General Game Playing

Adaptable, Multi-Game Playing Programs

Ricardo Filipe Amendoeira

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Nuno Cavaco Gomes Horta
Prof. João Paulo Carvalho

December 2015

Abstract

Artificial Intelligence research has lead to many systems that perform incredibly well for the specific task they were designed for, but are useless for anything else. For example, while Deep Blue could defeat Garry Kasparov, the world's chess champion, it was utterly incapable of playing checkers or even tic-tac-toe. Also, with these systems, all the interesting analysis of the game is done ahead of time and is done by the developers, not the system. General Game Playing (GGP) aims to create computer systems capable of playing any game without human assistance, being given only the game rules. Such systems must be highly adaptable and should be capable of high-level tasks such as reasoning, learning or deduction, among others. Player performance is dictated by how well it handles the following three tasks: Interpreting the game rules (efficiently representing this knowledge), analyzing the game rules (recognizing patterns and other features) and deriving a strategy for the given game (quickly searching through the possible actions).

In this work an overview of GGP is given and the state of the art solutions are presented.

Keywords: General Game Playing, Multi-Game Playing, Artificial Intelligence,

Resumo

O teu resumo aqui...

Palavras-Chave: as tuas palavras chave

Contents

List of Tables **x**

List of Figures **xiii**

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Multi-game playing . . . 3
1.2.1 Early attempts at multi-game playing . . . 4
1.2.1.1 Hoyle . . . 4
1.3 Objectives . . . 4
1.4 Planning . . . 5

2 Background **7**
2.1 The basics of General Game Playing . . . 7
2.1.1 Game Description Language . . . 8
2.1.2 Game Manager . . . 8
2.1.3 Game Player . . . 8
2.2 Markov Decision Problem . . . 8
2.3 Basics of a GDL player . . . 9
2.3.1 Measuring performance . . . 10
2.3.1.1 Agon Skill . . . 11
2.4 Reasoner Techniques . . . 11
2.4.1 Prolog Interpreters/Provers . . . 11
2.4.2 Propositional networks . . . 11
2.5 Game Planner Techniques . . . 12
2.5.1 Heuristic based techniques . . . 12
2.5.1.1 MinMax . . . 13
2.5.1.2 Alpha-Beta Pruning . . . 14
2.5.1.3 Iterative deepening depth-first search . . . 14
2.5.2 Monte Carlo Tree Search . . . 14
2.5.2.1 UCT . . . 15
2.5.2.2 Heuristics in MCTS . . . 15

3 State of the Art **17**
3.1 The world competition as a benchmark . . . 17
3.2 The Tiltyard game server as a benchmark . . . 17
3.3 Results and characteristics of a few well important players . . . 17
3.3.1 Galvanize . . . 18

3.3.2 Sancho . . . 18
3.3.3 TurboTurtle . . . 19
3.3.4 Ari . . . 19
3.3.5 FluxPlayer . . . 19
3.3.6 Random . . . 19
3.4 Comparison . . . 20

4 Preliminary Results **21**

Bibliography **22**

A GDL example: tic tac toe **25**

B Calculation of *Agon Skill* (in Python2) **29**

List of Tables

1.1 Expected dates for the dissertation objectives 5

3.1 Results and characteristics of a few important players 18

List of Figures

1.1 Gantt chart for the dissertation 5

2.1 Match Components 7
2.2 Simplified example of a GGP Player architecture 10
2.3 Basic components of propositional networks 12
2.4 Example of a propositional network with 2 inputs and 1 transition 12
2.5 Example of a MinMax game tree 13
2.6 Example of a pruned game tree 14

Chapter 1

Introduction

1.1 Motivation

Game playing has always been a fundamental part of Artificial Intelligence (AI) research, as it can be used to test strategy in a straightforward way. Different games can be created to test specific features or properties that are of research interest, such as opponent modeling.

Throughout the history of AI research there was always a big focus on the ability of playing specific games, like chess, well. This focus led to systems like Deep Blue (the computer system that defeated Garry Kasparov at chess in the 90's) that, while very advanced on the games they are designed for, delegate all the interesting analysis to the system designers. These systems are also completely useless for all games other than the one they were designed for (even if the difference is very small).

This over-specialization limits the usefulness of these systems. It is then of our interest to also be able to design more general systems, which could be closer to one of the most powerful features of human intelligence, adaptability:

> *"A human being should be able to (...) design a building, write a sonnet, balance accounts, build a wall, set a bone, comfort the dying, take orders, give orders, cooperate, act alone, solve equations, analyze a problem, pitch manure, program a computer, fight efficiently, die gallantly.*
> *Specialization is for insects."*
>
> - Robert A. Heinlein, *Time Enough for Love*

General Game Playing (GGP) aims to develop general game playing systems, systems that can play any game when given the rules, acting as a stepping stone for General Intelligence research. Artificial General Intelligence has been a topic for science-fiction stories and, if possible, can be the biggest revolution in human history, allowing for an unprecedented ability of problem-solving.

1.2 Multi-game playing

Multi-game playing is the concept of developing programs that can play not just one, but many different games. This is not to be mistaken with combining many programs into one, multi-game players should be able to adapt to games unknown to them as well. Doing this requires many traits shared by *Artificial Intelligence* research, such as deduction, reasoning, problem solving, intelligent search, knowledge representation, planning and learning.

General Game Playing was proposed in 2005 by Standford University as a platform for for multi-game playing research, along with a language for describing games, the Game Description Language (GDL). An online course was created by Standford University and there are several new GGP projects every year, as well as various competitions, the biggest one being the annual GGP competition, held at the AAAI conference.

More recently another platform also appeared, General Video-Game Playing (GGVP), which focuses on video games.

1.2.1 Early attempts at multi-game playing

Possibly the earliest work on the subject was done in 1969 in [1]. There were other notable works in the 90's, like SAL, Hoyle, Morph, METAGAMER (all sumarized in [2]) and Zillions of Games. Hoyle is described in more detail below.

1.2.1.1 Hoyle

A system developed in the 90's, using a training scheme called lesson and practice, where lessons are games played against an expert and practice is self-playing. Predates GDP and was developed for 2-player games. The system used a set of game independent Advisers, each specialized in a game aspect such as position. These Advisers could recommend moves that could then be chosen by higher tier advisers. There were 3-tiers, depending on specialization:

1st. These Advisers specialized in immediate consequences: they performed very shallow searches to avoid things like instant loss moves. These decisions were final.

