Master's thesis in the Department of Computer Science at
Freie Universität Berlin

Game Balancing Using Game Descriptions and Artificial Intelligence

Spielebalancing mit Hilfe von Spielbeschreibungen und Künstlicher Intelligenz

Armin Feistenauer

August 2012

Supervisor: Prof. Dr. Marco Block-Berlitz
Second Supervisor: Prof. Dr. Elfriede Fehr

Abstract

In this paper methods of general game playing, the process of playing previously unknown games without human intervention, are applied to social games in order to predict human playing behavior and use these findings to guide game balancing decisions. To achieve this goal a new game description language for social building simulation games, called Wooga Game Description Language (WGDL), has been introduced in this thesis and a prototype of a balancing tool playing these game descriptions has been implemented. To improve the results of the automatic game play several enhancements to the established search algorithm, called Monte Carlo Tree Search (MCTS), have been proposed and evaluated. The division of a game into several sub-games played consecutively improved the quality of the results by 35%, while the introduction of human knowledge about game moves by categorizing them increases the results by 23%. A comparison with human player data shows that the prototype is in some cases in the region of human players but does not yet consistently outperform most of them. Therefore, the approach analyzed in this thesis is promising, but has yet to be optimized for application in game development.

Abstract (German)

In dieser Arbeit werden Methoden des General Game Playing, das ist das automatisierte Spielen vorher unbekannter Spiele ohne menschliche Hilfe, auf das Spielen von Social Games angewandt um menschliches Spielverhalten vorherzusagen und diese Erkenntnisse für das Balancing der Spiele zu verwenden. Um dieses Ziel zu erreichen wurde im Rahmen dieser Masterarbeit eine neue Spielbeschreibungssprache, die Wooga Game Description Language (WGDL), entwickelt. Ein Prototyp eines Spielebalancing-Tools, welcher in WGDL beschriebene Spiele automatisiert spielen kann, wurde implementiert. Um die Ergebnisse des automatisierten Spielens zu verbessern wurden mehrere Erweiterungen für den etablierten Suchalgorithmus, die Monte Carlo Baumsuche (MCTS), entworfen und evaluiert. Die Aufteilung eines Spiels in mehrere Teilspiele, welche nacheinander gespielt werden, hat die Qualität der Ergebnisse um 35% verbessert. Das Einfügen menschlichen Wissens über Spielzüge, durch die Aufteilung derer in verschiedene Kategorien, bewirkte eine Verbesserung um 23%. Der Vergleich der Prototypergebnisse mit Spieldaten von menschlichen Spielern zeigt, dass er unter gewissen Bedingungen auf einem Niveau mit diesen ist aber sie noch nicht kontinuierlich übertrumpfen kann. Der hier analysierte Ansatz ist daher vielversprechend, muss vor der praktischen Anwendung für das Spielebalancing aber noch verbessert werden.

Acknowledgements

Before continuing with my thesis I want to thank some people that helped me write it during the last six month.
First of all I want to thank my supervisor Prof. Dr. Marco Block-Berlitz for his support and feedback and especially for getting me started with a clear structure early on. That helped a lot.
For her continuous support with a product managers perspective and for proof reading this thesis I would like to thank Alena Delacruz. Also at Wooga I want to thank Jesper Richter-Reichhelm for his engineering and game industry perspective, the feedback on my thesis and for supervising me at Wooga for the last months. This thesis would not have been possible without both their encouragement and support.
I am grateful to my fellow colleagues at Wooga, the thesis writing students as well as my team, the game eight team, and the people at the magic land team.
For bringing Wooga to my attention and bringing me on board I would like to thank Anne Seebach.
Last but not least I owe my deepest gratitude to my girlfriend Isabel who supported me while I was occupied with this thesis.

Armin Feistenauer

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröentlicht.

Berlin, den 09.08.2012

Armin Feistenauer

Contents

List of All Experiments

Nomenclature

Project Nomenclature

Balancing Tool	The prototype of a balancing tool has been implemented for this thesis. It is referred to as tool, balancing tool and prototype. When referenced as balancing tool the sentence does not only apply to the current implementation but to the idea behind it as well.
Goal	When the last goal is reached the game state is considered terminal. All previous goals are called subgoal.
Magic Land	Magic Land is a SBSG developed by Wooga. The player takes over control of a small kingdom and starts to expand it. This game is used as a reference for SBSGs in this thesis.
ML	See Magic Land.
Prototype	See Balancing Tool.
SBSG	A Social Building Simulation Game is a social game with focus on the building aspect of a game. Gardens, towns, kingdoms and many other things are built and expanded by the player.
SC	See Simple Coins.
Simple Coins	Simple Coins is an example game used to demonstrate the value of some search algorithm modifications. It allows to compare search results with the optimal solution because the optimal solution can be calculated.
Tool	See Balancing Tool.
Wooga	Wooga GmbH is the social and mobile game development company this thesis has been written at. It is currently the worlds fourth largest social gaming company on facebook.
XP	Short term for experience points, a common meassure for game progression.

