenbrot and Barzdin in 1973 [230], and rediscovered by Gold five years later [68]. In 1992, RPNI (Regular Positive and Negative Inference) algorithm was proposed independently by Oncina and Garcia [142], and Lang [106]. It guarantees that the obtained DFA is consistent, and is equivalent to the target DFA if S_+, S_- contains the so-called *characteristic* *set.* Both theoretical analysis and experiments support the thesis that the results obtained by RPNI are better, and it converges faster than TB/Gold algorithm [54]. The pseudocode and detailed description of RPNI algorithm is provided in Section 3.2.3. Several modifications of RPNI have been studied, e.g. incremental version [38], with faster convergence for some languages subfamilies [23], and suitable for PAC learning [145].

It is also worth mentioning that another approach, alternative to learning from labeled samples, was proposed by Angluin in 1987 [4]. This polynomial algorithm uses the concept of a teacher to learn the target DFA by asking membership and equivalence queries.

3.2 Boardgame Rules Inference

We are dealing with the situation where an agent observes a number of plays between some players and, by observing, it should learn the rules of this game. Assuming we restrict possible games to the class of Simplified Boardgames, where the main challenge consists of discovering the rules of piece movements, the problem can be stated as a specific variant of the Regular Language Inference.

On the one hand, this gives us possibility to take advantage of well known generally applicable algorithms. On the other hand, as the learning domain is narrow and characterized by certain properties, it should be possible to create more effective domain specific learning algorithms.

3.2.1 Problem Statement and Model

Let Σ^* be the set of moves for a given Simplified Boardgame, i.e. it contains words over the alphabet of $(\Delta x, \Delta y, on)$ triplets. Given data obtained by observing movements of a piece p, there exist the partition of Σ^* into the following languages: the language S_+ of observed legal moves, the language S_- of known illegal moves, the language S_0 containing all words that are impossible to perform from any square on the board, and $S_?$ containing all the other words. So, if $w \in S_?$, then it is theoretically possible that w belongs to L_p (the language of legal movements of p), yet there is no evidence in the data that it is legal or not. See Listing 3.1 for example of training data and partition languages.

Let us consider learning scenarios proposed in [12]. In the *single move known* scenario the S_- set is always empty, and $S_?$ may be nonempty. In the *all legals known* scenario S_- is not empty, yet we can still have words in $S_?$.

To meet the reality of Simplified Boardgames, the aforementioned languages have to satisfy the following properties. Languages S_+, S_- and $S_?$ are finite and closed on taking substrings, i.e. for any $w \in S_+ \cup S_- \cup S_?$ every substring of w also belongs to $S_+ \cup S_- \cup S_?$. Language S_0 is infinite, and such that for all $w \in S_0$ and $a \in \Sigma$, we have $aw \in S_0 \wedge wa \in S_0$. Also, there exist a procedure $\mathcal{O} : \Sigma^* \to \{T, F\}$ which for given w decide in time $\Theta(|w|)$ if $w \in S_0$. (We can iterate through the word summing relative distances and checking if the board size was exceeded.) To approximate size of all substring-closed sets we present the following observation. We use standard $|S|$ notation to denote the cardinality of the set S, and $||S||$ to denote the sum of the lengths of words in S.

Observation 1. *Given board of size $n \times n$, and $S = S_+ \cup S_- \cup S_?$. We have that:*

$$2 \sum_{k=1}^{n^2-1} 3^k \frac{(n^2-1)!}{(n^2-1-k)!} \leq |S| \leq n^2 \sum_{k=1}^{n^2-1} 3^k \frac{(n^2-1)!}{(n^2-1-k)!}, \tag{3.1}$$

which estimates the total number of moves that can be made on such a board.

Listing 3.1: Example of training data (all legals known) containing position from Figure 2.1.1. Consider moves of the white knight: $(-1, 2, e) \in S_+$ because it is listed as one of the legal moves; $(-1, 2, e)(-1, 2, p) \in S_-$ because this move can be performed on current position (capturing black knight) but is not listed as a legal one; $(-1, 2, e)^2 \in S_?$ because in given position we cannot decide its correctness; $(-1, 2, e)^4 \in S_0$ because move length exceeds the board size.

```
r.k....r
p..nb.p.
..b....p
.p.n.p..
..PP....
...Q.NB.
.P...PPP
R.....K.
43
P 1 1 (0,1,e)
P 1 1 (0,1,e)(0,5,e)(0,-4,e)
N 5 2 (2,1,e)
N 5 2 (-2,-1,e)
N 5 2 (1,2,e)
N 5 2 (-1,2,e)
N 5 2 (-1,-2,e)
Q 3 2 (1,1,e)
Q 3 2 (1,1,e)(1,1,p)
      ...
```

Notice that in the case of standard chess-like games, the number of legal moves (which is a superset of S_+) is very small comparing to the left hand side of (3.1). This causes that explicit occurrence search in S_- or $S_?$ is highly inefficient in comparison to checking set membership via $\mathcal{O}$ function.

For a given piece p, let $\mathcal{A}$ be a DFA approximating L_p based on given observations. It is required that $S_+ \subseteq L(\mathcal{A})$ and $L(\mathcal{A}) \cap S_- = \emptyset$. However, there is some freedom concerning relations between $L(\mathcal{A})$ and the remaining sets. Whether $L(\mathcal{A}) \cap S_0$ will be empty or not, in practice do not influence the results generated by $\mathcal{A}$. During the move generation phase, movements that are impossible to be made on the current board position are simply excluded.

The question whether $L(\mathcal{A})$ could contain some words (and which one) from $S_?$ depends on the chosen policy. It is a safe option to assume that every unobserved move is treated as illegal. This ensures that the player based on $\mathcal{A}$ will never produce a move that possibly could be illegal (which may cause e.g. an instant loss). On the other hand, admitting some words from $S_?$ allows to simplify the language definition and minimize produced DFA. Moreover, without extending the set of accepted words, it is impossible to obtain the optimal automaton given data containing only partial knowledge about L_p.

3.2.2 Björnsson's Algorithm Analysis

First, we analyze the algorithm for learning piece rules proposed by Björnsson in [12]. To make possible further references and comparisons we present the algorithm's pseudocode and provide its short description. Still, we refer the reader to [12] for all notions not defined here and the detailed description of the algorithm and results.

The construction of the algorithm consists of two parts: the consistency checking presented in Algorithm 3.2.1[1], and the automaton learning presented in Algorithm 3.2.2.

Algorithm 3.2.1 [12, Algorithm 1] consistent(Piecetype *pt*, DFA *dfa*, TrainingData *td*)

```
 1: for all {$pos \in td$} do
 2:     for all {$sq \in pos.board \mid pieceType(sq) = pt$} do
 3:         $movesDFA \leftarrow generateMoves(dfa, pos, sq)$
 4:         if $pos.moves(sq) \not\subseteq movesDFA$ then
 5:             return $False$
 6:         end if
 7:         if $pos.movelisting = all$ then
 8:             if $movesDFA \supset pos.moves(sq)$ then return $False$ end if
 9:         else { $pos.movelisting = some$ }
10:             for all {$pmp \in movesDFA \setminus U$} do
11:                 if $\mathfrak{F}(pmp) \not\subseteq U$ then
12:                     return $False$
13:                 end if
14:             end for
15:         end if
16:     end for
17: end for
18: return $True$
```

An accepted definition of *consistency* with the training data is that DFA should generate all moves known to be legal, and no moves known to be illegal. It is checked straightforwardly in the all legals known scenario. To handle the single move known scenario additional assumptions have been made. Let U be the union of all movement words observed in a given training data (regardless of the piece). Let $\mathfrak{F}(w)$ be the set of all subsequences of w of length $|w|-1$ treated as words. The consistency checking algorithm assumes that the word w represents a legal move if $w \in U$ or $\mathfrak{F}(w) \subseteq U$.

The learning DFA procedure begins with constructing for a given piece a *Prefix Tree Acceptor* (PTA) from the training data. The main part of the algorithm is based on the priority queue Q, containing candidates for the smallest consistent DFA. The maximal number of search steps is limited by the *MaxExpansions* constant. In each step, the smallest DFA is taken from the queue, and it is compared against the smallest DFA obtained so far (lines 9–12). The function *generalizeCandidates* returns pairs of states within a geodesic distance K, which are then collapsed (two states are merged into one) forming a new automaton. If this automaton passes the consistency test it is inserted into the priority queue (lines 16–19). Note that the collapsing procedure can return a non-deterministic automaton, so an additional determinization procedure is used (line 15). Thus, the theoretical time complexity of the *LearnDFA* algorithm is exponential.

Observation 2. *Let S_+ be the language accepted by the piece's prefix tree acceptor pt, and $\mathcal{C}(k, td)$ the complexity of consistency checking the given training data td and a DFA with k*

[1]In the form originally presented in [12] the algorithm immediately returns *True* if the first of the considered positions fulfills the desired property. Thus, the line 8 of algorithm containing „**return** $movesDFA \subseteq pos.moves(sq)$" has been rewritten to „**if** ($movesDFA \supset pos.moves(sq)$) **then return** $False$ **end if**". It is also worth noticing that, since the state collapsing operation can only enlarge the language, subset checking in lines 4–6 can be safely omitted.

Algorithm 3.2.2 [12, Algorithm 2] LearnDFA(Piecetype *pt*, TrainingData *td*)

```
1: $dfa \leftarrow constructPTA(pt, td)$
2: if not $consistent(pt, td)$ then
3:     return $null$
4: end if
5: $dfa_{min} \leftarrow minimizeDFA(dfa)$
6: $n \leftarrow 0$
7: $Q.insert(dfa_{min})$
8: while not $Q.empty()$ and $n < MaxExpansions$ do
9:     $dfa \leftarrow Q.pop()$
10:    if $|dfa| < |dfa_{min}|$ then
11:        $dfa_{min} \leftarrow dfa$
12:    end if
13:    $statepairs \leftarrow generalizeCandidates(dfa, K)$
14:    for all $(s, s') \in statepairs$ do
15:        $dfa' \leftarrow NFAtoDFA(collapse(dfa, s, s'))$
16:        if $consistent(pt, dfa', td)$ then
17:            $dfa' \leftarrow minimizeDFA(dfa')$
18:            $Q.insert(dfa')$
19:        end if
20:    end for
21:    $n \leftarrow n + 1$
22: end while
23: return $dfa_{min}$
```

states. Then, assuming that the $MaxExpansions$ and K parameters are constant, the complexity of LearnDFA(pt,td) may be bounded by

$$O(||S_+||(2^{||S_+||} + \mathcal{C}(||S_+||, td))).$$

The part $2^{||S_+||}$ is the worst case rarely achieved in practice. The dominant part is the consistency checking, which requires browsing through all the observed positions, generating movements, and performing subset checking operations.

Acceptance of illegal moves

We proceed to demonstrate that, in some situations, DFA's returned by Algorithm 3.2.2 generate illegal moves. To emphasize the real-application significance we restrict the proof of that fact to the specific kind of pieces used in fairy-chess variants.

Based on the definition of a *rider* piece (Section 2.1.1,[35]) we formally define a *limited rider* type of piece. Firstly, let define as a *limited ride* a tuple $\langle a, k, b\rangle \in \Sigma \times \mathbb{N} \times \Sigma$ representing a language

$$\bigcup_{j=1}^{k} a^j \cup \bigcup_{j=1}^{k} a^{j-1}b.$$

Then, the limited rider is a sum of some limited rides $\langle a_1, k_1, b_1\rangle, \ldots, \langle a_m, k_m, b_m\rangle$, such that for any $i, j \leq m$, $a_i \neq a_j$.

A simple example of such a piece is Short Rook, which is an ordinary rook limited up to 4 squares. Using the above notation we can describe Short Rook as $\langle(1, 0, e), 4, (1, 0, p)\rangle, \ldots,$ $\langle(0, -1, e), 4, (0, -1, p)\rangle$. Other examples such as Cloud Eagle, Forfer, or Lion Dog, can be found in various fairy-chess games [236].

Let $\mathcal{Q}$ be a limited rider, consisting of $m > 0$ limited rides. Let $\mathcal{G}$ be a game containing $\mathcal{Q}$, such that for some $i \leq m$, we have that $k_i > 2$ and $a_i^{k_i+1} \notin S_0 \wedge a_i^{k_i} b_i \notin S_0$, i.e. extending ride of one step does not fall off the board. Then let l be the minimal integer such that $a_i^l \in S_0 \vee a_i^{l-1} b_i \in S_0$. Let $\mathcal{D}$ be a training data in the *all legals* scenario satisfying the following conditions:

(a) $a_i^j \in S_+ \wedge a_i^{j-1} b_i \in S_+$ for all $0 < j \leq k_i$,

(b) $a_i^j \in S_? \wedge a_i^{j-1} b_i \in S_?$ for all $k_i < j < l$,

As we have to take into account the constants used by the algorithm, let us assume that $K \geq 1$ and *MaxExpansions*$\cdot (k_i - 2) > 2 + \sum_{j=1}^{m}(k_j - 1)$ (which is true for *MaxExpansions* value of 20 proposed in [12] and all real-life examples). Then we have the following:

Theorem 3.2.1. *If $\mathcal{Q}$ is a limited rider in a game $\mathcal{G}$, and $\mathcal{D}$ a training data described above, then the DFA returned by the Algorithm 3.2.2 generates illegal moves.*

Proof. Let $\mathcal{A}_\mathcal{Q}$ be the DFA constructed by minimizing prefix tree acceptor generated from the set of words S_+ observed in $\mathcal{D}$ for the piece $\mathcal{Q}$ (lines 1–5). As $\mathcal{Q}$ is a limited rider, $\mathcal{A}_\mathcal{Q}$ has exactly $2 + \sum_{j=1}^{m}(k_j - 1)$ states.

The main loop in lines 8–22 has at most *MaxExpansions* repetitions and in each repetition chooses a minimal (in the number of states) automaton from Q. When a pair of adjacent states connected with the letter a_i, and not containing the initial state, is considered in the inner loop (lines 14–20), its collapsing guarantees a reduction of $k_i - 2$ states.

Let $\mathcal{A}'$ be the resulting automaton. To be inserted in Q, $\mathcal{A}'$ has to be consistent (lines 16–19). Assume that it is not true. Then, there is a word $w \in L(\mathcal{A}')$ such that $w \in S_-$. The only words added from previous iteration of the loop have a form of a^j or $a^{j-1}b$. They are illegal for $j > k_i$, however, they belong to $S_?$ by the choice of $\mathcal{D}$. This is a contradiction, so $\mathcal{A}'$ can be inserted into Q.

If $2 + \sum_{j=1}^{m}(k_j - 1) < (k_i - 2) \cdot \mathit{MaxExpansions}$ then, by the pigeonhole principle, this reduction will be chosen as the best in at most *MaxExpansions* turns of the main loop. As, by the construction of $\mathcal{Q}$, this reduction does not contradict collapsing states from other limited rides, it will be preserved through the rest of the loop and returned in the final automaton resulting in acceptance of the illegal move.

Theorem 3.2.2. *Consider* single move known *scenario and a game $\mathcal{G}$ containing a limited rider $\mathcal{Q}$ and another piece $\mathcal{R}$ with unlimited ride, i.e. satisfying $\{a_i^j, a_i^{j-1}b_i\} \subseteq L_{\mathcal{R}} \cup S_0$ for all $j > 0$. Let training data $\mathcal{D}$ be such that all the moves of the form a_i^j and $a_i^{j-1}b_i$ are observed for the piece $\mathcal{R}$. Then, the DFA returned by Algorithm 3.2.2 generates illegal moves.*

Proof. The construction and argument for choice of $\mathcal{A}'$ are the same as in the previous case. The remaining part is to show that $\mathcal{A}'$ is consistent.

As the set U, consisting of all the observed movements (words), contains every a_i^j and $a_i^{j-1}b_i$ for $j < l$ (as there were observed for $\mathcal{R}$), the set defined in line 10 of Algorithm 3.2.1 is empty. This ensures that the procedure never returns *False*, and $\mathcal{A}'$ is consistent.

The practical consequence of Theorem 3.2.2 is that given any limited rider piece occurring next to a similar not limited rider (e.g. Short Rook and Rook, Lion Dog and Queen) the learned rules of these figures can be undistinguishable.

The problem addressed in Theorems 3.2.1 and 3.2.2 lies mainly in the construction of the provided training data, which is hard to detect and handle on the algorithm's side. However, by careful designing of learning and checking procedures, we should be able to guarantee more safety, and more intuitive restrictions on the produced DFA. We will present approaches that try to fix the problem from two sides.

First, we establish dependencies between actually discovered legal moves and hypothesis concerning additional moves from $S_?$. By that, we add additional safegurad and can restrict consistency checking to discard „highly improbable" candidates even if they are consistent with the given data. Secondly, we force consistency checking function to check only those automata we strongly believe they may be good, by using a proper automata learning algorithm, based on more sophisticated, heuristic strategy of merging states.

3.2.3 RPNI Analysis

An alternative solution is to use one of the existing polynomial algorithms for identification DFA's from samples, e.g. Gold [68] or RPNI [142]. Due to its better performance ([54]), we have chosen RPNI as our test algorithm for boardgame move learning.

The pseudocode of the RPNI algorithm, which arguments are the set of positive samples S_+ and the set of negative samples S_-, has been presented in Algorithm 3.2.3. Initially, the prefix tree acceptor $\mathcal{A}$ is constructed, which is a minimal (acyclic) DFA accepting S_+. The algorithm maintains two types of states: *red*, which are confirmed already examined states, and *blue*, which are the non red successors of the red states.

The algorithm searches for a pair of states, which one is blue and the other red, such that after merging these states the automaton does not accept any word from S_-. The procedure of *MergeAndFold* is the deterministic merge of two states p and q, disconnecting the state q and reconnecting all its incoming edges to p, and then folding the subtree rooted in q into the DFA starting in p. The complexity of the procedure is linear on the size of the folded subtree.

Algorithm 3.2.3 RPNI($S_+ \subseteq \Sigma^*, S_- \subseteq \Sigma^*$)

```
1: $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle \leftarrow PTA(S_+)$
2: $blue \leftarrow \{\delta(q_0, a) : a \in \Sigma\}$
3: $red \leftarrow \{q_0\}$
4: while $blue \neq \emptyset$ do
5:     choose $q \in blue$
6:     if $\exists p \in red.\ L(MergeAndFold(\mathcal{A}, p, q)) \cap S_- = \emptyset$ then
7:         $\mathcal{A} \leftarrow MergeAndFold(\mathcal{A}, p, q)$
8:     else
9:         $red \leftarrow red \cup \{q\}$
10:    end if
11:    $blue \leftarrow \{\delta(q, a) : q \in red, a \in \Sigma\} \setminus red$
12: end while
13: return $\mathcal{A}$
```

Given the sets of positive samples S_+ and negative samples S_-, the time complexity of RPNI is $O((||S_+|| + ||S_-||)||S_+||^2)$. However, considering application to Simplified Boardgames and given estimation from (3.1), this complexity is unpractical. For this reason we have to modify consistency checking part of the algorithm from simple iteration through S_- set to some more complex function (e.g. the one used in Björnsson's algorithm). Then we have

Observation 3. *Let S_+ be the language accepted by the piece's prefix tree acceptor pt, and $\mathcal{C}(k, td)$ the complexity of consistency check given training data td and DFA with k states. Then, the complexity of RPNI algorithm is*

$$O((||S_+|| + \mathcal{C}(||S_+||, td))||S_+||^2).$$

The algorithm remains polynomial on the size of the initial prefix tree acceptor, yet again the dominant part depends on construction of the consistency checking procedure. For that reason, in the next section, we propose alternative procedures focused on efficiency.

3.3 Efficient Consistency Checking

Our definition of consistency is that a language has to contain all positive samples (S_+) and reject the negative ones (S_-). In the case of boardgame movements learning, there are many interpretations of what the negative sample is. If we are able to observe all legal moves in any position, every unlisted move fitting within board have to be considered as negative. All the other moves that have not been rejected, can be labeled in any way.

This is the same problem of fitting into unknown target language like in the standard language inference problem, yet here we know the set S_0 that does not matter at all, and we can predict correct and incorrect moves basing on our boardgame related intuition.

3.3.1 Fast Consistency Check

Assume the scenario when our priority is to ensure our algorithm learns only correct moves, i.e. we treat $S_?$ as a subset of S_-. We are looking for the language L such that

$$S_+ \subseteq L \text{ and } L \cap (S_- \cup S_?) = \emptyset, \tag{3.2}$$

and the minimal DFA representing L has the smallest number of states.

For the given prefix tree acceptor T defining S_+ and DFA $\mathcal{A}$ approximating L, the optimal procedure checking consistency of $\mathcal{A}$ is described as Algorithm 3.3.1. Starting with the initial state of $\mathcal{A}$, and the root of T, we traverse through $\mathcal{A}$, simultaneously matching visited states with the states in T. The algorithm returns *False* if there is a mismatch in the state acceptance within the T or there is an accepting state outside T but within the board.

Algorithm 3.3.1 FastCheck($\mathcal{A}=\langle Q,\Sigma,\delta,q_0,F\rangle$, $x\in Q$, $T=\langle Q',\Sigma,\delta',q_0',F'\rangle$, $x'\in Q'$, $w\in\Sigma^*$)

```
1: for all $a \in \Sigma$ do
2:     if $\exists y, y'.\ \delta(x,a) = y \wedge \delta'(x',a) = y'$ then
3:         if $F(y) \neq F'(y')$ then return False end if
4:         if $\neg$FastCheck($\mathcal{A}$, $y$, $T$, $y'$, $wa$) then return False end if
5:     end if
6:     if $\mathcal{O}(wa)$ then continue end if
7:     if $F(\delta(x,a))$ then return False end if
8:     if $\neg$FastCheck($\mathcal{A}$, $\delta(x,a)$, $T$, null, $wa$) then return False end if
9: end for
10: return True
```

Let as notice, that in the case of Simplified Boardgames, complexity of the $\mathcal{O}$ function is additive, i.e. given w_1, w_2 and intermediate result of $\mathcal{O}(w_1)$, the value of $\mathcal{O}(w_1+w_2)$ can be computed in time $O(|w_2|)$. The conclusion is that the check in line 7 can be done in $O(1)$. So the runtime of the algorithm is linear on the $||L(\mathcal{A})\cap(S_+\cup S_?)||$, which is the upper bound for the worst case complexity of checking property (3.2).

3.3.2 Fractional Acceptance Consistency Check

Another reasonable assumptions is that if a DFA representing a language of piece movements is small and does not contradict given data, then with a high probability it is correct. This may not be entirely true when taking into account artificially generated rules, yet in the vast majority fairy chess pieces can be represented by automata with simple construction.

Basing on that, and assuming that we have „big enough" training data, we can easily extend *FastCheck* algorithm to allow some fraction of moves from $S_?$. For our *FractionalCheck*$_\alpha$ algorithm we assume that if it produces language L, then at most $(1-\alpha)|L\setminus S_0|$ generated words belong to $S_?$.

The consequence of this assumption is that if L_p is the language of legal moves and S_+ observed valid moves then, given $\frac{|S_+|}{|L_p|}\geq\alpha$, consistency check will allow optimal DFA despite acceptance of moves from $S_?$. On the other hand, if the sample is small and $\frac{|S_+|}{|L_p|}<\alpha$, the optimal automata will be rejected, as the evidences supporting its correctness are judged as too weak.

The procedure of *FractionalCheck*$_\alpha$ can be described by comparing with *FastCheck* as follows. Instead of returning *False* in case of finding accepted move from $S_?$, the algorithm has to track the number of such moves. If their number exceeds an established threshold, then the function has to return *False*, at best finishing immediately e.g. using the exception mechanism. In the *all legals* scenario, the additional consistency check with S_- is necessary. This requires browsing through all the observed positions and subset checking between observed and DFA-generated moves (which is equivalent to Algorithm 3.2.1, lines 7–8).

3.4 Spine Compaction Algorithm

Both RPNI and Björnsson's LearnDFA algorithms are in fact general purpose methods, i.e. they do not benefit in any way from the fact that they are applied to learning boardgame piece rules. We would like to present our approach that, in contrast, is entirely based on the assumption that it has to learn rules of the chess-like piece. The goal of this algorithm is fast learning of probable pieces movements by performing only specialized state merges and thus minimizing the number of required consistency checks.

The main idea uses the observation that piece PTA's usually have a form of multiple *spines* with similar subtrees attached (see Figure 3.4.1 to examine a rook-like piece example). Actual cases may be more complicated (e.g. spine's cycle period greater then one), yet the idea remains similar, also in the cases such as Checkers *man* piece.

Formally, given a (partial function-based) DFA $\mathcal{A}$ and a word $w \in \Sigma^*$, we define a *w-spine* as a path in $\mathcal{A}$ starting in some state q and going along the longest prefix v of w^* that determines a valid path in $\mathcal{A}$. In such a case w is called the *vertebra* of the spine, $|w|$ its *period*, and $|v|$ its length. Each spine is encoded in the triple $(\textit{vertebra}, \textit{initial state}, \textit{length})$ (see Figure 3.4.2). We are not interested in spines where $|v| < |w|$, and discard them as not proper.

The main procedure (Algorithm 3.4.1) consists of two crucial parts. The first one is responsible for finding all spines in a given prefix tree automata. The other one analyzes the obtained spines in proper order and find the pairs of states being candidates to merge. First we describe both these procedures, and then present detailed description of the outline algorithm.

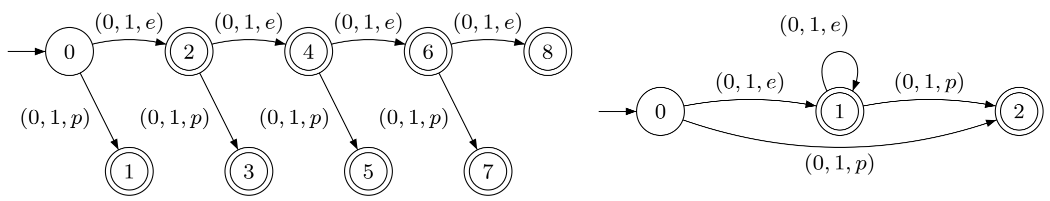


Figure 3.4.1: An example of a chess-like piece sliding only forward. On the left the prefix tree acceptor, on the right the optimal DFA for this piece (assuming board of height 5).

3.4.1 Spines Selection and Pairs Selection

Given a prefix tree acceptor T, the task of the *spine selection* is to find all proper spines with a period k. Starting from the root r and traversing T using depth-first order, reaching depth k provides the first candidate w for a vertebra. However, if $F(r) \neq F(\delta(r, w))$, then we drop the first letter of w and continue searching for the proper vertebra. Otherwise, we have to analyze outgoing edges. If one is labeled by the first letter of w (assume it is a), and $F(\delta(r, a)) \neq F(\delta(r, wa))$ we mark $(w, r, |w|)$ as a proper spine, and start a new search from the current node. If states acceptances match, we can continue extending the current spine as long as the traversed path matches w^*. For any other edge labeled by $b \neq a$, we create w' by dropping the first letter of w and appending b at the end, and proceed further down with w' as a new vertebra candidate.

The procedure works recursively for a given state considered as the actual root, candidate for vertebra, and length of the longest spine matching so far. Example of the outcome of the given procedure is shown in Figure 3.4.2. The time complexity is the same as in the case of the standard DFS, i.e. $\Theta(|Q|)$ given that we traverse only trees.

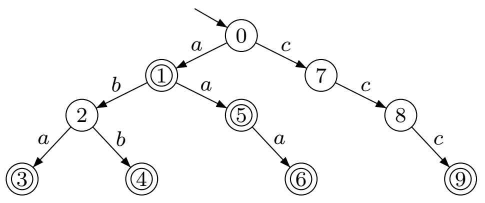


Figure 3.4.2: For the given PTA, the spine selection procedure returns spines as $(vertebra, initial\ state, length)$ triples. Result for $k = 1$: $(a, 1, 2)$, $(c, 0, 2)$. Result for $k = 2$: $(ab, 0, 3)$, $(bb, 1, 2)$, $(aa, 1, 2)$, $(cc, 0, 2)$.

Given a w-spine, the next task is to choose a pair of states (within geodesic distance $|w|$), which will be the best candidate for merging. Our goal is to minimize a chance that the merge result will be rejected by a consistency checking procedure, and simultaneously maximize the state reduction.

Let q and q' be two corresponding states in a spine (i.e. the length of the path from q to q' is a multiply of $|w|$), with outgoing vertebra edges labeled by a. In theory, it is safe to merge q and q', if the subtrees rooted in these states are equal except branches initiated with a. If the property is fulfilled for every two corresponding states in a spine, such merge do not add any new words to the language, except for the words longer then v.

In practice, it is enough to have this property only approximately fulfilled. First of all, it is too costly to check the equality of two subtrees. Instead, we check only the first level equality, i.e. if for every letter $b \neq a$, $(q, b) \in \mathrm{Dom}(\delta) \Leftrightarrow (q', b) \in \mathrm{Dom}(\delta) \wedge F(\delta(q, b)) \Leftrightarrow F(\delta(q', b))$. Moreover, we should be aware that given data might be sparse, and it is less probable that we will have valid data concerning longer moves. Given that, we apply a heuristic that allows corresponding subtrees more distance from the spine root to be smaller.

The overall *pair selection* procedure starts with a spine root r as a candidate for merging with $\delta(r, w)$. Then, we traverse through the spine comparing corresponding subtrees, and replace candidate for merging if we meet a larger corresponding subtree. Final candidate, if spine length remains longer then vertebra, is the base for selected pair. Figure 3.4.3 presents the visualization of the process. The complexity of the procedure is $\Theta(|v| \cdot d)$, assuming d is the maximal degree of node in spine.

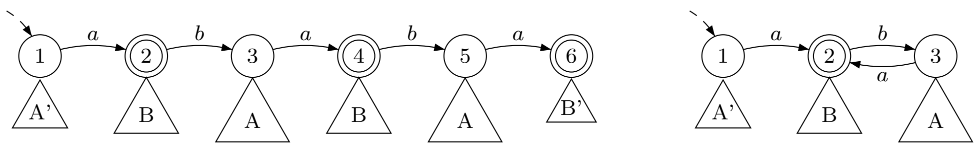


Figure 3.4.3: Example of the spine compaction process. On the left ab-spine rooted in 1, with subtrees such that $A' \subset A$ and $B' \subset B$. On the right the situation after compaction: pair $(1, 3)$ is not a safe candidate for merging, so the states 2 and 4 were selected and merged instead.

3.4.2 The Main Algorithm

The main part of the Spine Compaction procedure is presented as Algorithm 3.4.1. One of its arguments is an integer K, indicating, similarly as in Björnsson's algorithm, the maximal allowed length of the cycle.

After constructing initial prefix tree acceptor, in lines 2–7 we search for all spines not exceeding period K, starting in children of the root. Thus, we explicitly exclude PTA root for being selected as a spine root, to prevent it for being a part of a cycle (which is an assumption supported by our experiments). In lines 8–10, we investigate each spine to select pairs for further merging.

What remains, is to try to merge every pair and check for consistency (lines 13–18). Very important is the order of applying merge operations. Our strategy is to compact spines with shorter vertebra first, starting with the longest spines. The merging function (line 15) is a deterministic merge used in RPNI algorithm. At this moment, we operate on states being the sets of original states, to be able to track merging process. We maintain a set of pairs that resulted in nonconsistent automata (lines 12, 17), which is used to skip unnecessary repeated computations (line 14). Additionally, we prevent pair to create a multiloop, which is a experience-based heuristic, as fairy-chess pieces representations rarely have one. To check this condition it is enough to analyze δ values for states in current pair. Finally, we minimize resultant automaton (line 19).

Observation 4. *Let S_+ be the language accepted by the piece's PTA pt, and $\mathcal{C}(k, td)$ the complexity of consistency check given training data td and DFA with k states. We can estimate upper bound on the number of spines selected in Algorithm 3.4.1 by $O(||S_+||^2)$. Then, the complexity of the SpineCompaction algorithm is*

$$O((||S_+||^2 + \mathcal{C}(||S_+||, td))||S_+||^2).$$

Algorithm 3.4.1 SpineCompaction(Piece *pt*, TrainingData *td*, int K)

```
1: $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle \leftarrow constructPTA(pt, td)$
2: $spines \leftarrow \emptyset$
3: for all $a \in \Sigma$ if $\langle q_0, a\rangle \in \mathrm{Dom}(\delta)$ do
4:     for $k \leftarrow 1$ to $K$ do
5:         $spines \leftarrow spines \cup \mathcal{A}.SelectSpines(k, \delta(q_0, a), \varepsilon, 0)$
6:     end for
7: end for
8: $pairs \leftarrow \emptyset$
9: for all $s \in spines$ do
10:     $pairs \leftarrow pairs \cup \{\mathcal{A}.SelectPair(s)\}$
11: end for
12: $forbidden \leftarrow \emptyset$
13: for all $p \in pairs.OrderByPriority()$ do
14:     if $p \in forbidden \vee CreateMultiloop$ then continue end if
15:     $\mathcal{A}' \leftarrow MergeAndFold(\mathcal{A}, p)$
16:     if $consistent(pt, \mathcal{A}', td)$ then $\mathcal{A} \leftarrow \mathcal{A}'$
17:     else $forbidden \leftarrow forbidden \cup \{p\}$ end if
18: end for
19: return $minimize(\mathcal{A})$
```

3.5 Experiments and Empirical Evaluation

We have performed experiments to compare all the three presented learning algorithms paired with different consistency check functions in both the *single move known* and *all legals* scenarios.

For the *all legals* scenario we have prepared 20 data sets per game, each containing record of 50 plays generated using two random agents playing against each other. In the case of the *single move known* scenario, the number of play records in each data set was increased to 1000. The maximum game length was set to 80 move per player in all cases. Due to the symmetry of all games, we have performed learning only for the white player (results presented in [12, 69] show no significant difference for white and black in the case of such games). If not stated otherwise, the *MaxExpansions* parameter of Björnsson's algorithm was set to 20, and K parameter used in Björnsson's and Spine Compaction algorithms was set to 2. All experiments were run on 2.60GHz Intel Core i7-6700HQ processor.

3.5.1 Games

We have used as a testbed 8 fairy-chess games, containing 41 pieces altogether. Chosen games can be divided into three categories. The first one consisting simplified versions of known boardgames or their variants contains *Chess*, *Los Alamos* (small chess variants without bishops), *Tamerlane* (large 10×11 game containing many non-standard pieces like giraffe, camel, picket), and a breakthrough variant of *Checkers*. See [1, 12] for more detailed game descriptions.

Other three games are the result of procedural content generation algorithms. One game is presented in Section 4.2 and other were generated using RAPP-based evolver (Section 4.3). Thus, they all contain only non-standard pieces.

The remaining game was designed especially to fool unaware learning algorithms. It contains two special pieces that can move horizontally like rook without capturing moves, yet with limited movement lengths. The initial position is constructed such that every move exceeding these lengths is impossible due to the obstacles. Thus, the setup fulfill prerequisites given in the definition of the limited rider in Section 3.2.2, and there is no evidence of nonconsistency when assuming the $(1, 0, e)^* + (-1, 0, e)^*$ language for these pieces. With this assumption the fractions of uncertain (i.e. $\in S_?$) moves for these pieces are 0.1 and 0.6 respectively. The game rules are presented in Appendix C.

3.5.2 Results

Tables 3.5.1 and 3.5.2 show the results for the *all legals* and *single move known* scenarios (Spine Compaction algorithm is abbreviated as SC). For every *(learning algorithm, consistency check)* pair, we have computed the percent of generated automata with optimal size (which is the task of regular language inference), and the percent of automata generating consistency *errors*. By an *error*, in this context, we mean a non-empty intersection with S_- of the piece's true language. These data visualize the trade off between the rapid language expansion followed by the DFA size reduction and performing only minor adjustments to the initial prefix tree acceptor. Runtimes of all conducted experiments are presented in Table 3.5.3.

All legals scenario

The experiment using the *all legals* scenario shows very similar performance of all learning algorithms in determining the correct size of the DFA. However, the differences, when comparing the accuracy of learned automata, are visible. Best results are obtained by the Björnsson's algorithm. The reason behind the number of errors generated by RPNI lies in its merging strategy. As the states close to the root are considered first, it makes algorithm more susceptible to training data increasing chance of accepting „non-intuitive" automata (example of such behavior is illustrated in Figure 3.5.1).