2nd. Advisers in this tier chose moves according to certain goals. These decisions were also final.

3rd. Advisers in the last tier differed from the first tiers in an important way: The decision of each Adviser wasn't final, the final decision was decided by a process similar to taking a vote between the Advisers in this 3rd tier. Advisers votes were weighted in accordance to the lesson and practice results: Advisers that were more often correct during the training stage received bigger weights.

This process of weighting the Advisers was crucial to the performance of the system and could even be worse than a random player if done incorrectly. If none of the 3rd tier advisers were even remotely related to the game being played the results would also be disappointing. Having a varied pool of Advisers was for this reason vital but they were never, by definition, general enough to be useful for any game. Hoyle was tested in 18 two-player board games, its potential in complex games was never verified.

1.3 Objectives

Main Objectives:

- Research and compare current techniques and solutions for GGP players
- Improve or develop new techniques for game playing
- Use the improvements to make a new GGP player
- Benchmark the player against other existing GGP players

Secondary Objectives:

- Publish the results
- Take the online course on GGP
- Enter the annual GGP competition
- Enter other relevant competitions

1.4 Planning

Table 1.1: Planned dates for the objectives

Objective	Start date	End date
Research and compare current techniques and solutions for GGP players	18/Jan/2016	14/Feb/2016
Improve or develop new techniques for game playing	01/Feb/2016	10/Apr/2016
Use the improvements to make a new GGP player	11/Apr/2016	22/May/2016
Benchmark the player against other existing GGP players	23/May/2016	12/Jun/2016
Publish the results	20/Jun/2016	17/Jul/2016
Take the online course on GGP	28/Mar/2016	17/Jul/2016
Enter the annual GGP competition	06/Jun/2016	15/Jun/2016
Enter other relevant competitions	-	-

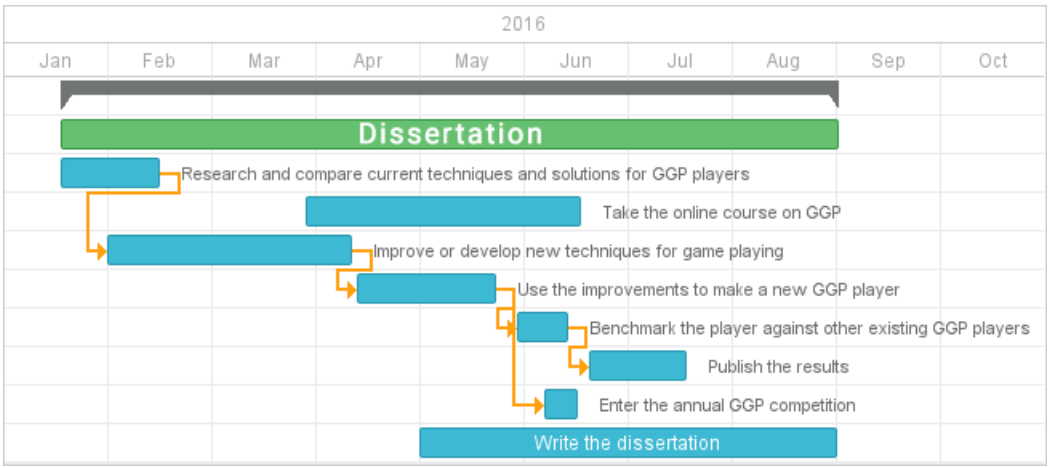

Figure 1.1: Gantt chart for the dissertation

Chapter 2

Background

2.1 The basics of General Game Playing

General Game Playing is a project of the Stanford Logic Group of Stanford University, California, which aims to create a platform for GGP. Since 2005, there have been annual GGP competitions at the AAAI Conference.

A GGP match consists of 3 major components:

- Game Description: The game rules, in GDL.
- Game Manager: This system acts as a referee and manages communication with the players and other systems like graphics for the spectators. *State Data* is usually part of the Game Manager.
- Players: Players are the most interesting component of a GGP game, they need.

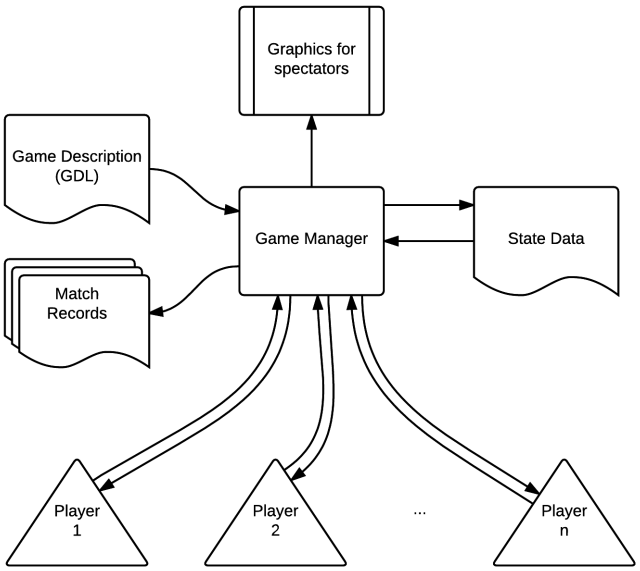

Figure 2.1: Match Components

At the beginning of a match, the Game Manager sends all Players a match identifier, the game description, the role of each player and the time limits (for preparation (*startclock*) and for each round (*playclock*)).

The match commences, after all players respond, with the Game Manager requesting a move from all players. Each round ends when all players send their moves or the time limit runs out (a random legal move is chosen for players that don't respond in time), after which the Game Manager will send each player another request along with all moves taken in the previous round.

2.1.1 Game Description Language

The GDL is the standard way of describing games in the GGP community. GGP players interpret the language using something called a *Reasoner*. Choosing a good way of interpreting the game rules is one of the keys to performance and so many players develop their own custom *reasoners*.

It can describe any finite deterministic move-based strategy game with an arbitrary number of players (most board games). GDL-II is an extension that has been made to allow for probabilistic games and incomplete information, like most card games.

Both GDL and GDL-II are variants of Datalog (query and rule language similar to prolog) and use first order logic. Since GDL is a very conceptual description of the rules their interpretation is very computationally expensive. Choosing a good way of doing this interpretation (components that do this are called reasoners) is therefore very important to player performance, even in the recent years.