Game Description Nomenclature

EBNF	The Extended Backus-Naur Form is a metasyntax for context-free grammars, used to define WGDL.
GDL	A Game Description Language invented in Stanford. Most commonly used to date.
SGDL	The Strategy Game Description Language is currently under development and the inspiration for WGDL.
WGDL	The Wooga Game Description Language is introduced in this thesis to describe social building simulation games.

Search Algorithm Nomenclature

Alpha-Beta Pruning	Alpha-Beta Pruning is an extension optimizing Minimax by cutting down the tree size that has to be searched.
GGP	General Game Playing is the process of playing games previously unknown to the program. It is the name of a project of the Stanford Logic Group.
MCTS	Monte Carlo Tree Search is an algorithm building up a search tree and evaluating the game states nodes with Monte Carlo methods.
Minimax	Minimax is the standard search algorithm for two player games with perfect information.
Monte Carlo Methods	A mathematical method to approximate values with many random experiments.
NMCS	Nested Monte Carlo Search uses a nested search algorithm with Monte Carlo methods instead of building up a search tree. It is therefore very economic regarding memory consumption.
Playout	A playout is the process of playing a game from a start game state until a terminal state is reached.
SP-MCTS	SP-MCTS is an UCB formula specifically designed to do well in single-player games.
UCB	Upper Confidence Bound is an algorithm balancing exploration of new path in the tree with exploiting the currently best options. It aims to maximize the result of the plays but promises to eventually find the best possible solution. Other algorithms with the same goal are listed under this term as well.

1. Motivation and Introduction

Few developments have transformed our daily life over the last few years as much as the success story of social networks. Three quarters of Germany's internet users are active on a social network, tendency rising. Most of them are registered with Facebook [6]. Since Facebook opened its platform to external vendors in 2007, other companies like games developers have been profiting from its rapid success as well. Their Social Games can reach tremendous player numbers in a short amount of time [8]. Social Games are defined as games that access data provided from a social graph, optimize their games' designs on short play sessions and monetize with virtual goods. A prominent example, CityVille[1], was played by over 100 million players in January 2011 and remains the biggest Facebook game to this day [49]. The Social Games development cycle is characterized by their early release and the addition of content and features over a longer period of time afterward. This provides new challenges for traditional game balancing:

- How long will the „new" content occupy its players?
- How will the new content affect the balance of the existing game?

A small revolution has taken place in the field of Artificial Intelligence (AI) in the past ten years as well. This thesis will explore whether AI can be a tool to solve these new challenges. Since the invention of the computer, ever better algorithms have been developed for playing games under the heading of Artificial Intelligence [46]. The game of chess received special attention because it could, amongst other things, not be solved with a brute force approach alone. In the year 1997 IBM succeeded at beating the acting chess world champion Garry Kasparov with their chess computer Deep Blue [35]. This huge success could not conceal the fact that this algorithmic approach mainly pursued in AI until then had two fundamental problems. Firstly, these specifically constructed AIs only showed good results in one type of game and could

Figure 1.1.: *Garry Kasparov vs Deep Blue Copyright Adam Nadel, AP*

[1] CityVille can be played at: http://apps.facebook.com/cityville/

not play any others. This made it very difficult to generalize or transfer any findings. Secondly, the AIs needed heuristics to evaluate game situations and these heuristics are not known or available for all games. This was abundantly clear in the game of Go, for which no heuristic is known. While the chess world champion was already beaten, Go programs could not even win against amateurs.

The Stanford Logic Group faced the problem of overspecialization by creating a new language for describing games, the Game Description Language (GDL) [20]. GDL allows the development of programs that can play all games described in this language. Since 2005, a yearly tournament[2] has been held in which programs have to compete in different games, described with GDL. The tournament was soon dominated by programs using Monte Carlo Methods instead of heuristics to evaluate game situations. These Monte Carlo Methods, which where first used in AI in 2006, have since improved the performance of Go playing programs considerably.