Table 3.5.1: Experiment results for the *all legals* scenario.

Consistency	Correct size (%)			Errors (%)		
Check	Bjö.	RPNI	SC.	Bjö.	RPNI	SC.
Björnsson	87.6	91.5	87.3	4.9	9.1	7.6
Fractional$_{0.5}$	87.6	89.9	84.9	4.9	6.3	7.6
Fractional$_{0.6}$	89.0	89.8	86.6	2.4	4.0	5.0
Fractional$_{0.7}$	87.1	87.1	87.1	2.4	3.5	3.2
Fractional$_{0.8}$	85.6	85.5	85.6	2.4	3.7	3.2
Fractional$_{0.9}$	73.7	73.5	73.7	2.4	2.7	3.9
Fast	69.3	69.3	69.3	0	0	0

Table 3.5.2: Experiment results for the *single move known* scenario.

Consistency	Correct size (%)			Errors (%)		
Check	Bjö.	RPNI	SC.	Bjö.	RPNI	SC.
Björnsson	73.7	70.6	73.7	5.4	19.8	9.6
Fractional$_{0.6}$	88.7	61.5	87.3	6.8	40.4	6.1
Fractional$_{0.7}$	89.9	65.7	89.8	4.9	37.6	2.8
Fractional$_{0.8}$	88.0	64.6	88.0	4.5	34.0	4.4
Fractional$_{0.9}$	73.3	71.5	73.3	3.7	14.4	3.4
Fast	68.5	68.5	68.5	0	0	0

Table 3.5.3: Learning times (in seconds).

Consistency	*all legals*			*single move known*		
Check	Bjö.	RPNI	SC.	Bjö.	RPNI	SC.
Björnsson	68.2	16.4	4.1	1598.6	454.1	82.4
Fractional$_{0.5}$	51.8	9.4	2.2			
Fractional$_{0.6}$	50.8	9.3	2.1	5.4	4.4	0.2
Fractional$_{0.7}$	50.8	9.2	2.0	5.6	3.3	0.2
Fractional$_{0.8}$	49.4	9.5	1.8	5.7	3.0	0.2
Fractional$_{0.9}$	42.8	10.4	1.5	5.7	2.6	0.2
Fast	4.0	4.6	0.4	4.6	5.3	0.4

When it comes to the consistency check functions comparison, Björnsson's algorithm is less restrictive than the others, which is reflected by the increased number of errors, but not necessarily in the reduction of the DFA size. As expected, increasing α results in increasing sizes of the output DFA and decreasing the number of false assumptions that translates into errors. For the given data set, the optimal α seems to be in the range $[0.7, 0.8]$, depending on the learning algorithm chosen.

Learning times clearly show the difference in performance of algorithms. *SpineCompaction* is usually about 4 times faster than RPNI, and 10–28 times faster than Björnsson's algorithm. The usage of *FastCheck* results in the fastest learning time. The performances of *FractionalCheck* are similar regardless of α value, although greater α usually implies lesser learning time. Due to the obligatory iterations through the entire training data, the performance of Björnsson's consistency check is the lowest.

Finally, we have to comment the K parameter occurring in both *SpineCompaction* and Björnsson's algorithms, which is a bound of the length of cycle the algorithm can make. Additional experiments performed for $K \in \{3, 4\}$ show no difference in the quality of obtained results, only in performance time. In the case of Björnsson's algorithm increasing K by one results in the increase of the learning time by about 15–20%. Surprisingly, increasing K in SC algorithm, have a marginal effect on performance (0.1 sec. even in the case of Björnsson's consistency check), which leads to the conclusion that the cost increase is constant rather than linear.

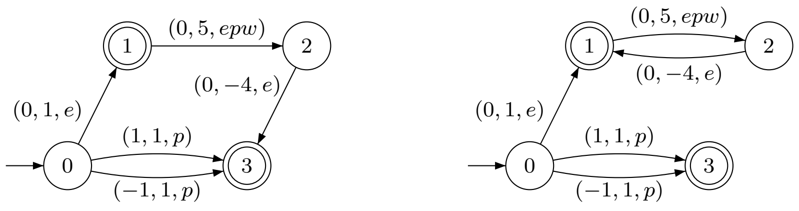


Figure 3.5.1: On the left the valid DFA representing Chess pawn, on the right the DFA learned by RPNI: invalid, yet fully consistent with any possible data.

Single move known scenario

In this case, the poor quality of automata learned by the RPNI algorithm is even more visible. The size of DFA returned by Björnsson's algorithm and *SpineCompaction* is nearly similar, however in all cases except Björnsson's consistency check SC returns more accurate automata. Clearly, SC is also the fastest learning algorithm regardless the consistency check used.

In the case of *single move known* scenario, the usage of the *FastCheck* algorithm seems totally ineffective. The data obtained is too sparse to produce small DFA's and necessity of traversing unreduced trees reflects on the overall performance. Björnsson's consistency check is again the most ineffective in the sense of learning time.

Results obtained by different consistency checks mostly depends on the accompanying learning function. For Björnsson's learning best α can be taken from the range $[0.8, 0.9]$. The overall best result is obtained by using *SpineCompaction* combined with *FractionalCheck* for $\alpha = 0.7$. In this case the heuristic strategy of merging states shows its advantage, allowing reduction of α.

3.6 Conclusions

In this chapter we continue the study of learning game rules started by Björnsson for the class of Simplified Boardgames. The problem is also considered from the point of view of regular language inference. We compare existing algorithms developed independently for these tasks, and provide our own contribution consisting of parameterizable consistency check function and a specialized heuristic DFA learning algorithm.

The experiments show that our *SpineCompaction* learning algorithm provides most accurate results in the *single move known* scenario of learning, and comparable results in the GGP-like *all legals* scenario. Moreover, it requires much less time for learning than the other tested algorithms. Experiments show that in practice gathered observations data are incomplete, which justify the need for a learning function based on some approximation method. For this reason, we have also proposed consistency check functions, determining the acceptance of the proposed DFA's on the basis of the fraction of unsafe moves. These are *FractionalCheck*$_\alpha$ and *FastCheck* (which is a special case of *FractionalCheck* with $\alpha = 1$). As Tables 3.5.1 and 3.5.2 show, we can select an α value giving us better results then Björnsson's consistency check, which has been our reference point.

Nevertheless, all the tested methods are efficient in comparison to game reasoners used in GGP. This supports the thesis that the problem of General Game Playing should be solved by detecting subclasses with more effective, domain based algorithms applicable [96]. As the boardgames are one of the most common classes of games used in GGP, the effort in this direction is relevant for the domain, and has possible practical applications in improvement of GGP systems [12].

There are a few directions of the future work. One, as already mentioned in [12], is to extend Simplified Boardgames class, mainly by adding side-effects of moves and more complex terminal conditions. As we have approached the task of learning by observing by using positive and negative samples only, the expected next step is to compare it against scenario allowing queries and tutors (as in Angluin algorithm [4]). Another interesting question that arose in our studies, is how big fraction of legal moves a player has to know to be able to play competitively. Considering the *single move known* scenario and a skilled opponent, it is a rational assumption that most of the time he/she will use only a subset of legal moves. It may be so, that the remaining moves are not so important and, when our goal is to learn quickly how to play well enough, we can safely omit them.

Finally, we have to point out that the data containing the complete move rules is in analogy with learning from a child's play, where each piece is moved carefully step by step pointing out every translation. Normally, a player moves its piece directly to the final square, so we do not know the whole word encoding the move. Removing this limitation is another important problem to solve.

4
Generation

The task of Procedural Content Generation (PCG) is to create digital content using the algorithmic methods. When applied to games, PCG is used for creating various elements, e.g. maps, textures, music, or enemies. From the General Game Playing point of view a case of the special interest is generation of the game rules. For a given general game description language we wish to generate valid playable games that will additionally hold some gameplay properties, making them appealing for a human player.

In this chapter, we focus on the generation of the complete games belonging to the Simplified Boardgames class (Section 2.1.2). This is one of the few approaches of the procedural generation of complete rules of the games, and it concentrates on the problem of rules evaluation. The question how to assess the quality of the game is hardly non-trivial, and it can be answered in numerous ways. As our motivation is to create games for the GGP competitions, so they can be played by computer programs, our generation method puts the emphasis on the strategic properties of the generated rules.

The next section provides a brief introduction into the broad domain of PCG, with the focus on describing systems to generate complete games. In Section 4.2, we present initial study to evolve fair, human-playable chess-like games, using game features extracted via simulations. Alternative approach, based on the indirect observation of the game behavior by using results of plays between various algorithms, is presented in Section 4.3

The content of this chapter is partially based on the papers [100, 101].

4.1 Procedural Content Generation in Games

The domain most benefiting from applying Procedural Content Generation is undeniably video games. PCG can be used for generating every single part of the game, from textures and items, through levels and music, to AI opponents [190, 223].

The main reason to generate game content is to reduce costs. Human designers and artists are expensive and slow, and many game designing tasks do not require top-level creativity. Modern video games usually take place in very big virtual areas that are nearly impossible to fill manually. This is especially true for the sandbox „AAA" games, where the emphasis is on the quantity, and the human-created and procedurally generated content can be easily mixed. When player attention is mainly focused on some carefully prepared plot tasks, the surroundings play only a minor role, and their main goal is to make a player fill the world is not empty and lifeless except his activity. PCG tools do not have to generate the exact final content, they can be also used by developers for helping them e.g. by creating initial content that is then revised and modified to fulfill the current needs.

Combining PCG with player modeling makes the games more player-adaptive and thus increase the players enjoyment. Such techniques are used to generate game content in real-time in response to estimated player state, and e.g. increase or decrease the number of enemies approaching.

Another important aspect is to use PCG for extending game's lifetime. Good map or mission generators can entertain players long after finishing the game's main campaign, as they constantly provide new challenges and make the game potentially endless.

Historically, in early eighties, the usage of Procedural Content Generation was forced by the limited capabilities of the home computers. The most notable example is the *Elite* game [18] that allows the player to visit eight galaxies, each containing 256 unique planets, that are generated based on the stored random generator seeds. Another classical 1980 game *Rogue* [229] is a dungeon-crawling game where new levels are randomly generated every time the new game starts.

Procedural CG techniques are widely used in independent (so called *indie*) games, when developers do not have time and resources for hand-crafting every component. One of the recent popular indie games extensively using PCG to generate an entire world including its content is *Minecraft* [134]. Another good example is the complex world generator used in *Dwarf Fortress* [3]. *Spelunky* [238], is a very successful 2D rogue-like platform game that uses PCG for making level variations. An example of adaptive PCG is the indie game *Galactic Arms Race* [73], where players can use weapons evolved for them based on their preferences and game history.

Let us also mention some notable examples of using procedural content generation techniques in modern commercial „AAA" games. Hack and slash role playing game *Diablo* [140] uses PCG not only for designing levels, but also for generation of number, type, and even names of monsters and items. This made the game very popular in multiplayer mode, where teams, mostly created by strangers, could play the same levels multiple times, each time in different circumstances. Epic turn-based strategy game *Civilization IV* [53] generate random earth-like worlds with respect to the latitude-climate relationship influencing placement of natural resources. First person shooter *Left 4 Dead* [28] use adaptive PCG to dynamically adjust game pace based on the player's *emotional intensity.* Current game difficulty is tuned to keep the player maximally engaged [15]. The best example of the state-of-the-art outer space generation is, released in 2016, *Elite*-inspired strategy game *Stellaris* [197], which generates enormous procedural galaxies containing thousands of planets.

4.1.1 PCG Techniques

As we have seen in previous examples, the most popular usage of PCG is to generate maps of any kind. Various types of maps requires various techniques, however, most of them are based on the evolutionary algorithms.

The dungeon-like maps (levels) are widely used in RPG games, action adventure games and platform games. Usually the valid levels have to fulfill some basic properties, like admissible goal square. The placement of all game elements influencing the gameplay (monsters, items, gold, NPCs) is theoretically unrestricted, yet it has to be well balanced for „good" levels, e.g. levels with too many opponents right from the start will be, in the most cases, unplayable.

Space partition is a technique of subdivision of 2D or 3D space into disjoint subsets so that any point is contained by one of this subsets. It is simple and popular method for creating the layout of the dungeons, especially in *binary space partitioning* version where space is recursively divided into two subsets. One of the alternatives is agent-based dungeon growing where a semi-random agent (or group of agents) create maze by digging tunnels and creating rooms. Also cellular automata can be used for that purpose. Applying a cellular automaton, which contains rules for creating filled cell given its neighbors, generates a surprisingly lifelike cave-rooms as shown in [81]. Lastly, it is worth to mention the usage of grammatical evolution, i.e. modeling dungeon creation process as a set of recursive rules of a generative grammar [36, 113]. Some high-level strategies, independent on PCG technique actually used, can be applied to improve quality of created maps. One of such a methods consists of simulating human designing sketch process, starting with a low quality prototype and iteratively refine levels by gradually improving its resolution and fidelity [112].

Other types of maps can be generated by various methods as they have to fulfill different criteria. Strategy games usually contain maps where landscape serve an important tactical role, and available resources and player positions have to be placed fairly. Multiobjective approach combining evaluation functions for base placement, resource placement, available paths, and choke points to generate *StarCraft* maps, has been presented in [226]. In a case of civilization-like world maps the data used to generate them can be based on real world's data, so the obtained maps correspond to existing landscapes and resource rich areas [6]. In [189], grammatical evolution approach has been used to produce platform game levels for *Infinite Mario Bros.* The method of generating racing tracks based on evolution of sequences of Bezier curves and modeling of player driving style has been described in [224]. An online tool for generation of tracks for two open-source 3D car racing games (*The Open-source Racing Car Simulator* [237] and *Speed Dreams*), combining procedural content generation with interactive evolution, has been recently presented in [24].

Another domain where PCG techniques have been successfully used is quest and story generation. Missions can be represented as graphs where each node have a specified type, e.g. entrance, task, goal, lock, key. Relations between these nodes corresponds to plot elements and their precondition-postcondition structure, i.e. a player needs a key to open the doors, but he can also bribe a guard to achieve the same result. Thus, generation of level structure can be seen not as a standalone task, but rather as making a geometric layout to complement the generated story [36].

Alternatively, one can perceive generating stories as a planning problem. Originally such approach named *procedural story generation*, arisen in 1970s, was focused on generating short text-base stories, not the game plots. One of the first story generation system *Tale Spin* [125] generate stories by simulating character behavior and then recount events that happened. Planning systems like STRIPS [42] or ADL [146] can be used to search through the space of possible plans/plots. If one wants to generate worlds and stories together, the next step is to translate given story into the game space. An approach using so called *space trees* has been proposed in [72]. Given that each plan step has assigned location, a space tree is an abstract representation of the game world, indicating game locations and their adjacency. After a proper space tree, consistent with the plan and fulfilling other evaluation criteria, has been found, it can be translated into e.g. 2D grid with the proper connections between each zone.

Lastly, we want to briefly point out how to create other generatable elements of the games. In many games there is a need for a high number of textures representing game objects sharing the same property. A good example are sprites of spaceships in 2D space shooter games [111].

Presented method describes evolving visually consistent spaceships using a genotype consisting of turtle commands (so called *L-systems*, which are a particular form of formal grammars), transformed into a spaceship image composed of human-authored sprite components.

Alternative approach to generate pictures is to use *compositional pattern producing networks* (CPPNs). CPPNs are a variation of *artificial neural networks*, but they can use other functions than Gaussian or sigmoids to activate neurons [196]. The method has been used to generate flowers in the Facebook game *Petalz* [163], in which players can share flowers they breed themselves with other players through a global marketplace. Purchased flowers can be used for breeding new flowers, which then can be sold or send as a gift. Consequent application of embedded evolution mechanism results in creation of a whole new lineage of aesthetically pleasing flowers. CPPN-based mechanism is also a core for evolving weapon projectiles in already mentioned *Galactic Arms Race* [73]. Instead of creating static images, this time CPPNs determine the behavior of each particle over time, as axis velocities are one of the network outputs.

Other useful generatable game content is vegetation. Many games use procedural vegetation generation and some software is available to support game developers, e.g. used in dozen of „AAA" games *SpeedTree* [78]. 2D plants evolution using L-systems has been presented in [141]. Nature-inspired fitness function using phototropism (height), bilateral symmetry, light gathering ability etc., makes possible to control the type of generated plants.

Music is an integral and very important part of nearly every video game. The music composer in such an interactive environment needs to create music that is dynamic and non-repetitive. Also, using music, game creators can influence player's emotions in indirect way separately on what is going on the screen, e.g. sudden change for gloomy sounds may cause player feeling insecure even on shiny open grassland (so called *foreshadowing*). Making music dynamically reflect current game state, as well as influencing player towards desired emotional state, is a field of study for music procedural generation in games.

An experiment investigating the relationship between procedurally generated mood-expressing music and player's experience on true and false foreshadowing trails has been described in [186]. *MetaCompose*, described in [187], is a music composition framework that generates harmonious, pleasant and interesting compositions using multi-objective optimization and feasible/infeasible 2-population genetic algorithm [89]. The method presented has been validated empirically, showing that all the components embedded are necessary, as they improve the quality of generated music.

4.1.2 Generation of Complete Games

The most challenging task for the Procedural Content Generation is to generate a complete game [137, 225, 241]. The core of that task concerns game rules, which will specify the environment, the player roles, and the plot. So far, several such attempts have been made, creating games belonging to some restricted classes of possible rules.

Embedding PCG into General Game Playing begins with the Pell's *Metagame* [147] described in details in Section 2.1.1. Although game generation mechanism is an inseparable part of the *Metagame*, generation is clearly not the main task of the system. It is based on the probability distribution tables, which are hand-crafted and provided for each point of the grammar that is a nondeterministic point of choice. This gives a limited control over the type of game we wish to be generated. Additional hardcoded rules are embedded to ensure some unwanted situations do not occur (e.g. instant win in the initial position) [149].

Some properties, like game balance, are hoped to achieve by game symmetry restriction itself. Others, like search complexity, decision complexity, or rule complexity, can be indirectly controlled by the various parameters. And some, like winnability (which is shown to be an NPcomplete problem), are simply not tested. Nevertheless it is up to the human to decide for every generated game is it good or not. An example of the generated Symmetric Chess-like game has been presented in [148].

The next step, taken by Browne in his *Ludi* system [19, 20], is to evaluate games and use the obtained information to improve them. The main question is: how to distinguish the good games from the poor quality games. The approach taken by *Ludi* is based on simulating the gameplay using a set of *policies*, which are game position evaluation functions. Policy is a linear combination of *advisors* that represent a single aspect of current position, e.g. material, mobility, number of possible captures, proximity to enemy, etc. For a given game, the policy optimization search is performed using self-play and evolutionary strategies.

The *Ludi* language is very rich and supports large number of boardgame-related concepts. Boards can have various tile types (e.g. triangular and hexagonal) and shapes. Pieces can contain additional information like state and flags. Multiple player orderings, endgame condition types and initial positions are available. This makes the language functional rather then declarative, and in contrast to GDL *Ludi* is a high level language where game designer combine large (yet limited) number of predefined functions rather then write everything from scratch

Rules are evolved using genetic programming, with incremental checking of expected game properties (well formed, not too slow, not drawish) before the final evaluation and advancement to the next population. The game evaluation function is mainly based on the self-play, with the assumption that good games should produce non trivial and interesting matches from the players' point of view. The main idea is similar to that presented in [76], where after a number of plays their course and result are analyzed to deduce balance. However, *Ludi* use a much broader range of 57 aesthetic measures. Apart from expected criteria like branching factor, balance, and drawishness, lot of important non-trivial criteria were mathematically formulated and measured, e.g. search penetration (average shape of the search tree), momentum (tendency for the leading player to extend his lead with subsequent moves) or drama (tendency to recover from seemingly bad positions).

Actually, *Ludi* is one of the most successful stories of Procedural Content Generation. Reported results of one week experiments describes obtaining 19 „playable" games (from overall 1309 evolved during that time), including two with exceptional quality. The most famous one, *Yavalath*, is hexagonal board 4-in-a-row to win, with additional rule that making 3-in-a-row is loss. *Yavalath* and the other game *Ndengrod* (renamed to *Pentalah*) were the first commercially published computer-invented games, and *Yavalath* has been ranked in the top #100 of abstract games ever invented[1].

Of course attempts of game generation do not end on boardgames, and some results concerning different genres have been published. Card Game Description Language (CGDL), using a context-free grammar to represent various card-games including poker variant Texas hold 'em, Blackjack and UNO, has been introduced in [50]. The syntax of CGDL allows to use grammar-guided genetic programming to generate new card games. Games are evaluated using simulation-based fitness function and MCTS agents. The results of the experiments, including evolved *The Ant and the Grasshopper* game, has been presented in [51]. (Detailed language description can be found in Section 2.4.3, and our translation from CGDL to GDL-II is presented in Section 6.1.)

In a series of experiments using Strategy Game Description Language (Section 2.4.1) elements of strategy games were evolved using genetic programming. An experiment concerning generation of balanced sets of units, where the main unit's attribute was a number of damage points it can inflict to every other unit, has been presented in [119]. Then the game rules were evaluated using simulation based approach and fitness function partially inspired by Browne [20]. The following

[1]BoardGameGeek database, June 2016 (`http://www.boardgamegeek.com`)

research focuses on generating balanced card sets for the commercial deck-building card game *Dominion* [120].

In [227], the authors introduce a psychology-based way of fun measurement based on the assumption that game rules should support learning as a way to improve player results [93]. The games from the domain of simple two-dimensional Pac-Man-like games are evolved using simple hill-climbing approach. The fitness function measures the *learnability* of a game using agents learned by the (5+5) Evolution Strategy algorithm. Games that are too hard or do not require any skill are classify low, while that which can be learned quickly are classify high.

Arcade-style video games become a domain of special interest. An initial work towards game rules generation for General Video Game Playing (Section 2.3) has been presented in [138] (see Section 4.3.1 for detailed method description). Recently, a level generation track of GVGAI competition has been opened. Competition details and simulation-based feasible/infeasible 2-population approach has been described in [88].

The ANGELINA is an ongoing project generating complete arcade-style 2D games including rules, levels, and game characteristics [27]. It uses a form of cooperative coevolution to evolve complete games. Each design element (e.g. ruleset, map, agent placement) is evaluated partly in isolation and then combined into full game and evaluated through the automated playouts.

4.2 Evolving Chess-like Games Using Game Features

In this section, we present initial research towards procedural generation of Simplified Boardgames. To generate playable, human readable, and balanced chess-like games, we use an adaptive evolutionary algorithm with the fitness function based on simulated playouts using „smart” (min-max based) and „stupid” (random) algorithms.

The information retrieved from the playouts is translated into nine features, which capture an approximation of a game behavior in a manner similar to presented in *Ludi* system [20]. From the feature values we reconstruct what properties the game likely has, and judge their usefulness to further guide the evolution process. Using the presented method we were able to obtain some game rules that are non-trivial and enjoyable for humans players, while being quite easy to remember.

This is also a step towards establishing Simplified Boardgames as a comparison class for General Game Playing agents (see Section 6.2 for further work on that topic).

4.2.1 The Evolutionary System

We decided to focus our research on chess-like games to simplify the representation and increase the chance of well balanced instances. By chess-like, in this case, we mean: initial position close to symmetrical with the line of weak pieces on the front of each army, fairy chess-like movements, and winning achieved by moving a piece to the opponent's backrank or taking all pieces of a given type.

The chromosome structure consists of three separate representations: for board, piece rules, and winning conditions. The board is a rectangular table containing in each square information about the piece's type and owner, or an empty square. Pieces are represented by a map from pieces symbols to its move rules. Winning conditions are encoded as a list of triples containing piece type, piece side, and the type of winning condition (capture or movement).

Generating initial population

At the beginning of the process, we generate building blocks for pieces movements. Based on some probability matrices we generate short words from Σ^* and optionally assign them modifiers that extend a single word to a set of words. A modifier can mirror the move to be horizontally or vertically symmetrical, rotate the move, or add the Kleene star to any letter of the word.

We have divided types of pieces into three classes varying on mobility (inspired by the Chess we have weak, light, and strong pieces). Using the building blocks we randomly build up sets of possible moves for every class. These sets are generated once for every evolutionary run.

The initial board position is randomized by a number of parameters. At first, a quarter of the board is generated with the probability of appearance of strong piece growing for the rows more distant from the opponent. Then, the generated quarter is mirrored (with some probability of change) to represent all pieces of the white player. After that, the white pieces are mirrored to the black ones, again with some probability of randomly regenerating a piece (with the constraint that it must be from the same class).

Generating rules for pieces is relatively simple. For every piece type (e.g. pawn, rook, griffon) we just randomly choose its moves from the precomputed set for the corresponding class. Winning conditions are obtained by drawing piece type and capture/movement condition type. Possibilities of multiple winning conditions and asymmetry are provided by additional parameters.

Fitness function

Before presenting the details of our evolutionary operators, we have to describe the method of evaluating each individual. As the main goal of evolution is very complex, and it cannot be easily expressed with a single function (which is a common problem with content generation for games), we use simulations to interpolate the features of the created game. The question is what are the features we want to acquire. Obviously, we want the game to be playable and as balanced as possible. Because generated games are chess-like, a large number of draws between players of comparable strength is permissible; however as we want the game to be strategic, we expect that better players should not have difficulty beating weaker ones (such games should also be much shorter). We also expect the games to be understandable and playable by humans, which requires that the rules are not too complex. Since we want them also to be playable by computers, the branching factor should also be restricted.

We apply two types of simulations (playouts): between two min-max algorithms, and between a min-max player against a random player (symmetrical for both roles). A min-max agent used for the comparison is the RBgPlayer, described in details in Section 6.2. We perform a fixed number of such simulations. From the results, the following game features are calculated:

- $P \in \{0, 1\}$ – *Playability*, which is 0 if the game is found unplayable, i.e. some simulation did not finish within the given timelimit, and 1 otherwise.
- $L_M \in [0, 1]$ – *Min-max playouts length*, which for ℓ_{mm} being the mean length of playouts between two min-max players divided by the game turnlimit is equal to $2\ell_{mm}$ for $\ell_{mm} < \frac{1}{2}$, and 1 otherwise.
- $L_D \in [0, 1]$ – *Playouts length difference* is calculated as follows. Let $z = \ell_{mm} - \ell_{rm}$, where ℓ_{rm} is the mean length of playouts between the min-max player and the random player, divided by the game turnlimit. The value of the feature is 1 if $z > \frac{1}{4}$, 0 if $z < 0$ and $4z$ otherwise.

- $W \in [-1, 1]$ – *Width* is a normal distribution probability density rescaled to peak at 1, i.e.

$$W = e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{4.1}$$

  where μ is equal to the given expected branching factor of the game, σ is equal to $\frac{\mu}{4}$, and x is the mean branching factor obtained from every state visited during all simulations.

- $B_M \in [-1, 1]$ – *Min-max balance* is the balance between both players in min-max playouts calculated as *white_wins* − *black_wins* divided by the number of playouts.

- $B_R \in [-1, 1]$ – *Random balance* is the balance in the case of min-max vs. random games, i.e. *white_vs_rand_wins* − *black_vs_rand_wins* divided by the number of min-max vs. random playouts.

- $S \in [0, 1]$ – *Strategy* describes chances of winning against the weak opponent, and is calculated as the number of wins of min-max vs. random player divided by the number of such playouts.

- $C \in [0, 1]$ – *Complexity* is calculated based on the size of the games rules compared to some desired size s (which is based on the size of Chess pieces descriptions and depends on the number and class of pieces). The feature is equal to the absolute value of (4.1), where μ is the desired game size, σ is equal to $\frac{\mu}{4}$, and x is the actual game size.

- $R \in [0, 1)$ – *Reducibility* means that not all pieces are really important for the game. Assume that there are k pieces, and u_i is the *usefulness* measure of i-th piece, calculated as the number of moves performed with this piece by the min-max players, divided by the number of all moves made by the min-max players. Then, this feature is equal to $\sum_{i=1}^{k}(\frac{1}{k} - u_i)^2$

Based on these features, the game's fitness is calculated using the following formula:

$$\frac{P}{15}\left(L_M + L_D + 2|W| + 3S + 2C + (6 - 3|B_M| - |B_R| - 2R)\right).$$

Evolving

We start with the initial population of size n. Every individual is evaluated, and if it is found unplayable it is removed from the population. The remaining games are ordered by $B_M + B_R$ measure. Parents are chosen by pairing games from the ends to the center, i.e. the first with the last, the second with the last but one, etc. Crossover creates two offsprings in the following way. We use two point crossover on a board columns, and a uniform crossover on pieces rules and winning conditions (the conditions may have different length for both parents).

We also mutate every individual from the population in an adaptive GA fashion. Let s be the sum of the board's dimensions. We mutate pieces rules and winning conditions with probability $\frac{1}{s}$, and the initial position otherwise. The initial position is mutated by randomly choosing a square and its new content (empty or a piece of random type). If the square is on the lower half of the board, the piece is white, otherwise it is black. In the case of mutating piece rules we have three options depending on R, C, and some constant weight. We can draw new rules for a piece with the lowest usability, or choose it using roulette wheel based on the complexity of the piece's rules. The last option is to swap the rules of two pieces of the same class. Mutation of terminal conditions is guided by balance values. With some constant weight, both sides have added or removed a winning condition. With a weight dependent on B_M and B_R, only one side's conditions are modified, to possibly improve game's balance.

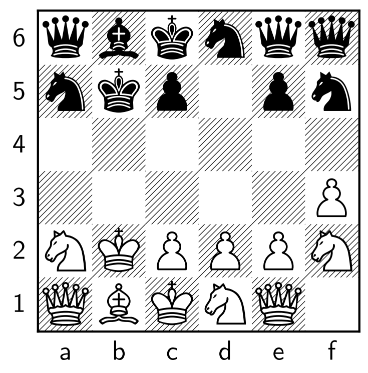
The mutants and offsprings are evaluated and removed if unplayable. For the next generation we take best n individuals from the joint set of parents, offsprings, and mutants. If there is only $k < n$ playable games, $n - k$ best individuals are duplicated.

4.2.2 Experiment Results

We performed several evolutionary runs with various board sizes (from 6×6 to 8×8), numbers of pieces per class, population sizes, and numbers of playouts. In the case with 6×6 board and population of size 50, we were able to evolve an entire population with the fitness above 0.98. With a smaller population and larger boards, the evolution usually stuck at a local maximum around 0.85–0.88. The two features that we found the most difficult to optimize were complexity and reducibility.

We present one of the evolved games, appeared in the 22-nd round of evolution, with fitness 0.9899. We have found this game interesting because of non-trivial asymmetry, simple and interesting pieces, and good reducibility. The game was also rated as fully balanced, with all playouts lost by the random player, and 20% wins for both white and black sides in the case of min-max vs. min-max playouts. Full description of the game is presented in Appendix B.

The game uses 6×6 board with two weak pieces (♙♔), two light pieces (♘♝), and one strong piece (♕). The turnlimit is 60 and there are no other winning conditions specified, which means that every player has to capture all enemy pieces. (We have found this type of victory surprisingly common among best-fitted individuals.) The initial position looks as follows:



The pieces with the biggest usefulness (0.28) are ♘ and ♕, while the least useful pieces are ♔ (0.12) and ♝ (0.13). The pieces rules are as follows:

♙ Moves and captures 1 cell diagonally ahead.

♔ Moves and captures 1 or 2 squares straight ahead, jumping the intervening square in the case of moving 2 squares.

♘ Moves and captures 1 cell diagonally, and it can jump orthogonally forward by 2 squares if the destination is empty.

♝ Moves like the orthodox bishop, and it can move and capture 1 square straight ahead.

♕ Slides any number orthogonally if all pieces on the way are the opponent's, and it can capture and self-capture 1 case diagonally ahead, 1 square straight-ahead, and 1 square straight-back.

4.2.3 Conclusions

This section presents an evolutionary system that automatically generates Simplified Boardgames. Generated games are evaluated using simulations with min-max and random reference players. Games are rated using several features measuring balance, strategy influence, pieces usefulness, game tree size, and rules complexity. An example of the evolved, balanced, chess-like, and human readable game is provided.

We plan to further improve the system and make it more general. In particular, it should be able to generate fully asymmetric non chess-like games. The issue of much importance is the fitness function, and in future versions we want it to be based on *learnablity* [227], defined as a potential of improving the level of play related to the number of games played. By adding to the evolver an algorithm rewriting boardgames into GDL, we are going to extend the experiments from [96] and test GGP agents on automatically generated games.

4.3 Evolving Chess-like Games Using Relative Algorithm Performance Profiles

As we deal with the problem of automatic generation of complete rules of an arbitrary game, this requires a generic and accurate evaluating function to score such games. In this section, we investigate the method of evaluation based on the assumption that games should be primarily sensitive to the skill of the players. We present an extension of the method called Relative Algorithm Performance Profiles (RAPP) [138], which makes use of score differences between strong and weak AI players to evaluate games.

We formalize this method into a generally application algorithm estimating game quality according to some set of model games with properties that we want to reproduce. We applied our method to evolve chess-like boardgames belonging to the class of Simplified Boardgames. The results show that we can obtain playable and balanced games of high quality.

Instead of computing the quality of a game basing just on the assumption that we should be interested in games for which good algorithms play significantly better than bad ones, we present more methodical and sophisticated approach. First, for a given set of player strategies, we train our evaluating function using a set of *model* games, i.e. games that have desired properties and should be considered interesting. Then, to reduce computational effort, we use a generic tactic to choose the subset of strategies that will best reflect relations observed for the model. This allows us to value the game according to the level of similarity between relations of the algorithms for the evaluated game and for the model. Finally, we have used our method to evolve 400 symmetric chess-like boardgames, and compared their values with evaluation of some human-written fairy-chess games and 200 randomly generated games.

4.3.1 Relative Algorithm Performance Profiles

The concept of player performance profiles has been used as a method for improving the level of game playing programs by comparing them with various opponent's strategies [80, 213]. The idea behind the RAPP is to use a comparison of playing agents not to evaluate their strategies, but to evaluate the game. This is an indirect approach, which focuses not on the fact how the game is built, but rather how it behaves. That makes RAPP a promising method for application in the GGP domains, where game descriptions are complicated and very sensitive (what actually applies to all GGP languages including GDL and VGDL).

The initial study concerning RAPP focuses on verifying if the concept is applicable in the domain of VGDL games [138]. The authors show that in human-designed games the differentiation between scores obtained by strong and weak algorithms is greater than in randomly generated or mutated games. This leads to the conclusion that good games should magnify the differences between the results of distinct algorithms.

More insightful research has been presented in [139], where RAPP has been used in a fitness function evaluating VGDL games. The function compares the scores of the following two algorithms: *DeepSearch* presented in details in the paper, and *DoNothing* which always returns the *null* action. A number of games were generated; many of them evaluated with near-perfect fitness, and some of them had interesting properties and features. Yet, for creating high quality games, comparable to human-designed ones, the necessity of refining the fitness function to identify more aspects of the game has been stated.

4.3.2 Method

RAPP is an approach to evaluate the quality of games by measuring results of various controllers playing them. To use this method, one needs a set of algorithms serving as controllers, and a set of approved games that will be used as the model set. Then, a game is evaluated by running the algorithms on it, and comparing their results with those obtained by playing the model set of games.

Games in the example set

As a set of exemplary, well-founded games, we have chosen ten human-written variants of fairy-chess (including Chess itself). Most of these games use the orthodox Chess pieces, have a similar starting state (the first line of pawns, one king), and allow pawns promotion.

Here we provide a brief description of each game from the example set. The detailed rules of each game can be seen in [1]. In some cases, we had to omit some of the special game rules (eg. castling, en passant, promotion) to fit the game within the Simplified Boardgames framework. The concept of the Chess check is replaced by the goal of capturing the king. We also changed promotion of the weakest piece into the winning condition. To each game we assigned a turn limit, whose exceeding causes a draw.

Gardner (boardsize: 5×5, turnlimit: 100). The starting positions look as in the orthodox Chess with removed columns f, g, h and rows 3, 4, 5. Besides that, the rules of the Chess apply.