An example of tic tac toe described in GDL with some syntax explanation can be seen in A

2.1.2 Game Manager

The purpose of the Game Manager is to be a single source of truth about what's happening in a match, and verify all moves taken by players. It must be able to interpret GDL, to verify these moves

Players communicate their moves to the Game Manager (via HTTP), who checks the validity of the moves. A random legal move is chosen if a player chooses an illegal move or doesn't respond in time.

It should also provide a way of archiving the match history (all game states and moves taken) and other useful features like an interface for spectating.

2.1.3 Game Player

Game Players are systems that can interpret a GDL game description, communicate with the Game Manager and devise strategies to maximize their result in a certain game. Their aim is to be as general as possible while also having reasonably good performance in any game, which is a surprisingly difficult feat. Suffice it to say, sophisticated AI techniques like heuristics are very hard to successfully apply in a general, domain independent, way.

2.2 Markov Decision Problem

The Markov Decision Problem (MDP) provides a way of mathematically modeling decision making and can help to formalize the task of GGP Players. In short, a Markov Decision Process is composed of 5 components:

- States - S : The set of all possible states of the problem.
- Actions - $A(s)$: The set of possible actions in state s.
- Model - $T(s, a, s') \sim Pr(s'|s, a)$: The laws of the universe in which the problem is contained. In other words, what is the outcome (state s') of taking an action a in state s. The model can be deterministic or stochastic ($Pr(s'|s, a)$), where multiple outcomes are possible.

- Reward - $R(s, a, s')$: What is the reward of taking action a in state s and transitioning to state s'.
- Discount Factor - γ : Reduces the importance of distant rewards. It's important in the common case of a stochastic problem, since distant rewards may become unreachable due to external influence.

A policy, π, defines what action $\pi(s)$ the agent will take when in state s. Policies are solutions to MDP's.

In the domain of General Game Playing each different match has it's own MDP, since the Model is defined not only by the game rules but also by the behavior of other players. Rewards and discount factors are also not solely dependent on the game rules (is it always a good move to take a Knight in chess?). Most GGP matches have, therefore, unknown rewards and probabilities from the point of view of the player, making them *reinforcement learning* problems. In these problems it is useful to define the function $Q(s, a)$, given by 2.1, and is an assessment of the quality of taking action a in state s.

$$Q(s, a) = \sum_{s'} Pr(s'|s, a)(R(s, a, s') + \gamma V(s')) \tag{2.1}$$

$V(s')$ is the value of being in state s', the discounted sum of the expected rewards to be earned (on average) by following the policy from state s' on-wards. There are several ways to implement this calculation.

If all the probabilities and rewards are known the policy can be solved recursively according to 2.2 and 2.3.

$$\pi(s) \equiv argmax_a \left\{ \sum_{s'} Pr(s'|s, a)(R(s, a, s') + \gamma V(s')) \right\} \tag{2.2}$$

$$V(s) \equiv \sum_{s'} Pr(s'|s, \pi(s))(R(s, \pi(s), s') + \gamma V(s')) \tag{2.3}$$

2.3 Basics of a GDL player

A simplified example of a GGP Player architecture can be seen in figure 2.2. As seen in the figure, the player must have some form of these components:

- HTTP Server: Interfaces with the Game Master
- GDL Interpreter (also known as *Reasoner*): Analyses the game rules, in some cases changing their representation into a more efficient data structure
- Game State: Holds any relevant information about the game, like the current state or past moves by opponents (to maybe model their strategy)
- Game Planner: Attempts to choose the best action for the current state of the game. If it uses simulations as part of its operation (very common) it may interface with the *reasoner* to discover what actions are available in each of the simulated states.

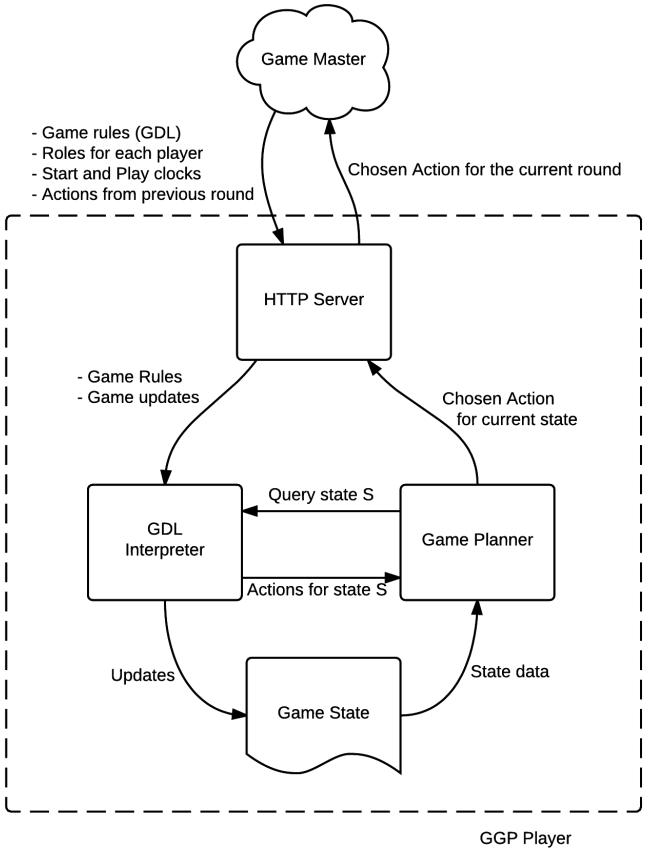

Figure 2.2: Simplified example of a GGP Player architecture

The Game Planner is the biggest factor for system performance, as it is both the component that chooses the strategy and the one that requires the most computing power. The Reasoner can also affect performance in a smaller but relevant way by using a significant portion of the computational power of the system. It becomes more important as more simulations are needed.

The currently most relevant solutions for both Game Planning and GDL interpretation are discussed in chapter 3.

2.3.1 Measuring performance

The measurement of game player performance is a hard task in itself. A simple method is playing against a random player (a player that randomly chooses one of the legal actions for each turn). It's no longer very useful because after a certain level of competence can player can beat the random player nearly all the time, so it becomes impossible to distinguish game players based on this metric alone.

What is more commonly done is competition with other existing game players. This should be done in as many different games and types of games as possible, since playing just a few games well is of little value in General Game Playing.