This master's thesis will bring both developments together and answer the following questions:

- How can Social Games, particularly building simulations, be described in a general way?
- Are Monte Carlo methods suited to play these games successfully?
- What adaptions have to be made to current Monte Carlo Search Algorithms to fulfill the demands of this new task?

At last the thesis should allow an assessment of whether this approach is viable to simulate human playing behavior and gain useful insights for social game balancing.

[2]Website of the AAAI Tournament: http://games.stanford.edu/

1.1. About Wooga

This master's thesis was written throughout an internship at Wooga GmbH[3], where the result of this work will be used in the development of new games. Wooga was, at the time of this writing, the world's fourth largest Social Games developer and was founded in 2009 by Jens Begemann, Philipp Moeser and Patrick Paulisch in Berlin. Since 2009, Wooga has released six titles and focused on the Facebook and iOS platforms. All currently released titles may be divided into two categories:

Figure 1.2.: Wooga Logo

1. Arcade Games: Brain Buddies, Bubble Island and Diamond Dash. These are fast-paced mini-games where the goal is to reach a highscore in a given amount of time.

2. Building Simulation Games: Monster World, Happy Hospital and Magic Land. These are games where the player regularly returns to his previous game and continues building-up his virtual property.

The tool described in this thesis is meant to be used for balancing this second kind of game. For a more detailed description of the game types see chapter 4.1.2.

1.2. Scope of the Thesis

This master's thesis deals with the intersection between Artificial Intelligence, Social Games and Game Balancing. Before the theoretical background of the topics covered in this paper are analyzed in more depth, the scope of this thesis regarding the main topics will be narrowed down.

1.2.1. Artificial Intelligence

John McCarthy, who coined the term Artificial Intelligence in 1956, describes AI as *„[...] the science and engineering of making intelligent machines, especially intelligent computer programs.“* [51]. Intelligence in turn is defined as *„[...] the computational part of the ability to achieve goals in the world.“* [51]. Therefore AIs are computer programs that have the ability to achieve goals. As the question of whether something is intelligent becomes one of degree, AI research tries to move programs into the direction of intelligence on different branches of the field. Some branches of AI deal with neural networks

[3]Website of Wooga: http://www.wooga.com

and machine learning, others with visual recognition, the ability to understand language, logical inferences and many more.

This thesis will mainly cover the AI field of „search“ that deals with efficiently searching a large possibility space. As a prerequisite for searching it will also touch on some concepts of „representation“, a field that deals with how to represent facts about the world. Methods from these fields are applied on the task of playing different games with one program.

This specific research area has recently become its own active field of study under the term General Game Playing (GGP), coined by the General Game Playing Project of the Stanford Logic Group of Stanford University[20].

1.2.2. Social Games

Social Games are a fairly new phenomenon in the games market. Even at the moment there remains no dominant definition of what is and what isn't a Social Game. The main discussion at the moment being, what kind of interaction a game has to offer to be classified as social. For the purpose of this thesis it will be enough to settle on a very broad definition of the term.

In chapter 4.1.2 a subclass of Social Games that shall be describable by the new Game Description Language is defined.

In „Inside Virtual Goods - The Future of Social Gaming 2011“ [8] the following three core elements of Social Games are extracted:

Use Of Social Graph Data It has to use relationships in a social network to connect players, to play together, send and receive items or get new players to play a game. This feature introduces a new quality of virality, allowing games to spread enormously fast through the social graph.

Game Play Designed For Short Session Length Most Social Games are designed to be played in many small sessions, which fit into breaks or visits to a social network.

Free To Play Games Monetized Largely By Virtual Goods Social Games are not purchased, but played for free. This lowers the entrance barrier dramatically but has the effect that games have to be fun from the start, because the user has no investment in the game and might stop playing anytime. These games then monetize by smaller transactions over the whole lifetime of the player.

1.2.3. Game Balancing

Game Balancing is the process of fine-tuning a game's rules and settings. There are different balancing goals for various game types but in general, game balancing tries to achieve the goals listed below [50]:

Balance Game Features Human players have the ability to optimize their game play habits, i. e. learn how to best play a game. Poor balancing encourages players to optimize around the wrong features. Weak features won't be used whereas strong features will have disproportionate influence.

Progression Balancing influences the speed with which players advance to new levels, territories, weapons, and other possibilities. This is used to affect player motivation and game duration.

Fairness In multi-player games the different players should have an equal likelihood, or likelihood depending on their skill level, to win.

Game Balancing is a task every team faces during game development. This thesis shows how General Game Playing, henceforth referred to as GGP, can help to estimate player progression and maybe even balance game features .