Action Man's Chess (boardsize: 5×6, turnlimit: 120). This variant does not have the king, and has spearman (moves one step forward or backward, captures one step forward) instead of pawns. The goal is to capture all pieces of the opponent.

Petty Chess (boardsize: 5×6, turnlimit: 120). This is a variant published in British Chess Magazine in 1930. It contains all types of pieces from the orthodox Chess. There is no initial double-step move of pawns.

Half Chess (boardsize: 4×8, turnlimit: 100). There are bishops and knights instead of pawns in the first line. The second line contains rooks, a queen, and the king. The only goal is to capture the opponent's king.

Demi-chess (boardsize: 4×8, turnlimit: 100). This is a variant similar to Half Chess. Behind the line of pawns, there are king, bishop, knight and rook. The same rules as in orthodox Chess apply.

Los Alamos Chess (boardsize: 6×6, turnlimit: 120). This minichess variant has no bishops. It became famous as the first chess-like game played by a computer program (MANIAC I in 1956).
Cannons and Crabs (boardsize: 6×7, turnlimit: 160). This is an unusual variant with two types of pawns: the orthodox pawn without initial double-step move, and the crab, which can move one step forward and capture or move one square diagonally forward. There is also the cannon, which can move or capture one or two squares in any direction, and also can jump over the pieces.
Small-Deacon Chess (boardsize: 7×7, turnlimit: 160). This another unusual variant contains Alfil-Knight (jumps in $(2, 1)$ pattern like knight and in $(2, 2)$ pattern like alfil), Archdeacon (bishop that can move one step horizontally and vertically), and Dabbabah-Knight (jumps in $(2, 1)$ knight pattern and in $(2, 0)$ dabbabah pattern). In the simplified boardgames version, Archdeacon (which can bounce off edges) is turned into the orthodox bishop.
Shatranj (boardsize: 8×8, turnlimit: 200). This variant comes from the 7th century Persia. It uses the general, which moves one square diagonally, instead of queen, and the elephant (a $(2, 2)$-jumper like alfil) instead of bishop.
Chess (boardsize: 8×8, turnlimit: 200). This is the orthodox Chess game with the initial double-step move of pawns, but without other rules beyond the class of Simplified Boardgames (i.e. castling, en passant, check, stalemate, the fifty-move rule, and repetition of moves).

Evaluating algorithms

As the evaluating algorithms we have used a min-max search with a constant depth and various heuristic functions evaluating game states. In total, there are 16 distinct player profiles, which differ by strategic aspects they cover. Because such a big number of profiles is impractical to evaluate a large number of games, by analyzing their behavior on the example games, we narrowed this set to a subset of algorithms that produce most characteristic results.

All the heuristic functions are sums of weighted game features. The primary set of features contains material and positional features, which are general approaches to evaluate a state in a chess-like game [37]. For a given type of piece, the material feature is the difference between the numbers of pieces of the two players. The value of a positional feature for a given piece and square is 1 if such piece occupy the square, and 0 otherwise. Thus, the weight of this feature determines the willingness of the player to put a piece of that type on the square.

Implemented profiles use two strategies of assigning weights to material and positional features. The first, **Constant**, assigns to every material feature the square root of the number of squares on the board, and zero to all positional features. The alternative strategy is **Weighted**, which bases on the mobility of each piece as presented in [96]. For every available move of a piece, a probability that this move will be legal is estimated. The weight of the positional features is the sum of the probabilities of all moves that are legal from the given square divided by the number of squares on the board. The weight of the material features is the sum of all positional features for the given piece.

This primary set of algorithms can be improved by using the following more subtle heuristics:
Mobility: This computes for each square the square root of the number of legal moves ending on this square, and adds to the score of the game state the difference between the sum of these values between players. The aim of this heuristic is to promote expansion of pieces and covering a large area of the board.
Control: This is a similar, yet in some sense opposite strategy to the previous one. For every square, it computes which player wins a maximal sequence of captures, i.e. who has more capturing moves, assuming that the square is occupied by an opponent's piece. The score of the game state is modified by the difference between the numbers of squares controlled by the players.

This strategy assists protection of own pieces and points out holes in opponent's defense, posing a threat to unprotected pieces.
Goal: This modifies the values of pieces and squares that are crucial to win, according to the method presented in Section 6.2.2. The weight of pieces occurring in the game's terminal conditions are increased depending on their numbers in the initial state. The larger it is, the less important is an individual piece. Moreover, for the pieces whose aim is to reach some squares, the weights of positional features are increased for the nearby squares (using the number of moves required to move from a square to another as the distance measure). This heuristic, in contrast to Mobility and Control, is computed at the beginning of the game and does not tune weights during the gameplay.

We have used Constant or Weighted material heuristics with all combinations of the three positional ones. From this point, we will use shortened names to identify these combinations; for example *CGM* stands for Constant+Goal+Mobility, while *WC* denotes Weighted+Control.

Results

We have tested the performance of all sixteen algorithms by playing against each other the games from the example set. For every game, each pair of the heuristics played 100 times using three-ply deep min-max search, which gives 12,000 plays to create a single game profile. The gathered data is the matrix of the average scores between every pair of the heuristics. A win of a play was counted as 1 point, a draw as 0.5, and a loss as 0.

To evaluate how good each heuristic is, we took the average of its results from playing against all other heuristics. We illustrate the performance of the algorithms taking its average score for the games from the example set. This is presented by the bars of *example games* in Figure 4.3.1.

From this point, we can see a clear tendency in the cases of some algorithms. Heuristic *CG* is undoubtedly the worst, with *C* being the second worst. All heuristics based on weighted material and position values performed above the average score of 0.5. *WGC* is counted as the best heuristic, yet the differences between the top three are very small (*WGC* – 0.66360, *WC* – 0.64927, *WCM* – 0.64780). However, the standard deviation shows that the set of example games is not consistent, and some games have very different profiles than the others.

4.3.3 Selection of the Model

The results from the previous experiment show us that some games from the example set behave substantially different in the set of example algorithms. For example, the performance of *CM* and *WGM* in *Petty chess* (0.653 and 0.745, respectively) is vitally better than in all other games, while the performance of *CGM* for that game is the lowest (0.12). Hence, our first task is to narrow the set of example games into a smaller set, which will contain games with similar behavior. Then it will be used as the point of reference – the model for generating game rules.

Due to the computational cost, we also needed to narrow the set of algorithms used as evaluation profiles. Choosing the set of representative heuristics for a given model set of games is the second task covered in this section.

Selecting model games

For two games $\mathcal{G}$ and $\mathcal{H}$, given their profile matrices $P_\mathcal{G}$, $P_\mathcal{H}$ obtained by running n player profiles, we can calculate their level of resemblance in a standard way as the pairwise distance between these matrices:

$$\mathrm{dist}(\mathcal{G},\mathcal{H}) = \frac{\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(P_\mathcal{G}[i,j]-P_\mathcal{H}[i,j])^2}{n(n-1)/2}. \tag{4.2}$$

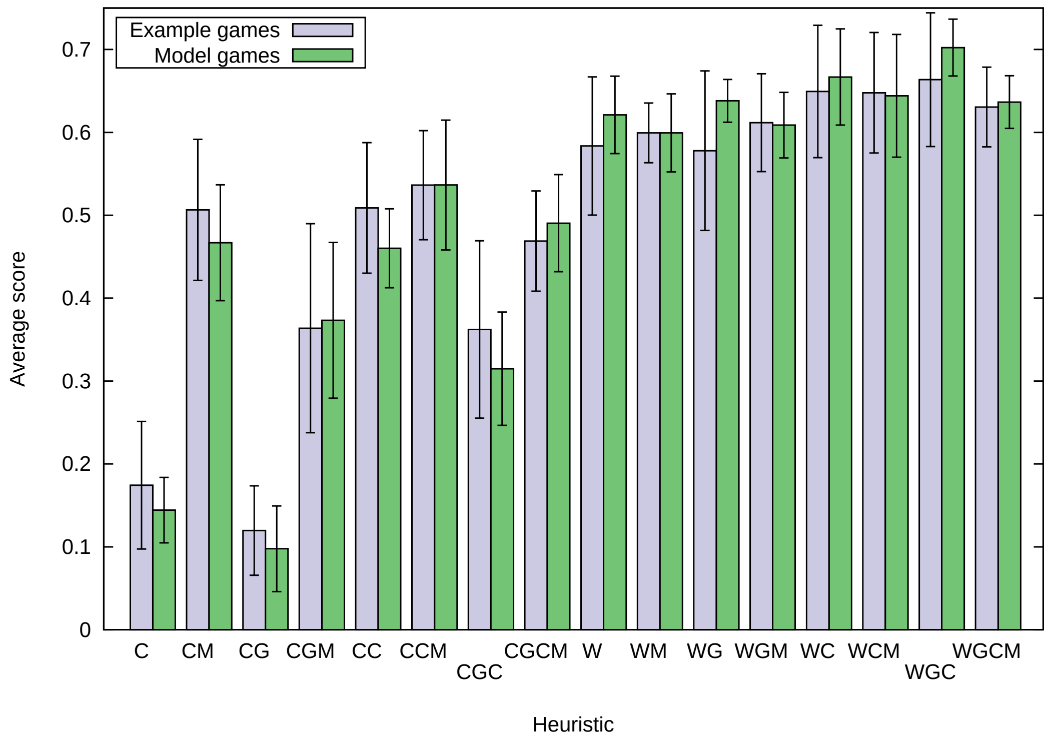


Figure 4.3.1: The average score and the standard deviation of the sixteen algorithms on the sets of example and model games.

We can extend this definition to sets of games in the way that the distance of a set is the sum of the distances between every pair of games from this set.

We computed distances of all subsets of the example games, and decided to divide the example set into two subsets. The larger set in the best partition is *Action Man's Chess*, *Cannons And Crabs*, *Chess*, *Los Alamos*, *Shatranj* and *Small-Deacon Chess*. This subset was evaluated as tenth among the sets containing six games. On the other hand, the set of the remaining games is one of the worst four element sets. From the obtained results, we draw a conclusion that the larger subset represents a type of games that we are interested in. These games are significantly different, yet share enough similarities to be relatively close in terms of performance of the algorithms. The set also contains *Chess*, which is arguably one of most popular board games with very desired strategic properties. Thus, we decided to use this set as our *model set* of games.

The difference in performances of particular algorithms between playing the model set and the full example set of games is presented in Figure 4.3.1. There are significant variations in scores of some algorithms and, as expected, in most cases the standard deviation is smaller.

Selecting evaluation algorithms

Having the model set, we can evaluate how similarly a given game behave to the model by comparing the relative performances of the heuristics. The more heuristics we will use, the more accurate evaluation will be. The drawback is that the required number of tests grows quadratically in the number of heuristics. For this reason, we needed to find a method of

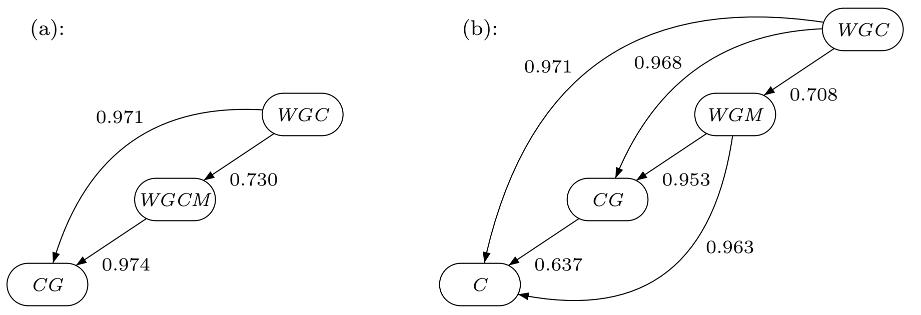


Figure 4.3.2: Representation graphs of the relative performances in the models with (a) 3 algorithms and (b) 4 algorithms. The arrows are directed from the stronger toward the weaker algorithm, and their labels contain the average score.

selecting a small number of algorithms, which stand as the best representatives for the model games.

We have decided to selected the algorithms that are most distant to each other in terms of their relative performances. The reasoning behind this idea is twofold. First, the selected heuristics are substantially different, which allows further evaluation to incorporate more game features. Second, we take into account only the most essential differences in performance, which are easier to capture during evaluation using a smaller number of test plays. If the difference in the performance of two algorithms is small, it could be easily misjudged depending on the number of test plays.

Let P_{model} be the matrix containing the average relative performances on the model games of some k algorithms. We can define its spread as follows:

$$\text{spread}(P_{\text{model}}) = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} |P_{\text{model}}[i,j] - 0.5|^2. \tag{4.3}$$

Then, for the given k, we can compute the matrices for all k-subsets of the example algorithms and calculate their spreads. We performed this computation for $k \in \{3, 4, 5\}$, and found the subsets with the largest spreads. For $k = 3$ this is *CG*, *WGC*, and *WGCM*; for $k = 4$ this is *C*, *CG*, *WGM*, and *WGC*; for $k = 5$ this is *C*, *CG*, *WGM*, *WGC*, and *WGCM*. In further experiments, we will use two models, using the best 3-subset and the best 4-subset, respectively. The visualization of the relative performance of the algorithms in these models is shown in Figure 4.3.2.

Fitness function

According to our RAPP method assumptions, the fitness of the game should be based on the distance between the matrix of the algorithm performances for that game, and the average performance matrix of the model games. This, however, is not enough to ensure that desired game properties will be fulfilled. There are several features that may be important for a good boardgame, e.g. balance between players, complexity of the rules, branching factor, game length, and usage of the pieces [76]. Our goal was to keep the fitness formula as simple and general as possible, so we decided to choose two features we perceive as the most crucial in the case of the Simplified Boardgames: balance between the players and game length being not too short.

Let $score_w$ be the percent of points scored by the white player during p test plays, and $score_b$ be the number of points scored by the black player. Then $B = |score_w - score_b|/p$ is the balance between the players. Let say that a play is *too short* if it ends in 10 turns. If during the test s games were qualified as *too short*, then $Q = \frac{s}{p}$ is the measure of the game's *quickness*. At last, let D be the modified formula (4.2) of distance between the game's matrix of the algorithm performances and the model matrix, where the absolute value of the difference is used instead of the power of two. Then, our fitness function is calculated as follows:

$$f = \begin{cases} (1-D)(1-B)(1-Q) & \text{if the game is } \textit{playable}, \\ -1 & \text{otherwise.} \end{cases} \tag{4.4}$$

The game is considered as *playable* if during the test all plays fit within the given timelimit. This function roughly matches with our intuition about that when a given game should be considered better than the others. We made some experiments using the unmodified formula (4.2), yet because it usually returns small values, the overall fitness was high, and the differences between good games and slightly-less-good games were very small. For this reason, we decided to stricter results to increase the impact of D in the formula, and make the differences in fitness more visible.

4.3.4 Evolving New Games

Our goal is to generate games that have similar properties, in the sense of efficiency of strategies, to the games in the model. To achieve that, we use RAPP and evolutionary search over a constrained subset of the Simplified Boardgames.

Generating games

For the sake of efficiency and reduction of the search space, we restricted our generating mechanism to produce only chess-like games. By that, we mean the games fulfilling the following additional conditions:

- The initial position is symmetric and contains two rows of pieces for both players. The front row contains only the pieces called *pawns* or empty squares. The back row contains, among other pieces, one piece of the *king* type.
- The terminal conditions are symmetric. A player wins by capturing the enemy king, or by reaching the opponent's back row with a pawn.

To generate a new game, we start by selecting randomly parameters *width* and *height* of the board, the number of *non-winning* types of figures (in addition to always present *king* and *pawn*). For the purpose of our experiments, *width* and *height* are taken from $\{6, 7, 8\}$, and the number of figures from $\{3, 4, 5\}$. Parameter *turnlimit* is computed by the formula $3 \times width \times height + r$, where r is a uniformly random number taken from $\{0, \ldots, 19\}$. Generating the initial position is straightforward: In the front row every square can be either empty with probability 0.1 or occupied by a pawn. The king is placed uniformly at random in a column of the back row, and the remaining squares are either empty with probability 0.1 or filled by a non-winning piece chosen uniformly at random.

The remaining part, generating regular expressions for descriptions of piece rules, is more challenging. First, we generate a lot of *raw* move patterns; they will be used as building blocks for piece rules. A single raw move pattern is a list of tuples $(\Delta x, \Delta y, on, star)$, where Δx and Δy are signed integers that define the horizontal and vertical shifts of the moves, *on* is

Table 4.3.1: Exemplary probability parameters used in generating games for picking at random: (a) relative coordinates, (b) square content, (c) pattern modifiers.

$\Delta y \backslash \Delta x$	0	+1	+2	+3
+3	0.6%	0.6%	0.6%	0.6%
+2	7%	7%	3.5%	0.6%
+1	20%	14%	5%	0.6%
0	0%	16%	7%	0.6%
-1	7%	7%	1.4%	0.6%

(a)

on	*e*	*p*	*w*	*ep*	*ew*	*pw*
normal	42%	14%	14%	7%	7%	14%
last	33%	33%	0%	33%	0%	0%

(b)

modifier	*FB*	*ROT*	*STAR*
probability	50%	25%	25%

(c)

from $\{e, p, w, ep, ew, pw\}$ and defines allowed contents on the destination square (e – empty, p – enemy piece, w – own piece), and *star* is a logical value indicating whether this tuple can be repeated with Kleene star. There is also a set of modifiers extending the pattern: FB mirrors the pattern across the horizontal axis, and ROT rotates the pattern through 90°, 180°, and 270° about the relative origin $(0, 0)$. For example, the orthodox rook can be described as $(0, 1, e, \mathit{true})(0, 1, ep, \mathit{false})[ROT]$.

Generating raw move patterns is based on a number of parameters. The process begins with picking at random a list of $(\Delta x, \Delta y)$ pairs. Exemplary probability values for each possible pair are listed in Table 4.3.1(a). New elements are added to the list as long as another probability test is passed. Next, we extend the list by adding *on* values. The exemplary probability table used for that purpose is shown in Table 4.3.1(b). We use a separate set of probabilities to determine the content of the destination square in the last part of a pattern. In particular, we do not allow self capturing, while it is possible to step over own figures during the movement. The last step is to add modifiers. As long as the probability test of the *modifier_prob* parameter (e.g. 0.1) is passed, one modifier is added to the pattern (see Table 4.3.1(c) for exemplary values). When $STAR$ modifier is drawn, it is applied to a random tuple in the list by changing the *star* value to true. Also, if $\Delta x > 0$ occurs in the pattern, we mirror it across the vertical axis to avoid side asymmetry. This completes generating raw move patterns.

Next, we generate each piece by choosing raw patterns, whose sum will be used as the movement language. This process is guided by the following two parameters: the chance of adding a new raw pattern to the set, and the maximal mobility of the piece. Because the values of these parameters are different for king and pawn, and non-wining pieces, we can enforce limited mobility for kings and pawns, to make resulting games more chess-like.

A weak point of this approach is that it heavily relies on human designed values. We have used four different settings, including the one presented here. The main differences in the settings we used were in probabilities of long-range jumps, probabilities of backward moves (also with additional possibility of $\Delta y = -2$), chances of applying modifiers, and probability of enlarging a raw move pattern.

Genetic operators

Let n be an even population size. Given fitness values, we select $n/2$ pairs of parents using the roulette-wheel method. Although it never happened in practice, in this method if all games from the population have fitness less than or equal to zero, the uniform selection is used instead. Every pair of parents produce two offspring using the uniform crossover, which independently swaps some squares of the initial position (except the squares containing kings) and some piece rules.

There are two types of mutation, both independently modifying an offspring with some small probability. The *piece mutation* regenerates the rule of a random piece. The *position mutation* changes the content of a random square. If the square belongs to the first row, a pawn is replaced by the empty square and vice versa. If the square is occupied by the king, it is swapped with some other second row square. In the remaining cases, the square content is replaced by a random non-winning piece or the square is left empty (with uniform probability).

The next generation is created by choosing the best n games from the population of parents and children. The evolution process stops when the maximum number of generations is done.

Classification

Apart from the fitness function, which is used for the purpose of evolution, we developed a notation classifying a game usefulness from the human point of view.

We call the games *unplayable* if their fitness value is negative. All the other games, whose balance factor B of the fitness function is less than 0.2 and whose Q factor is less than 0.05, are considered as *promising*. They may not have a great score taking into account the set of model games, but their properties assure that they should be at least human playable.

We have used this classification model to judge the progress of evolution in terms of reducing the number of bad individuals. It also gives a clear view how different settings of parameters influence the population. From the four of our settings, two result in about 12% of *promising* games, while the other two in about 50%. However, it is worth noticing that the differences in the number of *promising* individuals do not necessarily influence the score of the best individual, and some of the best generated games were produced by the less reliable settings.

4.3.5 Experiments

We tested our method using the extracted set of six model games and the two variants using 3 and 4 algorithms. We generated 200 games (using four sets of parameters), and evaluated each of them by 3 or 4 algorithms. In all our tests, we have used 50 min-max plays (25 per side) with depth 3 to compare every pair of algorithms and obtain the values to the profile matrix of the game. In the same way, we also evaluated all the example games.

The last test used evolution. For both variants with 3 and 4 algorithms we made 12 runs of evolution with populations of size 10, and both mutation rates equal to 5%. Additionally, we made 12 runs for the variant with 4 algorithms, and increased population size to 16 and mutation rates to 10%. In all these cases, the maximal number of generations was 20.

Overview

For all sets of games (example, generated, and evolved), we used two measures of comparison: the maximal fitness within the set and the average of the fitness of *promising* games. We are convinced that the last one is very important, as it shows the potential of the set. It may happen that the top game is not the best one from the human point of view; then the other candidates should be as good as possible in terms of accordance with the model. At last, we show the percent of *promising* games within the sets. These results are presented in Table 4.3.2.

As we can see, in all variants the best individual score was obtained by the evolutionary approach. Moreover, the average score of all final populations is also the highest. On the other hand, a pure generation can create a game even better than the best one from the example set (as in the variant with 3 algorithms), yet it is not so likely. The average score is low, as it is the fraction of promising games. Thus, the pure generation is too general and too random to be

Table 4.3.2: Comparison of game evaluation between the sets of example games, randomly generated games, and evolved games. The last column contains the results of the test with increased population size and mutation rate.

Variant	3 algs.			4 algs.			4 algs., population 16		
	Max.	Avg.	Promising	Max.	Avg.	Promising	Max.	Avg.	Promising
Evolved	0.971	0.911	100%	0.959	0.916	100%	0.978	0.922	100%
Generated	0.907	0.671	29%	0.942	0.704	34%			
Example	0.858	0.811	90%	0.959	0.859	100%			

seen as a proper method just by itself, and there is a necessity of combining it with some score-improving method as evolution process. The score of the example games is not so high as one might think; this is due to the fact that the model reflects the average relations observed between the games, so every individual is likely to be distinct to that average. However, the games from the model set were usually rated better than the other example games that are outside the model. There are two important observations. One is that a larger number of algorithms indeed yields a better ordering of games and evaluations closer to expected, as the score is obtained using a more detailed model. The second is that the evaluation method is very play-sensitive, i.e. the scores of one game obtained by consecutive tests can vary. Although the general tendency, whether a game is good, average, or poor, usually remains clear, the detailed results may be significantly different. This can influence the ordering of games and so the evolution process. However, to achieve better stability, it should be sufficient to increase the number of plays used for game evaluation.

Evolution results

Figure 4.3.3 presents the course of the evolution in the tested variants. They all look similar, however in the variant with 3 algorithms the density of final scores distribution is visibly lower.

It can be seen that the process of evolution can increase the quality of the population to a decent level even when the initial population is poor. What is to be expected, the improvement of the best individual takes place stepwise, as it is not so easy to obtain a better game in every generation. On the other hand, the improvement of the average population is more smooth. Also, usually except the first few generations, all games remaining in the populations are *promising*, so dysfunctional individuals are quickly discarded.

The example of an evolved game is presented in Figure 4.3.4 (the full rules are shown in Appendix B). This game appeared in the 16-th generation and obtained score 0.9538. Its rules are human-readable and not too complicated, and it requires a non-trivial strategy since the beginning. Note that an opening using 🐟 blocks the pawn advance of the opponent. Thus, to be able to move out the other pieces, and simultaneously prevent opponent's expansion, the player has to move his own pawns in a clever way.

Although pawns seem to be powerful and advance rapidly, it is impossible to move them to the opponent's backrank, so capturing the king remains the only reachable goal. It is common between best-scored games that pawn usefulness is very limited, e.g. they cannot advance but only move sideways. It seems that pawn movements are so crucial and hard to control that it is easier to gain better game flow stability by reducing their impact on the game than by finding a set of moves that is not too strong yet still substantially useful.

Also, we have found that the rules that seem pretty understandable in their regular expression form are often very hard to be translated on the board and remembered during a play. It seems

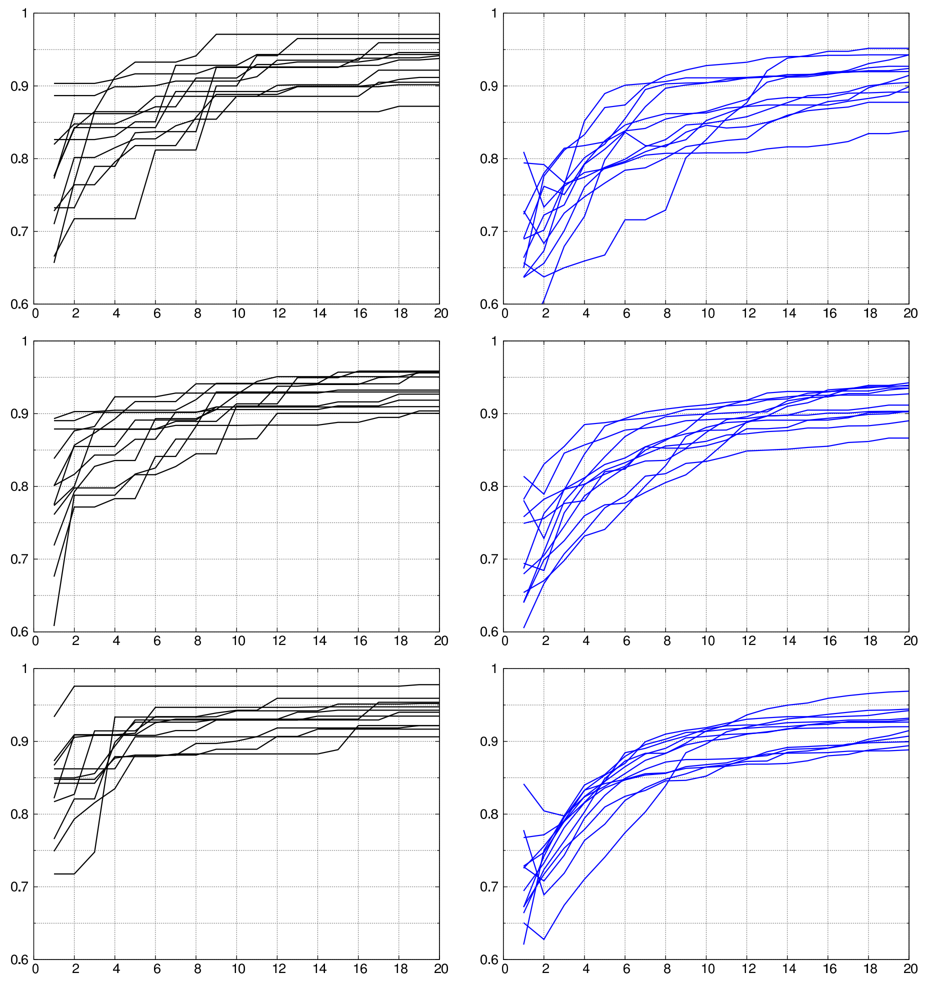


Figure 4.3.3: Visualization of our evolutionary runs. First row is generated for test with 3 algorithms, the second describes test with 4 algorithms (population size 10), the third row describes the variant with 4 algorithms and population of size 16. The left and right columns show respectively the maximum and average fitness values of playable games in every iteration.

that by generating we can easily produce movement rules that are reasonable yet over-complicated from the human point of view. This observation addresses the problem of differences between generating game rules for humans and for AI players (e.g. to be used during GGP competitions).

4.3.6 Conclusions

We proposed an extension of the RAPP method evaluating the quality of a game using the performances of game-playing algorithms. The original approach was based on the assumption that in well-designed games strong algorithms should significantly outperform weak ones.

In this section, we presented a more sophisticated solution, which uses more detailed information concerning the mutual relations of the given algorithms, and thus it is able to catch and evaluate more subtle aspects of the games. However, to make this working, one needs a set of

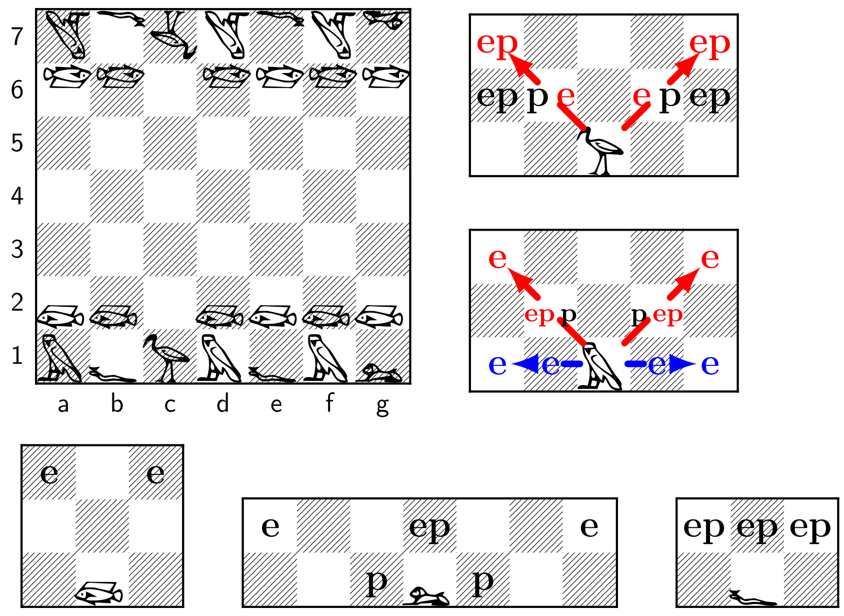


Figure 4.3.4: Initial position and piece rules for the evolved game. Destination square content (from $\{e, p, w\}$) designates legal moves. Moves with consecutive requirements are distinguished by colors and arrows. The pawn symbol is , and the king symbol is . The turnlimit is 160.

model games, which have desired strategic properties, and use it to actually train the evaluation function.

The previous application of RAPP concerns VGDL games [139], which are one-player puzzles focusing on maximizing the player's score. We have shown its application in the domain of two-player, zero-sum games belonging to the class of Simplified Boardgames. We also tested it with various numbers of algorithms used for performance comparison in the evaluation function. In contrast with the previous study using only 2 algorithms, we provided an evaluation function in a more general form, which can be applied with any number of strategies, and allowed us to perform tests with 3 and 4 algorithms.

Finally, we applied our method to an evolutionary system, which was able to produce in some cases even „more model" games than these actually used in the model set. Not all of the top rated games could be judged as well-designed by human players. The reason for that is mostly abnormality and large complexity of the movement rules. However, this issue can be solved in two ways: by improving move generation techniques and their parameterization, or by extending the fitness function to take into account features like complexity of the rules or pieces usefulness.

A nice property of the RAPP method is its generality. The presented methods of extracting the set of model games and model algorithms are domain independent, and can be used for any types of games. Unfortunately, the task of finding sets of example games and algorithms requires human expert knowledge on the subject. Making these tasks knowledge-free is a potential field of further study. In the case of example algorithms, the simplest (yet probably of low quality) approach is to use the MCTS algorithm, where the strength of a player is determined by its timelimit.

We confirm the conclusions from [139], that the best use for a RAPP-based game generation is being a fine sieve, with a human intervention at the last stage, where a man can choose the best game from the given results. Also, we confirm that the game evaluation, as it requires a significant number of simulations, is very time-consuming. However, given very promising results, it is a choice of preferring quality over quantity. The expected practical usage of such generated games, concerning e.g. providing interesting and novel games for the GVGAI Competition [153], requires obtaining just over a dozen high quality games per year.

5 Language Extension

For the sake of the International General Game Playing Competition the Stanford's Game Description Language (GDL) has been developed as a high-level knowledge representation formalism able to describe any finite, n-player, turn-based, deterministic, full-information game (Section 2.2.1). The last two restrictions were removed by the later extension called GDL-II (Section 2.2.3). In our opinion, if GDL is to be a general artificial intelligence language it has still too many restrictions. In particular, one of such restrictions is the requirement that the game has to be turn-based.

In this chapter, we present our extension of GDL, called Real-time Game Description Language (rtGDL), that makes it possible to describe a large variety of games involving a real-time factor. We follow the general idea of [217] to extend substantially the class of described games with the minimal modifications to the language and communication protocol. Real-time GDL can represent games involving time-dependent events, where the exact moment of an action is relevant. This concerns most current video games (such as RTS, FPS and RPG) for which computers are programmed to play using specially designed algorithms [75, 143].

We add only two new keywords and an additional argument for the other three standard GDL relations. What we have achieved is that no real number arithmetic is required inside the rule engine, which unlike some other proposed extensions, e.g. [222], preserve the pure declarative character of GDL. We present the formal syntax and semantics of this language, showing that this extension ensures finite derivability and effectiveness required for practical use. We consider its effectiveness and expressiveness arguing that this is a promising direction of research in the field of General Game Playing. We also introduce a construction of the union of rtGDL and GDL-II, which yields the most universal game description language in the GDL family.

In the next section, we introduce rtGDL as an extension of GDL. Section 5.2 provides formal semantics of Real-time GDL, while Section 5.3 covers the subject of the execution model. In the following section, the extension is illustrated by a series of examples. Section 5.5 contains a sketch of joining rtGDL with GDL-II yielding a language capable of describing games with both the imperfect information and the real-time factor. Concluding remarks are given in Section 5.6.

The content of this chapter is partially based on the papers [95], [97].

5.1 From GDL to rtGDL

The extension of GDL presented in this chapter is intended to remove important restriction of GDL games that they need to be turn-based. We preserve the purely declarative style, so that the state computing inference engine can remain unchanged. In general, dealing with real-time games may involve real number arithmetic, which cannot be encoded in pure GDL in a simple way. The proposed approach requires real arithmetic on the level of search engine reasoning only. On the level of the rule engine, numbers are treated like standard terms without additional semantics (in a manner similar to natural numbers in a `goal` relation).

`init(T,F)`	`F` is true in the initial state for the time `T`
`true(T,F)`	`F` is true in the current state for the time `T`
`next(T,F)`	`F` is true in the next state for the time `T`
`infinity`	stands for ∞ when used as a time value
`expired(F)`	holds when `F` becomes obsolete

Table 5.1.1: The rtGDL keywords: the top three are modified GDL keywords, while the last two are new keywords

Changes between rtGDL and GDL keywords are summarized in Table 5.1.1. Keywords that are not listed: `role`, `legal`, `does`, `terminal` and `goal` remain unchanged (see Table 2.2.1). Parameter `T`$\in \mathbb{R}^+ \cup \{\infty\}$ linked with the currently holding fact `F` encodes the *lifetime* of this fact, that is the time for which this fact will be true. After the time is over, the state update is performed and the additional relation `expired(F)` indicates that `F` has just expired and allows changes based on this knowledge.

The `expired` keyword can be seen as just a syntactic sugar for the expression `true(0,F)`, however its introduction simplifies games rules (there is no more need for conditioning time value) and provides unambiguous semantics for the obsolete facts. Thus, syntactic restrictions guarantees that 0 time value never occurs in `true` relation and `expired` is used instead.

State updates are performed after a player makes a move or when some fact becomes obsolete. Before such an update is computed, all times assigned to the set of holding facts are properly updated according to the time since the last update. The detailed semantics of rtGDL language is provided in the next two sections. Before going into formal details, the reader may want to see first examples in Section 5.4.1.

5.1.1 Formal Syntax Restrictions

Descriptions of rtGDL games use the standard syntax of logic programs, including negation and inequality (denoted as `distinct`). For the sake of readability, we will use Prolog convention rather than standard GDL code convention, that is clauses are written in infix form, variables are uppercase letters, and function names start with lowercase letters.