A very useful service is provided by ggp.org, the Tiltyard game server (tiltyard.ggp.org). It constantly arranges matches between the registered players and assigns an Agon skill rating to each player, based on the results. The Agon skill rating is very similar to the EloRank in Chess, except it also takes into account the perceived difficulty of the game and of playing a certain rule in that game.

2.3.1.1 Agon Skill

The Agon rating system computes a skill rating for each player, and a difficulty rating for each role in each game, based on historical match data. This system is based on the Elo rating system for Chess, expanding it in ways that are important for GGP: support for single-player games, many-player games, asymmetric games, cross-game skill ratings, and so on.

For a more detailed understanding of how Agon Skill is calculated, see appendix B.

2.4 Reasoner Techniques

A GDL description by itself isn't sufficient to know what actions are possible at each game state, such features must be deduced from the game rules. There are two main ways of dealing with this: Interpreting the rules whenever necessary or converting the game rules to an efficient data structure. The first solution is generally simpler to implement but has sub-par performance. There are, of course, various possible data structures to store the game rules. *Propositional Networks* are currently the most promising ones and have become quite popular in recent works.

2.4.1 Prolog Interpreters/Provers

Since GDL is a variant of Datalog, which is itself a syntactical subset of Prolog, it's fairly simple to translate GDL rules into Prolog data, which can then be inserted into a Prolog Environment to allow queries from the game planner. This approach is used mostly for it's simplicity in implementation, since it doesn't analyze (or simplify the analysis of) the game rules.

2.4.2 Propositional networks

Propositional networks are bidirectional graphs similar to simple logic circuits. This representation has 2 main benefits: compactness and simplified game analysis. The compactness comes from the fact that the number of represented states can be exponential to the number of propositions, n propositions can represent 2^n states, being much more compact than a state machine.

Propositional networks exploit the fact that, in the vast majority of games, there is limited influence to each action: each game symbol is affected only by a few other symbols, not all of them. For example, moving a pawn in Chess only changes the position of the moved pawn and perhaps an opponents piece is taken, the remaining pieces keep their previous state.

The simplified analysis comes from the fact that it becomes simpler to recognize things like symmetries, dead states, composite games (a game composed of multiple games that don't affect each other) and other properties.

Detecting composite games can be especially important: If a game is, for example, playing 3 board games concurrently and trying to win at least 2 out of the 3, it can lead a player to evaluate all 3 boards when a piece is moved on a single one. However, if the player can detect that the game is composed of 3 smaller games it can factorize it, playing each of the three board games as independent of each other, which is a much more efficient way of handling it.

The components of a basic propositional network can be seen in 2.3.

Propositions are boolean valued and can represent things like action legality, if a piece is in a certain location, etc. A proposition is an *input* if it's value is independent of the rest of the network and a *view* if it's the result of a connective. Connectives are transformations applied to propositions and be a transition

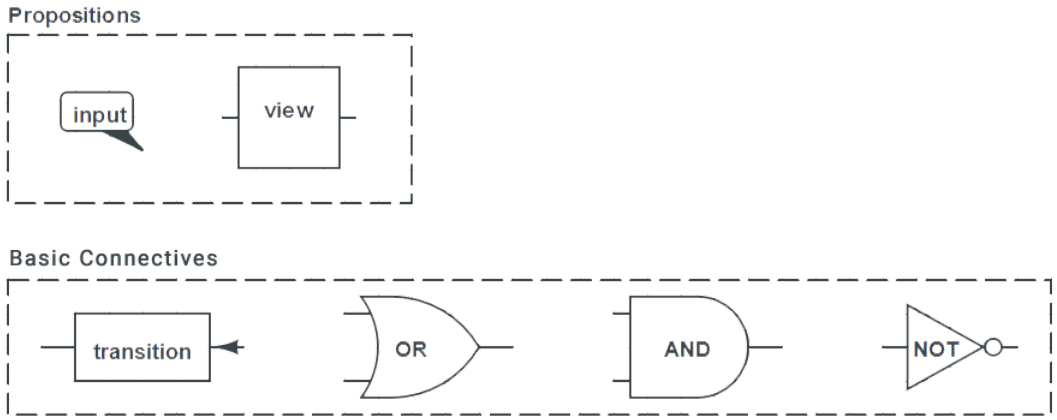

Figure 2.3: Basic components of propositional networks

or one of the 3 basic logic functions: *and, or* or *not*. A transition acts similarly to a register, its output does not changes until the game moves to the next state. A *view* after a transition is called a *base*.

A simple example of a propositional network can be seen in 2.4.

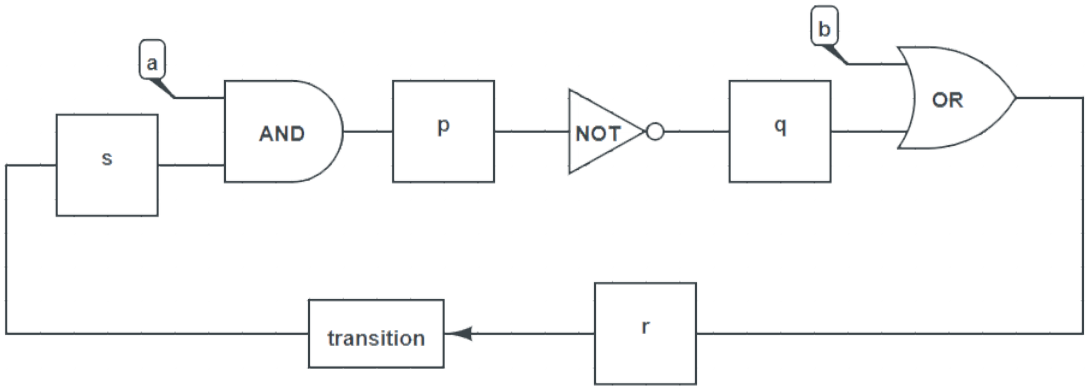

Figure 2.4: Example of a propositional network with 2 inputs and 1 transition

2.5 Game Planner Techniques

Some of the most widely used techniques for game planners are summarized bellow. Note that several of the top players actually use more than one technique, for example using an heuristic based approach for single player games, where the player is in charge of all actions.

2.5.1 Heuristic based techniques

Heuristics are methods that simplify or accelerate problem solving by not guaranteeing optimal solutions but satisfying approximations. Heuristics can be very useful whenever finding an optimal solution is impossible or impractical.