1.3. Structure of the Work

In chapter 2 the current state of research in the field of Game Descriptions (2.1) and Search Algorithms (2.2) is shown and methods used in this thesis are explained. The concepts of Software Engineering (2.3) that were applied during implementation are listed as well.

Chapter 3 states the goals of the prototype that was created during the course of this internship. These goals were a major factor when decisions, concerning features and their implementation, had to be made. This chapter also includes a section on the structure of the prototype.

Chapter 4 is the first of two major parts of this work. It introduces a new Game Description Language suited to describe a subclass of Social Games and discusses the fortitude and limitations of its expressiveness.

The second major part is discussed in Chapter 5. Social games are played using Monte Carlo Tree Search Algorithms. The adaptions made to this algorithm in order for it to play successfully are shown and different alternatives are considered.

In the next chapter on Experiments and Evaluation (Chapter 6) several variants of the algorithm are put to the test.

Finally the summary in Chapter 7 shows what has been achieved and what advancements have been made. This chapter also features an outlook where future research projects could continue this work.

2. Theory and Related Works

The research in GGP is currently focused on game descriptions and search algorithms. In this chapter the methods this work builds upon are explained and the current state of the research is shown.

2.1. Game Descriptions

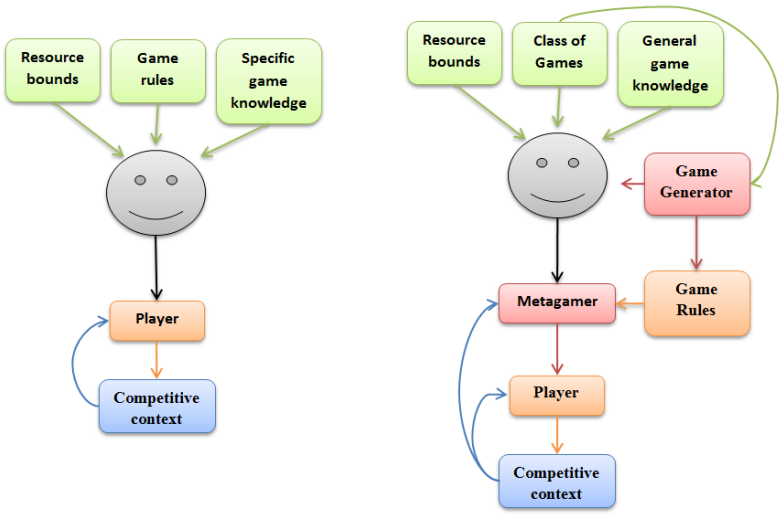
Game descriptions are formal languages to describe classes of games. These languages are a prerequisite for GGP, because they allow the construction of programs that are able to interpret and then play all games described with them. The game description therefore acts like a common interface, playing and searching algorithms can adhere to.

First attempts on GGP had been made as early as 1968, when Jacques Pitrat proposed his *„Realization of a general game-playing program“* [43]. But the idea did not take off and was revived only in 1992 by Barney Pell. He named the idea, that programs play games which are given to them as an input in a given language, *Metagame* [42, 41] and argued that this would force game analysis to be made by programs rather than programmers. The concept is explained in figure 2.1.

His implementation called *Metagamer* [40] was able to play symmetric chess-like games and played reasonably well against specialized programs in checkers and chess, while clearly loosing against the best of these programs.

Since then a string of new proposals of how to represent games have been made [37, 20, 4, 13, 7, 10, 28]. With *Zillions of Games*[1] even a commercial *„universal gaming engine“* was published in 1998. The main difference between these proposals being the class of games that can be expressed with them. Before the two most influential game description languages to this thesis (GDL see 2.1.2 and SGDL see 2.1.3) are presented, some basic terms for classes of games are explained.

[1] Zillions of Games can be found on the Website: http://www.zillions-of-games.com/



(a) Programming AIs for specific games (b) Programming a meta gamer for a class of games

Figure 2.1.: In Figure 2.1a the programmer (black smiley) uses knowledge about a specific game to create an AI playing this game.
The Figure 2.1b illustrates the programming of a meta gamer. The programmer can no longer introduce his knowledge directly into the player, but has to create a „Metagamer“ able to play games described in game rules, which are created by a game generator.
Figure recreated after Pell [40].