As in the case of the standard GDL, certain syntactic restrictions have to be fulfilled to ensure that the game specification has an unambiguous interpretation, and all derivations are finite. First, exactly the same restrictions as in GDL and GDL-II are imposed on the keywords (a new keyword `expired` is restricted in the same way as `true`).

- `role` only appears in the head of ground atomic sentences;
- `init` only appears in the head of clauses and does not depend on `true`, `legal`, `does`, `next`, `terminal`, `goal` or `expired`;

- `true` only appears in the body of clauses;
- `does` only appears in the body of clauses and does not depend on `legal`, `terminal` or `goal`;
- `next` only appears in the head of clauses;
- `expired` only appears in the body of clauses.

Also, we adopt the convention that to be considered as *valid* rtGDL game description must be *stratified* [5], *allowed* [115], and satisfy the general *recursion restriction* (see [181, Definition 3.]). These restrictions ensure that the game rules, despite introducing time-based values, which can be arbitrary real numbers, can be effectively and unambiguously interpreted by a state transition system, and all relevant derivations remain finite and decidable (see Theorem 5.2.2).

5.2 Semantics: A Game Model for rtGDL

We show that rtGDL may be seen as a straightforward extension of GDL. As in GDL, any game description defines a finite set of domain-dependent function symbols and constants, which determines a (usually infinite) set of ground symbolic expressions Σ. These ground terms can represent players, moves or parts of the game state. However, unlike in the standard GDL, the game states are not simply subsets of Σ. Every individual feature of the game state has assigned a *lifetime*, which is a real number value or the infinity symbol. Such feature holds for this period of time, and after that (unless other events occur) it disappears from the state. Moreover, initially declared real numbers that occur in Σ are not the only ones. Any real number, given by the outside environment as updated lifetime, can appear as a constant later during the game.

Therefore, we define an rtGDL game state to be a finite subset of $\mathbb{R}_+^\infty \times \Gamma$, where $\mathbb{R}_+^\infty = (0, \infty]$ (including ∞), and $\Gamma \subseteq \Sigma(\mathbb{R}_+^\infty)$ where $\Sigma(\mathbb{R}_+^\infty)$ is a set of ground terms of the game, with the set of constants extended by $\mathbb{R}_+^\infty$.

Now, in introducing the elements of the game model, we follow the constructions for GDL and GDL-II given in [181]

- $R \subseteq \Sigma$ (the *roles*);
- $s_0 \subseteq \mathbb{R}_+^\infty \times \Sigma$ (the *initial position*);
- $t \subseteq \mathcal{P}_{\text{fin}}(\mathbb{R}_+^\infty \times \Gamma)$ (the *terminal positions*);
- $l \subseteq R \times \Gamma \times \mathcal{P}_{\text{fin}}(\mathbb{R}_+^\infty \times \Gamma)$ (the *legality relation*);
- $u : \mathbb{R}_+^\infty \times (R \nrightarrow \Gamma) \times \mathcal{P}_{\text{fin}}(\mathbb{R}_+^\infty \times \Gamma) \mapsto \mathcal{P}_{\text{fin}}(\mathbb{R}_+^\infty \times \Gamma)$ (the *update function*);
- $g \subseteq R \times \mathbb{N} \times \mathcal{P}_{\text{fin}}(\mathbb{R}_+^\infty \times \Gamma)$ (the *goal relation*).

(where $\mathcal{P}_{\text{fin}}(A)$ denotes the finite subsets of A) The legality relation $l(r, m, S)$, states that in the position S, a player r can perform move m. Given a time Δt since the last state update, and a partial function of the joint moves performed by the players $M : (R \nrightarrow \Gamma)$, the state update function $u(t, M, S)$ determines the updated position. Finally, the relation $g(r, n, S)$ defines the payoff of a player r in the game state S.

5.2.1 State Transition System

Let $\mathcal{G}$ be a valid game description of an n-player rtGDL game. The players' roles are defined by the derivable instances of `role(R)`. A *game state* $S = \{(t_1, f_1), \ldots, (t_k, f_k)\}$ is encoded as a set of facts with assigned lifetimes. At the beginning of the game, the state is composed of the derivable instances of `init(T,F)`. We use the keyword `true` to extend the game description $\mathcal{G}$ by the facts resulting from S, which form the set

$$S^{\texttt{true}} \overset{\text{def}}{=} \{\texttt{true(}t_1\texttt{,}f_1\texttt{)}, \ldots, \texttt{true(}t_k\texttt{,}f_k\texttt{)}\}.$$

Instances of `goal(R,N)` derivable from $\mathcal{G} \cup S^{\texttt{true}}$ assign to every player `R` its goal value `N` in this state. If `terminal` is derivable from $\mathcal{G} \cup S^{\texttt{true}}$, then the state S is a terminal position. Computing legal actions, requires derivable instances of `legal(R,M)`, for every role `R` and move `M`.

The state of the game remains unchanged until an *event* occurs. There are two types of events, one caused by the players sending their moves, and the other resulting from the expiration of some facts in $S^{\texttt{true}}$. Let Δt be the time to the first occurrence of an event.

When a subset of players $r_{i_1}, \ldots, r_{i_l}$, where $\forall j\ i_j \in \{1, \ldots, n\}$, perform moves $m_{i_1}, \ldots, m_{i_l}$, then

$$M^{\texttt{does}} \overset{\text{def}}{=} \{\texttt{does(}r_{i_1}\texttt{,}m_{i_1}\texttt{)}, \ldots, \texttt{does(}r_{i_l}\texttt{,}m_{i_l}\texttt{)}\}.$$

Let us notice that $M^{\texttt{does}} = \emptyset$ if no player sends a move.

Performing state update requires updating lifetime of every holding fact, and information about the facts that expired. These two sets are defined in a following way:

$$\mu(S, \Delta t) = \{\texttt{true(}t_i - \Delta t\texttt{,}f_i\texttt{)} : \texttt{true(}t_i\texttt{,}f_i\texttt{)} \in S^{\texttt{true}} \ \wedge\ t_i > \Delta t\}$$

$$S^{\texttt{exp}}_{\Delta t} \overset{\text{def}}{=} \{\texttt{expired(}f_i\texttt{)} : \texttt{true(}t_i\texttt{,}f_i\texttt{)} \in S^{\texttt{true}} \ \wedge\ t_i \leq \Delta t\}$$

where $\mu(S, \Delta t)$ updates the facts lifetimes and $S^{\texttt{exp}}_{\Delta t}$ contains expired facts. The updated position is composed of the instances of `next(T,F)` derivable from $\mathcal{G} \cup M^{\texttt{does}} \cup \mu(S, \Delta t) \cup S^{\texttt{exp}}_{\Delta t}$.

In order to summarize above in a formal definition, as in [181], we make use of the fact that any stratified set of clauses $\mathcal{G}$ has a unique *stable model* [58]. We denote it SM[$\mathcal{G}$]. Due to the syntax restriction in rtGDL, it is finite ([118]).

Definition 5.2.1. *Let $\mathcal{G}$ be a valid rtGDL specification whose signature determines the set of ground terms Σ and $\Gamma \subseteq \Sigma(\mathbb{R}^{\infty}_{+})$. The semantics of $\mathcal{G}$ is the state transition system (R, s_0, t, l, u, g) given by*

- $R = \{r \in \Sigma : \texttt{role(}r\texttt{)} \in \mathrm{SM}[\mathcal{G}]\}$;
- $s_0 = \{(t, f) \in \mathbb{R}^{\infty}_{+} \times \Sigma : \texttt{init(}t\texttt{,}f\texttt{)} \in \mathrm{SM}[\mathcal{G}]\}$;
- $t = \{S \in 2^{\mathbb{R}^{\infty}_{+} \times \Gamma} : \texttt{terminal} \in \mathrm{SM}[\mathcal{G} \cup S^{true}]\}$;
- $l = \{(r, m, S) : \texttt{legal(}r\texttt{,}m\texttt{)} \in \mathrm{SM}[\mathcal{G} \cup S^{true}]\}$, *for all* $r \in R$, $m \in \Gamma$ *and* $S \in \mathcal{P}_{\text{fin}}(\mathbb{R}^{\infty}_{+} \times \Gamma)$;
- $u(\Delta t, M, S) = \{(t, f) : \texttt{next(}t\texttt{,}f\texttt{)} \in \mathrm{SM}[\mathcal{G} \cup M^{does} \cup \mu(S, \Delta t) \cup S^{exp}_{\Delta t}]\}$, *for all* $M : (R \nrightarrow \Gamma)$, $S \in \mathcal{P}_{\text{fin}}(\mathbb{R}^{\infty}_{+} \times \Gamma)$, *and minimal* Δt *such that* $M^{does} \cup S^{exp}_{\Delta t} \neq \emptyset$;
- $g = \{(r, n, S) : \texttt{goal(}r\texttt{,}n\texttt{)} \in \mathrm{SM}[\mathcal{G} \cup S^{true}]\}$, *for all* $r \in R$, $n \in \mathbb{N}$ *and* $S \in \mathcal{P}_{\text{fin}}(\mathbb{R}^{\infty}_{+} \times \Gamma)$.

This defines a formal semantics for an abstract game model of rtGDL. The calculation of function μ takes place outside the state computation and is a part of the execution model, which is the subject of the next section

The model in the above definition is uncountably infinite, and cannot be used itself as an effective description of the game. Yet, it is *locally finite* in the sense that all stable models involved are finite and effectively computable. This is in spite of that rtGDL games involve a real-time factor and incomplete information (on time when events occur). In other words, our extension preserves the crucial property of „finite derivability", which for rtGDL setting can be specified in the following way. A *state of the game* is the set of all facts (ground terms) holding in a given moment of the game. Each state that can be achieved from the initial state by a finite sequence of moves of the players performed in certain times from the start of the game is called a *reachable state.* The last state occurring before the present state S in this sequence is called the *preceding state* for S (and the given sequence of moves). Note that the same state can be reached by various sequences of moves.

Theorem 5.2.2. *Let $\mathcal{G}$ be a valid rtGDL game description. Then, each reachable state is finite and can be effectively computed from any preceding state, given the joint move and the time passed.*

Proof. The proof is similar as in case of GDL and GDL-II, based on results on logic programming. First, since $\mathcal{G}$ is stratified, by [58] it admits a unique stable model SM[$\mathcal{G}$] as the declarative semantics, which is the same a the „iterated fixed point" model M_P in [5]. Moreover, by the allowedness and the general recursion restriction this model is finite [118, 179]. It can be effectively computed using e.g., the iterative procedure described in [5]. This implies in particular that the initial state is finite and can be effectively computed. Now the proof is by induction. Assuming that a reached state is finite, the joint move of the players contains a finitely many new real time constants, and therefore the stable model $\mathrm{SM}[\mathcal{G} \cup M^{\mathtt{does}} \cup \mu(S, \Delta t) \cup S^{\mathtt{exp}}_{\Delta t}]$ is finite. It follows that there are finitely many instances of `next(T,F)` derivable from $\mathcal{G} \cup M^{\mathtt{does}} \cup \mu(S, \Delta t) \cup S^{\mathtt{exp}}_{\Delta t}$, and as before they can be effectively computed.

The result above means, in particular, that any finite part of the game tree may be effectively computed on the basis of the moves and times they occur.

5.3 Execution Model

An execution model of rtGDL is designed to handle real-time events, which makes it a bit more complex than the standard execution model of GDL. There are various possible implementations to consider. The simple model we present here requires very little deviation from the standard execution model of GDL [118].

After the initial state s_0 is computed, the timer is turned on and the game starts. The initial state is treated as the current one until some event occurs that stops the timer. This event can be triggered by players who sent their moves, or it can be scheduled, known in advance, expiration of some state features. At the moment of the event, the state update is performed according to the game model semantics with Δt equal to time indicate by the timer. The timer is then turned on again, and the whole process repeats until the game reaches a terminal position. The final scores of the players are defined by the goal relation.

It should be noticed, that scheduled updates caused by fact expirations are silent, i.e. there is no message to the players indicating that the update took place. Such messages are unnecessary due to the fact that players have all the information to perform such updates by themselves, based on control times sent by the Game Manager.

The straightforward implementation of a Game Manager according to this model looks as follows:

1. Send to the players the rtGDL code containing game description with information about their roles and binding timelimits. Set $S := S_0$. Turn the timer on.
2. After the time for the players to familiarize with game has passed, inform the players of the game beginning.
3. Restart the timer. Calculate t_u which is the minimal time for some position feature from S to become obsolete.
4. Wait until some players send their moves or the timer is equal to t_u. Set $M :=$ 'players moves', $\Delta t :=$ 'timer indications'.
5. If $M \neq \emptyset$ inform the players about the performed moves M and the current game time (since the beginning of the game).
6. Perform state update by setting $S := u(\Delta t, M, S)$.
7. If S is not a terminal state go to step 3. Otherwise, compute the payoff for every player accordingly to g relation and finish the game.

The game flow is unequivocally dictated according to the Game Manager's timer. In a case when a move sent by a player is not legal in the current state, update is not performed. There are no restrictions on the number of moves the players can send, and all of them should be considered by the Game Manager accordingly to their arrival times.

5.3.1 Communication Protocol

There are slight differences between the GDL and rtGDL communicates, but the main idea and architecture remains the same. For the detailed description of the GDL communication protocol we refer to [65].

In rtGDL protocol there are no new types of messages, and all player replies remain unchanged. In the Game Manager message of type `START`, the semantic of `PLAYCLOCK` parameter has changed. Now it is used as a multiplier of fact lifetimes, i.e. to get the number of seconds before a fact becomes obsolete, its current lifetime has to by multiply by `PLAYCLOCK`. With `PLAYCLOCK` 20, fact `true(1.5,control(white))` becomes expired after 30 seconds, while fact `true(0.5,cooldown)` after 10 seconds. This preserves a very desirable possibility of controlling the game pace without changes in the game code. The semantics of the `STARTCLOCK` parameter remain unchanged.

The `PLAY` and `STOP` messages are extended, and now their last (third) parameter is a game time as `PLAYCLOCK` multiplyer, i.e. the number of seconds (as a floating point number) from the beginning of the game to the moment when the message was sent by the Game Manager divided by the current `PLAYCLOCK` value.

The second argument of `PLAY`/`STOP` message is a list of actions taken by all players. This list can be set to `NIL` in two special cases: when this is the initial game message indicating the end of the preparation phase, or it is an answer to a player who sent an illegal move (other players should not receive this message). In a real-time gaming the Game Manager is receiving players' moves sequentially rather than simultaneously, which results that the lists of performed actions sent in `PLAY`/`STOP` messages are only partially filled. If a player on a position k made no move his move is simply denoted as `NIL`, which allows that the length of the list remains constant and

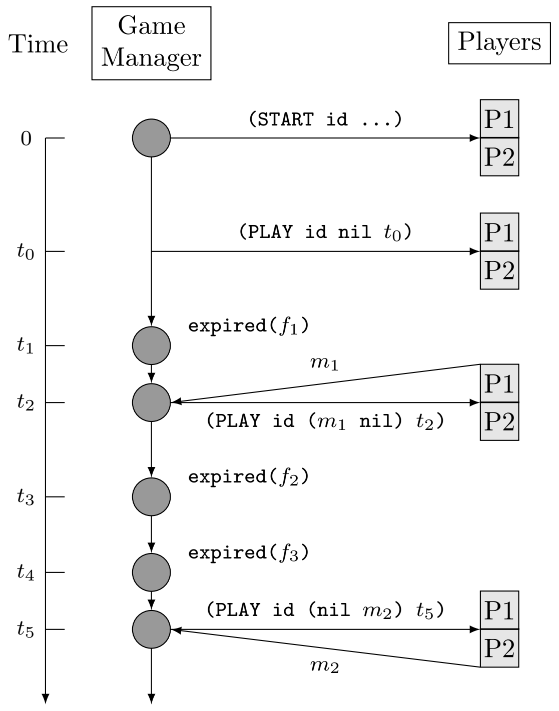


Figure 5.3.1: Visualization of communication during some rtGDL game. Circles in the Game Manager's column indicates a state update.

equal to number of the players in the game. (In fact, the lists may be replaced by *role–move* pairs.) An example of communication between the Game Manager and players during exemplary rtGDL game with two players is presented in Figure 5.3.1.

5.3.2 Model Analysis

The above simple execution model does not address the problem of possible lags during communication. It may be treated as suitable in the scenario when rtGDL programs compete on the same server, and communication delay may be ignored. Yet, in case of games played through the GGP website, lags may cause practical problems with having up-to-date knowledge about the current state of the game. One direction to deal with the delay problem is to divide the time axis into small intervals and admit communication only at the end points of these intervals. The intervals should be large enough to provide time for indispensable computation. Another possible approach is to allow for the Game Manager to stop the time of the game in the situations when updating the state requires more time. The choice between models should depend on the possible applications.

Another problem concerns correcting information about the current state of the game in the situation when a message about an action-based event just arrived and makes some knowledge obsolete. The point is that an action-based event may change lifetimes of other events.

Consider a message m notifying about an action-based event with timestamp t, received at time $t' > t$. The player, having access to all previous messages, and thus to the move history, has full knowledge about the state s at time t (including lifetimes of all facts). Thus, the player

Listing 5.1: rtGDL rules of Game of Chicken.

```
role(white).   role(black).
init(infinity,dir(white,straight)).
init(infinity,dir(black,straight)).
init(1.0,timer).

next(T,timer) ⇐ true(T,timer).
next(infinity,dir(R,swerve)) ⇐ does(R,swerve).
next(T,dir(R,D)) ⇐ true(T,dir(R,D)) ∧ ¬ does(R,swerve).
legal(R,swerve)  ⇐ true(T,dir(R,straight)).

terminal ⇐ expired(timer).
goal(R,0) ⇐ true(T,dir(R,straight)) ∧ true(S,dir(P,straight))
  ∧ distinct(R,P).
goal(R,80) ⇐ true(T,dir(R,swerve)) ∧ true(S,dir(P,straight)).
goal(R,90) ⇐ true(T,dir(R,swerve)) ∧ true(S,dir(P,swerve))
  ∧ distinct(R,P).
goal(P,100) ⇐ true(T,dir(R,swerve)) ∧ true(S,dir(P,straight)).
```

can backtrack to this state and, given m, perform the suitable state update.

Assuming stable time difference between a player and the Game Manager, it is possible for the player, after performing at least one move, to synchronize with the Game Manager and be able to send every later move so that it arrives at the desired time.

Indeed, assume a player sent a move at time t_s. As an answer, he get a `PLAY` message at time t_p, with parameter t_r describing the time when the Game Manager received his move. Now, the player can put $\Delta t = t_r - t_s$, and for every move which should be received by the Game Manager at time t, the player can send it at the time $t - \Delta t$ on its own clock. The estimation Δt can be adjusted to be more reliable as the game continues.

This problem recalls the known practical problem in the GGP competitions of sending moves a little bit earlier to make sure they arrive to the Game Manager in time. Yet, in rtGDL setting, it is more essential to adjust the time as exact as possible.

We may summarize all the above:

Theorem 5.3.1. *For each Real-time GDL game taking place through the website, there exists a time delay Δt such that, for any time t_0, the players are able to compute fully the state of the game at time t_0 with delay not larger than Δt.*

5.4 Expressiveness of rtGDL

To illustrate how the game rules are constructed, we provide an rtGDL code of the *Game of Chicken* (Figure 5.1). It is a well known game, where two players pretend to be tough, and do not want to swerve from the colliding path waiting for the opponent to do this.
The game lasts for the one unit of time. Each player starts riding straight ahead, and until the `timer` holds, the player can decide to `swerve`. After swerving, no more legal moves are allowed.

The payoff matrix is defined as follows. In the case of a crash, both players' reward is 0. If only one player decides to swerve his reward is 80, while his opponent's reward is 100. When both players decide to swerve they gain 90 points each.

We now proceed to show that, in principle, all GDL games can be expressed in rtGDL. To this end we need the following definition.

Definition 5.4.1. *We say that an rtGDL game $\mathcal{G}'$ has an* equivalent semantics *to a GDL game $\mathcal{G}$ if there is a procedure converting any program for $\mathcal{G}$ to a program for $\mathcal{G}'$, so that the result of any rtGDL competition C' between converted programs is the same as the result of GDL competition C of the original GDL programs.*

The definition could be stated more formally, but it is sufficient for the sketch of the proof provided below.

Theorem 5.4.2. *For every GDL game $\mathcal{G}$ there exist real-time GDL game $\mathcal{G}'$ with equivalent semantic.*

Proof. Considering $\mathcal{G}$ as a GDL specification, we convert it, step by step, in an rtGDL specification. First, all occurrences of `true`, `init`, and `next` are modified to fit the rtGDL syntax by endowing each with the lifetime parameter `infinity`. We introduce three new function constants `made(R,M)`, `clock`, and `moved(R)` with intention to postpone the state update until playclock `clock` expires, and to mark down that the player `R` has performed his move. Accordingly, all occurrences of `does(`r`,`m`)` are replaced by `true(infinity, made(`r`,`m`))`, and every rule of the form `next(infinity,`f`)`$\Leftarrow$ ϕ is replaced by `next(infinity,`f`)`$\Leftarrow$ ϕ $\wedge$ `expired(clock)`. In addition we add a set of new rules:

```
next(infinity,moved(R)) ⇐ does(R,M).
next(infinity,made(R,M)) ⇐ does(R,M) ∧ ¬ true(infinity,moved(R)).
next(infinity,F) ⇐ true(T,clock) ∧ true(infinity,F).
```

The intended meaning of the constant `clock` is ensured by the following:

```
init(1.0,clock).
next(1.0,clock) ⇐ expired(clock).
next(t,clock) ⇐ true(t,clock).
```

The presented code allows the players to perform precisely one move during the unit `clock` interval. The optimal strategy of the players (based on the given strategy for the corresponding GDL game) should be to make use of all the timelimit to compute their single move as in GDL and send it to the Game Manager. Then, the result of the competition will be the same.

Here we need to make a theoretical assumption that, as a result of such strategy, the players send their moves at the same time, at the end of the timelimit, and in consequence, at the same time are informed about their joint move. Another theoretical assumption is that players do not send illegal moves, because the rtGDL Game Manager handles illegal moves in a different way than the GDL one.

In practice, the games $\mathcal{G}$ and $\mathcal{G}'$ would not be fully equivalent, especially when it is essential that players have no knowledge about the moves done by other players during the same turn. Also, the error handling would require a special solution. (The simplest one would be to treat illegal moves as the loss.)

It is natural to compare rtGDL games with Extensive Form Continuous-time Games [194]. Let us recall that in such games the players act on the interval $R = [0, 1]$. Let A be the joint set of all possible actions of all players. The *decision node* is a pair $\langle 0, \emptyset \rangle$ or $\langle t, h \rangle$, where $t \in (0, 1]$ and h is a function from $[0, t)$ to A, representing the history of the game up to time t. Now, since every rtGDL game has to be finite in terms of match duration, it is always possible to map rtGDL lifetime units to $(0, 1]$ such that every match do not go beyond R. The construction is straightforward, and we may see that every rtGDL game can be represented in the form of the Extensive Form Continuous-time Game.

However, it should be noted, that rtGDL nodes allows only finite number of legal actions. This restriction adopted to preserve finite derivability property of GDL games is, at the same time, a serious limitation. In particular, it does not allow players to use real numbers in their move messages, which is a natural action in many games.

5.4.1 Examples

We also provide a series of short examples describing real-time elements appearing in many games.

Example 1. Turn-based games

First, we demonstrate how to translate a turn-based GDL game into rtGDL. For this purpose we use the standard Tic-tac-toe game. The following code describes the most important parts of the suitable rtGDL description:

```
role(xplayer).
role(oplayer).
init(infinity,cell(1,1,blank)).
...
init(infinity,cell(3,3,blank)).
init(1.0,control(xplayer))

next(1.0,control(S))
    ⇐ does(R,M)
    ∧ role(S)
    ∧ distinct(R,S).

next(infinity,cell(M,N,R))
    ⇐ does(R,mark(M,N))
    ∧ true(infinity,cell(M,N,blank)).
next(infinity,cell(M,N,C))
    ⇐ true(infinity,cell(M,N,C)).
    ∧ distinct(C,blank).
next(infinity,cell(M,N,blank))
    ⇐ true(infinity,cell(M,N,blank))
    ∧ does(R,mark(J,K))
    ∧ (distinct(J,M) ∨ distinct(K,N)).

legal(R,mark(M,N))
    ⇐ true(infinity,cell(M,N,blank))
    ∧ true(T,control(R)).

terminal ⇐ expired(control(R)).

goal(R,0) ⇐ expired(control(R)).
goal(S,100)
    ⇐ expired(control(R))
    ∧ role(S)
    ∧ distinct(R,S).

terminal ⇐ ¬boardopen.
terminal ⇐ line(R).
goal(R,100) ⇐ line(R)).
...
```

Information about the board is stored as a set of forever holding facts, while the time for making a move is determined by the lifetime of the `control` fact. State update is performed only in two cases, after the current player performs a move – then the board should be accordingly updated and control switched to the other player; or after the current player exceeds timeout, which cause expiration of `control` and immediate end of the game.

Other terminal and goal conditions, omitted in the above listing, remain as in the standard Tic-tac-toe, with `boardopen` holding when there is some blank space left on the board, and `line(R)` holding when player `R` composed a line of his symbols. It is worth to note that in the presented game translation (which is not the only one possible) there is no need for the special `NOOP` move by the player without control.

Example 2. Chess clock

Instead of allocating a constant time for every move, rtGDL makes it possible to implement a Chess clock, and count accumulated move time for each player.

```
role(white).
role(black).
init(infinity,timer(R,120.0)) ⇐ role(R).
init(infinity,control(white)).
init(120.0,clock).

next(infinity,timer(R,T))
    ⇐ does(R,M)
    ∧ true(T,clock).
next(infinity,timer(S,T))
    ⇐ true(infinity,timer(S,T))
    ∧ does(R,M)
    ∧ distinct(R,S).

next(T,clock)
    ⇐ true(infinity,timer(S,T))
    ∧ does(R,M)
    ∧ distinct(R,S).

terminal ⇐ expired(clock).
goal(R,0)
    ⇐ expired(clock)
    ∧ true(infinity,control(R)).
```

The remaining times for players are stored in the `timer` relation, while the passing time is counted using the lifetime of the `clock`. After every turn, the remaining `clock` time is copied to the `timer` of the player with control, and it is set as the remaining time of the other player.

Example 3. Self-appearing and disappearing objects

The following rules specify an entity that appears and disappears in constant amounts of time.

```
init(1.0,appear(cheshireCat))

next(3.0,disappear(C)) ⇐ expired(appear(C)).
next(T,disappear(C)) ⇐ true(T,disappear(C)).

next(1.0,appear(C)) ⇐ expired(disappear(C)).
next(T,appear(C)) ⇐ true(T,appear(C)).
```

Example 4. Respawning objects

Slightly more complicated case describes respawning objects, e.g. items lying in the game area and reappearing some time after a player takes them, which is very popular concept in FPS games.

```
init(infinity,onMap(2,3,yellowArmor,5.5))
init(infinity,onMap(11,11,redArmor,10.0))

next(T,toRespawn(X,Y,A,T))
    ⇐ true(infinity,onMap(X,Y,A,T))
    ∧ playerAtPosition(X,Y).
next(T,toRespawn(X,Y,A,S))
    ⇐ true(T,toRespawn(X,Y,A,S)).

next(infinity,onMap(X,Y,A,T))
    ⇐ true(infinity,onMap(X,Y,A,T))
    ∧ ¬playerAtPosition(X,Y).
next(infinity,onMap(X,Y,A,T))
    ⇐ expired(toRespawn(X,Y,A,T))
```

Example 5. Self-moving objects

Another example contains an object traveling through the discrete space grid with fixed speed and refresh rate (`plus` is the + relation defined on a finite subset of natural numbers).

```
speed(xwing,2,2,1)
init(infinity,space(35,86,40,xwing))
init(0.5,refresh)

next(0.5,refresh) ⇐ expired(refresh)
next(T,refresh) ⇐ true(T,refresh)

next(infinity,space(F,G,H,S))
    ⇐ true(infinity,space(X,Y,Z,S))
    ∧ expired(refresh)
    ∧ speed(S,A,B,C)
    ∧ plus(X,A,F)
    ∧ plus(Y,B,G)
    ∧ plus(Z,C,H).
next(infinity,space(X,Y,Z,S))
    ⇐ true(infinity,space(X,Y,Z,S))
    ∧ ¬expired(refresh).
```

Example 6. Player ordering time-taking events

In many games, after giving an order, the player is free to make other actions while the execution of the order needs some time to complete. As an example of such situation we present a partial code of an RTS game where the player can order to build a barracks or a blacksmith (`greaterEqual` is the $\geq$ relation defined on a finite subset of natural numbers).

```
cost(barracks,160,6.0).
cost(blacksmith,140,7.0).
init(infinity,gold(500)).

legal(player,build(B))
    ⇐ true(infinity,gold(G))
    ∧ cost(B,F,T)
    ∧ greaterEqual(G,F).
next(T,underConstruction(B))
    ⇐ does(player,build(B))
    ∧ cost(B,G,T).
next(T,underConstruction(B))
    ⇐ true(T,underConstruction(B)).

next(infinity,constructed(B))
    ⇐ expired(underConstruction(B)).
next(infinity,constructed(B))
    ⇐ true(infinity,constructed(B)).
```

5.5 Real-time GDL with Imperfect Information

Following the extension from GDL to GDL-II proposed by Thielscher in [217], we can also drop the deterministic and perfect information restrictions. Below we sketch the rules of another GDL extension, called rtGDL-II, which is a combination of Real-time GDL and GDL with Incomplete Information, and in consequence, the most universal language in the GDL family.

5.5.1 Nondeterminism

The nondeterminism introduced by GDL-II rules relies on the special `random` role operated by the Game Manager. Real-time GDL can be extended in a similar way, however in this case the randomness may be introduced at two dimensions. We may describe not only *which* action is to be drawn (like in GDL-II), but also *when* it should be drawn.

The first case can be solved in a straightforward way, by assuming that if in some state the `random` role has a non-empty set of legal moves, then immediately one of these moves should be made with uniform probability and the state update should be performed. Such usage of randomness is suitable for generating discrete random variables, e.g. rolling a dice or shuffling a deck of cards.

For the latter case we may adopt the following solution. If the `random` player's move drawn by the Game Manager is a relation with arity greater then 0, and the first argument of this relation is a non-negative real number t, then the state update should be performed after a random time t' obtained using the continuous uniform distribution $\mathcal{U}(0, t)$.

As an example consider four legal actions of `random`: `wait`, `move(right)`, `shot(0,pistol)`, `shot(2,blaster)`. Every action has a 25% chance to be chosen. If one of the first three actions will is to be made, then the Game Manager should perform state update immediately. In the remaining case, a random value t' generated with distribution $\mathcal{U}(0, 2)$ should be drawn. After the time t', if the move is still legal, the state update should be performed. In the case when the move is no longer legal (due to the players' actions or some other events) no state update is performed and the action is wasted.

5.5.2 Imperfect Information

In orded to introduce imperfect information we follow directly Thiescher's approach. The relation `sees` serves as a container for players' percepts, and the Game Manager as a feedback for the player sends his percepts instead of the joint move.

However, as the game is not turn-based any more, the Game Manager cannot simply send everyone messages after obtaining some player's move, because this will give everyone a clue that someone made a move. We may want to avoid this in games where just knowing that other player made an action is a crucial piece of information.

Therfore we may adopt the following solution. For every player two sets of percepts are computed. One containing percepts after updating a state according to the game rules, and the other with the percepts from the same moment, but with the assumption that the state update did not occur. The latter is possible, because apart from the players' actions the Game Manager can predict the state in any future moment. A message containing the updated percept is send to the player only in the case when this two sets differ.

5.6 Conclusions

With the introduction of GDL-II it was claimed that the GDL language can be considered complete [181, 217], and additional elements can only serve for simplifying description or will be forcing setting extensions far beyond the concept of General Game Playing (e.g. open-world games or physical games like in General Video Game Playing [154]).

Our position is contrary. We argue that our real-time extension preserves the core idea of GDL – a concise purely logical description of the game and a simple execution model. At the same time it is a larger extension than GDL-II, allowing a game to have an infinite number of states and the players to have an infinite number of actions. By introducing time based events and by giving relevance to the move ordering it makes possible to describe properly many real world situations. In particular, it allows to model many elements of the popular computer games, which are currently used as a testbed for dedicated AI players, e.g. Super Mario Bros [228], Unreal Tournament 2004 [75], TORCS [116], or Starcraft [143]. Beyond the usage in GGP, the correlations between the GDL and the game theory [220] and Multiagent Systems [179] are often pointed out. A goal for GDL is to become a universal description language that can describe as large class of game-like problems as possible, at the same time remaining compact, high-level, and machine-processable. We presented the next step into such a generalization of problems description, which brings the class of games covered by the GDL family closer to real-time game theory (e.g. Extensive Games in Continuous Time [194], Continuous Time Repeated Games [11], Differential Games [79], and Real Time Multiagent Systems [82]). In this chapter, we have introduced a new language rtGDL with possible extension to rtGDL-II, and addressed a new ambitious challenge for General Game Playing systems. Taking into account response time of message arrival, it creates for the players an „act-wait dilemma", i.e. given more computation time the game tree could be explored better, however this may cause some promising paths to no longer be available due to the in-game events or the other player actions. This scenario requires developing some new general-case solutions, like for example usage of real-time Monte Carlo Tree Search [151]. It makes real-time general gaming a very hard but interesting area of further General Game Playing research.

6 Translations

In this chapter, we address the concept of translating games between various representations. There are few existing translations between Stanford's GDL and other languages. As mentioned in Section 2.4.2, GDL games can be rewritten as structure rewriting games and used by the Toss system [87]. A complete embedding of GDL into an action language called $\mathcal{C}+$ has been presented in [221]. The translation is proved to be correct and allows applying to the GDL-based players known results from reasoning about actions domain. In [162], the authors describe a translation of a given single-player GDL game (puzzle) into Planning Domain Definition Language. Conducted experiments support the practical application of this method, which for many cases allows efficient puzzle solving by using planning algorithms to generate the game's solution.

We present two projects based on the translations and comparisons between general game description languages. Firstly, we make a link between a specialized context free language expressing the rules of variety of card games, called the Card Game Description Language (Sections 2.4.3 and 6.1), and the most known general-purpose game description language GDL-II (Section 2.2.3). We present a systematic translation between the languages, show the translation to be correct, and analyze the complexity of resulting code.

Secondly, we compare the level of GGP programs in two domains: Stanford's GGP and Simplified Boardgames (Section 2.1.2). A few boardgames were rewritten into GDL format and played by the two top GGP players against a sample Simplified Boardgame player which, having knowledge about the game chess-likeness, used temporal difference learning to learn evaluation function feature weights.

The content of this chapter is partially based on the papers [94, 96].

6.1 Embedding CGDL into GDL-II

The Card Game Description Language developed recently in [50] has a unique feature – it allows to use genetic programming to evolve game rules and so to create the new games. CGDL is a high level language with a lot of domain specific commands to describe n-player, standard deck card games with possibility of bets (like in poker). In this section, we define a direct translation from CGDL language into GDL-II. Then, we compare both languages as representatives of two different approaches to game description: very general one but with difficult semantic influencing efficiency of playing algorithms, and narrow one with the compact and flexible description.

We implement our translation and compare the complexity of CGDL and GDL-II both theoretically and empirically. We also distinguish features that can possibly be transferred from CGDL into GDL/GDL-II to improve this, currently the most widely used, family of general game description languages.

6.1.1 Card Game Description Language

Card Game Description Language (CGDL), introduced in [50], is a context free language designed to define a rich subset of possible card games and allow to perform genetic operations to create new or evolve existing games [51]. The language domain is narrowed to n player, standard deck card games with a possibility of coin bets. All language constructions are strictly domain-dependent and use concepts of a card, number, suit, token, etc. These concepts, same as arithmetic and boolean operators, are defined a priori and used without explicit declaration.