In the case of GGP these techniques try to learn or identify features of the game, so the player can more efficiently approximate the quality of the available actions. For example, a simple heuristic for Checkers would be to count the difference in pieces between the two players. One of the biggest

drawbacks of this type of technique is that heuristics become less useful the more general they become, meaning that, ideally, specialized heuristics should be created at run time, which is a very complex problem.

By using heuristics to attach a value to game states and/or actions, it is then possible to use search algorithms to try to find the best strategy. It's important to note that the performance of these search algorithms is completely dependent on the quality of the heuristics on top of which they run.

These techniques can be used in a game player by themselves but they can also be used to improve Monte Carlo Tree Simulation (MCTS) based players.

2.5.1.1 MinMax

MiniMax is a depth-first heuristic search method for adversary games. It is centered around the simple assumption that opponents try to minimize our score while we try to maximize it.

A MinMax game tree is represented in figure 2.5, where *square* plays against *circle*. The value in each node is the value of the state for *square* (based on the chosen heuristic). Squares represent states where *square* has control, similarly for circles.

Running the MinMax algorithm for *square*, we'll start with a depth-fisrt traversal of the tree, first finding the leaves on the left side, with values 5 and 6. Since in the previous turn the opponent (*circle*) is in control, 5 will be the chosen state if the game is ever in the parent state, since *circle* is trying to minimize *square*'s value. The value 5 is then propagated to the parent of both leaves, and the same method is applied to all other leaves and direct parents. We now do a similar process for all the nodes in the 4th level, as an example we can use the leftmost nodes, 5 and 4: Since these nodes represent *circle*'s turn, that means in the previous turn *square* is in control, and will try to maximize his value, choosing the action that leads to a state with value 5. After following these maximizing and minimizing rules, we end up with the maximum attainable value at the root (current state) node.

This algorithm can be easily generalized to games with more than one opponent, by assuming all opponents are working together against us and representing them as a single opponent that takes multiple consecutive turns. This is very pessimistic model of the opponents, since they are probably competing among themselves as well. The performance of the algorithm will of course depend on how the deep the search can go, on how close the opponent behaves to this model and the heuristics correlation between the heuristics used and game results.

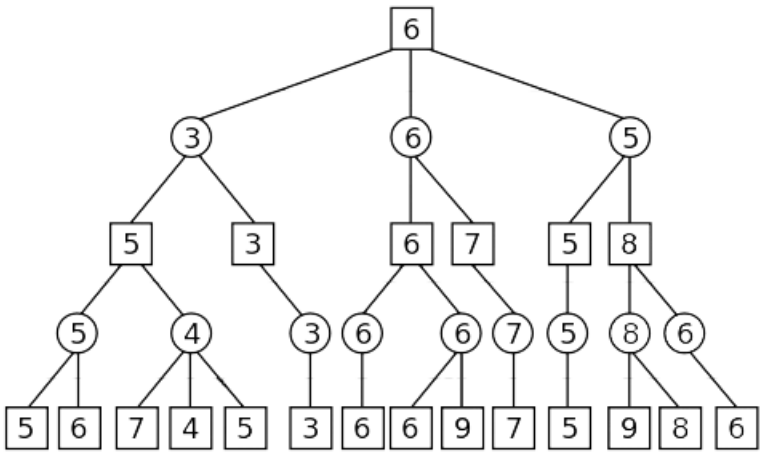

Figure 2.5: Example of a MinMax game tree

2.5.1.2 Alpha-Beta Pruning

MinMax can be very expensive as it tries to search through the whole game tree. Alpha-Beta Pruning is a very effective enhancement to MinMax, that attempts to reduce the amount of nodes that need to be visited.

It works by identifying nodes that lead to sub-trees that cannot change the current game value. An example can be seen in figure 2.6: Looking at the right side of the tree we can see that the opponent can choose a state with value 5, meaning whatever values are found in the other children of that state, the value attainable by *square* will not be higher than 5. Since *square* can already get a value of 6 by going for the middle choice, there's no point in searching the other children of the rightmost node because that choice cannot lead to a better score than 5. This allows for the pruning of many irrelevant sub-trees by simply storing a lower-bound, α, and an upper-bound, β, for each level of the tree.

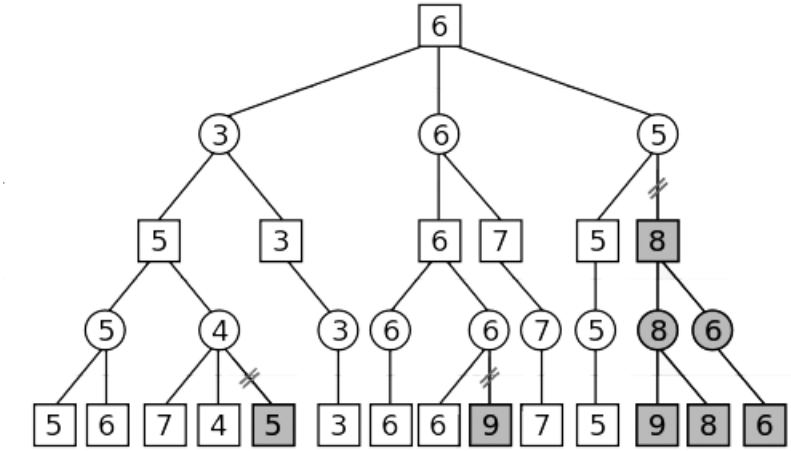

Figure 2.6: Example of a pruned game tree

2.5.1.3 Iterative deepening depth-first search

Iterative deepening depth-first search (IDDFS) is a technique for more dynamic game tree searches, it raises the depth limit iteratively while a goal state isn't found. The depth can be controlled by a cost function and it can be different for each branch, called *non-uniform depth-first search*, which has the added benefit of prioritizing the branches that seem most valuable. This technique increases the depth to which the game tree is searched, especially with big branching-factors, increasing the likelihood of finding terminal states.

2.5.2 Monte Carlo Tree Search

Introduced to the GGP competition by CADIA player in 2007 [3], MCTS is currently the most used and successful method in GGP. Starting from the current state, the algorithm traverses the tree until the move timer ends, doing as many iterations as possible.

The main advantages of MCTS are:

- It parallelizes well, as opposed to techniques like Alpha-Beta pruning
- It's domain independent, the creation of heuristics is optional
- It can be interrupted and resumed as needed, without spending time on restarting the simulations. This is very useful because of the time limits for each turn

One of the main issues of MCTS, however, is that simulations are more time consuming than heuristic based-methods, although this is attenuated by the parallization.