2.1.1. Classes of Games

There is no single language or framework for classifying different types of games. But we need some common understanding of terms to discuss differences in descriptive power between game description languages. The terms used to describe game classes in this paper are therefore defined as follows:

Finite Vs. Infinite Finite games have a beginning and an end, whereas infinite games need not have either of these.
e. g. Connect Four vs. Conway's Game of Life

Discrete Vs. Continuous Discrete games have a finite and countable number of moves and possibilities. Continuous games allow to choose from a continuous field of possibilities.
e. g. Connect Four vs. Counter-Strike[2]

Deterministic Vs. Non-deterministic In deterministic games, the outcome of any event is predetermined, whereas non-deterministic games allow for random events.
e. g. Chess vs. Poker

[2] Counter-Strike is a first-person shooter, where terrorists and counter-terrorists compete against each other. Website: http://www.counter-strike.net/

Multi-player Vs. Single-player In multi-player games several players play together, either cooperating or competing. In single-player games there is only one player
e. g. Counter-Strike vs. Pinball

Complete Information Vs. Incomplete Information In games of complete information every player knows everything about the current game state. In games of incomplete information the knowledge of different players may vary.
e. g. Chess vs. Poker

Zero Sum Vs. Non-zero-sum In zero sum games gains of one player are equal to the loses of the other player(s) [47]. Therefore players compete for the same resources and none are created or destroyed.
e. g. Liquid War[3] *vs. FarmVille*

Symmetric Vs. Asymmetric Symmetric games give the same possibilities to every player, whereas asymmetric games may treat players differently.
e. g. Go vs. StarCraft[4]

Game	**Type**
Chess	finite, discrete, deterministic, symmetric, zero-sum, multi-player game with complete information
Poker	finite, discrete, non-deterministic, symmetric, zero-sum, multi-player game with incomplete information
StarCraft	finite, discrete, deterministic, asymmetric, non-zero-sum, multi-player game with incomplete information
Counter-Strike	finite, continuous, deterministic, asymmetric, non-zero-sum, multi-player game with incomplete information

Table 2.1.: Game examples listed with their classes

[3] Liquid War is a popular multi-player linux game, where the player has to surround the enemies liquid with his own. Surrounded liquid changes its allegiance to the surrounding player. The overall amount of liquid stays the same, what makes it a zero sum game.
[4] StarCraft is a real-time strategy game in a science fiction setting. Three different races fight each other. Website: http://eu.blizzard.com/de-de/games/sc/

2.1.2. Game Description Language

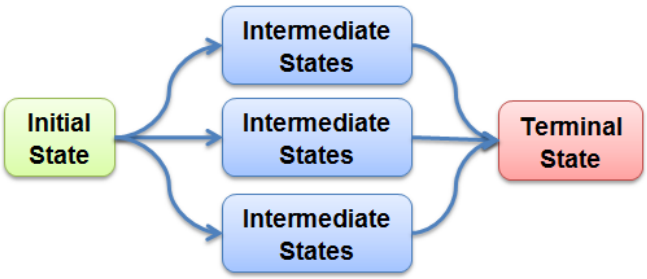


Figure 2.2.: From an initial state various intermediate states can be reached by applying different moves. After various moves the game reaches a terminal state. Initial state, terminal states and possible moves have to be defined in a game description.

The Game Description Language (GDL) is a logical language able to describe *„finite, discrete, deterministic multi-player games of complete information“* [20]. It allows reasoning by giving a number of facts that are either true or false in a given state. A set of true facts therefore describes the world of the game at any time. The initial state of the game and the terminal state(s) are especially marked. A game description does not have to contain all possible states, because the state space is defined by the initial state, the possible moves and the terminal conditions. From these all intermediate states can be calculated, by applying consecutive player moves on the state. A move is a relation, that has preconditions, which need to be true before it can be made and it describes what the next state looks like in relation to the current state. By applying a move the initial state is therefore transformed to an intermediate state. When a terminal state has been reached by any player the goal relations are executed and return a score for every player of the game. This is explained in Figure 2.2.

GDL is based on the programming language Datalog and has some game related relations predefined, listed in Table 2.2.

Name of Relation	**Description**
(role r)	r is a player in the game
true(fact)	the fact holds true in this state
init(fact)	the fact holds true in the initial state
next(fact)	the fact will hold true in the next state
legal(role, move)	when the conditions behind this legal relation are met the player *role* is allowed to play the move *move*
does(role, move)	indicates which move a player actually performs
goal(role, score)	When the conditions for the goal relation hold the player *role* has achieved a goal value of *score*
terminal	when the conditions of this relation are true the game ends

Table 2.2.: List of GDL-relations and their purpose

The Game Description Language is the standard by default of game description languages at the moment. Most research on General Game Playing uses or extends it[5]. This may in part be because of the infrastructure provided for it by the scientific community. In addition to the language itself, its' inventors designed a Game Master service that defines an interface these programs must conform to and allows different players to compete against each other. The Computational Logic Group at TU-Dresden has implemented a General Game Playing Server (GGP Server) as an Open Source Project[6] and hosts a public instance where anyone can test his players[7].