A valid CGDL game is defined as follows. P is the number of players. Every player i has his private hand location (named Hi) and two areas for placing coins: private $Ki0$ with player's coins, and public $Ki1$ for placing bets. T is the number of virtual table locations, where cards can be placed face up. The set of game rules is organized into sequentially ordered stages, containing rules. Every stage is played in a round-robin order until all players are *out* of the game or decide to end the current stage (status set to *done*). Every rule has a form of $modifier$ `if` *condition* `then` *action*. The current player can perform any *action* from the set of current stage rules only if the rule *condition* is satisfied and the *modifier* limitations are met. Possible rule *modifiers* are: *computer* – must be played by the computer at the beginning of a stage, *mandatory* – must be played by a player at the beginning of a stage (after computer rules applied), *once* – can be applied only once, *optional* – no restrictions. The complete list of possible *conditions* contains

- λ – no condition to satisfy, always true;
- `sum,`LA`,`R`,`LB – sums values of cards in both locations LA and LB, evaluates to true if the restriction $R \in \{<, >, =, \leq, \geq, \lambda\}$ is satisfied;
- `tokens,`KA`,`R`,`KB – compares the number of tokens in KA and KB, true if restriction R is satisfied;
- `have,`C – check if the player's hand contains given card combination C, e.g. king of hearts, three diamonds and one heart, etc;
- `draw` – draw one card from the deck and place it in the current player's hand location;
- `play,`LA`,`R`,`LB – compares the play in the card locations LA and LB, true if restriction R is satisfied;
- `show,`R`,`LA – show a set of cards from player's hand that satisfy restriction R compared to the cards from LA.

Set of available *actions* is a superset of

- `pifr,`LA`,`A`,`F – draw a given amount A of cards from location LA to player's hand, cards are openly visible if face F is *up*;
- `bet,`R`,`KA – bet an amount of tokens if number satisfies a restriction R compared to token location KA;
- `done` – the player status is set to *done*;
- `out` – the player status is set to *out*;
- `win` – the player instantly wins the game;

- `give`,P,A – *computer* only, give A tokens to the set of players P;
- `deal`,P,A – *computer*, deal A cards from the deck to the players P.

Special abbreviations are available like HA – hands of all players, KA bets of all players, HX hand of current player, KX bets of current player. Also `<allplayers>` is a syntactic sugar for multiplying rule for all possible players.

Rules can also define a value mapping for every card and some combination of cards (called plays). Game is over when all players are *out*, one of the players performs `win` action or the last stage is over. Player's score is calculated by points for every possessed token (`t`), card (`c`) and "not being `out`" bonus (`s`). If player ends by choosing `win` action he got 100 and other 0.

As an example, rules in Listing 6.1 describe a codification of game *Blackjack*. For more detailed language specification, including other games examples and lists of all possible conditions and actions we refer to [50].

Listing 6.1: CGDL codification of the game *Blackjack*.

```
[SETTINGS] P=3, T=0
[STAGES]
  Stage 0
COMPUTER deal, <allplayers>, 2
COMPUTER give, <allplayers>, 99
  Stage 1
MANDATORY if λ then bet, λ , λ
  Stage 2
OPTIONAL if λ then pifr, D, 1, up
OPTIONAL if λ then done
  Stage 3
MANDATORY if sum, HX, >, 21  then out
MANDATORY if sum, HX, <=, 21  then done
  Stage 4
MANDATORY if sum, HX, >, HA  then gain, KA
[RANKING] 2:2, ..., King:10, Ace:11, Ace:1
[POINTS] t=1, c=0, s=0
```

6.1.2 Translation

In this section, we present details of our translation. We constructed the translation function $\mathcal{F}$ that takes a CGDL game description $\mathcal{G}$ and returns the same game encoded in GDL-II language. As the CGDL language is created by context-free grammar, construction is grammar based. For every game description subtree we compute all GDL rules necessary to encode that subtree. After this we remove duplicate definitions of the predicates and save the final result.

To organize the description, we divided it into subsections covering creation of all parts of GDL game specified in Definition 2.2.1. As an example we translate CGDL codification of game *Blackjack* from Listing 6.1 to equivalent GDL-II rules partially presented in Listing 6.2. References to line numbers (if not stated otherwise) refers to Listing 6.2.

Listing 6.2: Partial GDL-II code of translated CGDL game *Blackjack*.

```
(role random) (role player1) ... (role player3)
...
(init (Stage ShuffleDeck COMPUTER))
(init (Shuffled Top))
(<= (init (UnShuffled ?c)) (card ?c ?num ?suit))
(init (ActionAvailable 0 s0a0 random COMPUTER))...
...
(<= (next (Token ?l2 ?n3)) (true (Token ?l2 ?n1))
  (movecoin ?n2 ?l1 ?l2) (asum ?n1 ?n2 ?n3))
(<= (next (Token ?l1 ?n3)) (true (Token ?l1 ?n1))
  (movecoin ?n2 ?l1 ?l2) (asub ?n1 ?n2 ?n3))
(<= (next (ActionAvailable ?stage ?id1 ?p ?type))
  (true (ActionAvailable ?stage ?id1 ?p ?type))
  (does ?player (action ?id2 ?vis ?cond ?act))
  (distinct ?id1 ?id2))
(<= (next (Won ?player))
  (does ?player (action ?id ?vis ?cond win)))
...
(<= (legal ?player ?act) (tmplegal ?player ?act))
(<= (tmplegal ?player (action s4a0 visible (sum HX gt HA) (gain KA)))
  (true (Stage 4 MANDATORY))
  (true (ActionAvailable 4 s4a0 ?player MANDATORY))
  (true (CurrentPlayer ?player))
  (not (true (PlayerStatus ?player aDONE)))
  (not (true (PlayerStatus ?player aOUT)))
  (handlocation ?player ?hand) (rsum ?hand ?n)
  (handlocation ?p1 ?h1) (distinct ?p1 ?player)
  (rsum ?h1 ?n1) (handlocation ?p2 ?h2)
  (rsum ?h2 ?n2)
  (distinct ?p2 ?player) (distinct ?p2 ?p1)
  (or (bgt ?n ?n1) (true (PlayerStatus ?p1 aOUT)))
  (or (bgt ?n ?n2) (true (PlayerStatus ?p2 aOUT))))
(<= (legal random NOOP)
  (true (Stage ?n ?t)) (distinct ?t COMPUTER))
...
(<= (movecoin ?n ?bloc1 ?hloc)
  (does ?player (action ?id ?vi ?cond (gain KA)))
  (betlocation ?player ?hloc ?bloc)
  (betlocation ?p ?hloc1 ?bloc1) (rKA ?n))
...
(<= (sees ?player (deltacoins ?n ?loc1 ?loc2))
  (does ?p (action ?id visible ?cond ?action))
  (movecoin ?n ?loc1 ?loc2))
(<= (sees ?player (deltacoins ?n ?loc1 ?loc2))
  (does ?player (action ?id ?vis ?cond ?action))
  (movecoin ?n ?loc1 ?loc2))
...
(<= terminal endstage (true (Stage ?n ?t))
  (_stagesorder (Stage ?n ?t) (Stage EndGame none)))
(<= (goal ?player 100) (true (Won ?player)))
(<= (goal ?player 0) (true (Won ?p))
  (role ?player) (not (true (Won ?player))))
...
```

```
557 (<= (rKA ?s3)
558   (true (Token K11 ?n1)) (true (Token K21 ?n2))
559   (true (Token K31 ?n3)) (asum 0 ?n1 ?s1)
560   (asum ?s1 ?n2 ?s2) (asum ?s2 ?n3 ?s3))
561 (<= (rsum ?loc ?s12)
562   (location ?loc) (numberofcards ?loc 2)
563   (hold2cards ?loc ?c1 ?c2)
564   (card ?c1 ?num1 ?suit1) (value ?num1 ?val1)
565   (card ?c2 ?num2 ?suit2) (value ?num2 ?val2)
566   (asum 0 ?val1 ?s1) (asum ?s1 ?val2 ?s12))
...
694 (location D) ... (location H3)
695 (handlocation player1 H1) ...
696 (betlocation player1 K10 K11) ...
697 (card 2OfHearts 2 Hearts) ...
698 (value 2 2) ... (value Ace 11) (value Ace 1)
699 (stagesorder (Stage ShuffleDeck COMPUTER) (Stage 0 COMPUTER))
700                  ...
701 (stagesorder (Stage 4 MANDATORY) (Stage EndGame none))
...
819 (aplus1 0 1) ... (aplus1 100 101)
820 (<= (asum ?n 0 ?n)     (aplus1 ?n ?m))
821 (<= (asum ?n1 ?n3 ?n5) (aplus1 ?n2 ?n3) (aplus1 ?n4 ?n5) (asum ?n1 ?n2 ?n4))
822 (<= (bleq ?n ?n)       (aplus1 ?n ?m))
823 (<= (bleq ?n1 ?n3)     (aplus1 ?n1 ?n2) (bleq ?n2 ?n3))
```

Constants

Constants define relations that hold throughout the entire game (line 1 and 694–701). The predicate `role ?role` declares set of players `player1`, ...`player`P, `random`; `location ?loc` defines set of possible card locations: `D` for the deck, `H1`, ..., `H`P for players' hands and `T0`, ..., `T(`T–1`)` for table locations; `handlocation ?player ?hand` maps `player`i to his hand location `H`i. The predicate `tokenlocation ?loc` defines all token locations `K10`, ..., `K`P`0`, `K11`, ..., `K`P`1` and `betlocation ?player ?privloc ?betloc` maps these locations from player `player`i to private token location `K`i`0` and bet location `K`i`1`. List of all cards is stored in `card ?card ?number ?suit`. The card value mapping is stored in predicate `value ?card ?n`. Round-robin ordering of P players is simply defined as `playersorder ?prev ?next` relation. The predicate `stagesorder ?prev ?next` is constructed based on ordering detected in CGDL game codification.

Game state

Definition 6.1.1. *The game state is the minimal set of data necessary to distinguish that two game positions are different. CGDL game state consists of: the contents of the deck and all table and token locations; the number of current stage and player; the status of every player and the information about available computer/mandatory/once actions.*

A game state is covered by a constant number of GDL-II predicates forming $S^{\mathtt{true}}$ sets and storing information necessary to encode the CGDL state according to Definition 6.1.1.

- `Stage ?id ?type` – identifier of the current stage and a type of the sub-stage containing information about the allowed actions type (*computer*, *mandatory*, *optional*); one such predicate is true at the time;

- `ActionAvailable ?stagenumber ?actionID ?player ?type` – stores all actions that can be performed by the players, if a non-optional action was made it is removed from this set and cannot be used again;
- `Token ?location ?amount` – for every `tokenlocation` stores the number of tokens in this location;
- `Table ?location ?card` – for every `tablelocation` except the deck stores the cards in this location;
- `Deck ?nextcard ?prevcard` – contains cards in the deck arranged in an order (special constant `Top` marks top of the deck);
- `CurrentPlayer ?player` – identifies the current player if it is not the dealer's turn, maximum one such predicate is true at the time;
- `PlayerStatus ?player ?status` – for every player remembers his status if necessary; status can be `aDONE` if the player decided to end the current stage, `aOUT` if he is out of the game or `mDONE` if he has no mandatory moves but is not *done*;
- `Won ?player` – true if some player performed the `win` action;
- `UnShuffled ?card` – a special predicate used at the beginning of the game to remember cards that are not yet shuffled into the deck;
- `Shuffled ?card` – a special game-beginning predicate to remember the last card chosen by the dealer to be placed at the bottom of the deck.

Definition 6.1.2. *A state is called* ***technical*** *in one of the following cases: the predicate `Stage ShuffleDeck COMPUTER` is true – which means that the virtual dealer prepares random ordering of cards in the deck; or the `CurrentPlayer` status is "out", "done" or he has no actions with fulfilled conditions but it is his turn in round-robin ordering – then every player makes the `NOOP` move and the `CurrentPlayer` shifts to the next player in order (which may lead to a next technical state).*

Initial state

The initial $\mathcal{F}(\mathcal{G})$ game state s_0 (line 26) is technical, because of necessity to randomize the card deck. Every game begins with `(Stage ShuffleDeck COMPUTER)` $\in S^{\mathtt{true}}$ and all cards identifiers marked as `UnShuffled`. For the next 52 turns, the `random` player choose every still `UnShuffled` card with the uniform probability and put on the bottom of the actual deck. After this, `Stage` changes to the first stage from the original $\mathcal{G}$ game. Also possible players' actions are put into the `AvailableAction` predicate.

Legal actions

Managing legal actions can be divided into several cases. When `CurrentPlayer` is set and there is some non-`COMPUTER` `Stage`, the player can choose legal action from existing `tmplegal ?player (action ?id ?visibility ?condition ?action)` relations. Such relation corresponds to exactly one rule from $\mathcal{G}$. Variables `?condition` and `?action` matches CGDL rule condition and action; `?id` is an artificial identifier matching the id from `AvailableActions` relation, and `?visibility` serves to determine $\mathcal{I}$ function.

In our example, the rule from CGDL codification line 15 is translated into the `tmplegal` rule in GDL-II code line 315. Initial queries are checking stage, action availability and player's status. Then queries matching *condition* are set. In our example these are restricting sum of values of player's cards against the values of all other players cards.

Another case occurs when no `tmplegal` relation is true, due to not fulfilled conditions or exhaustion of `AvailableActions`. Then player can make only special `NOOP` move that does not change the game state. This move is also used in the following cases: for player that as not marked as `CurrentPlayers`; for player who is marked as current but he is actually *done* or *out* and for all players when there is a `COMPUTER` type stage.

For managing changes in cards and tokens possession two special predicates were introduced: `movecard` and `movecoin`. They are filling the gap between performed actions A^{does} and state update function u. As the main idea between both predicates is similar we present here only a `movecoin` example (line 432). If relation `movecoin ?n ?from ?to` holds, this means that `?n` coins are added to `?to` coin location and subtracted from `?from` location. Multiple such relations can hold at the same time, but then their `?n` and `?to` arguments are the same. CGDL limitations implies that moving coins from multiple locations forces them to be empty, so in such cases `?n` is always set to sum of tokens in all `?from` locations. It is safe due to natural number arithmetic (subtracting from 0 is 0).

State update

Updating the current state given players' moves, i.e. defining u function is the most complex part of the translation (partial `next` code is shown in line 161). Every base predicate has its own set of updating rules, mostly using additional helper predicates. Sketch of the mechanics looks as follows. End of a stage occurs when all players have adequate statuses (`aOUT`, `aDONE` or `mDONE`), there is no player with any `ActionAvailable` left for the stage, or this is last turn of `ShuffleDeck` stage. If `endstage` holds, stage changes to the next stage in `stagesorder`. Similarly `CurrentPlayer` changes when `endplayer` holds. This depends on the player's actions performed, status and availability of the current stage actions.

Content of `Table`, `Deck` and `Token` is updated based on `movecard` and `movecoin` semantics. This requires several cases to examine, especially concerning taking cards from the deck without destroying its structure. If in the last turn, the player performed non-`OPTIONAL` action with some identifier, it is removed from the `ActionAvailable` set.

Updating `Shuffled` and `UnShuffled` predicates takes place ony during `ShuffleDeck` stage. If `does random (?shuffle ?card)` holds, then `?card` is subtracted from `UnShuffled` set and remembered as last `Shuffled` card. Predicate `Won` becomes true only when some player perform a `win` action.

Arithmetic and restrictions

Simulating CGDL requires declaring natural number arithmetic for numbers not greater then 100, which is upper bound for a goal value. We also assume that no card value or play value extends that number (if so, we can use the optional translation function parameter to increase maximal declared value). Every created GDL-II description contains the following arithmetic and boolean functions: `aplus1`, `asum`, `asub`, `amult`, `blt`, `bgt`, `bleq`, `bgeq`, `beq`, `bneq` (line 819).

Depends on *conditions* in $\mathcal{G}$, `tmplegal` declaration rules contain queries checking various *restrictions*. Such restrictions are helper functions encoding subtrees of CGDL conditions. For example `rKA ?n` holds if `?n` is the cumulative bet of all players (line 557), `rsamesuit ?card ?loc1 ?loc2` holds for every card from `?loc1` that has corresponding card with same suit in `?loc2` and `rsum ?loc ?n` holds if values of cards in location `?loc` sum up to `?n`. Function `rsum` is particularly complex and requires other helper predicates: `numberofcards ?loc ?n` which holds if in `?loc` there exactly `?n` cards, `holdcards_arity ?loc ?n` satisfied when `?loc` contains at least `?n` cards, and finally a whole predicate family `hold`n`cards ?loc ?card1 ... ?card`n which holds if there are n different cards in `?loc`. The complexity of solution rises from a need of reasoning about number of satisfied predicates. Example of `rsum` rule for $n = 2$ is shown in line 561.

Predicates `sees`, `terminal` and `goal`

The predicate `sees ?player ?percept` defines $\mathcal{I}$ relation, i.e. predicates perceptible by given player. Every player has full knowledge about his hand, his private coin location, current stage, current player and all players' statuses. Table locations and bet locations are visible to all players. Knowledge which action was performed by last player is also common. Details of the action (e.g. what cards player took from the deck or facts from `movecard`/`movecoin` relations) are perceived only by a player who made the action or by everyone if action visibility is set to `visible` (line 471).

Definition of `terminal` relation depends on several rules checking if: all players are *out*, some player made `win` action or `Stage EndGame none` is reached (line 487). Computing g function is divided into two cases. If player had won using `win` action he got 100 and all other players (including `random`) got 0 (line 490). Otherwise the player's score is computed according to CGDL game specification using game-dependent values for every possessed token (`t`) and card (`c`) plus bonus for not being *out* (`s`).

6.1.3 Translation Properties

We state the main properties of the translation, concerning its correctness and complexity.

Correctness

Definition 6.1.3. *Let S be the state of CGDL game $\mathcal{G}$. Then state S' of game $\mathcal{F}(\mathcal{G})$ is called **corresponding**, i.e. $S \doteq S'$ if: both states have the same content and ordering of deck and every hand location, table location and token location; actual number and type of stage; current player; player's statuses and set of available actions.*

Theorem 6.1.4. *For every CGDL game $\mathcal{G}$ presented construction of translation $\mathcal{F}$ satisfies the following.*

1. *The game $\mathcal{F}(\mathcal{G})$ meet all syntactic requirements of valid GDL-II description.*

2. *The first non-technical state of $\mathcal{F}(\mathcal{G})$ is corresponding to the first state of $\mathcal{G}$ assuming identical deck ordering.*

3. *For every non-technical game states $S \doteq S'$ there is an isomorphism between joint legal actions from both states.*

4. *For every non-technical game states $S \doteq S'$ and joint actions $A \doteq A'$, for all sequences of joint moves $\langle A', A'_2, \ldots, A'_k \rangle$ such that $S_i = u(A'_i, \ldots u(A', S') \ldots)$ and S_k is non-technical but for all $i < k$, S_i is technical, S_k is corresponding to state S after applying actions A. Such sequence always exists.*

5. *If $S \doteq S'$, S is terminal iff $S' \in t$, goal values match for corresponding players.*

Table 6.1.1: Results of example games transformation. Visualize dependence between complexity of CGDL description (number of players, table locations, stages and rules) and resulting GDL-II code. As measurement we took number of rules and predicates used to express the game. The number of predicates and rules for base predicates are shown separately.

Game	CGDL code				GDL-II code			
					predicates		rules	
	P	T	stages	rules	base	all	base	all
Uno	2	1	2	7	10	61	38	245
Uno	3	1	2	8	10	61	39	254
Uno	3	2	2	8	10	61	39	256
Jednadvacet	1	1	3	8	10	60	39	255
Kuku	3	1	2	9	10	61	40	251
Blackjack	3	0	5	12	10	63	43	361
Blackjack	3	1	5	12	10	63	43	363
Poker	3	2	13	30	10	68	61	426
Poker	4	2	13	32	10	68	63	437
Poker	5	2	13	34	10	68	65	448

Proof. It is easy to verify that the translated code meets all syntactical requirements of proper GDL-II game including stratification and allowance. Also that, assuming identical result of deck shuffling, initial game states (first non technical state in case of $\mathcal{F}(\mathcal{G})$) are corresponding. Let us take any state S. Because the translation is constructed in a way that base predicates of $\mathcal{F}(\mathcal{G})$ are strict equivalents of all information needed to encode S, a corresponding state S' can be created by setting $S'^{\texttt{true}}$ properly.

The set of joint actions legal from state S is in fact the set of current player's or computer's actions that conditions are fulfilled. For this player/computer every action has its own GDL codification with the condition codification in the legal rule body. An action will be legal for this player iff it is available to him in S, where equivalent of computer is `random`. As in S' every player needs to make action, all the other players' actions are set to `NOOP`. After updating state S' given joint actions A' resulting state can be technical. But this means a special case occurs, e.g. there should be move of an *out* player. If so, all legal joint actions from $u(A', S')$ leads to the same next state. If it is technical we continue this process, which will always end. Proof that the first non-technical state is corresponding to S updated using A proceeds by analyzing the construction of `next` rules following possible cases.

Complexity

We implemented a program that applies the translation function for given CGDL game description. To measure practical complexity of resulting game we provided a series of experiments. We applied transformation for *Poker*, *Blackjack* and *Uno* games from [50] (with slightly different changes) and for games *Kuku*[1] – simple game about transferring cards between players to get 3 cards with same suit or same number, and *Jednadvacet*[2] – a game similar to *Blackjack* but with different card values and special rule that having two aces gives instant win. The results of the experiments are shown in Table 6.1.1.

[1]http://boardgamegeek.com/boardgame/110483/kuku
[2]http://cs.wikipedia.org/wiki/Jednadvacet

Sizes of translated games gives a picture of complexity of their rules, which also affects speed of computing GDL game states. The data also show that a number of predicates is independent on number of players and table locations, number of base predicates rules depends linear on number of players and both these values have influence on overall number of rules.

We state that theoretical complexity of a translated game is described by

Theorem 6.1.5. *Let $\mathcal{G}$ be a P player CGDL game description of the length N (where length is the sum of the number of stages, rules, and card/plays value mapping entries) with T table locations. Then the number of rules in the GDL-II game $\mathcal{F}(\mathcal{G})$ is $O(P+T+N)$ and the number of predicates can be bounded by a constant.*

Proof. The number of rules to encode arithmetic expressions depends on the maximal possible number occurring in game (even if it exceed default 100 it is still bounded by the maximal computer integer value). This means that we can bound the number of arithmetic rules (and thereby number of predicates) by a constant. The number of rules computing conditions and restrictions depends on the constructions used in game. Because there is a finite number of such combinations, and duplicates do not increase the number of rules, this also can be bounded by a constant. Handling the base game state requires a constant number of predicates and the number of rules depends on game length. It is enough to use 4 predicates to handle legal moves, and the number of rules is linear on the number of rules in original game. The number of additional helper predicates (like `movecard` or `movecoin`) can be bounded by a constant and there is constant number of possible rules to handle them. Predicate `sees` requires the number of rules linear on the number of table locations. There is fixed number of predicates that are constant, i.e. roles, declaration of card values, locations, etc. Each of these predicates needs a linear number of rules to be declared. The rest of the generated code, including declarations of predicates `terminal` and `goal`, uses a constant number of rules. As every case has been examined we can state that the number of predicates in translated game is $O(1)$ and the number of created rules is $O(P+T+N)$.

6.1.4 Summary

The quality of solutions for GGP problems heavily depends on the language that describes the game. Although the target should be to describe as many games as possible, very general languages causes two major problems. First is that providing a good playing algorithm is much harder if the type of the game is unknown. Second, that understanding and maintaining game description is also far more complicated. Other extreme case is a language that can describe only certain type of strictly declared games, but it is high level and uses domain-dependent constructions. In this case maintaining game description and implementing better playing algorithms is simpler.

In this section, we have studied the relation between both of these approaches. On the one hand we took the most popular and the most general first order logic General Description Language with Incomplete Information, which can describe any finite, turn-based, n-player game. On the other hand we took recently developed Card Game Description Language, which is high level language to describe card games with bets in a way that allows a genetic manipulation on the game structure.

We constructed a translation from any game in CGDL language into the corresponding game in GDL-II, described details of this translation, proved its correctness and checked its complexity both theoretically and empirically. Although the size of translated game description is linear (in the sense of the game rules) complexity of simulating the game basing on pure GDL engine without compilation [99] or calling external code [192] can be computationally too hard to be performed in reasonable time, due to complicated form of queries.

To improve especially difficult cases like randomizing long sequences or computing amount of satisfied relations, the meta-language definitions with at least arithmetic expressions should be provided. This is not desired solution because pure declarativeness is seen as GDL strength. Such approach was successfully applied in *Toss* language (Section 2.4.2). We need to realize that some limitations of declarative approaches exist not in theoretical aspect of language power but complexity of state computations. Most GDL engines are very vulnerable to things like style of predicates declarations or even queries order.

Our translation can be used as benchmark tool for improving GDL players by comparison with CGDL playing algorithms. If some successful solutions could be transferred to more general approach it would be a step to reduce the gap between GDL-based and game-specific reasoners.

An interesting feature is a possibility of performing genetic operations on CGDL game code. Although CGDL was designed specifically for this case, we think it is also possible to develop this feature to GDL. GDL is a complicated language in terms of validity and semantics, but syntax rules are simple enough to make it feasible. In fact, artificially evolved GDL games could have interesting influence on creating GGP agents. They should from now on be able to play really *any* game, even codified in strange style and without common sense, not only well behaved adaptations of human games.

6.2 Comparing Stanford's GGP and Simplified Boardgames

In this section, we assess the progress of General Game Playing by comparing some state-of-the-art GGP players with an exemplary program dedicated to playing Simplified Boardgames using temporal-difference learning. The research presented is also the first step in creating a standard test class for measuring performance of GGP players.

Through a decade of developing general problem-solving approaches, a great advancement has been achieved. New approaches, algorithms and data structures have been used to improve players' efficiency. Despite that fact, still GGP players are not a match for the specialized programs designed to master one game only. This, however, shows a lack of some „in-between" problem that can be used as comparison, being on the edge of GGP programs capabilities. This should be a mid-range class of games, much tighter than that described by GDL, but wide enough to be challenging on its own.

In this research we are using Simplified Boardgames as a comparison class for GGP players. We designed a program that can play previously unknown boardgames in a manner defined by the GGP protocol. Our player uses the alpha-beta min-max algorithm and evaluates states by material and position analysis using piece tables and piece-square tables. Proper weight assignment for the evaluation function, which was stated as an open problem for Metagamer player ([150]) is solved here by applying temporal-difference learning [200]. This was one of the ideas mentioned by Pell, and it was already applied for the material analysis in some non standard Chess variants [37], as well as for standard Chess [8]. We provided several experiments opposing our simplified boardgames player against some of the top GGP players, and comparing efficiency for various styles of learning.

6.2.1 Reinforcement Learning in Boardgames

The key idea of *reinforcement learning* [200, 201] applied to games, is to learn the evaluation function by increasing the weights of the features that positively contributed to good choices of moves, and decreasing the weights of features that lead to bad moves. The question of which moves are good and which moves are bad is solved in a straightforward way by the assumption that all moves leading to a won game are good, while all moves leading to a lost game are bad. If a learning set is big enough, the statistics guarantees that favorable positions will occur more frequently in won games thus, the features that describes them will have higher weights.

Evaluation function

The standard approach to evaluate a state of a game is to use some set of features $\mathcal{F}$ with a linear evaluation function of a form

$$E(p) = \sum_{f \in \mathcal{F}} w_f \cdot f(p), \tag{6.1}$$

where each individual feature f is a function assigning to each position p a real number, and w_f is a weight assigned to that feature. The task of learning the evaluation function for a given set of features is reduced to finding the weight values.

In our experiments we use two types of features that can be safely applied to all games in the Simplified Boardgames domain: material (piece values) and position (piece-square values). For every player, we create a *material feature* for each available type of piece, counting the number of pieces of that type for each side. Since in Simplified Boardgames the initial position does not have to be symmetric, we assume that the importance of the same piece for each player can differ, thus it should be evaluated separately. For both players, for each type of piece we create $n \cdot m$ *position features* (assuming the board size is $n \times m$). The feature value is 1 if the appropriate square is occupied by a piece of that type, and 0 otherwise.

To evaluate a given position from the perspective of the player who has to make a move, we sum weighted values of his pieces' features and subtract weighted values of the opponent's pieces' features. For example, if a white-to-move game position p contains two white rooks on squares (x_1, y_1) and (x_2, y_2), and a black rook on the square (x_3, y_3), then the position evaluation for white is:

$$E(p) = 2 \cdot w_{WR} - w_{BR} + w_{WR(x_1,y_1)} + w_{WR(x_2,y_2)} - w_{BR(x_3,y_3)},$$

where WR is a material feature containing the number of white rooks (analogously BR contains the number of black rooks), and $WR(x, y)$ is a position feature equal to one only if a white rook stands on the square (x, y) (analogously $BR(x, y)$).

Temporal-difference learning

The reinforcement learning algorithm most frequently used in game playing is *temporal-difference learning* [214]. The most famous example of its successful usage is Tesauro's backgammon player which achieved a world champion level by learning only from self-play [215]. Faster convergence of TD(λ) algorithm is achieved by learning using differences between successive position estimations rather than the final score only [144]. More precisely, for given feature f, its weight update at time t is computed by the following formula:

$$\Delta w_{t,f} = \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_{w_f} P_k, \tag{6.2}$$

where P_t is a position evaluation at time t, α is a global learning rate, $\nabla_{w_f} P_k$ is the partial derivative of P_k with respect to w_f, and λ is a parameter providing an exponentially decaying weight for more distant outcomes, minimizing their influence on the current position.

We want our predictions to reflect the probability of success, so we transform our raw evaluation E given by (6.1) into a sigmoid function

$$P(p) = \frac{1}{1 + e^{-\omega E(p)}}, \tag{6.3}$$

with parameter ω influencing the curve steepness. Then, the weight update $\nabla_{w_f} P_k$ for the feature f can be computed using the formula

$$\nabla_{w_f} P_k = f \cdot P_k \cdot (1 - P_k). \tag{6.4}$$

In our experiments we use the TDLeaf(λ) algorithm [8] which was designed to be effective on chess-like games where move selection is based on deep searches. The key idea of this algorithm is to take the principal variation leaf node obtained from the search tree rooted in p rather then position p itself. We also use the Temporal Coherence algorithm [9] to dynamically adjust learning rate α for each feature separately.

6.2.2 Experimental Setup

We have developed a program that can play any given, previously unknown, regular boardgame (as we sometimes call Simplified Boardgames to emphasize its regularity) according to the General Game Playing framework. This allows us to compare our *Regular Boardgame Player* (*RBg-Player*) with GGP players. We have chosen a few games and encoded them both in GDL and in regular boardgame format. Then we run matches using a standard GGP game controller. Players had some *startclock* time to prepare themselves to play and after that, they had to alternately perform moves, not exceeding the given *playclock* time. During one player turn the other one could perform some computations, but it had to sent the special `NOOP` move (which was the only legal move in GDL version) within the timelimit.

Games

Experiments have been performed using the following games, differentiated due to their size (depth and branching factor) and importance of material/position advantage:

Regular Chess is a standard game of Chess cut to the Simplified Boardgames framework. Thus there is no promotion nor check constraints; 50-move rules and draw by position repetition do not apply; special moves such as castling, en passant, two step pawn move are not allowed. To win the game, one has to capture the opponent's king or reach the opponent's backrank with a pawn. The game is considered as draw after 200 turns.

Regular Gardner is one of the minichess variants, played on a 5×5 board with a 100 turn limit. The detailed description and game rules are presented in Section 2.1.2.

Regular Asymmetric Los Alamos is a slight modification of the standard Los Alamos 6×6 minichess variant. White pieces are the same as in the standard version (backrank from left: rook, knight, queen, king, knight, rook), but the black player has two bishops instead of two knights. Winning conditions and other rules are the same as for the Regular Gardner, with the exception that the turnlimit is set to 120.

Regular Escort Breakthrough is a modified version of the standard 8×8 Breakthrough. Every player has one king on his backrank, and a row of pawns before it. A pawn may move one space straight or diagonally forward if the target square is empty, and only diagonally forward if it is occupied by enemy's piece. A king may move as the standard Chess king, except that it cannot move backwards. To win, one has to reach opponent's backrank with the king. If no one wins within 60 turns, the game ends in a draw.

Search engine

The basics of the search algorithm used in our RBgPlayer is negamax, which is a variant of standard alpha-beta that relies on the zero-sum property. We used the parallel iterative deepening algorithm complemented with a quiescence search and the killer-move heuristic. We use non-persistent transposition tables, removing unattainable positions after captures.

During the RBgPlayer's turn, we start the iterative deepening search from depth 3, with parallelization occurring on the highest level only. When the time limit is reached, the embedded timer terminates searching and the result from the last fully computed depth is returned.

Learning features weights

The learning phase occurs during the opponent's turn and in preparation time before the match begins. We perform a series of constant depth playouts, without quiescence search, updating the weights after every playout. We learn weight updates separately for the white and black player, and we use the average as the final update.

We do not use randomness during the move searching phase to prevent game repetition, which is often an important issue during self-play learning. The main reason is that this would significantly slow down searching in situation where the time is crucial. Also, our learning function is self adapting during the game's course, depending on the opponent's choice of moves, which makes the self-play part safer for over-fitting errors.

If the time and game tree allow, we increase the playout depth (starting from 2), remembering times of longest playouts. This allows us to run more accurate simulations with assurance there will be enough of them (our algorithm tries to have at least 20 simulations per turn).

The real danger during the learning process, making playout repetition most probable, is too fast minimization of the learning value α, set by the Temporal Coherence algorithm. To prevent this, we update α values not after every simulation, but after every ten simulations. According to our observations, this is enough to ensure that decreasing α is a result of a real weight convergence, not only coincidental cases.

Computing initial weights

Normally, computing proper weights takes a very long time. For example, according to experiments conducted in [37], piece values stabilized after 250-500 full matches, depending on the game. Because during a GGP match we do not have time for such an exhaustive learning process, we have developed a heuristic algorithm to compute initial weights based mainly on piece mobility and winning conditions.

As the first step, we identify how aggressive the game is, by counting ratio between displacement moves, captures and self captures; and then predict the content of the board during a middle game. The result of this prediction is given in a form of probability that a square will be empty, occupied by white or occupied by black. Then, for a given move rule, we calculate its score, equal to the probability that this move will be valid. This is done by multiplying probabilities for all steps except the last one. If the final step is capture, we multiply the move score by 2, assuming that the possibility of capturing enemy pieces can always give advantage.

The initial weight for a position feature of a given piece is the sum of the scores of the moves that can be performed by the piece from the given square, divided by the size of the board $(n \times m)$. The material feature weight of a piece is just the sum of all its position features weights.

The weights computed so far are now modified taking into account winning conditions. Let s be the constant describing the winning score in min-max nodes evaluations. If some player wins by capturing k pieces of a given type, then this piece's value is increased by $\sqrt[k]{s}$. This results in better protection of pieces which losing has more severe consequences. If some player wins by reaching a goal square by a piece of certain type, and he has j such pieces in the initial position, then the value of this piece is increased by $\sqrt[j]{s}$. In this case, we also modify the piece-square values. The values of goal squares are increased by s. The value of each square from which a goal square can be reached in $l \leq \frac{\max(n,m)}{2}$ moves, for each move, is increased by $\frac{\sqrt[l]{s}}{nm}$ multiplied by the probability of the move. This rewards advancing such pieces. However, because bonuses are not too large and added only in a limited board area, it prevents the blind march, which usually ends in capturing these lonely, valuable pieces.

Reference GGP players

We compare our min-max based RBgPlayer with two GGP programs, both using variants of the Monte Carlo Tree Search algorithm which, since its first success in IGGPC 2007, is the most popular algorithm for GGP players.

Dumalion is one of the top 8 players in 2014 International General Game Playing Competition. It is based on the GDL compilation mechanism (described in Chapter 7) and parallelization of the UCT algorithm over the cluster of computers. The main program is written in Java and running on a computer with Intel(R) Core(TM) 2 Quad CPU Q9300 2.50GHz 4 cores and 4GB of RAM. GDL reasoners are compiled into the C++ language and during this test run on 25 computers with Intel(R) Core(TM) i7-2600 CPU 3.40GHz 4 cores and 16GB of RAM.