Each iteration has four steps: selection, expansion, simulation and back propagation:

1. Selection: Some technique is used to select which already traversed node to start from for a search. The most common technique is Upper Confidence Bounds applied for Trees (UCT), described in section 2.5.2.1.
2. Expansion: Adding a node with the first unvisited state yet to the tree, meaning a state that wasn't already in the tree.
3. Simulation: Perform a random simulation until a terminal game state is reached.
4. Back-Propagation: The scores obtained by all players at the end of the simulation are back-propagated to all nodes traveled in the selection and expansion stages. This is how the rewards of non-terminal states is computed.

2.5.2.1 UCT

Upper Confidence Bounds applied for Trees (UCT) is a way of selecting what node should be searched next and it's widely used for the *Selection* stage of MCTS. It uses a constant that can be tweaked to favor more or less exploration of non visited branches. The UCT algorithm is described by 2.4:

$$a^* = arg \max_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\frac{\ln |N(s)|}{N(s,a)}} \right\} \tag{2.4}$$

Where a^* is the selected node, $a \in A(s)$ means an action that contained in the set of possible actions in the current state s, $Q(s,a)$ is an assessment of performing a in state s, C is the exploration ratio constant, $N(s)$ is the number of previous visits to state s and $N(s,a)$ is the number of times a has been sampled in state s.

2.5.2.2 Heuristics in MCTS

There have been several suggested improvements to the basic MCTS. One of the most interesting ones is Simulation Heuristics, proposed by MINI-Player [4], which aims to add some form of learning to the standard MCTS algorithm. The heuristics proposed are very light-weight and are the following:

- Random: The standard MCTS
- History Heuristic: Tries to identify globally good actions (generally good regardless of state)
- Mobility: Favors actions that lead to states with more move options relatively to other players
- Approximate Goal Evaluation: Tries to calculate the degree of satisfaction of a GDL goal rule
- Exploration: Measures the difference between states as a way to do a diverse exploration
- Statistical Symbol Counting: Before the start clock simulations are done to calculate the correlation between game score and certain game symbols (moves, pieces, board locations, etc). Symbols that do not change much are then ignored to allow more computation to be made on the more relevant ones

Chapter 3

State of the Art

3.1 The world competition as a benchmark

The General Game Playing annual world competition, held at the AAAI conference since 2005, has been the community accepted way to know what are the state of the art players. As of 2015, it uses a double-elimination tournament between the qualified players. The qualification phase is open to anyone who wants to participate and consists of several tests designed to test stability and ability. All the players that surpass a threshold score are entered into the main tournament.

3.2 The Tiltyard game server as a benchmark

The Tiltyard game server, mentioned in section 2.3.1, is also a great way of comparing performance, since it constantly runs matches between the registered players and ranks them according to the *Agon Skill* metric (explained in section 2.3.1.1). This allows for more thorough testing of each player, as over time they are matched against more players and with more games than on a single competition. One of the downsides is that, unlike competitions, the games are not kept secret. However this is likely a small issue since there are no rewards for being the top player and as the game library grows it becomes increasingly impractical to make specializations for the games used by the Tiltyard game server.

3.3 Results and characteristics of a few well important players

The choice for what players were presented was based on the following:

- Should include the current top performers
- There should be a good amount of information about the systems, from the authors themselves
- As a group, they should allow for comparisons on the performance of various techniques

  Galvanize is the current world champion. Sancho won the 2014 annual competition and currently holds 1st place in the Tiltyard server.

  Sancho and Galvanize hold the 3 top spots on the Tiltyard server (they have multiple entries, like many other projects, because they have registered multiple versions on the site)

  Ari and FluxPlayer are former world champions that are well documented and serve as a base for comparison of different techniques.

Random is included to provide a notion on the *Agon Skill* of an incredibly player, as it currently has the second to worst rating on the Tiltyard game server.

Table 3.1: Results and characteristics of a few important players

Name	2015 GGP competition placement	Agon Skill (as of 12/2015)	Reasoner	Planner	Additional Heuristics	Hardware
Galvanise	1st	192.72	Prop. Network	MCTS with UCT	Yes, determined during game setup time	1 Xeon (AWS)
Sancho	2nd	213.18	Prop. Network	MCTS with UCT	Yes, determined during game setup time	8 core i7 with 32GB RAM
TurboTurtle	3rd	Not registered in Tiltyard g.s.	Prop. Network	MCTS with UCT	No	?
Ari	Didn't participate	50.54	Prolog-based	MCTS with UCT	No	?
FluxPlayer	Didn't participate	34.05	Prolog-based	Heuristic-based	Yes, determined during game setup time	?
Random	Didn't participate	-359.75	Prolog-based	Random	No	Not relevant

3.3.1 Galvanize

The current (2015) champion and the second place in the Tiltyard game server. It's very similar to Sancho (3.3.2), at least from a high-level perspective.

It also uses MCTS with Upper Confidence Bounds applied for Trees (UCT) as the base for it's planner, with some dynamic techniques to balance the amount of computation given to the *search* and *backpropagation* steps of MCTS. It seems to use a normal Monte Carlo Tree, unlike Sancho's more general graph (not mentioned by the developer). It's GDL Reasoner is also based on Propositional Networks, it's capable of game factorization and can optimize for inputs that keep switching between *on* and *off*, by storing both results instead of recalculating.

It's written in a mixture of Python and C++ (with all optimizations turned on), for the 2015 world competition it ran on an Amazon Web Services server with a single Intel Xeon processor.

Created by Richard Emslie.

3.3.2 Sancho

Champion in 2014 and currently the first place in the Tiltyard game server.

Like most recent game players, it uses MCTS with UCT as the basis for it's game planner. The game planner has a few notable features:

- Replacement of the MCTS tree with a more general graph, which allows for transitions between lines of play without duplication of nodes for any given state.
- Active trimming of lines that have fully-determined outcomes, propagating those outcomes up the MCTS structure to the highest level at which they remain fully determined (with best play).
- An efficient infrastructure for parallelization across threads.

- Use of heuristics, determined during game setup time, to aid MCTS selection in preferentially expanding some branches.

  For it's GDL Reasoner it uses Propositional Networks and it's capable of identifying disjunctive factoring opportunities and significant latches (latches are game properties that after becoming *true* can never change value throughout the rest of the game).