On the servers of the University of Stanford or TU-Dresden programmers can prepare for the yearly GGP Tournament that features a prize of $10,000 for the winner.

Extending the Game Description Language

Several attempts have been made to extend GDL to allow non-deterministic games and games of non-complete information. This would increase the range of games, describable with GDL significantly. Random events are crucial to dice games, most card games and many complex systems. Hiding the result of random events to other players creates a new challenge for general game players. They then have to model the view of the game from their opponents perspective as well as their own.

A first approach by Kulick at FU-Berlin in 2009 added random events, non-complete information, real-time capability in contrast to the turn based system used in GDL and a library system allowing for easy reuse of game description parts [13, 14]. This language was called World Description Language (WDL). WDL kept backwards compatibility to GDL and provided a World Controller based on the GGP Server of TU-Dresden[8].

In 2010 Thielscher published similar but less ambitious research and proposed a new dialect of GDL called Game Description Language with Incomplete Information (GDL-II) [7]. It kept backwards compatibility to GDL and introduced two new relations. One is called *random* and acts like an independent player that chooses random moves. This relation can be used to simulate non-deterministic events like dice rolling. The second one is called *sees* and controls the visibility of information. All information is unknown to the players until it is revealed to them with a *sees* relation. These additions suffice to describe *finite, discrete, non-deterministic multi-player games of incomplete information.* The GGP Server is able to hold matches in GDL-II and the language has therefore already been used in competitions[9].

[5] This literature list is a good starting point when searching for research on General Game Playing: http://www.general-game-playing.de/literature.html

[6] The project can be found on sourceforge: http://sourceforge.net/projects/ggpserver/

[7] A public instance of GGP Server from the TU-Dresden: http://ggpserver.general-game-playing.de/

[8] Executable and source code of the World Controller are available under: http://gameai.mi.fu-berlin.de/ggp/resources.html

[9] The GDL-II track of the German open in GGP 2011 is available here: http://www.tzi.de/~kissmann/ggp/go-ggp/gdl-ii/

It is too early to tell which extension will spread more widely but GDL-II definitely has the head start.

2.1.3. Strategy Game Description Language

In contrast to the GDL the Strategy Game Description Language (SGDL) is not aimed at being as general as possible, but concentrates on the specific needs in describing strategy games. It was invented by Mahlmann, Togelius and Yannakakis at the IT University of Copenhagen in 2011 and is still under active development [4, 1, 5]. Their goal was not to create sophisticated AIs for existing games, but to allow procedural creation of new strategy games through evolutionary algorithms. This has affected the design of SGDL, to use tree based representations for describing game mechanics because tree representations are widely used in evolutionary algorithms.

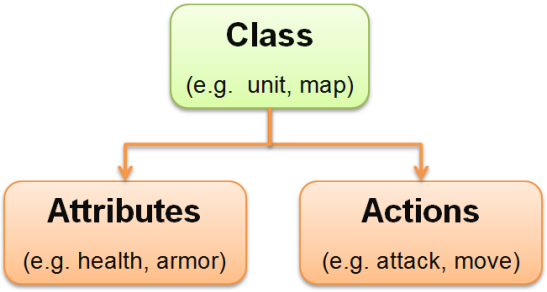


Figure 2.3.: In SGDL object classes consist of attributes and actions.

The elements of a SGDL game are called object classes. Every object class defines some attributes and actions, which belong to every object of this class (see figure 2.3). The different classes, like units, map and buildings can therefore have different properties and abilities. Attributes themselves have a name and a value. Actions are more complex, having conditions which have to be fulfilled for their consequences to happen. This corresponds to relatively simple if ... then ... statements. Conditions and consequences alike are again represented in a tree structure composed of different kinds of nodes: Actions, comparators, operators and constants:

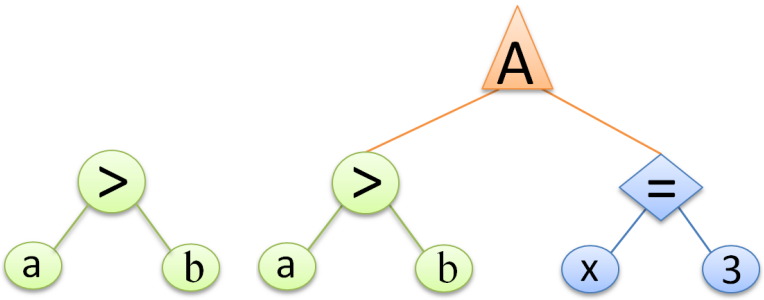
Action Checks all conditions for their value. When all return true the consequences are applied.