Sancho, by Steve Draper[3] and Andrew Rose, is the winner of the 2014 International General Game Playing Competition. The program uses the UCT algorithm and propositional networks [31] as a reasoning engine. It is written in Java, and operates on Intel(R) Core(TM) i7-4770K 3.50GHz CPU 4 cores computer with 32GB of RAM.

Unfortunately, no publications are available revealing more details about this player and describing its method. Therefore, we provide a short description. The Sancho player has implemented a piece detection method, and after estimating piece values, it uses this information to help Monte Carlo searching. The algorithm detects predicates that may represent a piece and correlates the number of such pieces with the goal score. If a correlation is above certain threshold, the player treats the predicate as a piece. A number of random playouts are run to establish the pieces values. The obtained piece value depends on its mobility and correlation between a number of pieces and playouts results.

A heuristic evaluation function using pieces values is used as the initial score for a new node added to UCT tree. The node's visit counter is initialized according to some weight function. After new node creation process, all simulations and score's updates behave like in the standard UCT algorithm. This makes that the initial distribution of playouts heavily depends on the heuristic knowledge. If the initial heuristic reflects the real importance of the pieces, it helps to guide search into promising parts of the tree and not to waste simulations. If not, the values of the scores contradictory to the observations are gradually revised by the successive simulations.

To complement the technical information above, we also describe our RBgPlayer specification. The program is written in C# using .NET Framework 4.5, and runs on a single machine with Intel(R) Core(TM) i5-2410M CPU 2.30GHz 4 cores and 12GB of RAM.

6.2.3 Results

We ran two sets of experiments, one of them using precomputed initial weights, while the other used only pure learning to determine the proper weights. In the latter case, all material features

[3]I would like to thank Steve Draper for very useful comments, opinions and great support during the experiments.

weights were initially set to 1, while all positional features weights were set to 0. We tested our RBgPlayer against Dumalion, Sancho without piece detection, and Sancho with piece detection enabled. We used two time setups: one similar to those that are used during GGP competitions for large games, with 3 minutes startclock and 1 minute playclock; and another one with shorter times (60 seconds startclock, 20 seconds playclock) used only for experiments with precomputed weights. To test every setup we ran 20 matches (10 with RBgPlayer as white, and 10 as black).

Table 6.2.1 shows the results of experiments with RBgPlayer using precomputed weights, while Table 6.2.2 contains the results of with pure learning experiments. A single result consists of three numbers: the first is the percent of RBgPlayer wins, the second the percent of draws, and the third is the percent of RBgPlayer loses. The last column contains the results against Sancho with piece detection on, while the column before, with piece detection off.

Table 6.2.1: Precomputed initial weights (wins:draws:loses percent)

Game	startclock, playclock	Dumalion	Sancho	
			det. off	det. on
Chess	180, 60	100 : 0 : 0	100 : 0 : 0	15 : 5 : 80
	60, 20	100 : 0 : 0	95 : 5 : 0	25 : 5 : 70
Asymmetric Los Alamos	180, 60	100 : 0 : 0	100 : 0 : 0	50 : 0 : 50
	60, 20	100 : 0 : 0	100 : 0 : 0	75 : 0 : 25
Gardner	180, 60	100 : 0 : 0	40 : 5 : 55	20 : 15 : 65
	60, 20	100 : 0 : 0	45 : 10 : 45	25 : 5 : 70
Escort Breakthrough	180, 60	55 : 20 : 25	75 : 10 : 15	35 : 30 : 35
	60, 20	35 : 15 : 50	85 : 10 : 5	35 : 5 : 60

Table 6.2.2: Pure learning (wins:draws:loses percent)

Game	startclock, playclock	Dumalion	Sancho	
			det. off	det. on
Chess	180, 60	100 : 0 : 0	95 : 0 : 5	0 : 0 : 100
Asymmetric Los Alamos	180, 60	100 : 0 : 0	80 : 0 : 20	5 : 0 : 95
Gardner	180, 60	95 : 0 : 5	10 : 0 : 90	15 : 10 : 75
Escort Breakthrough	180, 60	15 : 55 : 30	0 : 40 : 60	5 : 20 : 75

At first, one may observe that the table reflects precisely the playing level of GGP programs, which shows that Simplified Boardgames form a good base for a testing class for measuring the performance of GGP players. An expected trend observed in the table is that the GGP agents are playing worse as the game tree size grows. Escort Breakthrough has a larger branching factor than Gardner at the beginning, but after that, both values are similar. However, the tree depth in Breakthrough is much smaller, which means that random MCTS playouts are nearly two times faster achieving faster convergence and leading to definitely better results. The differences between Gardner, Los Alamos and Chess lie both in branching factors and game tree depths. These differences are reflected in our results. We can also observe that smaller timeouts, combined with the game tree depth, favor RBgPlayer. It seems that in such games with complex rules short lookahead based on the heuristic position evaluation leads to better results than the small number of simulations (especially in an endgame phase, which is often crucial).

If, as our main reference point, we take GGP champion Sancho in the mode with piece detection off, we see that Gardner is the game with probability of winning close to equal for this player and the dedicated regular boardgames player. In slightly larger Los Alamos, the GGP player chances to win plummet to zero. Gardner is still a complex, but relatively small game in human terms. A tempting conclusion that any larger game leads to dramatic drop in GGP players efficiency is not encouraging in terms of domain success. However, the fact that GGP players behave with every game like they are seeing it for the first time should be taken into account. Comparing the efficiency of Sancho against RBgPlayer without precomputed weights we can see a significant improvement in GGP agent scores. Without game-specific knowledge, RBgPlayer has too little time to gather reliable information about piece behavior to construct an accurate evaluation function.

The use of heuristic knowledge about piece mobility makes Sancho with piece detection behave more like a dedicated player. Note that Sancho possesses only very limited knowledge. It does not detect board or other concepts but only the pieces. Moreover, there may be games where pieces are not detected properly because of the fixed threshold in the predicate-score correlation test. Also, weight values may be challenged as based only on the random simulations. In spite of all these, this limited knowledge combined with engineering effort to make player computationally efficient turns out to be more then enough to have clear advantage over RBgPlayer in most experiments. Initial heuristic guidance over the game tree to explore its most promising areas without costly computations put Monte Carlo ahead of min-max with simple material and positional evaluation.

It should be also pointed out that the influence of heuristic evaluation strongly depends on the game tree size. In small games, sampling quickly dominates the initial heuristic values while in large ones, with computationally expensive simulations, heuristic evaluation remains dominant. Since UCT needs help exactly for large games this is very welcome behavior.

There is however no assurance that the method used by Sancho is accurate and safe from producing wrong results. What we have learned from our experiments, the weight assignment algorithm seems to have a tendency for overestimating the values of some types of pieces. It seems also to have problems with inequitable initial positions. For Asymmetric Los Alamos the heuristic does not work so well as in other cases.

Feature weights

Table 6.2.3 shows the comparison between the values of pieces (material features weights) calculated by the RBgPlayer's initial weight computation algorithm and the Sancho's simulation-based algorithm. For every game, the obtained values have been normalized by the assumption that the pawn value is 1. The values calculated by RBgPlayer are sums of the scores based on mobility and goal conditions. The goal condition part has nonzero value only for pawns and kings, and in such cases this value is put in the braces (thus for Chess the pawn score 1.0 is obtained by the goal condition evaluated for 0.38 and the mobility evaluated for 0.62). The Sancho player has made different evaluations for the Asymmetric Los Alamos rook depending on the side: for white the weight is 1.5, while for black it is 1.2.

Although weights assigned to the pieces differ in both approaches, they mostly coincide with the human judgment of which piece type is better. Scores relative to the pawn are usually smaller then given by humans (e.g. in Chess), but this is a result of changes in the rules of the game. The fact, that pawn reaching the opponent's backrank causes an instant win increases the pawn's weight. The Sancho player has a tendency to overrate knights, and in Asymmetric Los Alamos this even influences the rook score, as the simulations probably discover knight+rook as a complementary combination of pieces.

Table 6.2.3: Estimated piece values.

Piece	Chess		Assymetric Los Alamos		Gardner	
	RBg	Sancho	RBg	Sancho	RBg	Sancho
Pawn	1 (0.38)	1.00	1 (0.47)	1.00	1 (0.54)	1.00
Knight	2.50	3.69	1.97	2.80	1.57	4.34
Bishop	3.00	2.69	2.01	1.40	1.51	1.24
Rook	4.18	2.59	2.88	1.5/1.2	2.20	1.25
Queen	7.16	5.73	4.90	3.30	3.70	2.16
King	161.60 (158.48)	2.99	150.46 (147.71)	2.00	138.59 (136.24)	1.12

In Escort Breakthrough, the precomputed heuristic grants a very high value to the king, as it is the only piece allowing victory. This prevents RBgPlayer from losing its king to a much greater extent than in the pure learning case. Also, because without a king, a match normally goes on until turnlimit is reached, the small search horizon is enough to compete against UCT, which requires full length playouts.

Learning

The main difference between the process of learning weights during our experiments and that presented in [8, 37] is that in the GGP scenario in which we work, the weights values are updated from turn to turn, and a new game always begins without knowledge about previous experience. Thus, the setup is more difficult, but there are also advantages. For example, our player has possibilities to adapt to the current situation.

It is well known that, e.g. in Chess, weights of some pieces are different depending on the game phase (beginning, middle, endgame) or other pieces on board (e.g. two bishop bonus which is usually a part of a Chess evaluation function). In the learning setup based on the material and positional evaluation, it is impossible to embed such nuances. However, as learning proceeds after every turn, the weights can automatically adapt to better suit the current situation. During our experiments, we have observed that material evaluation for pieces has been changing between turns to reflect their power in the current state. A practical problem is time for such learning to be sufficient. In Chess experiments, during the startclock RBgPlayer was able to perform 10-30 learning simulations, and as the vast majority of them ended with a draw no really significant weight changes were done.

This is not a problem in the precomputed initial weight values scenario, but in the learning only scenario piece values in Chess began usually to differentiate around the 8th-12th turn. Earlier, all piece types were evaluated as nearly equal in strength. During the middle game only a few Chess simulations can be done during the learning phase. The number of middle phase simulations in Garder was about 15-20. This is not enough to obtain the best suited values, but enough to slightly push the weights in the right direction, and this works well in the scenario with precomputed initial weights. An interesting issue concerns endgame situations with some unused pieces. In this case, such pieces tend to have negative weights, which is often reasonable, because they block the mobility of the other pieces (although still contradictory with human evaluation of such situations). This occurs only when weights depend on pure learning. With precomputed weights, the values are larger and all material features are more stable.

Other issues

An interesting aspect of a dynamically learned evaluation function is the usage of transposition tables. Although transposition tables speed up computations, the retrieved values are often outdated due to the reevaluation of weights. However, the value stored for a state has a search depth level assigned, and it is used only if the current searching depth is not greater then that. This ensures that, during the move searching phase, the state evaluations will be recomputed using the current heuristic, and they will override values stored during shallow learning playouts. Experimentally we did not observe disadvantages of using transposition tables, while using them usually results in reaching a deeper level in iterative deepening alpha-beta used by RBgPlayer.

Another important issue is GDL code efficiency. In GDL, there are many ways to encode a single game, and the way it is done has an essential impact on the number of computations needed to calculate a state, and so, on the performance of GGP players. During our experiments we have been using the optimized style of codifying chess-like games [105], presented by Alex Landau the author of the Alloy GGP Player. At the beginning we had been using a less efficient coding style, which resulted in larger number of losses by Sancho (in some cases even twice).

6.2.4 Conclusions

In this section, we assess the progress of General Game Playing by comparing some state-of-the-art GGP players with an exemplary program dedicated to play in a smaller class of games called Simplified Boardgames. We have developed a min-max based player and confront it with two top GGP players on a number of games. Our player takes advantage of the fact that the game played is always a regular boardgame and evaluates game positions using material and positional evaluation functions, with weights learned during the play.

The obtained results lead to the conclusion that GGP programs play at an acceptable level on smaller games. Due to the careful, efficiency-oriented design, they are able to produce millions of simulations which pay-off when using MCTS methods. However, this approach disappoints when a game tree becomes bigger, in which case a relatively simple knowledge-based approach achieves better results. The experiment shows, where a limit of GGP players capabilities lies.

The outstanding results of Sancho with implemented boardgame detection demonstrate the benefit of having knowledge about essential game properties. This contradicts conclusions formulated in [206]. Based on this observation, we think that a key to GGP players improvement is a proper game type detection. It is worth noting that from the very beginning of GGP the natural division of games into single player and multiplayer is applied, and nearly all GGP players use special strategies for single player games (DFS, reduction to ASP [216]), different from those for multiplayer ones.

It may appear somewhat against the idea of generality, but defining GDL class as a union of specialized subclasses seems to be a promising research direction for improving GGP players efficiency. Such an approach shifts the burden into the following tasks: defining the proper set of classes, detecting game class membership, developing algorithms for particular classes. A proper class of games should be narrow enough, to allow players to use specific game type knowledge, but wide enough to cover a large part of the GDL class. It should be also possible to detect whether a game belongs to this class based on the properties of GDL predicates.

It may even be argued that the general approach based on straightforward UCT heuristics (like MAST, PAST, etc. [47]) and software engineering improvements will reach its limits soon. Starting to classify games using a multilevel hierarchy, and introducing a mechanism for detecting game classes seems unavoidable for getting further progress. It is rather obvious that, like in Problem Solving, the more general the method the weaker the performance: a general purpose algorithm will always be worse than a good algorithm designed specially for a given task. Of course, detecting game classes may be suitably combined with the general knowledge-based approaches (like that in [74]).

In accordance with the long term research goals stated above, we would like to improve the boardgames detection and playing algorithms in MCTS GDL-based players. In particular, we plan to make more efficient use of the knowledge about pieces in MCTS search.

We also would like to establish the Simplified Boardgames class as a standard test class for GGP. To enable easy testing, we are going to provide an automatic translation system from Simplified Boardgames into GDL (in a manner similar to that recently proposed for Card Game Description Language, see Section 6.1) producing efficient GDL code. The present research may be considered as the first step in this direction.

Finally, there are several interesting issues about improving regular boardgames player, including generation of opening/endgame libraries, and using more sophisticated evaluation function (including e.g. threats counting).

7 Compilation

For the General Game Playing systems it is crucial to have an efficient reasoning engine able to perform fast game state updates, and thus fast game simulations. The correlation between the strength of play and the amount of the state space examined per turn is well known. Especially the quality of the Monte Carlo based systems strictly depends on the number of simulations performed.

In this chapter we present our approach for compilation of the Stanford's Game Description Language (Section 2.2.1). Reasoning in pure GDL requires expensive logical inference usually done by the high-level interpreter or built-in Prolog engine. Our method translates GDL code into an intermediate structure describing detailed computation order, which is then translated into C++. In general, mentioned intermediate structure can be straightforwardly translated into any target language. Because of used optimizations, it produces very effective reasoners that can compute game states faster than other so far known approaches. Presented compiler is a part of GGP player Dumalion, which has been classified in the top eight during International General Game Playing Competition in 2014.

The next section provides necessary background and describes current state of the art in development of the GDL reasoners. Step by step details of our construction are presented in Section 7.2. Section 7.3 contains overview of experimental results. We conclude in Section 7.4.

The content of this chapter is partially based on the paper [99].

7.1 Fast Reasoning in GDL

Increasing the performance of logic inference engine required to successfully play GDL-based games is a non-trivial yet very profitable task. Right optimizations and careful engineering can dictate which GGP player is better, even regardless of the (slightly) worse AI algorithm.

Currently most wide-spread simulation based approach [44, 45] relies on the Monte Carlo Tree Search algorithm. It uses random game simulations, which means that its effectiveness in approximating min-max game tree strongly depends on the number of the simulations performed. To achieve faster convergence, domain-independent knowledge may be used to guide the simulations [191]. However, even some of the non-MCTS knowledge-based approaches made benefits from game simulations, e.g. in [26] where simulations are used to tune heuristic evaluation functions.

Let us recall that GDL is a strictly declarative and rule based language, which means that gaining information about a game state is equivalent to applying rules and extending the set of holding (true) facts. Listing 7.1 shows a part of the *Goldrush* game description from Dresden GGP Server [70], which we will be using as an example throughout the rest of the chapter (the complete rules of the game are presented in Appendix D).

Listing 7.1: Part of the *Goldrush* game GDL code.

```
(role Green) (role Red)
(init (OnMap Green 1 1)) (init (OnMap Red 7 7))
(init (OnMap Obstacle 1 6)) (init (OnMap Obstacle 2 4)) ...
(init (OnMap (Gold 2) 1 7)) (init (OnMap (Gold 1) 3 6)) ...
(init (OnMap (Item Blaster 3) 7 4)) ...
(init (OnMap (Item Stoneplacer 3) 1 4)) ...
(<= (legal ?r (Move ?nx ?y))
    (role ?r) (true (OnMap ?r ?x ?y)) (InBoard ?nx))
    (or (+ ?x 1 ?nx) (- ?x 1 ?nx))
(<= (next (OnMap ?r ?x ?y))
    (role ?r) (does ?r (Move ?x ?y)))
(+ 0 0 0) (+ 1 0 1) (+ 2 0 2) (+ 3 0 3) ...
(<= (- ?x ?y ?z) (+ ?y ?z ?x))
(InBoard 1) (InBoard 2) (InBoard 3) ... (InBoard 7)
```

Predicates that are arguments of `init`, `true`, and `next` can be considered as a minimal set of predicates enough to restore all information about the state. We will call them as the *base predicates*. This means that the full game state (the *view* of a state) is a set of facts closed under application on the *base facts*, and the next state is computed based on the previous full state and the players actions.

This leads us to the notion of the reasoner. This is an essential part of every player, allowing it to shift the state based on the information from the Game Manager. During competition, GM sends to a player only moves made by all players, so computing the next state, legal moves and so forth should be made at the player's side. In other words the reasoner is an implementation of the game loop (Fig. 7.1.1) described by the game rules.

7.1.1 Reasoner Implementations

The most common way to implement the reasoner is to use a Prolog engine built into another programming language. In [14, 176], various approaches of GDL to Prolog conversion are examined based on the example GGP players: CadiaPlayer [13, 48] written in C++ and using YAP Prolog [29] for state reasoning, and Fluxplayer [177] written almost entirely in ECLiPSe Prolog. The benefit from such Prolog-based approach is that rewriting GDL into Prolog requires only syntactic modifications, so it results in simplicity of implementation. However, using a very general inference engine to compute the rules of strictly given form is a drawback causing lack of computation speed.

The problem of playing general games is so computationally difficult that many of the players use distributed architecture to support parallel computations on multiple machines [77, 126, 127, 135]. Complementary approach to parallelism is speeding up the process of reasoning itself.

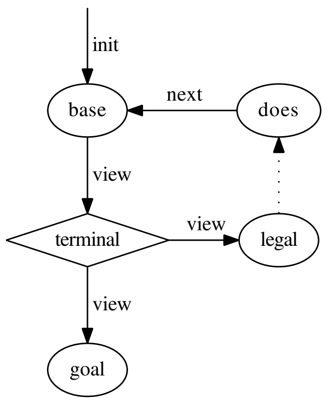


Figure 7.1.1: Game loop: the *view* of a state is computed based on the current *base* facts. Then either the game ends or, given the actions of the players, the next *base* should be computed.

Some approaches have been made to embed logic resolution engine into higher-level language and use it to interpret the given GDL code. This does not necessarily results in better efficiency. If the resolution engine is not well optimized, the overhead from the interpretation process can overcome the potential benefits, thus the engine can be less efficient then Prolog.

The most obvious method of increasing the performance of the reasoning engine is to compile GDL directly into the other language. Straightforward rewriting of GDL code to C++ language, done in a top-down manner, was described in [235]. Reported speedup factor in comparison to YAP Prolog has reached 22 for some games. However, the system is not able to handle game descriptions with recursive terms. Java-based version of this approach, embedded in the GGP player Centurio, can be found in [135]. Speedup factors up to three, compared to the Prolog based version of Centurio, have been reported by the authors.

The other project GaDeLaC, the forward chaining GDL to OCaml compiler with some optimizations, was presented in detail in [167]. The results were mixed, as for some games GaDeLaC performed considerably worse than YAP Prolog. This approach has been further extended in [184], where the significant improvement of the results has been reported. GDL interpreter using OR-AND trees, a data structure to perform optimized logical operations, and producing C++ code has been described in [203, 204].

Except the compilation, other worth mentioning methods of improving reasoner efficiency consists of using propositional networks and instantiation. Propositional networks [31, 182], while result in a significant increase of representation size, can speed up reasoning by several orders of magnitude [195]. This approach is widely used by the students during Coursera online GGP course [62], and it is embedded in the GGP-Base project [185].

Instantiation of the game description is the process of grounding all possible terms to reduce query time during the reasoning. Comparison of two methods for obtaining the supersets of reachable state atoms, moves, and axioms, using Prolog and dependency graphs, has been presented in [91]. In [232, 233] it has been demonstrated that performance of instatiation can be improved using tabling engine built in a Prolog interpreter. Efficient grounding technique, by transferring GDL code into an answer set program ([7]), has been recently presented in [175].

In the following section we will describe in details our method of constructing GDL compiler. We use combination of a few ideas such us careful ordering of computations, optimizing control flow, and designing dedicated structures to perform queries. All of these, result in a significant improvement in efficiency compared to the methods used before.

7.2 GDL Compiler

Our compiler takes as an input the game rules written in GDL and outputs a structure called *compilation plan*, decoding the computation strategy for the reasoner. The plan can be then translated to some efficient low-level programming language like C/C++. In this section, we describe all major steps of constructing the plan. Our general aim is to achieve better efficiency of computing game states by the resulted reasoner. Looking to a process of playing GDL game, it can be considered as an usage of a database. The predicates are containers with some facts and we have to perform reading (queries) and writing (insertions) to these containers. Our method uses such techniques as flattening domains, optimizing data structures for containers, and reordering operations, which are mostly apart from the target language.

7.2.1 Calculating Domains and Flattening

The first thing we need to do, after parsing GDL to some abstract tree structure, is to compute domains of the predicates' arguments. Let P be a predicate. Because predicates in GDL can be nested, every occurrence of P form a tree of its arguments. In that case we can describe such occurrence as a function from vectors encoding positions in tree to arguments symbols, so e.g. position of `3` in `OnMap (Item Blaster 3) 7 4` can be described as $\langle 0, 2\rangle$ (positions at every tree level are enumerated from 0). We want to calculate domain as a function that takes a pair of a predicate P and a tree position $\tilde{p}$ (possible for P), and returns a set of symbols that can occur in this position. Such domains are in fact a supersets of the real predicates' domains and also lose information about dependencies between arguments. However this is enough for the further calculations.

The method proposed in [91] requires computing set of dependencies where $(P, \tilde{p}) \rhd (Q, \tilde{q})$ ($\rhd$ is "depends on" relation) if and only if there exists a rule with P in the head and Q in the body, where at the positions $\tilde{p}$ and $\tilde{q}$ respectively, the same variable occurs. This means that every symbol in domain of $(Q, \tilde{q})$ should be also in domain of $(P, \tilde{p})$. In this approach calculating domains means resolving dependencies by extending appropriate domains until a fixpoint is obtained.

We improved this method to handle nested predicates and compute smaller domains. In our case extending domains include also domains of every subtree of variable occurrence. If $(P, \tilde{p}) \rhd (Q, \tilde{q})$ then for every $\tilde{q}''$ that has $\tilde{q}$ as a prefix (so $\tilde{q}'' = \tilde{q} + \tilde{q}'$ for some position vector $\tilde{q}'$), also $(P, \tilde{p} + \tilde{q}') \rhd (Q, \tilde{q}'')$. These dependencies must be dynamically computed because during the algorithm new predicates positions can be found.

Instead of $\rhd$ we use relation $\rhd_R$ where $(P, \tilde{p}) \rhd_R (Q, \tilde{q})$ if dependency is created by rule R in CNF form (note that every game described in GDL can be easily converted do CNF). Let $\odot$ be the operator of domain conjunction defined as $(\odot\, d_1 \ldots d_n)\ v = d_1(v) \cap \ldots \cap d_n(v)$. For every rule R we create set $\pi^R_{(P,\tilde{p})}$ containing every $(Q, \tilde{q})$ such that $(P, \tilde{p}) \rhd_R (Q, \tilde{q})$ holds. Then for every $\pi^R_{(P,\tilde{p})}$ we extend domain of $(P, \tilde{p})$ by $\odot\, d_i$ for $d_i \in \pi^R_{(P,\tilde{p})}$. This simulates conjunction that takes place in GDL rules and prevent domains from containing symbols unused in practice. The procedure loops for every pair $(P, \tilde{p})$ and finishes when a fixpoint is found.

To illustrate this algorithm, consider a subset of game rules shown in Listing 7.1. Calculating domains based on this example appoints the following domain of predicate `OnMap`:

$(\texttt{OnMap}, \langle 0 \rangle) \rightarrow$ `{Green,Red,Obstacle}`	$(\texttt{OnMap}, \langle 0,2 \rangle) \rightarrow$ `{3}`
$(\texttt{OnMap}, \langle 0,0 \rangle) \rightarrow$ `{Gold,Item}`	$(\texttt{OnMap}, \langle 1 \rangle) \rightarrow$ `{1,2,3,4,5,6,7}`
$(\texttt{OnMap}, \langle 0,1 \rangle) \rightarrow$ `{1,2,Blaster,Stoneplacer}`	$(\texttt{OnMap}, \langle 2 \rangle) \rightarrow$ `{1,2,3,4,5,6,7}`

Because of arguments nesting, the number of leaves in parse tree can vary for one predicate. But to effectively perform queries, we need to have predicates without nesting and with a fixed arity. To achieve this, we developed a notion of *flattened* predicate and algorithms to convert standard (nested) predicate to flattened form and to perform reversed conversion.

The arity of a flattened predicate is the number of leaves in the widest of arguments assignments found in domain calculating phase. Tighter occurrences of the predicate are then stretched using special non-GDL symbol `#nil`, and each variable occurrence is extended by introducing new variables with added suffixes to avoid ambiguity. From now on, each mentioned predicate is flattened. Conversion from flattened predicate to its standard GDL form is necessary when the player needs to send a move to the Game Manager, having a flattened move given by the reasoner. As we made proper algorithms, we can stand:

Lemma 7.2.1. *For every occurrence of a valid GDL predicate with a fixed domain, there exist its unique flattened form. There exists an algorithm that converts these forms.*

Proof. We know that in GDL each predicate (in its nested form) has a fixed arity. Let P be a predicate with domain considered as a mapping from the positions of arguments $\tilde{p}_i$ to the sets of possible arguments. The positions of the arguments (encoded as vectors of numbers) form a tree T_P.

Consider any occurrence $\mathcal{P}$ of P where the first symbol is x on the position $\tilde{q}$. The proof will be inductive over the height of $\tilde{q}$ in T_P (distance from the farthest leaf in a subtree). If the height of $\tilde{q}$ in T_P is 0, then x is a leaf. This means that it is constant so it can be rewritten as the first argument of flattened form $\mathcal{P}^*$. Then we can treat $P\ x$ as a new predicate with arity equal to the arity of P minus 1, and repeat this procedure.

Assume that we can convert to the flattened form all arguments to the height at most h. Let the height of $\tilde{q}$ be $h+1$. Let T_P^q be the subtree of T_P with q as the root, k be the number of leaves in T_P^q and n be the arity of x ($n \leq k$). We can rewrite x as the first argument of the flattened form and then use the induction assumption to flatten all its arguments and place them in a row. The number of already set arguments of $\mathcal{P}^*$ is some $r \leq k$. The next $k-r$ arguments will be set to `#nil`, because the flattened representation of nested argument must have the length equal to the number of leaves in its domain subtree. Again, the first argument of P can be cut and the presented reasoning can be repeated until all arguments from $\mathcal{P}$ have their flattened counterparts.

Converting from the flattened form to the nested form means rebuilding the tree matching arguments positions $\tilde{p}_i$ to symbols, and then adding necessary parentheses. Let $\mathcal{P}^*$ be a flattened occurrence of predicate P with the first argument x of arity n. If n is 0 then the matching argument position is $\langle 0 \rangle$ and no parentheses are needed. In other case, x matches the position of $\langle 0, 0 \rangle$. This means we can always match a nested position of the first argument. Now we can prove uniqueness of deflattening using induction over the number of arguments. The base step is done. Now let $\tilde{q}$ be the position of the previously found argument, and y be the current symbol. We have multiple cases depending on y type and arity of the preceding predicate.

If the arity of some symbol z at the position $\langle \tilde{q}[1], \ldots, \tilde{q}[|\tilde{q}|-1], 0 \rangle$ (where $\tilde{q}[i]$ is the i-th element of the vector $\tilde{q}$) is not greater than $\tilde{q}[|\tilde{q}|]$, we can say that the level of y is stable and it is the next argument of z. In this case we have the following. If y is a GDL constant, then its position must be $\tilde{q}$ with the last element incremented. If it is a nested predicate with a non-zero arity, then a new level of braces must be added, so the position of y is equal to $\langle\tilde{q}[1],\ldots,\tilde{q}[|\tilde{q}|]+1,0\rangle$. The last possibility in this case is that y is `#nil`. Then the argument behaves like a constant but its position is not actually put into the rebuilt tree.

Another case is when y is not stable and cannot be an argument of z. Then we consider position vector $\langle\tilde{q}[1],\ldots,\tilde{q}[|\tilde{q}|-1]\rangle$ and arity of symbol z' at $\langle\tilde{q}[1],\ldots,\tilde{q}[|\tilde{q}|-2],0\rangle$. We can repeat this procedure and at some turn r we will find that y is stable for some z^r (or we reach the root level which means $z^R = P$). We found that at this point r nested predicates „ends", so in GDL representation of $\mathcal{P}^*$, y should be preceded by r closing brackets. Now the place for y can be found using procedure from previous paragraph with assumption that the position of previously found argument is $\langle\tilde{q}[1],\ldots,\tilde{q}[|\tilde{q}|-r]\rangle$ instead of $\tilde{q}$.

A small part of flattened *Goldrush* game is shown in Listing 7.2 as an example. As it shows at line 10, it can create rules with unbound variables, but the values of these variables are explicitly set to `#nil` during further calculations.

Listing 7.2: Flattened GDL code.

```
 2 (init (OnMap Green #nil #nil 1 1)) ...
 3 (init (OnMap Obstacle #nil 1 6)) ...
 4 (init (OnMap Gold 2 #nil 1 7)) ...
 5 (init (OnMap Item Blaster 3 7 4)) ...
 6 (init (OnMap Item Stoneplacer 3 1 4)) ...

10 (<= (next (OnMap ?r0 ?r1 ?r2 ?x0 ?y0))
11     ( role ?r0 ) ( does ?r0 ( Move ?x0 ?y0 #nil ) ) )
```

7.2.2 Predicates Dependency Graph and Layering

Let say that a predicate P depends on Q if there exists a game rule R such that P is the head of R and the body of R contains Q. We consider the *dependency graph*, which is a directed graph representing the dependency relation of predicates.

After the complete dependency graph is built, it is split up to subgraphs representing each of the game *phases*, depending on what we are going to compute. The phases are: *init*, *term*, *goal*, *legal*, and *next*, and they correspond to the solid arrows from the game loop visualization (Fig. 7.1.1), where *term*, *goal* and *legal* belong to *view*. In such a way, the dependency graph gives us information about the predicates usage.

We can get rid of all predicates that are not needed to compute any of `legal`, `terminal`, `goal`, `next`. *Constant* predicates can be fully precomputed during the initialization phase and they stay unchanged during the rest of the game. These and base predicates belong to the *init* phase. The predicates reachable in the reversed dependency graph from `terminal`, `goal`, `legal` and `next` belong to corresponding phases respectively. There is one exception: predicates reachable from both `goal` and `legal` are put in the *term* phase. We note that it is not necessary required to compute the `goal` predicate while the state is non-terminal. This loses possible information about scores in non-terminal states, but it saves computation time and in Monte Carlo approach simulations go to the end anyway, so checking the `goal` values in non-terminals can be avoided.

All proper GDL games must be stratified, which means that for all predicates P and Q, if P depends on `not` Q then P must be in a higher stratum, and all facts of a lower stratum

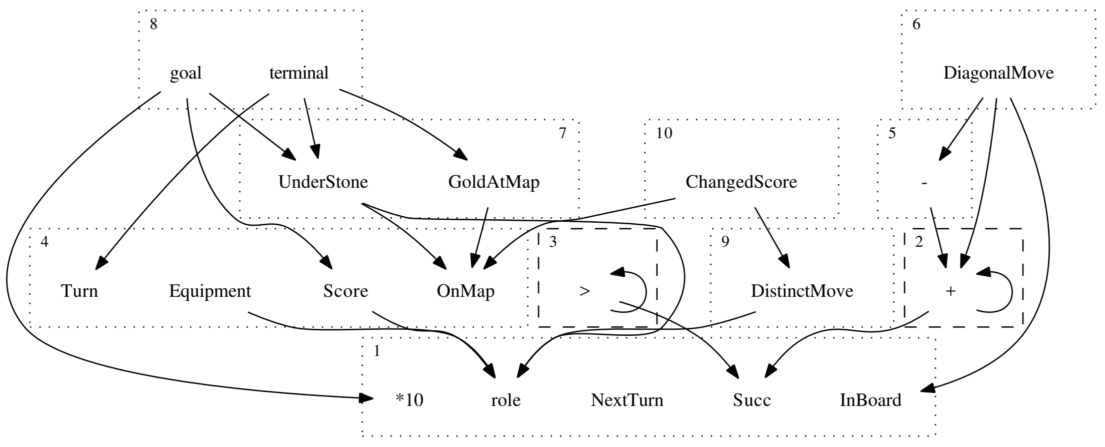


Figure 7.2.1: Dependency graph of the game *Goldrush* (predicates `legal` and `next` are omitted due to visibility) with cyclic layers marked with dashed border.

should be deducted before deduction of the upper stratum starts (which is always possible). This mechanism allows to treat GDL deduction as continuously adding facts to a database, without worrying of withdrawing them when the computations are made in the proper order. In a top-down approach the right computation order is for free, but in a bottom-up approach the ordering is more flexible and can lead to better efficiency.

Despite stratification as a result of negations placement, we consider *layering*. This is a more general and a more complex approach based on dependency graphs. Each layer corresponds to a set of strongly connected components of the dependency graph. There are two types of layers:

Acyclic layer is a set of predicates such that there is no path in the dependency graph between any two predicates from this set. This means that, if only all the lower layers are computed, all the rules with these predicates in the head can be computed simultaneously and only once.

Cyclic layer is a set of predicates that are reachable from any other from this set (by using at least one edge). In this case the number of rules applications to deduct these predicates is unknown, and computations must take place until a fixpoint is reached (no new fact is added after an iteration).

Partitioning of the dependency graph to layers should be done in a way, that acyclic layers should be as large as possible, and cyclic layers as small as possible (which reduces the number of computations). Currently, we create the layers incrementally from the nodes without ingoing edges (so the first layer contains all „leaves" of graph). If there is a choice which layer, cyclic or acyclic, should be considered as a lower, the lower (first to compute) goes the cyclic one.

7.2.3 Defining the Rules Computation Order

Mapping from predicates to layers does not make ordering of rule computation unambiguous. Consider a rule R, and let L_h^R be the layer where the head of the rule belongs, and let $L_{b\,\max}^R$ be the maximal (the highest) layer of predicates in body of R. This means that R must be computed before layer $L_h^R + 1$ and after layer $L_{b\,\max}^R$, so the rule can be placed in any layer between these values. An exceptional situation is when $L_h^R = L_{b\,\max}^R$, which happens only for some rules from cyclic layers. In this case the rule placing is unambiguous.

Intelligent rule placement can lead to some speed improvement in two main cases. First, when the predicate in the head of a rule is from a cyclic layer but the body is not ($L_h^R < L_b^R{}_{\max}$) then the rule can be computed cheaper, because it is computed only once within a lower (assuming non cyclic) layer. Sometimes even special layers for such rules can be created. The second reason to move rules is that in some layers in admissible range there are rules similar in construction and some sharing computations between them can be done.