  It's written in Java and for the 2015 world competition it ran on a computer with an 8 core Intel i7 with 32GB of RAM.

Created by Steve Draper and Andrew Rose.

3.3.3 TurboTurtle

The winner of world GGP competition in 2011 and 2013.

Like Sancho and Galvanize, it uses MCTS with UCT for it's game planner.

It's GDL reasoner tries to maximize the number of simulations that are made instead of focusing on rule analysis. It does this by converting the rules to a Propositional Network and then attempting to compile into a programmatic representation of the game state machine in C++. When this fails simpler, less efficient, state machines are used, such as a Prover state machine that operates directly on the game rules instead of using a Propositional Network as an intermediate form. One other useful technique is that, when faced with a single-player game (i.e. a puzzle), it will convert game rules into an Answer Set Program and then use an off-the-shelf ASP solver to find the optimal set of moves to solve the puzzle. This approach works well for small puzzles but is usually innefective for big puzzles, like Sudoku.

The main competitive strength that TurboTurtle currently has is the efficiency of its Propositional Network converted do C++ approach. Whenever this technique cannot be used it's results are sub-par.

Created by Sam Schreiber.

3.3.4 Ari

The winner of world GGP competition in 2009 and 2010.

It's GDL reasoner translates the description to Prolog and it's game planner is based on MCTS with UCT without other notable additions.

Created by Jean Méhat (Paris 8 University).

3.3.5 FluxPlayer

The winner of world GGP competition in 2006, it has an heuristic-based planner that uses non-uniform iterative deepening depth-first search with Alpha-Beta pruning. It's heuristics use fuzzy logic to determine how close to terminal a certain state is and also try to identify game structure by automated analysis of game specifications. The GDL reasoner translates the description to Prolog.

It uses a mixture of Prolog and Flux, a language designed to write AI agents that relies on fuzzy logic.

Created by Stephan Schiffel and Michael Thielscher (Dresden University of Technology).

3.3.6 Random

A very simple player, included only for the sake of comparison, it evaluates what actions are legal for each turn and chooses one at random.

3.4 Comparison

As can be seen in table 3.1, as well as on [5], the State of the Art solutions for GGP use MCTS with UCT for game planning and Propositional Networks for GDL reasoning. Furthermore, the current top players also do some rule analysis to determine game patterns and factorization opportunities, which helps reduce the search space and improves the search itself by the use of heuristics.

On the hardware side it should be noticed that while players are already leveraging multi-core processors to improve parallelism, no distributed systems or CPU+FPGA hybrids have yet been created.

Chapter 4

Preliminary Results

To get a better understanding of the GGP platform, a few tests were run, using the application *Kiosk* as a game manager (provided by ggp.org) and a few different game players. Games on the *Kiosk* application have graphical user interfaces and even allow a user to play directly against a player. A few test players were also created, mixing and matching different components already provided by ggp.org.

Overall, it was quite useful to get some practical knowledge on the field but performance comparisons proved to be quite laborious (many different games have to be tested for the results to have meaning) and so were avoided since better options already exist, namely the Tiltyard game server.

Bibliography

[1] J. Pitrat, “Realization of a General Game Playing Program,” *Pascal, Institut Blaise*, 1969.

[2] J. Mandziuk, Y. Ong, and K. Waledzik, “Multi-game playing - A challenge for computational intelligence,” *Computational Intelligence . . .*, pp. 17–24, 2013.

[3] H. Finnsson, *CADIA-Player: A General Game Playing Agent*. M.sc. thesis, Reykjavík University - School of Computer Science, 2007.

[4] M. Swiechowski, “Specialized vs . Multi-Game Approaches to AI in Games,” pp. 1–12, 2014.

[5] M. Swiechowski, H. Park, J. Mandziuk, and K.-j. Kim, “Recent Advances in General Game Playing,” *The Scientific World Journal*, vol. 2015, 2015.

[6] Y. Björnsson, “Learning Rules of Simplified Boardgames by Observing,” *Ecai*, pp. 175–180, 2012.

[7] J. E. Clune, “Heuristic Evaluation Functions for General Game Playing,” *KI - Künstliche Intelligenz*, vol. 25, pp. 73–74, 2011.

[8] M. Genesereth, N. Love, and B. Pell, “General game playing: Overview of the AAAI competition,” *AI Magazine*, vol. 26, no. 2, pp. 62–72, 2005.

[9] J. Mandziuk and M. Swiechowski, “Generic Heuristic Approach to General Game Playing,” *Sofsem 2012, Lncs 7147*, pp. 649–660, 2012.

[10] S. Schiffel and M. Thielscher, “Fluxplayer : A Successful General Game Player,” *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*, vol. 22, no. 2, pp. 1191–1196, 2007.

[11] M. Jean and T. Cazenave, “Ary , a general game playing program,” *Board Games Studies Colloquium*, pp. 1–9, 2010.

[12] M. Buro, H. Finnsson, and A. Saffidine, “Alpha-Beta Pruning for Games with Simultaneous Moves,” *Aaai*, pp. 556–562, 2012.

[13] M. Thielscher, “General game playing in AI research and education,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7006 LNAI, pp. 26–37, 2011.

Appendix A

GDL example: tic tac toe

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Tictactoe - GDL Description
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Roles - Each player is assigned one of these roles at the beginning of the match
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (role xplayer)
  (role oplayer)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Initial State - With the init keyword one can define initial states  of the game.
;; the control keyword defines which player or players have the first move
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (init (cell 1 1 b)) ;; Initialization of the cells with the value '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 xplayer)) ;; xplayer has the first move

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Dynamic Components - Components that change throughout the game, like the cells
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Cell

;; variables are written with '?' in front of them

  (<= (next (cell ?m ?n x)) ;; the cell with coordinates m,n changes to 'x' if...
      (does xplayer (mark ?m ?n)) ;; xplayer chooses to mark it...
      (true (cell ?m ?n b))) ;; and it's still in state 'b' (blank)

  (<= (next (cell ?m ?n o)) ;; same thing for oplayer
      (does oplayer (mark ?m ?n))
      (true (cell ?m ?n b)))
```

```
(<= (next (cell ?m ?n ?w)) ;; the cell changes to state ?w if...
    (true (cell ?m ?n ?w)) ;; it's already in state ?w
    (distinct ?w b)) ;; and ?w isn't blank