Comparator A boolean logic term that returns a boolean value.

Operator Either assigns a value or performs a mathematical operation on its child nodes.

Constant Is a leaf node that contains a single value.

The SGDL is not yet fully documented and no implementation is publicly available. But the inventors have already added more functionality to allow object creation, non-deterministic events and differentiation between the global game state and the player



(a) A condition (b) A condition combined with a consequence

Figure 2.4.: A condition tree and an action tree with one condition on the left and a consequence on the right.

game state [3]. SGDL is not nearly as mature as GDL and it remains to be seen what can be learned from this new game description language.

2.2. Search Algorithms for Games

While game descriptions are necessary to define a games' rules and objects GGP additionally needs agents playing these games. Playing a game can be described as making a series of decisions until the game ends. GGP is therefore the art of making good decisions in very different games. This leads to the question of how programs decide which move to take next. There are only two viable options.

On the one hand the program can analyze the current game state and decide based on rules, like an opening book [19], or completely at random.

On the other hand programs may try to analyze future game states, by trying out different moves in their memory and then deciding which of these is best.

Both strategies are applied by humans when playing. In a chess game a human player might conform to the rule to open with *pawn to e4* every time he has white and is therefore first to move. But when reacting on situations that the player can not look up in an opening book, he will most likely think about different moves, some steps ahead, and then play the move that seems to lead him into the best situation even if his opponent plays his optimal moves.

Chess or Go programs combine these strategies most of the time. A set of rules, provided by human experts or by learning from previous games and moves, is the foundation for all of their decisions. To this information, a search strategy that plays several steps ahead or even to the end of the game, is added. The combination of searching by rules and looking into future steps is called a search algorithm for games.

Why brute force might not be enough

When regarding AIs performance in games one can identify two largely different groups, depending on their so called tree space. As different choices of moves starting from an initial position can be represented as a tree, we call the number of nodes in this game tree: the tree space of a game. One group of games has relatively few different moves and paths to the end. Tic-Tac-Toe or Connect Four are good examples for these games. They can be extensively searched by AIs and may therefore be considered solved. Creating an AI for these games is a straightforward task. Play all possible moves and calculate the terminal states of the games. Then play the move that will most certainly lead to a win for your side. This is no challenge to AI any more. The other group of games still presents a challenge to AI. These games have a huge tree space and can therefore not be extensively searched in a reasonable amount of time. AIs can therefore explore only parts of his tree space's possibilities and then choose which to explore. Chess and Go are the pre-emminent examples for these kind of games.

How large the tree space for a game is can be calculated with very little information about the game. It depends on m, the number of moves available to the player at every turn and l, the number of moves a game may last. An upper limit for the tree space is then:

$$limit = m^l$$

This term gets obviously very large for long games with high m or games with many different moves l.

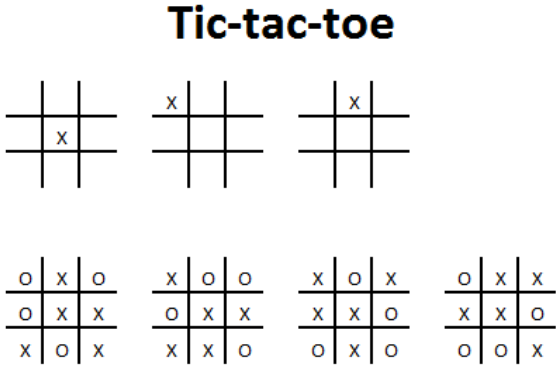


Figure 2.5.: *Examples of Tic-tac-toe situations*
First row: possible first moves of player X
Second row: the same draw end state in four symmetric versions