7.2.4 Filter Trees

With computed order of the rule computations we can produce the final plan. In such a plan the symbols and predicates get their unique id, and each predicate has bound information about its (flattened) domain and container type. To appoint exact ordering of game state computations, structures called *filter trees* are created.

Filter trees contain nodes that can have child nodes. The whole computation process is just traversing the tree and performing actions according to the types of nodes. During computation a set of local variables and a set of containers are maintained. In the root the sets are empty. A variable (similarly container) defined in a node has scope bounded to the children nodes. A variable defined in a parent node is bound in the children and cannot change its value. The nodes have the following types:

- *Sequence* A node with many children which should be executed in the order.
- *Query* Queries the specified container for a given subset of facts. There can be either symbols or bound and unbound variables. For each fact in the container that matches the query, define unbound variables by setting the values to match the fact and go into the child node. For optimization purposes a query can have additional explicit domain filter.
- *Accept* Inserts a fact to the specified container. All of the variables used in an insertion must be bound. If a new fact is added, a special *repeat* flag is set to inform a cyclic layer to be repeated.
- *Repeat* Repeats computation of its subtree until the *repeat* flag is unset. The flag is checked and cleared each time between iterations.
- *If* It has three children. The child called *test* is executed first until it is finished or a special *Return* node is reached. If *Return* was reached the child *true* is executed, otherwise *false.*

A GDL rule can be simply transformed to a filter tree without any significant modifications. A conjunction is simply nested children, an alternative can be decoded as *Sequence* and negated terms can be put in *If* filter with *Accept* in *false* subtree. Distinct between two variables is converted to a *Query* with unspecified container but with a list of distinct variables, while distinct with a variable and a symbol is just *Query* with reduced variable's domain.

A careful way of constructing filter trees can reduce much of computations. At first, if the same query occurs in two rules in the same layer it can be shared. Because every nested query can potentially cut out variable domains, the right order of nested queries can also improve efficiency. The last main optimization takes place when all variables from the heads of the rules are already defined in some query. Since the added fact is fully defined it remains only to check if the rest of queries can be satisfied. This can be realized by putting into *If* the rest nested queries, so a single positive pass through them is sufficient to immediately return and insert a fact.

We observe that only the values of variables that can reach *Insert* or *Return* node are necessary to be considered. We can restrict the domain in advance in queries defining these variables, instead of filtering them by nested queries.

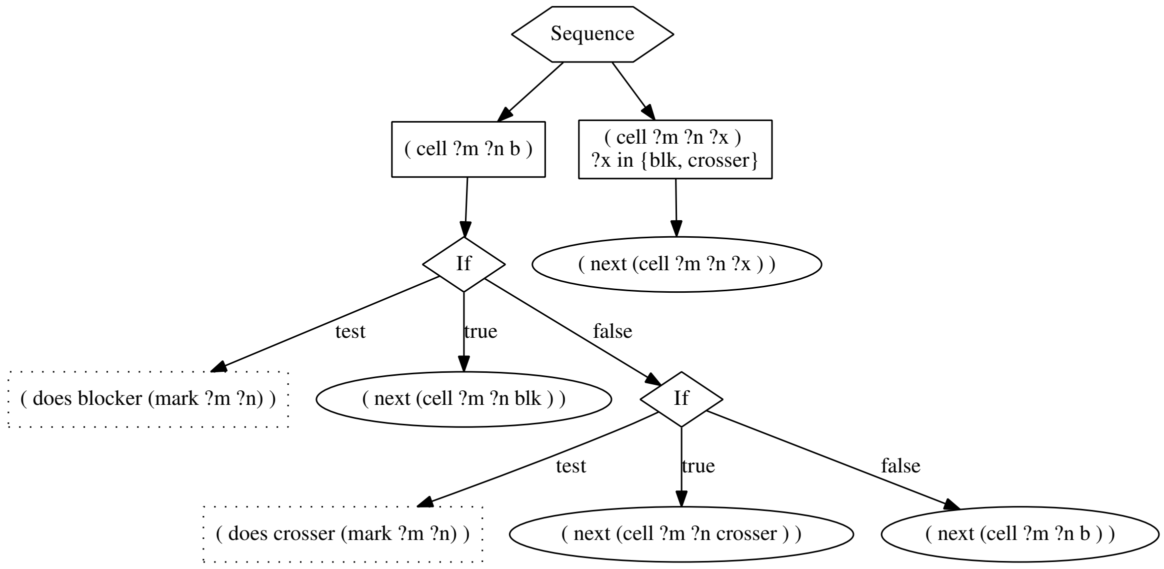

Sequence
(cell ?m ?n b)
(cell ?m ?n ?x)
?x in {blk, crosser}
(next (cell ?m ?n ?x))
If
test
true
false
(does blocker (mark ?m ?n))
(next (cell ?m ?n blk))
If
test
true
false
(does crosser (mark ?m ?n))
(next (cell ?m ?n crosser))
(next (cell ?m ?n b))

Figure 7.2.2: *Next* filter tree of the game *blocker*. Box nodes represent *Queries* while ellipses are *Accept* filters. Every *test* branch of *If* ends with unmarked *Return* node.

We create five filter trees, one for each of the phases. An example filter tree for *next* phase of the game *blocker* [70] is presented in Fig. 7.2.2. Creating the filter trees finishes our construction, allowing generation of the final code.

7.2.5 Data Structures for Containers

Queries can have very different shapes. They consist of a predicate and a fixed number of arguments, depending on the predicate's arity. The arguments can be constants or variables. The simplest are those asking about existence of a particular fact like `cell 1 3 b`. More complicated are queries mixing both constants and variables, including bound variables like in `cell ?x ?x ?t`. Efficiency of performing queries depends on the data structure used to implement the container for a given predicate. An elementary analysis can be used to estimate query and insertion costs for various data structures. Although more deep optimizations can take care of the proportion between queries and insertions, and occurrences of query shapes.

In our compiler we use a few different data structures. We describe them here and perform a simple efficiency analysis. Consider a container and assume that d is the arity of facts in the container. Then let $c_1, \ldots, c_d$ be the sizes of the domains of the arguments, that is the i-th argument can take one of c_i possible values. Thus the container can hold at most $C = \prod_{i=1}^{d} c_i$ facts. Assume that n is the number of currently stored facts in the container. We assume that d is a fixed constant and comparing any two values (symbols) takes $O(1)$ time.

Querying a subset of facts in the container takes time at least $\Omega(s)$, where s is the size of the queried subset, because of further processing these facts. For simplicity we do not consider the cases with duplicated unbound variables in a single query, which are also rare cases. Thus, an optimal data structure would take $O(s)$ time performing a query, and $O(1)$ time performing an insertion. A special careful should be taken for querying for a particular fact, because such an operation is often required during insertions to prevent storing duplicated facts in the container.

One of the simplest data structures is a standard dynamically-sized *vector*. Inserting to a vector takes $O(1)$ time, but querying for a particular fact or a subset of facts can take $O(n)$ time in the worst case. In opposition to the vector there is a complete *lookup array* with the fixed size C. Each allowed fact has a fixed position in the array storing a flag indicating if the fact is in the container. Insertion to an array, as well as querying for a particular fact, takes $O(1)$ time. However querying for a larger subset of facts can take $O(C)$ time, depending on the number of possible facts matching the query. Thus vectors are better for larger domains and smaller number of stored facts, and arrays conversely.

Hash and tree sets are quite efficient for insertions and querying for a particular fact, which take $O(1)$ time in a hash set and $O(\log n)$ in a balanced tree set with lexicographically ordering of facts. However querying for a subset of facts may take $O(n)$ in a hash set, and as well in a tree set if the first argument is an unbound variable.

A *trie* is a more complex tree-like structure. Levels correspond to the arguments. At the level h, each node contains d_h pointers to nodes at the height $h+1$ (except the last). These correspond to all d_h allowed values of the h-th argument. A fact is encoded by a path from the root to a leaf. A pointer is *null* if there are no facts with the corresponding value. A trie grows as more facts are added. An insertion takes $O(\sum_{i=1}^{d} c_i)$ time in the worst case (empty trie), because we must create a node at each level. Querying a particular fact takes $O(1)$ time. Efficiency of a trie in querying for a specified subset depends on the order of the arguments. A query can cost from $O(s)$ time when the constant arguments are at the beginning, up to $O(n)$ time when they are at the end. We consider tries as an universal balanced structure, since it performs quite well in most cases.

With an assumption that we have a constant number of query shapes, we developed two nearly optimal *composed structures*. Composed structure consist of a set of other structures made especially for efficient maintaining of different query shapes. The first such structure is the *trie-composed* structure based on tries, and the second is the *tree-composed* structure based on balanced tree sets. It seems that the *trie-composed* structure is better, because queries usually occur more frequently than insertions, and it generally has a lower constant factor.

Lemma 7.2.2. *The* trie-composed *structure takes $O(s)$ time for a query and $O(c_1 + \ldots + c_d)$ time for an insertion.*

Proof. For each of the query shapes occurring for the container, we create an optimal trie for this shape. In an optimal trie the arguments are reordered so that the bound arguments are placed at the beginning, and each node corresponding to an unbounded argument is provided with an additional vector.

The additional vectors (lists) store the values for which there are non-null corresponding pointers. At the level h this speeds up querying for a subset of facts with the unbounded h-th argument, because we do not have to iterate through all of the d_h possibilities.

When querying, we use the optimal trie designed to the query shape. Because of the provided vectors in nodes it would take $O(s)$ time. An insertion to the structure is just an insertion to each of the tries.

Lemma 7.2.3. *The* tree-composed *structure takes $O(s + \log n)$ time for a query and $O(\log n)$ time for an insertion.*

Proof. For each query shape, the bound arguments should be put at the beginning of the tree lexicographical ordering. In this way we can perform a query by searching for a lower and an upper fact in the tree in $O(\log n)$ time. All the facts between the bounds will be in the resulted set. It is possible to provide the trees with additional pointers to successors allowing us to iterate them in $O(s)$ time.

Table 7.3.1: The numbers of performed random simulations and visited game states per second for various games.

Game	The reasoner of Dumalion			Prolog	
	Compilation	Simulations	States	Simulations	States
Tic-tac-toe	0.736 s	331,647	2,860,888	2,076	15,829
Blocker	0.645 s	194,628	1,729,674	1,020	8,049
Connect Four	0.857 s	15,092	353,283	287	6,424
Breakthrough	1.074 s	3,086	200,901	55	3,553
Checkers	10.482 s	211	21,291	12	1,186
Skirmish	7.810 s	71	7,114	5	518

7.2.6 Final Code Generation

We have implemented GDL compilation to C++. The compiled reasoner is just a module allowing to maintain game states. In particular, the phases are functions initializing the reasoner and computing the *init state*, computing a *view state* given a *base state*, or computing a next *base state* given a *base state*, *view state* and the moves of the players. It also provides an interface for answering if a state is terminal, getting the goal or the legal moves.

Each node of the filter tree is directly inlined in the code. In this way many technical optimizations are possible by using the context, since for example each query can have different set of domains for the variables, and we can perform explicit iteration through them.

7.3 Experimental Results

We have implemented the compiler as a Java program producing a reasoner in C++ as output. Our benchmark results are presented in Table 7.3.1. The resulted reasoners were compiled by g++. The main program performed uniformly random simulations of the games. Comparative reasoners uses *ECLiPSe Prolog* system. The benchmarks were done on Intel(R) Core(TM) i7-3610QM 2.3GHz with 8GB of RAM.

Although differences between computation speed between a simple Prolog engine and the optimized and compiled code are outstanding as expected, an interesting observation is that the improvement factor is less for more complicated games (such a tendency is also visible in benchmarks from [135, 167]). In other hand, while the improvement factor is smaller, the numbers of performed simulations and visited states were increased several time, and this is especially crucial for very difficult games when computing states is hard and even small speed up can give a big advantage.

Because of hardware differences and chosen method of benchmarking it is hard to make a straightforward comparison between our compiler and other approaches described in [135, 167, 203, 235]. But after recalculating all the results to a common base simulations over a second, a roughly comparison shows that the reasoner of Dumalion can compute simulations from 2 to about 10 times faster (depending on the game) than the compiling methods described so far (Table 7.3.2), and from 10 to 160 times faster than a standard Prolog engine.

7.4 Conclusions

Using a compiler generator to create reasoners requires far more work than running a Prolog engine on syntactically changed GDL code, but the benefit in computation speed is significant.

Table 7.3.2: Comparison of improvement factors between various compilation approaches and basic Prolog engines (depend on Prolog engine used).

Game	Dumalion	gdlcc tt [235]	Centurio [135]	GaDeLaC [167]	MINI-Player [203]
Tic-tac-toe	159.75	22.48	2.66	1.46	3.90
Connect Four	52.58	2.46	2.48	0.30	4.05
Breakthrough	56.11	–	–	0.24	2.46
Checkers	17.58	1.80	2.16	–	2.26
Skirmish	14.2	–	2.46	–	–

We mention here a few inconveniences of our method. At first the produced reasoner must be compiled into a native code. This can take quite long if the game is complicated. The second problem is that we lose all the structure information, for example we cannot ask about a specified predicate defined in the original GDL, since it is possible that the corresponding container does not exist at all due to optimizations. Another drawback is that the process of compilation itself make the whole GGP system more complicated and harder to handle, especially if it should support parallelism.

There are many other ways of further optimizations of the plan. They include introducing new temporary containers, reordering of arguments, more careful selection of container types, reordering of queries and splitting them. As the future work, we have plan to construct GGP architecture with the aim of efficient maintain compiled code in a scalable, parallel system.

8 Discussion and Future Work

In this thesis, we have presented our results concerning various aspects of general game description languages. We now want to summarize each area of concern by discussing the remaining open questions and presenting tasks that can be further explored.

The Procedural Content Generation of game rules is still a novel domain without many contributions. Our method of adapting an RAPP approach to evaluate the strategic properties of games seems to be very promising, providing a possible answer to the main question: „how can we judge if a game is good or bad?”. However, one of the drawbacks of our method is that it requires a set of model games and a set of handcrafted player profiles. The latter can be solved by replacing domain-based specific algorithms with general MCTS-based players. We can estimate the „strategic complexity” curve of a game using the results of MCTS within different time limits. Initial research towards this approach shows that game similarities calculated using the domain-based method do not coincide with the ones computed using MCTS-based evaluation. Actually, this is to be expected, as the domain-free method is just an approximation of the game complexity and does not distinguish certain strategic properties of the game. Nevertheless, we plan to test the behavior of this knowledge-free RAPP approach on some existing GGP domains.

Another issue, concerning the generation of chess-like games, is to ensure that the rules will not be too complex for a human player. We would like to provide a „one-game challenge” website where people could play against each other in a GGP manner, i.e. with the rules of an unknown chess-like game. This, however, requires the game rules to be easy to understand and remember, while being dissimilar enough to Chess to give the feeling of novelty. The intended next step is to create a description-complexity measure, which will evaluate each piece and simultaneously build a grammar-based natural language description, based on similarities to the known chess-like figures and movement patterns [35].

Another goal is to use the RAPP method (knowledge-based or knowledge-free) for generating Stanford's GDL games. The logic-based structure of the game rules makes it suitable for the genetic programming approach. However, due to the constraints of syntax and semantics, there are a number of problems to be solved on the way. There are no published approaches applying PCG to Stanford's GGP domain at the moment.

There are a few interesting tasks that should be completed in the future for the domain of Simplified Boardgames itself. One is to significantly extend the language in order to be able to represent complex chess-like rules like promotions, castling or en passant, without losing the properties of regularity. We have prepared an initial version of such a language (called Regular Boardgames). Let us briefly point out its most important components. Firstly, the *on* field in Σ can contain any subset of available pieces, not just *own* or *opponent*. Secondly, we extend Σ by adding the *off* field containing what can be left after the piece movement. This allows promotions (multiple choice is available), and jump captures like in checkers.

One of the biggest changes is that we have assumed an „everything is a piece" design philosophy. The board is always fully covered with pieces (one per square), i.e. *empty* is no longer a reserved keyword but rather some kind of piece that no one can move. What follows is that there is no strict ownership restriction, e.g. players can move the same piece even using different rules. (The rules are defined for the player not for the piece type.) Our new language also supports more than two players, consecutive moves (i.e. the order of moves can change during the play), priority-ordered rules, and extended goal definitions. We also allow a movement pattern (i.e. some $w \in \Sigma^*$) to be a termination condition. This is an intuitive approach for games like Tic-tac-toe, where the goal is to arrange some pattern on the board. We took special care so that the rules defining the games are still defined mostly as a sum of regular languages.

The second major task concerning Simplified Boardgames is to develop efficient game-playing algorithms. Min-max based zero-sum game algorithms are well studied, especially for the chess-like domain, but bringing them to a higher level of abstraction should create many challenges. Apart from basic heuristics like mobility or material, games, especially generated ones, can possess some non-trivial properties that are more difficult to exploit. Ad hoc discovery of such properties, based only on the language of piece movements, is a non-trivial knowledge-extraction task. Lastly, let us point out that there is the possibility of generating books of openings and endings, which can substantially improve the player's performance.

Future work concerning Real-time GDL should be mainly focused on establishing it as a new domain of GGP competition. This requires providing potential players with a detailed specification and the base package, containing an efficient implementation of Game Manager, an example player and a number of game rules. The nature of competition forces some refinements concerning the presented theoretical model. Let us consider two events e_1, e_2 occurring at times t_1, t_2, where $t_2 = t_1 + \epsilon$. Given arbitrary small ϵ, these events should cause two separate state updates. However, for obvious reasons, such behavior cannot be guaranteed in any practical implementation. Moreover, if the minimal ϵ making events distinguishable is not available for the game designer, he cannot predict what semantics his game will have. Solving this issue requires additional assumptions and restrictions on the Game Manager.

Another interesting open problem about rtGDL is the representation of arithmetic operators. Real number arithmetic is a crucial element in most real-time games. Combined with comparison operations, real number arithmetic is required in order to navigate through space, update players/units/npc's statistics, etc. Although rtGDL rules themselves do not have the ability to express arithmetic (in a way other than static definitions of every possible equation like in plain GDL), this issue can be solved using passing game time and the Game Manager's ability to reduce certain in-game values. This mechanism allows real-number addition, subtraction, multiplication by a natural number, and a floor of division to be represented. The cost of such operations can be measured in the game time devoted to the operation and the number of additionally required state updates.

Learning game rules by observing can be further explored in different GGP domains. As it seems to be an extremely difficult task for an arbitrary GDL game, this may be a promising area of research for GVGAI. During the General Video Game Playing AI Competition, players are provided only with the forward-model of the game, and not its description. Normally, the model is used to simulate the game and search for the optimal agents' moves. However, we can restate our goal and create an agent that will systematically reconstruct the rules of the game based on the observed behavior of the environment. This would require not only a method for reverse engineering the game engine, but also a searching algorithm focused on discovering unseen parts of the game and clarifying ambiguities in rules.

Although the presented method of GDL compilation is very efficient, it may be further improved, as there are some cases where another organization of computations is more effective. Generally, the methods of computation can be split into *top-down* and *bottom-up*, and all of the developed reasoners can be assigned to one of them. Our current algorithm uses the bottom-up approach. It should be possible to compound these two methods and use both of them in a single reasoner. We need to estimate probabilities of fact occurrences, create an optimized structure of queries, and thus calculate expected time complexity. Then, based these estimations, the algorithm could decide whether the top-down or the bottom-up approach is better in a particular place of computations. This could lead to reasoners joining benefits from both approaches, and so become the most effective ones for any game.

A
Simplified Boardgames Grammar

The formal grammar in EBNF, following our definition from [98], is presented below. C-like comments can appear anywhere in the game definition: "//" starts a line comment and every next character in the line is ignored. "/*" starts a multiline comment and every character is ignored until the first occurrence of "*/".

The start non-terminal symbol is "sbg". The "nat" non-terminal stands for a natural number (thus a non-empty sequence of digits), while "int" stands for a signed integer (thus it is "nat" optionally preceded by "-"). The "alphanumspace" non-terminal generates all alphanumerical characters or a space.

⟨*sbg*⟩ ::= '`<<`' ⟨*name*⟩ '`>>`' '`<BOARD>`' ⟨*board*⟩ '`<PIECES>`' ⟨*pieces*⟩ '`<GOALS>`' ⟨*goals*⟩

⟨*name*⟩ ::= alphanumspace {alphanumspace}

⟨*board*⟩ ::= ⟨*nat*⟩ ⟨*nat*⟩ { ⟨*row*⟩ }

⟨*row*⟩ ::= '`|`' {`[.a-zA-Z]`} '`|`'

⟨*pieces*⟩ ::= { `[A-Z]` ⟨*regexp*⟩ '`&`' }

⟨*regexp*⟩ ::= ⟨*rsc*⟩ | ⟨*regexp*⟩ ⟨*regexp*⟩ | ⟨*regexp*⟩ '`+`' ⟨*regexp*⟩ | '`(`' ⟨*regexp*⟩ '`)`' [⟨*power*⟩]

⟨*rsc*⟩ ::= '`(`' ⟨*int*⟩ '`,`' ⟨*int*⟩ '`,`' ⟨*on*⟩ '`)`' [⟨*power*⟩]

⟨*power*⟩ ::= '`^`' ⟨*nat*⟩ | '`^`' '`*`'

⟨*on*⟩ ::= `[epw]`

⟨*goals*⟩ ::= ⟨*nat*⟩ '`&`' { ⟨*goal*⟩ '`&`' }

$\langle goal \rangle$ ::= ‘`#`’ $\langle letter \rangle$ $\langle nat \rangle$ | ‘`@`’ $\langle letter \rangle$ $\langle squares \rangle$

$\langle letter \rangle$ ::= `[a-zA-Z]`

$\langle squares \rangle$ ::= $\langle nat \rangle$ $\langle nat \rangle$ { ‘`,`’ $\langle nat \rangle$ $\langle nat \rangle$ }

B
Evolved Simplified Boardgames

B.1 Using Game Features

Listing B.1: The rules of the game presented as an example in Section 4.2.

```
<<GenP4>>

<BOARD>
6 6

|qbknqq|
|nkp.pn|
|......|
|.....P|
|NKPPPN|
|QBKNQ.|

<PIECES>

// 1040 <2.78> (4 legals) :1,1,ep,_:LR
P (1,1,e) + (-1,1,e) + (1,1,p) + (-1,1,p) &

// 1021 <3.00> (4 legals) :0,2,ep,_: + 0,1,ep,_:
K (0,2,e) + (0,2,p) + (0,1,e) + (0,1,p) &

// 2043 <6.44> (10 legals) :2,2,e,_:LR + 1,-1,ep,_:LR,ROT
N (2,2,e) + (-2,2,e) + (1,-1,e) + (-1,-1,e) + (-1,1,e) +
  (1,1,e) + (1,-1,p) + (-1,-1,p) + (-1,1,p) + (1,1,p) &
```

```
// 2009 <7.22> (10 legals) :1,1,ep,*:FB,LR + 0,1,ep,_:
B (1,1,e)^1 + (1,1,e)^2 + (1,1,e)^3 + (1,1,e)^4 + (1,1,e)^5 +
  (-1,1,e)^1 + (-1,1,e)^2 + (-1,1,e)^3 + (-1,1,e)^4 + (-1,1,e)^5 +
  (1,-1,e)^1 + (1,-1,e)^2 + (1,-1,e)^3 + (1,-1,e)^4 + (1,-1,e)^5 +
  (-1,-1,e)^1 + (-1,-1,e)^2 + (-1,-1,e)^3 + (-1,-1,e)^4 + (-1,-1,e)^5 +
  (1,1,p)^1 +     (1,1,p)^2 +   (1,1,p)^3 +   (1,1,p)^4 +   (1,1,p)^5 +
  (-1,1,p)^1 +   (-1,1,p)^2 +  (-1,1,p)^3 +  (-1,1,p)^4 +  (-1,1,p)^5 +
  (1,-1,p)^1 +   (1,-1,p)^2 +  (1,-1,p)^3 +  (1,-1,p)^4 +  (1,-1,p)^5 +
  (-1,-1,p)^1 + (-1,-1,p)^2 + (-1,-1,p)^3 + (-1,-1,p)^4 + (-1,-1,p)^5 +
  (0,1,e) + (0,1,p) &

// 3001 <10.28> (13 legals) :0,-1,p,*:FB,LR,ROT +
//                            0,1,ew,_:FB + 0,1,ep,_: + 0,1,ep,_: + 1,1,pw,_:LR
Q (0,-1,p)^1 +(0,-1,p)^2 + (0,-1,p)^3 +(0,-1,p)^4 +(0,-1,p)^5 +
   (-1,0,p)^1 +(-1,0,p)^2 + (-1,0,p)^3 +(-1,0,p)^4 +(-1,0,p)^5 +
   (0,1,p)^1 + (0,1,p)^2 +  (0,1,p)^3 + (0,1,p)^4 + (0,1,p)^5 +
   (1,0,p)^1 + (1,0,p)^2 +  (1,0,p)^3 + (1,0,p)^4 + (1,0,p)^5 +
   (0,1,e) + (0,-1,e) + (0,1,w) + (0,-1,w) + (0,1,p) +
   (1,1,p) + (-1,1,p) + (1,1,w) + (-1,1,w) &

<GOALS>
160 &
```

B.2 Using Relative Algorithm Performance Profiles

Listing B.2: The rules of the game presented as an example in Section 4.3.

```
<<The Legacy of Ibis>> // RAPP Evolved - 4x16 Dm#16-06 (0.9538)

<BOARD>
7 7

|osiosol|
|ff.ffff|
|.......|
|.......|
|.......|
|FF.FFFF|
|OSIOSOL|

<PIECES>

// 3016 <4.94> (8 legals) :1,1,p,_:LR + 1,1,ep,_;1,1,e,_:LR + 1,0,e,_;1,0,e,_:LR
O (1,1,p) + (-1,1,p) + (1,0,e)(1,0,e) + (-1,0,e)(-1,0,e) +
  (1,1,e)(1,1,e) + (-1,1,e)(-1,1,e) + (1,1,p)(1,1,e) + (-1,1,p)(-1,1,e) & // Owl
```

```
// 3017 <4.65> (6 legals) :0,1,ep,_: + 0,1,p,_: + 1,1,ep,_:LR
S (0,1,e) + (0,1,p) + (1,1,e) + (-1,1,e) + (1,1,p) + (-1,1,p) & // Snake

// 3018 <5.96> (10 legals) :1,1,p,_:LR + 2,1,ep,_:LR + 2,1,ep,_:LR + 1,1,e,_;1,1,ep,_:LR
I (1,1,p) + (-1,1,p) + (2,1,e) + (-2,1,e) + (2,1,p) + (-2,1,p) +
  (1,1,e)(1,1,e) + (-1,1,e)(-1,1,e) + (1,1,e)(1,1,p) + (-1,1,e)(-1,1,p) & // Ibis

// 2007 <4.41> (6 legals) :1,0,p,_:LR + 3,1,e,_:LR + 0,1,ep,_:
L (1,0,p) + (-1,0,p) + (3,1,e) + (-3,1,e) + (0,1,e) + (0,1,p) & // Lion

// 1007 <1.22> (2 legals) :1,2,e,_:LR
F (1,2,e) + (-1,2,e) & // Fish

<GOALS>
160 &
@F 0 6, 1 6, 2 6, 3 6, 4 6, 5 6, 6 6 &
@f 0 0, 1 0, 2 0, 3 0, 4 0, 5 0, 6 0 &
#L 0 &
#l 0 &
```

C
Example Game for Learning by Observing

Listing C.1: The rules of the game used as a hard learning case in experiments described in Chapter 3.

```
<<Tricky Learning>>

<BOARD>
11 8

|s.........s|
|t....xxxxxx|
|..p.p......|
|...........|
|...........|
|.P.P.......|
|T....XXXXXX|
|S.........S|

<PIECES>

P (0,1,e) + (-1,1,p) + (1,1,p) &

X (1,0,e) &

S (1,0,e)   + (-1,0,e)   +
  (1,0,e)^2 + (-1,0,e)^2 +
  (1,0,e)^3 + (-1,0,e)^3 +
  (1,0,e)^4 + (-1,0,e)^4 +
  (1,0,e)^5 + (-1,0,e)^5 +
  (1,0,e)^6 + (-1,0,e)^6 +
  (1,0,e)^7 + (-1,0,e)^7 +
  (1,0,e)^8 + (-1,0,e)^8 +
  (1,0,e)^9 + (-1,0,e)^9 &

T (1,0,e)   + (-1,0,e)   +
  (1,0,e)^2 + (-1,0,e)^2 +
  (1,0,e)^3 + (-1,0,e)^3 +
  (1,0,e)^4 + (-1,0,e)^4 &


<GOALS>
160 &
@P 0 5, 1 5, 2 5, 3 5, 4 5, 5 5, 6 5, 5 5 &
@p 0 2, 1 2, 2 2, 3 2, 4 2, 5 2, 6 2, 7 2 &
```

D
Example GDL Game for Compilation

Listing D.1: Full GDL rules of the game Goldrush, used as an example in Chapter 7.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Goldrush
; v. 1.0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Brief description:
; Two players walk in a labirynth and search for gold.
; Each piece of gold on the map is worth some points, they all sum to 100.
; Players can destroy obstacles and place new ones using the items,
; which can be found in the labirynth.
; If both players step on the same item/gold no one gets it.
; A player being on a field with an obstacle immediately lose.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Map:
; G  s O2
;   OO1 O
;   b
; bO 4 Ob
;     b
; O 100
; 2O s   R
;
; G - Green player; R - Red player; O - Obstacle; b - blaster; s - stoneplacer
; 1, 2 - gold worth 10x points; 4 - gold under the obstacle worth 40 points
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       ROLE section
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(role Green)
(role Red)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;        INIT section
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (OnMap Green 1 1))
(init (OnMap Red 7 7))

(init (OnMap Obstacle 1 6))
(init (OnMap Obstacle 2 4))
...
(init (OnMap Obstacle 7 2))

(init (OnMap (Gold 1) 3 6))
...
(init (OnMap (Gold 4) 4 4))

(init (OnMap (Item Blaster 1) 3 3))
(init (OnMap (Item Blaster 1) 5 5))
(init (OnMap (Item Blaster 3) 4 1))
(init (OnMap (Item Blaster 3) 7 4))
(init (OnMap (Item Stoneplacer 3) 1 4))
(init (OnMap (Item Stoneplacer 3) 7 4))

(<= (init (Equipment ?r Blaster 2))
    (role ?r))
(<= (init (Equipment ?r Stoneplacer 1))
    (role ?r))

(<= (init (Score ?r 0))
    (role ?r))

(init (Turn 1))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;        LEGAL section
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal ?r (Move ?nx ?y))
    (true (OnMap ?r ?x ?y))
    (role ?r)
    (or (+ ?x 1 ?nx)
        (- ?x 1 ?nx))
    (InBoard ?nx))

(<= (legal ?r (Move ?x ?ny))
    (true (OnMap ?r ?x ?y))
    (role ?r)
    (or (+ ?y 1 ?ny)
        (- ?y 1 ?ny))
    (InBoard ?ny))
```

```

(<= (legal ?r (Use ?e ?nx ?ny))
    (true (Equipment ?r ?e ?q))
    (> ?q 0)
    (true (OnMap ?r ?x ?y))
    (role ?r)
    (DiagonalMove ?x ?y ?nx ?ny))

(<= (DiagonalMove ?x ?y ?nx ?ny)
    (InBoard ?x)
    (InBoard ?y)
    (+ ?x 1 ?nx)
    (InBoard ?nx)
    (or (+ ?y 1 ?ny)
        (- ?y 1 ?ny))
    (InBoard ?ny))

(<= (DiagonalMove ?x ?y ?nx ?ny)
    (InBoard ?x)
    (InBoard ?y)
    (- ?x 1 ?nx)
    (InBoard ?nx)
    (or (+ ?y 1 ?ny)
        (- ?y 1 ?ny))
    (InBoard ?ny))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       NEXT section
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;    OnMap
; Players
(<= (next (OnMap ?r ?x ?y))
    (role ?r)
    (does ?r (Move ?x ?y)))
(<= (next (OnMap ?r ?x ?y))
    (true (OnMap ?r ?x ?y))
    (role ?r)
    (does ?r (Use ?e ?nx ?ny)))
; Obstacles
(<= (next (OnMap Obstacle ?x ?y))
    (does ?r (Use Stoneplacer ?x ?y)))
(<= (next (OnMap Obstacle ?x ?y))
    (true (OnMap Obstacle ?x ?y))
    (not (does Green (Use Blaster ?x ?y)))
    (not (does Red (Use Blaster ?x ?y))))
; Gold
(<= (next (OnMap (Gold ?q) ?x ?y))
    (true (OnMap (Gold ?q) ?x ?y))
    (not (does Green (Move ?x ?y)))
    (not (does Red (Move ?x ?y))))
; Items
```

```
(<= (next (OnMap (Item ?e ?q) ?x ?y))
    (true (OnMap (Item ?e ?q) ?x ?y))
    (not (does Green (Move ?x ?y)))
    (not (does Red (Move ?x ?y))))

;    Equipment
(<= (next (Equipment ?r ?e ?q))
    (true (Equipment ?r ?e ?q))
    (does ?r (Move ?x ?y)))
(<= (next (Equipment ?r ?e1 ?q))
    (true (Equipment ?r ?e1 ?q))
    (does ?r (Use ?e2 ?x ?y))
    (distinct ?e1 ?e2))
(<= (next (Equipment ?r ?e ?nq))
    (does ?r (Use ?e ?x ?y))
    (true (Equipment ?r ?e ?q))
    (- ?q 1 ?nq))
(<= (next (Equipment ?r ?e ?q3))
    (DistinctMove ?r ?x ?y)
    (true (OnMap (Item ?e ?q1) ?x ?y))
    (true (Equipment ?r ?e ?q2))
    (+ ?q1 ?q2 ?q3))

;    Score
(<= (next (Score ?r ?s3))
    (DistinctMove ?r ?x ?y)
    (true (OnMap (Gold ?s1) ?x ?y))
    (true (Score ?r ?s2))
    (+ ?s1 ?s2 ?s3))
(<= (next (Score ?r ?s))
    (true (Score ?r ?s))
    (not (ChangedScore ?r)))

(<= (DistinctMove ?r1 ?x ?y)
    (role ?r1)
    (role ?r2)
    (distinct ?r1 ?r2)
    (does ?r1 (Move ?x ?y))
    (not (does ?r2 (Move ?x ?y))))

(<= (ChangedScore ?r)
    (DistinctMove ?r ?x ?y)
    (true (OnMap (Gold ?s1) ?x ?y)))

;    Turn
(<= (next (Turn ?nt))
    (true (Turn ?t))
    (NextTurn ?t ?nt))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       GOAL section
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```

(<= (goal ?r 0)
    (UnderStone ?r))
(<= (goal ?r ?s10)
    (true (Score ?r ?s))
    (not (UnderStone ?r))
    (*10 ?s ?s10))
(<= (UnderStone ?r)
    (role ?r)
    (true (OnMap ?r ?x ?y))
    (true (OnMap Obstacle ?x ?y)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       TERM section
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<=  terminal
    (true (Turn 50)))
(<=  terminal
    (UnderStone ?r))
(<=  terminal
    (not GoldAtMap))
(<=  GoldAtMap
    (true (OnMap (Gold ?q) ?x ?y)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       CONSTS section
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(InBoard 1) (InBoard 2) ... (InBoard 7)

(*10 0 0) (*10 1 10) ... (*10 10 100)

(NextTurn 0 1) (NextTurn 1 2) ... (NextTurn 49 50)

(Succ 0 1) (Succ 1 2) ... (Succ 14 15)

(+ 0 0 0) (+ 1 0 1) ... (+ 15 0 15)

(<= (+ ?x ?sy ?sz)
    (Succ ?y ?sy)
    (Succ ?z ?sz)
    (+ ?x ?y ?z))

(<= (- ?x ?y ?z)
    (+ ?y ?z ?x))