(<= (next (cell ?m ?n b)) ;; the cell ?m ?n changes to 'b' if...
    (does ?w (mark ?j ?k)) ;; player ?w marks ?j ?k...
    (true (cell ?m ?n b)) ;; the cell is already state 'b' or...
    (or (distinct ?m ?j) (distinct ?n ?k))) ;; ?w marked another cell (?m,?n != ?j,?k)

(<= (next (control xplayer)) ;; if oplayer is in control, xplayer takes control
    (true (control oplayer)))

(<= (next (control oplayer)) ;; and vice-versa
    (true (control xplayer)))


(<= (row ?m ?x) ;; a row means 3 cells in the same line with the same state ?x
    (true (cell ?m 1 ?x))
    (true (cell ?m 2 ?x))
    (true (cell ?m 3 ?x)))

(<= (column ?n ?x) ;; a column means 3 cells in the same column with the same state ?x
    (true (cell 1 ?n ?x))
    (true (cell 2 ?n ?x))
    (true (cell 3 ?n ?x)))

(<= (diagonal ?x) ;; same idea for each diagonal
    (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) (row ?m ?x)) ;; a line of ?x exists if there is a row of ?x
(<= (line ?x) (column ?m ?x)) ;; ...or a column of ?x
(<= (line ?x) (diagonal ?x)) ;; ...or a diagonal of ?x


(<= open ;; the game is open if there's at least one cell with the state 'b'
    (true (cell ?m ?n b)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal ?w (mark ?x ?y)) ;; player ?w can mark ?x ?y if...
    (true (cell ?x ?y b)) ;; the cell is blank
    (true (control ?w))) ;; player ?w is in control

(<= (legal xplayer noop) ;; player xplayer can do a 'noop' move (do nothing) if...
    (true (control oplayer))) ;; the oplayer is in control

(<= (legal oplayer noop) ;; and vice-versa
    (true (control xplayer)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
  (<= (goal xplayer 100) ;; xplayes gets 100 points if there is a line of 'x'
      (line x))

  ;; xplayer gets 50 points if there are no open cells and no lines of 'x' or 'o'
  (<= (goal xplayer 50)
      (not (line x))
      (not (line o))
      (not open))

  (<= (goal xplayer 0) ;; xplayer gets 0 points if oplayes gets a line
      (line o))

  (<= (goal oplayer 100) ;; same thing for oplayer
      (line o))

  (<= (goal oplayer 50)
      (not (line x))
      (not (line o))
      (not open))

  (<= (goal oplayer 0)
      (line x))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (<= terminal ;; the game ends if there is a line of 'x'
      (line x))

  (<= terminal ;; ...or a line of 'o'
      (line o))

  (<= terminal ;; ...or if the game is no longer open
      (not open))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Listing A.1: tic tac toe described in GDL

Appendix B

Calculation of *Agon Skill* (in Python2)

```
import json

dataPoints = []

f = open("matchesFrom2011", 'r')
for line in f.readlines():
  # Load the match data in JSON format.
  match = json.loads(line)['data']
  # Discard matches that aren't signed.
  if not 'matchHostPK' in match:
    continue
  # Discard matches that aren't signed by Tiltyard.
  if hash(match['matchHostPK']) != -859967508381652683:
    continue
  # Discard matches that didn't record player names.
  if not 'playerNamesFromHost' in match:
    continue
  # Discard matches that didn't complete successfully.
  if not match['isCompleted']:
    continue
  # Discard matches where a player had errors.
  if 'errors' in match:
    hasErrors = False
    for errorsForTurn in match['errors']:
      for error in errorsForTurn:
        if len(error) > 0:
          hasErrors = True
    if hasErrors:
      continue
  # Store the relevant parts for computing ratings: when the match started,
  # the players involved, and the final scores.
  dataPoints.append((match['startTime'], match['gameMetaURL'], match['playerNamesFromHost
    '], match['goalValues']))

# Agon rating, like Elo rating, is order dependant: if a player that's currently weak
# beats a player that's currently strong, that's more important than if a player that
# was once weak (but is now average) beats a player that was once strong (but is now
# also average). Thus we need to process matches in the order in which they occurred.
# This is done by sorting them by start time.
dataPoints.sort()

# Ratings will be tracked in this map.
```

```
agonRating = {}

# An essential part of Agon rating, like Elo rating, is determining the expected score
# for players when they're matched against each other, based on their current ratings.
# This is done exactly as it is done in ordinary Elo rating.
def getExpectedScore(aPlayer, bPlayer):
  if not aPlayer in agonRating:
    agonRating[aPlayer] = 0
  if not bPlayer in agonRating:
    agonRating[bPlayer] = 0
  RA = agonRating[aPlayer]
  RB = agonRating[bPlayer]
  QA = pow(10.0, RA / 400.0)
  QB = pow(10.0, RB / 400.0)
  return QA / (QA + QB)

# Updating ratings also works exactly like in Elo rating: compute the expected score,
# and then increase ratings if the players exceeded that score, and decrease ratings
# if the players fell below the expected score.
def updateRating(aPlayer, bPlayer, aScore, bScore):
  if aScore + bScore != 100:
    return
  EA = getExpectedScore(aPlayer, bPlayer)
  EB = 1.0 - EA
  agonRating[aPlayer] = agonRating[aPlayer] + (aScore/100.0 - EA)
  agonRating[bPlayer] = agonRating[bPlayer] + (bScore/100.0 - EB)

# For every recorded match, we do pairwise ratings updates between all of the players
# involved in the match, *and* we do a ratings update between each player in the match
# and their role in the game, representing the roles in the game as distinct players
# with their own ratings that vary just like player ratings.
for dataPoint in dataPoints:
  gameURL = dataPoint[1]
  playerNames = dataPoint[2]
  goalValues = dataPoint[3]
  for i in range(len(goalValues)):
    updateRating(playerNames[i], gameURL + '_role' + str(i), goalValues[i], 100 -
    goalValues[i])
    for j in range(i+1,len(goalValues)):
      updateRating(playerNames[i], playerNames[j], goalValues[i], goalValues[j])

# Display a list of (rating,player) sorted by rating in ascending order.
ratingsForPlayers = [ (i,j) for (j,i) in agonRating.items() ]
ratingsForPlayers.sort()
for rating, playerName in ratingsForPlayers:
  print str(rating).rjust(20), playerName
```

Listing B.1: Calculation of the *Agon Skill* rating system, in Python2