(<= (> ?x ?y)
    (Succ ?y ?x))
(<= (> ?sx ?y)
    (Succ ?x ?sx)
    (> ?x ?y))
```

Bibliography

[1] The Chess Variant Pages. http://www.chessvariants.org/, 2015.

[2] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases: The Logical Level.* Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1995.

[3] T. Adams. *Dwarf Fortress.* Bay 12 Games, 2006.

[4] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[5] K. R. Apt, H. A. Blair, and A. Walker. Foundations of deductive databases and logic programming. chapter Towards a Theory of Declarative Knowledge, pages 89–148. 1988.

[6] Barros G. A. B. and J. Togelius. Balanced civilization map generation based on Open Data. In *IEEE Congress on Evolutionary Computation*, pages 1482–1489, 2015.

[7] C. Baral. *Knowledge representation, reasoning and declarative problem solving.* Cambridge University Press, 2003.

[8] J. Baxter, A. Tridgell, and L. Weaver. Learning to Play Chess Using Temporal Differences. *Machine Learning*, 40(3):243–263, 2000.

[9] D. F. Beal and M. C. Smith. Temporal Coherence and Prediction Decay in TD Learning. In *International Joint Conference on Artifical Intelligence*, pages 564–569, 1999.

[10] M. G Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[11] J. Bergin and W. B. MacLeod. Continuous time repeated games. *International Economic Review*, pages 21–37, 1993.

[12] Y. Björnsson. Learning Rules of Simplified Boardgames by Observing. In *European Conference on Artificial Intelligence*, volume 242 of *FAIA*, pages 175–180. 2012.

[13] Y. Björnsson and H. Finnsson. CadiaPlayer: A Simulation-Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15, 2009.

[14] Y. Björnsson and S. Schiffel. Comparison of GDL Reasoners. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'13)*, pages 55–62, 2013.

[15] M. Booth. The AI systems of Left 4 Dead. In *Keynote, Fifth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2009.

[16] B. Bouzy. Old-fashioned computer Go vs Monte-Carlo Go. IEEE Symposium on Computational Intelligence in Games (Invited Tutorial), 2007.

[17] B. Bouzy and T. Cazenave. Computer Go: an AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[18] D. Braben and I. Bell. *Elite*. Acornsoft, Firebird, Imagineer, 1984.

[19] C. Browne. *Automatic Generation and Evaluation of Recombination Games*. PhD thesis, Queensland University of Technology, 2008.

[20] C. Browne and F. Maire. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):1–16, 2010.

[21] C. B. Browne, E Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[22] M. Campbell, A. J. Hoane, and F. Hsu. Deep Blue. *Artificial intelligence*, 134(1):57–83, 2002.

[23] A. Cano, J. Ruiz, and P. García. Inferring Subclasses of Regular Languages Faster Using RPNI and Forbidden Configurations. In *Grammatical Inference: Algorithms and Applications*, volume 2484 of *LNCS*, pages 28–36. 2002.

[24] L. Cardamone, P.L. Lanzi, and D. Loiacono. TrackGen: An interactive track generator for TORCS and Speed-Dreams. *Applied Soft Computing*, 28:550–558, 2015.

[25] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 216–217, 2008.

[26] J. Clune. Heuristic Evaluation Functions for General Game Playing. In *AAAI Conference on Artificial Intelligence*, pages 1134–1139, 2007.

[27] M. Cook and S. Colton. Multi-faceted evolution of simple arcade games. In *IEEE Conference on Computational Intelligence and Games*, pages 289–296, 2011.

[28] Valve Corporation. *Left 4 Dead*. Valve Corporation, 2008.

[29] V. S. Costa, R. Rocha, and L. Damas. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12(1-2):5, 2012.

[30] R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games*, pages 72–83, 2007.

[31] E. Cox, E. Schkufza, R. Madsen, and M. Genesereth. Factoring General Games using Propositional Automata. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'09)*, 2009.

[32] S. Cresswell and P. Gregory. Generalised Domain Model Acquisition from Action Traces. In *International Conference on Automated Planning and Scheduling*, pages 42–49, 2011.

[33] S. N. Cresswell, T.L. McCluskey, and M. M. West. Acquisition of Object-Centred Domain Models from Planning Examples. In *International Conference on Automated Planning and Scheduling*, pages 338–341, 2009.

[34] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars.* Cambridge University Press, 2010.

[35] A. Dickins. *A Guide to Fairy Chess.* Dover, 1971.

[36] J. Dormans. Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. In *Workshop on Procedural Content Generation in Games*, PCGames '10, pages 1:1–1:8, 2010.

[37] S. Droste and J. Fürnkranz. Learning the piece values for three chess variants. *International Computer Games Association Journal*, pages 209–233, 2008.

[38] P. Dupont. Incremental regular inference. In *Grammatical Interference: Learning Syntax from Sentences*, volume 1147 of *LNCS*, pages 222–237. 1996.

[39] M. Ebner, J. Levine, S. M. Lucas, T. Schaul, T. Thompson, and J. Togelius. Towards a video game description language. In *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-ups*, pages 85–100. 2013.

[40] S. L. Epstein. Toward an ideal trainer. *Machine Learning*, 15(3):251–277, 1994.

[41] D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. Prager, et al. Building Watson: An overview of the DeepQA project. *AI magazine*, 31(3):59–79, 2010.

[42] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208, 1971.

[43] H. Finnsson. Generalized Monte-Carlo Tree Search Extensions for General Game Playing. In *AAAI Conference on Artificial Intelligence*, pages 1550–1556, 2012.

[44] H. Finnsson. *Simulation-Based General Game Playing.* PhD thesis, Reykjavík University, 2012.

[45] H. Finnsson and Y. Björnsson. Simulation-based Approach to General Game Playing. In *AAAI Conference on Artificial Intelligence*, 2008.

[46] H. Finnsson and Y. Björnsson. Simulation control in general game playing agents. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'09)*, pages 21–26, 2009.

[47] H. Finnsson and Y. Björnsson. Learning Simulation Control in General Game Playing Agents. In *AAAI Conference on Artificial Intelligence*, pages 954–959, 2010.

[48] H. Finnsson and Y. Björnsson. CadiaPlayer: Search-Control Techniques. *KI 2011: Advances in Artificial Intelligence*, 25(1):9–16, 2011.

[49] H. Finnsson and Y. Björnsson. Game-Tree Properties and MCTS Performance. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'11)*, pages 23–30, 2011.

[50] J. M. Font, T. Mahlmann, D. Manrique, and J. Togelius. A Card Game Description Language. In *Applications of Evolutionary Computation*, volume 7835 of *LNCS*, pages 254–263. 2013.

[51] J. M. Font, T. Mahlmann, D. Manrique, and J. Togelius. Towards the automatic generation of card games through grammar-guided genetic programming. In *International Conference on the Foundations of Digital Games*, pages 360–363, 2013.

[52] F. Frydenberg, K. R. Andersen, S. Risi, and J. Togelius. Investigating MCTS modifications in general video game playing. In *IEEE Conference on Computational Intelligence and Games*, pages 107–113, 2015.

[53] Firaxis Games. *Civilization IV.* 2K Games & Aspyr, 2005.

[54] P. García, A. Cano, and J. Ruiz. A Comparative Study of Two Algorithms for Automata Identification. In *Grammatical Inference: Algorithms and Applications*, volume 1891 of *LNCS*, pages 115–126. 2000.

[55] T. Geffner and H. Geffner. Width-based Planning for General Video-Game Playing. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'15)*, pages 15–21, 2015.

[56] F. Geißer, T. Keller, and R. Mattmüller. Past, Present, and Future: An Optimal Online Algorithm for Single-Player GDL-II Games. In *ECAI*, pages 357–362, 2014.

[57] M. Gelfond. Answer Sets. In *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 285 – 316. 2008.

[58] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

[59] S. Gelly and D. Silver. Combining Online and Offline Knowledge in UCT. In *International Conference on Machine Learning*, pages 273–280, 2007.

[60] S. Gelly and D. Silver. Achieving Master Level Play in 9 x 9 Computer Go. In *AAAI*, volume 8, pages 1537–1540, 2008.

[61] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo go. Technical Report RR-6062, INRIA, 2006.

[62] M. Genesereth. Stanford University General Game Playing Coursera online course. `https://www.coursera.org/course/ggp`, 2014.

[63] M. Genesereth and Y. Björnsson. The International General Game Playing Competition. *AI Magazine*, 34(2):107–111, 2013.

[64] M. Genesereth and R. E. Fikes. Knowledge Interchange Format Version 3.0 Reference Manual. Technical Report LG-1992-01, Stanford Logic Group, 1992.

[65] M. Genesereth, N. Love, and B. Pell. General Game Playing: Overview of the AAAI Competition. *AI Magazine*, 26:62–72, 2005.

[66] M. Genesereth and M. Thielscher. *General Game Playing.* Morgan & Claypool, 2014.

[67] M. Gherrity. *A Game-Learning Machine.* PhD thesis, University of California, San Diego, 1993.

[68] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[69] P. Gregory, Y. Björnsson, and S. Schiffel. The GRL System: Learning Board Game Rules With Piece-Move Interactions. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'15)*, pages 55–62, 2015.

[70] M. Gunther and S. Schiffel. Dresden General Game Playing Server. `http://ggpserver.general-game-playing.de`, 2013.

[71] M. Günther, S. Schiffel, and M. Thielscher. Factoring General Games. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'09)*, pages 27–33, 2009.

[72] K. Hartsook, A. Zook, S. Das, and M. O. Riedl. Toward supporting stories with procedurally generated game worlds. In *IEEE Conference on Computational Intelligence and Games*, pages 297–304, 2011.

[73] E. J. Hastings, R. K. Guha, and K. O. Stanley. Evolving content in the Galactic Arms Race video game. In *IEEE Symposium on Computational Intelligence and Games*, pages 241–248, 2009.

[74] S. Haufe, D. Michulke, S. Schiffel, and M. Thielscher. Knowledge-Based General Game Playing. *KI 2011: Advances in Artificial Intelligence*, 25(1):25–33, 2011.

[75] P. Hingston. A new design for a turing test for bots. In *IEEE Symposium on Computational Intelligence and Games*, pages 345–350. IEEE, 2010.

[76] V. Hom and J. Marks. Automatic design of balanced board games. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 25–30, 2007.

[77] A. Hufschmitt, J Méhat, and J.-N. Vittaut. MCTS Playouts Parallelization with a MPPA Architecture. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'15)*, pages 63–69, 2015.

[78] Inc. Interactive Data Visualization. *SpeedTree for Games v7*, 2016.

[79] R. Isaacs. *Differential games: a mathematical theory with applications to warfare and pursuit, control and optimization.* Courier Corporation, 1999.

[80] W. Jaśkowski, P. Liskowski, M. Szubert, and K. Krawiec. Improving Coevolution by Random Sampling. In *15th Annual Conference on Genetic and Evolutionary Computation*, pages 1141–1148, 2013.

[81] L. Johnson, G.N. Yannakakis, and J. Togelius. Cellular Automata for Real-time Generation of Infinite Cave Levels. In *Workshop on Procedural Content Generation in Games*, PCGames '10, pages 10:1–10:4, 2010.

[82] V. Julian and V. Botti. Developing real-time multi-agent systems. *Integrated Computer-Aided Engineering*, 11(2):135–149, 2004.

[83] Ł. Kaiser. Synthesis for Structure Rewriting Systems. In *International Symposium on Mathematical Foundations of Computer Science*, volume 5734 of *LNCS*, pages 415–427, 2009.

[84] Ł. Kaiser. Learning Games from Videos Guided by Descriptive Complexity. In *AAAI Conference on Artificial Intelligence*, pages 963–969, 2012.

[85] Ł. Kaiser and Ł. Stafiniak. Playing Structure Rewriting Games. In *Artificial General Intelligence*, 2010.

[86] Ł. Kaiser and Ł. Stafiniak. First-Order Logic with Counting for General Game Playing. In *AAAI Conference on Artificial Intelligence*, pages 791–796, 2011.

[87] Ł. Kaiser and Ł. Stafiniak. Translating the Game Description Langauge to Toss. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'11)*, pages 91–98, 2011.

[88] A. Khalifa, D. Perez, S. Lucas, and J. Togelius. General Video Game Level Generation. In *Genetic and Evolutionary Computation Conference*, 2016.

[89] S. O. Kimbrough, G. J. Koehler, M. Lu, and D. H. Wood. On a Feasible–Infeasible Two-Population (FI-2Pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch. *European Journal of Operational Research*, 190(2):310–327, 2008.

[90] J. Kirk and J. E. Laird. Interactive task learning for simple games. *Advances in Cognitive Systems*, 3:11–28, 2014.

[91] P. Kissmann and S. Edelkamp. Instantiating General Games Using Prolog or Dependency Graphs. In *KI 2010: Advances in Artificial Intelligence*, volume 6359 of *LNCS*, pages 255–262. 2010.

[92] L. Kocsis and C. Szepesvári. Bandit Based Monte-carlo Planning. In *European Conference on Machine Learning*, pages 282–293, 2006.

[93] R. Koster and W. Wright. *A Theory of Fun for Game Design*. Paraglyph Press, 2004.

[94] J. Kowalski. Embedding a Card Game Language into a General Game Playing Language. In *Proceedings of the 7th European Starting AI Researcher Symposium*, pages 161–170, 2014.

[95] J. Kowalski and A. Kisielewicz. Game Description Language for Real-time Games. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'15)*, pages 23–30, 2015.

[96] J. Kowalski and A. Kisielewicz. Testing General Game Players Against a Simplified Boardgames Player Using Temporal-difference Learning. In *IEEE Congress on Evolutionary Computation*, pages 1466–1473, 2015.

[97] J. Kowalski and A. Kisielewicz. Towards a Real-time Game Description Language. In *International Conference on Agents and Artificial Intelligence*, volume 2, pages 494–499, 2016.

[98] J. Kowalski, J. Sutowicz, and M. Szykuła. Simplified Boardgames. arXiv:1606.02645 [cs.AI], 2016.

[99] J. Kowalski and M. Szykuła. Game Description Language Compiler Construction. In *AI 2013: Advances in Artificial Intelligence*, volume 8272 of *LNCS*, pages 234–245. 2013.

[100] J. Kowalski and M. Szykuła. Procedural Content Generation for GDL Descriptions of Simplified Boardgames. arXiv:1108.1494 [cs.AI], 2015.

[101] J. Kowalski and M. Szykuła. Evolving Chesslike Games Using Relative Algorithm Performance Profiles. In *Applications of Evolutionary Computation*, volume 9597 of *LNCS*, pages 574–589. 2016.

[102] T. Kozelek. Methods of MCTS and the game Arimaa. Master's thesis, Charles University in Prague, 2009.

[103] G. Kuhlmann, K. Dresner, and P. Stone. Automatic Heuristic Construction in a Complete General Game Player. In *AAAI Conference on Artificial Intelligence*, pages 1457–1462, 2006.

[104] G. Kuhlmann and P. Stone. Graph-Based Domain Mapping for Transfer Learning in General Games. In *European Conference on Machine Learning*, pages 188–200, 2007.

[105] A. Landau. Rewriting chess (again). Alloy GGP blog, may 2014.

[106] K. Lang. Random DFA's can be Approximately Learned from Sparse Uniform Examples. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, pages 45–52, 1992.

[107] J. Levine, C. B. Congdon, M. Ebner, G. Kendall, S. M. Lucas, R. Miikkulainen, T. Schaul, and T. Thompson. General Video Game Playing. In *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 77–83. 2013.

[108] R. Levinson. Towards Domain-Independent Machine Intelligence. In *Proceedings on Conceptual Graphs for Knowledge Representation*, ICCS, pages 254–273, 1993.

[109] G. M Levitt. *The Turk, Chess Automation.* McFarland & Company, Incorporated Publishers, 2000.

[110] D. Levy and M. Newborn. *How computers play chess.* Computer Science Press, Inc., 1991.

[111] A. Liapis. Exploring the Visual Styles of Arcade Game Assets. In *Evolutionary and Biologically Inspired Music, Sound, Art and Design*, pages 92–109. 2016.

[112] A. Liapis and G. N Yannakakis. Refining the Paradigm of Sketching in AI-Based Level Design. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 128–134, 2015.

[113] R.v.d. Linden, R. Lopes, and R. Bidarra. Designing Procedurally Generated Levels. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013.

[114] N. Lipovetzky and H. Geffner. Width and Serialization of Classical Planning Problems. In *European Conference on Artificial Intelligence*, pages 540–545, 2012.

[115] J. W. Lloyd and R. W. Topor. A basis for deductive database systems II. *The Journal of Logic Programming*, 3(1):55–67, 1986.

[116] D. Loiacono, P. L. Lanzi, J. Togelius, E. Onieva, D. A. Pelta, M. V. Butz, T. D. Lonneker, L. Cardamone, D. Perez, Y. Sáez, et al. The 2009 simulated car racing championship. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):131–147, 2010.

[117] R. J. Lorentz. *Amazons Discover Monte-Carlo*, pages 13–24. 2008.

[118] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General Game Playing: Game Description Language Specification. Technical report, Stanford Logic Group, 2006.

[119] T. Mahlmann, J. Togelius, and G. N. Yannakakis. Towards Procedural Strategy Game Generation: Evolving Complementary Unit Types. In *Applications of Evolutionary Computation*, volume 6624 of *LNCS*, pages 93–102, 2011.

[120] T. Mahlmann, J. Togelius, and G. N. Yannakakis. Evolving card sets towards balancing dominion. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2012.

[121] T. Mahlmann, J. Togelius, and G. N. Yannakakis. Spicing up map generation. In *European Conference on the Applications of Evolutionary Computation*, pages 224–233, 2012.

[122] T. Mahlmann, J. Togelius, and G.N. Yannakakis. Modelling and evaluation of complex scenarios with the Strategy Game Description Language. In *IEEE Conference on Computational Intelligence and Games*, pages 174–181, 2011.

[123] J. Mańdziuk, Y. S. Ong, and K. Walędzik. Multi-game playing – A challenge for computational intelligence. In *IEEE Symposium on Computational Intelligence for Human-like Intelligence*, pages 17–24, 2013.

[124] J. Mańdziuk and M. Świechowski. Generic Heuristic Approach to General Game Playing. In *International Conference on Current Trends in Theory and Practice of Computer Science*, volume 7147 of *LNCS*, pages 649–660, 2012.

[125] J. R. Meehan. *The Metanovel: Writing Stories by Computer.* PhD thesis, Department of Computer Science, Yale University, 1976.

[126] J. Méhat and T. Cazenave. A Parallel General Game Player. *KI 2011: Advances in Artificial Intelligence*, 25(1):43–47, 2011.

[127] J. Méhat and T. Cazenave. Tree Parallelization of Ary on a Cluster. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'11)*, pages 39–43, 2011.

[128] M. Mhalla and F. Prost. Gardner's minichess variant is solved. 2013. arXiv:1307.7118 [cs.GT].

[129] D. Michulke. Neural Networks for High-Resolution State Evaluation in General Game Playing. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'11)*, pages 31–37, 2011.

[130] D. Michulke and S. Schiffel. Distance Features for General Game Playing Agents. In *International Conference on Agents and Artificial Intelligence*, 2012.

[131] D. Michulke and S. Schiffel. Admissible Distance Heuristics for General Games. In *Agents and Artificial Intelligence*, volume 358, pages 188–203. 2013.

[132] D. Michulke and M. Thielscher. Neural Networks for State Evaluation in General Game Playing. In *European Conference on Machine Learning*, pages 95–110, 2009.

[133] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[134] Mojang. *Minecraft.* Mojang, Microsoft Studios, 2011.

[135] M. Möller, M. Schneider, M. Wegner, and T. Schaub. Centurio, a General Game Player: Parallel, Java- and ASP-based. *KI 2011: Advances in Artificial Intelligence*, 25(1):17–24, 2011.

[136] S. Muggleton, A. Paes, V. Santos Costa, and G. Zaverucha. Chess Revision: Acquiring the Rules of Chess Variants through FOL Theory Revision from Examples. In *Inductive Logic Programming*, volume 5989 of *LNCS*, pages 123–130, 2010.

[137] M. J. Nelson and M. Mateas. Towards Automated Game Design. In *AI* IA: Artificial Intelligence and Human-Oriented Computing*, pages 626–637, 2007.

[138] T. S. Nielsen, G. A. B. Barros, J. Togelius, and M. J. Nelson. General Video Game Evaluation Using Relative Algorithm Performance Profiles. In *Applications of Evolutionary Computation*, volume 9028 of *LNCS*, pages 369–380. 2015.

[139] T. S. Nielsen, G. A. B. Barros, J. Togelius, and M. J. Nelson. Towards generating arcade game rules with VGDL. In *IEEE Conference on Computational Intelligence and Games*, pages 185–192, 2015.

[140] Blizzard North. *Diablo*. Blizzard Entertainment, Ubisoft, Electronic Arts, 1997.

[141] G. Ochoa. On genetic algorithms and Lindenmayer systems. In *Parallel Problem Solving from Nature*, pages 335–344, 1998.

[142] J. Oncina and P. Garcia. Identifying Regular Languages In Polynomial Time. In *Pattern Recognition and Image Analysis*, pages 49–61. 1992.

[143] S. Ontanón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss. A survey of real-time strategy game AI research and competition in Starcraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(4):293–311, 2013.

[144] D. Osman and J. Mańdziuk. Comparison of TDLeaf(λ) and TD(λ) Learning in Game Playing Domain. In *Neural Information Processing*, volume 3316 of *LNCS*, pages 549–554, 2004.

[145] R. Parekh and V. Honavar. Learning DFA from Simple Examples. *Machine Learning*, 44(1):9–35, 2001.

[146] E. P. D. Pednault. Formulating Multi-Agent, Dynamic-World Problems in the Classical Planning Framework. In *Workshop on Reasoning about Actions and Plans*, pages 47–82, 1986.

[147] B. Pell. METAGAME: A New Challenge for Games and Learning. In *Heuristic Programming in Artificial Intelligence: The Third Computer Olympiad.*, 1992.

[148] B. Pell. METAGAME in Symmetric Chess-Like Games. In *Heuristic Programming in Artificial Intelligence: The Third Computer Olympiad.*, 1992.

[149] B. Pell. *Strategy Generation and Evaluation for Meta-Game Playing*. PhD thesis, Computer Laboratory, University of Cambridge, 1993.

[150] B. Pell. A Strategic Metagame Player For General Chess-Like Games. *Computational Intelligence*, 12(1):177–198, 1996.

[151] T. Pepels, M. HM. Winands, and M. Lanctot. Real-time Monte Carlo Tree Search in Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):245–257, 2014.

[152] D. Perez, J. Dieskau, M. Hünermund, S. Mostaghim, and S. M. Lucas. Open Loop Search for General Video Game Playing. In *Conference on Genetic and Evolutionary Computation*, pages 337–344, 2015.

[153] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. Lucas, A. Couëtoux, J. Lee, C. Lim, and T. Thompson. The 2014 General Video Game Playing Competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 2015.

[154] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, and S. M. Lucas. General Video Game AI: Competition, Challenges and Opportunities. In *AAAI Conference on Artificial Intelligence*, pages 4335–4337, 2016.

[155] Diego Perez, Spyridon Samothrakis, and Simon Lucas. Knowledge-based fast evolutionary MCTS for general video game playing. In *IEEE Conference on Computational Intelligence and Games*, pages 68–75, 2014.

[156] J. Pitrat. Realization of a general game-playing program. In *IFIP Congress*, pages 1570–1574, 1968.

[157] J. Pitrat. A general game playing program. In *Artificial Intelligence and Heuristic Programming*, pages 125–155. 1971.

[158] L. Pitt and M. K. Warmuth. The Minimum Consistent DFA Problem Cannot Be Approximated Within Any Polynomial. *Journal of the Association for Computing Machinery*, 40(1):95–142, 1993.

[159] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1):255–293, 1996.

[160] R. Poli, W.B. Langdon, and N. F. McPhee. *A field guide to genetic programming.* Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[161] E. Rasmusen. *Games and Information: An Introduction to Game Theory.* Blackwell, 4th ed., 2007.

[162] T. Rauber, P. Kissmann, and J. Hoffmann. *Translating Single-Player GDL into PDDL*, volume 8077 of *LNCS*, pages 188–199. 2013.

[163] S. Risi, J. Lehman, D. B. D'ambrosio, R. Hall, and K. O. Stanley. Combining search-based procedural content generation and social gaming in the petalz video game. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 63–68, 2012.

[164] J. Romero, A. Saffidine, and M. Thielscher. Solving the Inferential Frame Problem in the General Game Description Language. In *AAAI Conference on Artificial Intelligence*, 2014.

[165] J. Ruan and M. Thielscher. Strategic and Epistemic Reasoning for the Game Description Language GDL-II. In *European Conference on Artificial Intelligence*, pages 696–701, 2012.

[166] A. Saffidine. The Game Description Language Is Turing Complete. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):320–324, 2014.

[167] A. Saffidine and T. Cazenave. A Forward Chaining Based Game Description Language Compiler. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'11)*, pages 69–75, 2011.

[168] S. Samothrakis, D. Perez, S. M. Lucas, and M. Fasli. Neuroevolution for General Video Game Playing. In *IEEE Conference on Computational Intelligence and Games*, pages 200–207, 2015.

[169] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[170] J. Schaeffer and J. van den Herik. *Chips Challenging Champions: Games, Computers and Artificial Intelligence*. Elsevier, 2002.

[171] T. Schaul. A video game description language for model-based or interactive learning. In *IEEE Conference on Computational Intelligence and Games*, pages 1–8, 2013.

[172] T. Schaul, T. Glasmachers, and J. Schmidhuber. High dimensions and heavy tails for natural evolution strategies. In *Conference on Genetic and Evolutionary Computation*, pages 845–852, 2011.

[173] R. B. Scherl and H. J. Levesque. Knowledge, action, and the frame problem. *Artificial Intelligence*, 144(1):1–39, 2003.

[174] S. Schiffel. Symmetry Detection in General Game Playing. In *AAAI Conference on Artificial Intelligence*, pages 980–985, 2010.

[175] S. Schiffel. Grounding GDL Game Descriptions. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'16)*, pages 15–22, 2016.

[176] S Schiffel and Y. Björnsson. Efficiency of GDL Reasoners. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):343–354, 2014.

[177] S. Schiffel and M. Thielscher. Fluxplayer: A Successful General Game Player. In *AAAI Conference on Artificial Intelligence*, pages 1191–1196, 2007.

[178] S. Schiffel and M. Thielscher. Automated Theorem Proving for General Game Playing. In *International Joint Conference on Artificial Intelligence*, pages 911–916, 2009.

[179] S. Schiffel and M. Thielscher. A Multiagent Semantics for the Game Description Language. In *Agents and Artificial Intelligence*, volume 67 of *Communications in Computer and Information Science*, pages 44–55. 2010.

[180] S. Schiffel and M. Thielscher. Reasoning about general games described in GDL-II. In *AAAI Conference on Artificial Intelligence*, pages 846–851, 2011.

[181] S. Schiffel and M. Thielscher. Representing and Reasoning About the Rules of General Games With Imperfect Information. *Journal of Artificial Intelligence Research*, 49:171–206, 2014.

[182] E. Schkufza, N. Love, and M. Genesereth. Propositional Automata and Cell Automata: Representational Frameworks for Discrete Dynamic Systems. In *AI 2008: Advances in Artificial Intelligence*, volume 5360 of *LNCS*, pages 56–66. 2008.

[183] M. Schofield, T. Cerexhe, and M. Thielscher. HyperPlay: A Solution to General Game Playing with Imperfect Information. In *AAAI Conference on Artificial Intelligence*, pages 1606–1612, 2012.

[184] M. Schofield and A. Saffidine. High Speed Forward Chaining for General Game Playing. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'13)*, pages 31–38.

[185] S. Schreiber. The general game playing base package. `http://code.google.com/p/ggp-base/`, 2013.

[186] M. Scirea, Y-G. Cheong, M. J. Nelson, and B-C. Bae. Evaluating Musical Foreshadowing of Videogame Narrative Experiences. In *Audio Mostly: A Conference on Interaction With Sound*, AM '14, pages 8:1–8:7, 2014.

[187] M. Scirea, J. Togelius, P. Eklund, and S. Risi. MetaCompose: A Compositional Evolutionary Music Composer. In *Evolutionary and Biologically Inspired Music, Sound, Art and Design*, pages 202–217. 2016.

[188] S. Sebastian Haufe, S. Schiffel, and M. Thielscher. Automated verification of state sequence invariants in general game playing. *Artificial Intelligence*, 187:1 – 30, 2012.

[189] N. Shaker, M. Nicolau, G.N. Yannakakis, J. Togelius, and M. O'Neill. Evolving levels for Super Mario Bros using grammatical evolution. In *IEEE Conference on Computational Intelligence and Games*, pages 304–311, 2012.

[190] N. Shaker, J. Togelius, and M.J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer, 2015.

[191] S. Sharma, Z. Kobti, and S. Goodwin. Knowledge Generation for Improving Simulations in UCT for General Game Playing. In *AI 2008: Advances in Artificial Intelligence*, volume 5360 of *LNCS*, pages 49–55. 2008.

[192] X. Sheng and D. Thuente. Extending the General Game Playing Framework to Other Languages. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'11)*, 2011.

[193] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[194] L. K. Simon and M. B. Stinchcombe. Extensive Form Games in Continuous Time: Pure Strategies. *Econometrica*, 57(5):1171–1214, 1989.

[195] C. F. Sironi and M. H. M. Winands. Optimizing Propositional Networks. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'16)*, pages 7–14, 2016.

[196] K. O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8(2):131–162, 2007.

[197] Paradox Development Studios. *Stellaris.* Paradox Interactive, 2016.

[198] Westwood Studios. *Dune II*, 1992.

[199] N. R. Sturtevant. *An Analysis of UCT in Multi-player Games*, pages 37–49. 2008.

[200] R. S. Sutton. Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3(1):9–44, 1988.

[201] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning.* MIT Press, 1998.

[202] M. Świechowski. *Adaptive simulation-based meta-heuristic methods in synchronous multi-player games.* PhD thesis, Polish Academy of Sciences, 2015.

[203] M. Świechowski and J. Mańdziuk. Fast interpreter for logical reasoning in general game playing. *Journal of Logic and Computation*, 2014.

[204] M. Świechowski and J. Mańdziuk. Prolog versus specialized logic inference engine in General Game Playing. In *IEEE Conference on Computational Intelligence and Games*, pages 1–8, 2014.

[205] M. Świechowski and J. Mańdziuk. Self-Adaptation of Playing Strategies in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):367–381, 2014.

[206] M. Świechowski and J. Mańdziuk. Specialized vs. Multi-game Approaches to AI in Games. In *IEEE International Conference on Intelligent Systems*, pages 243–254, 2014.

[207] M. Świechowski and J. Mańdziuk. *A Hybrid Approach to Parallelization of Monte Carlo Tree Search in General Game Playing*, pages 199–215. 2016.

[208] M. Świechowski, J. Mańdziuk, and Y. S. Ong. Specialization of a UCT-based General Game Playing Program to Single-Player Games. *IEEE Transactions on Computational Intelligence and AI in Games*, (99), 2015.

[209] M. Świechowski, K. Merrick, J. Mańdziuk, and H. Abbass. *Human-Machine Cooperation Loop in Game Playing*, volume 8, pages 310–323. 2015.

[210] M. Świechowski, H. Park, J. Mańdziuk, and K. Kim. Recent Advances in General Game Playing. *The Scientific World Journal*, 2015, 2015.

[211] O. Syed and A. Syed. Arimaa – a new game designed to be difficult for computers. *ICGA*, 26(2):138–139, 2003.

[212] I. Szita, G. Chaslot, and P. Spronck. Monte-Carlo Tree Search in Settlers of Catan. In *International Conference on Advances in Computer Games*, pages 21–32, 2010.

[213] M. Szubert, W. Jaśkowski, P. Liskowski, and K. Krawiec. The Role of Behavioral Diversity and Difficulty of Opponents in Coevolving Game-Playing Agents. In *Applications of Evolutionary Computation*, volume 9028 of *LNCS*, pages 394–405, 2015.

[214] G. Tesauro. Practical Issues in Temporal Difference Learning. *Machine Learning*, 8(3-4):257–277, 1992.

[215] G. Tesauro. TD-Gammon, a Self-teaching Backgammon Program, Achieves Master-level Play. *Neural Computation*, 6(2):215–219, 1994.

[216] M. Thielscher. Answer Set Programming for Single-Player Games in General Game Playing. In *Logic Programming*, volume 5649 of *LNCS*, pages 327–341. 2009.

[217] M. Thielscher. A General Game Description Language for Incomplete Information Games. In *AAAI Conference on Artificial Intelligence*, pages 994–999, 2010.

[218] M. Thielscher. GDL-II. *Künstliche Intelligenz*, 25:63–66, 2011.

[219] M. Thielscher. General Game Playing in AI Research and Education. In *KI 2011: Advances in Artificial Intelligence*, volume 7006 of *LNCS*, pages 26–37. 2011.

[220] M. Thielscher. The General Game Playing Description Language is Universal. In *International Joint Conference on Artificial Intelligence*, pages 1107–1112, 2011.

[221] M. Thielscher. *Translating General Game Descriptions into an Action Language*, volume 6565 of *LNCS*, pages 300–314. 2011.

[222] M. Thielscher and D. Zhang. From general game descriptions to a market specification language for general trading agents. In *Agent-Mediated Electronic Commerce. Designing Trading Strategies and Mechanisms for Electronic Markets*, pages 259–274. 2010.

[223] J. Togelius, A. J. Champandard, P. L. Lanzi, M. Mateas, A. Paiva, M. Preuss, and K. O. Stanley. Procedural Content Generation: Goals, Challenges and Actionable Steps. In *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 61–75. 2013.

[224] J. Togelius, R. De Nardi, and S. M. Lucas. Towards automatic personalised content creation for racing games. In *IEEE Symposium on Computational Intelligence and Games*, pages 252–259, 2007.

[225] J. Togelius, M. J. Nelson, and A. Liapis. Characteristics of Generatable Games. In *FDG Workshop on Procedural Content Generation*, 2014.

[226] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, G. N. Yannakakis, and C. Grappiolo. Controllable procedural map generation via multiobjective evolution. *Genetic Programming and Evolvable Machines*, 14(2):245–277, 2013.

[227] J. Togelius and J. Schmidhuber. An experiment in automatic game design. In *IEEE Symposium on Computational Intelligence and Games*, pages 111–118, 2008.

[228] J. Togelius, N. Shaker, S. Karakovskiy, and G. N. Yannakakis. The Mario AI Championship 2009-2012. *AI Magazine*, 34(3):89–92, 2013.

[229] M. Toy, G. Wichman, and K. Arnold. *Rogue*, 1980.

[230] B. A. Trakhtenbrot and Ya M Barzdin. *Finite automata: Behavior and Synthesis.* American Elsevier Publishing Company, 1973.

[231] M. Trutman and S. Schiffel. *Creating Action Heuristics for General Game Playing Agents*, volume 614 of *CCIS*, pages 149–164. 2016.

[232] J.-N. Vittaut and J Méhat. Efficient Grounding of Game Descriptions with Tabling. In *Computer Games*, volume 504 of *CCIS*, pages 105–118. 2014.

[233] J.-N. Vittaut and J Méhat. Fast Instantiation of GGP Game Descriptions Using Prolog with Tabling. In *European Conference on Artificial Intelligence* , volume 263 of *FAIA*, pages 1121–1122, 2014.

[234] K. Walędzik and J. Mańdziuk. An Automatically Generated Evaluation Function in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):258–270, 2014.

[235] K. Waugh. Faster State Manipulation in General Games using Generated Code. In *IJCAI Workshop on General Intelligence in Game-Playing Agents(GIGA'09)*, 2009.

[236] Wikipedia. Fairy chess piece — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/wiki/Fairy_chess_piece&oldid=711408491`, April 2016.

[237] B Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner. *TORCS: The Open Racing Car Simulator, v1.3.5*, 2013.

[238] D. Yu and A. Hull. *Spelunky.* Mossmouth, 2009.

[239] H. Zhang, D. Liu, and W. Li. Space-Consistent Game Equivalence Detection in General Game Playing. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'15)*, pages 47–53, 2015.

[240] D. Zhao, S. Schiffel, and M. Thielscher. Decomposition of Multi-Player Games. In *Australasian Joint Conference on Artificial Intelligence*, volume 5866, pages 475–484, 2009.

[241] A. Zook and M. O. Riedl. Automatic Game Design via Mechanic Generation. In *AAAI Conference on Artificial Intelligence*, pages 530–537, 2014.