In Tic-tac-toe there are never more than nine possible moves and the game only lasts up to nine turns. The upper limit for the tree space of Tic-tac-toe is therefore $9^9 = 387,420,489$ which is deceptive. While 9 is the number of moves in the first turn this possibility then decreases by one for every following move. When calculating the possibilities for every move this leads from $9^9 = 9 \times 9 \times 9 \times 9 \times 9 \times 9 \times 9 \times 9 \times 9 = 387,420,489$

to $9! = 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 362,880$. This number, though a lot smaller, is still to high, because Tic-tac-toe uses a symmetric board. The first mark may be set in a corner (4 similar moves), in the middle of a side line (4 similar moves) or in the center (1 move). As it doesn't matter in which order moves are made and there are different move combinations leading to the same game situation later on, the state space is reduced even further. The state space is defined as the number of different game states or game positions that might occur in a game. When reasoning about game AIs it might be reasonable to add some simple rules players will apply. One common rule is to play a terminal move if the player is able to do so, because these are relatively easy to spot. In his analysis of Tic-tac-toe's state and tree space in 2002 Schaefer [52] added these and some more considerations and found a reasonable state space of 230 positions, leading to 138 terminal positions with a tree space of 1,145 different paths to these terminal positions. Tic-tac-toe can therefore easily be searched extensively.

For chess it's a whole different story. On the first move white has 20 possible moves to choose from. Pawns one step ahead (8 moves), two steps ahead (8 moves) and four positions for the Horses to go to (4 moves). The possibilities increase rather than decrease when the game moves on and there is no relevant symmetry. The average number of valid moves was calculated by De Groote in 1946 to be around 30 [48]. This calculation was used by Shannon in 1950 to estimate the game tree complexity of chess to be around 10^{120}. In 1994 Allis based his game tree complexity evaluation of 10^{123} on an average of 35 valid moves and 80 moves per game, 40 for white and 40 for black. When calculating with the speed of Deep Blue, the chess playing computer that on average calculated 126 million moves in a second [35], it would need $\approx 8 \times 10^{114}$ seconds or $\approx 2.5 \times 10^{107}$ years. In comparison the age of the universe is estimated to be approximately 4.32×10^{17} seconds, which is *only* 1.37×10^{10} years[10]. A brute force attempt for the whole game is therefore out of the question and an analysis of all steps has to limit itself to a very shallow depth.

[10] The age of the universe has been obtained from Wolfram Alpha: http://www.wolframalpha.com/input/?i=age+of+the+universe

Tree Based Search Algorithms

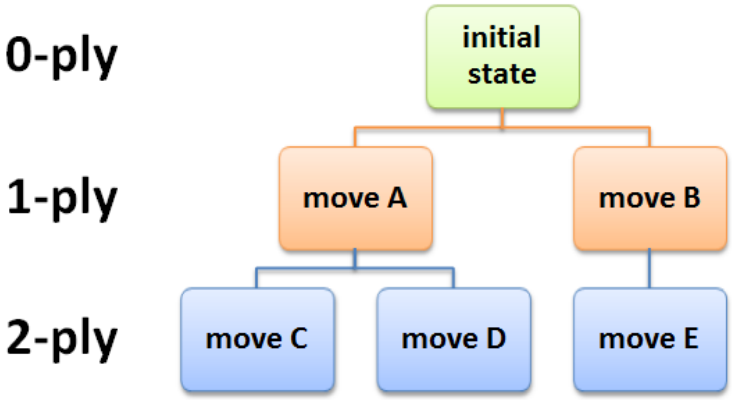


Figure 2.6.: Example of a game tree

To find the best moves, programs have to create a game tree with a node for each state and edges for moves leading to these states. While doing this every game playing algorithm has to face two problems.

Deciding Which Move To Take The algorithm has to decide which nodes to expand and which moves to explore until it reaches some limit. This limit can be a time constraint or a depth limit of how far the algorithm is supposed to look ahead. This depth limit is measured in plies, where a ply is one move of one player.

Evaluating The Game State Besides deciding which nodes to expand, these nodes, which represent game states, have to be evaluated. This evaluation is easy for extensively searched games, because it can be calculated backwards from the terminal nodes. When stopping the search before a terminal node is reached the game state must either be evaluated based on a heuristic (see 2.2.1) or based on statistical playouts (see 2.2.2).

2.2.1. Minimax and Alpha Beta Search

In deterministic single player games the AI may choose a move from the current node that promises the highest reward. This is different in competitive two player games, where both players will try to achieve the best possible result for themselves. In zero sum games, where players compete for the same result, the opponents choices have to be considered. Simply taking the move with the highest possible result at the end might lead to situations where the opponents move will stop the player from achieving this result.

Minimax

Minimax is the standard algorithm for two player games with perfect information [32]. It *maximizes* the *minimal* result the player will reach. Hence it is called Minimax. Figure

2.7 shows an example. A short description of the algorithm:

1. The search tree is explored until the maximum number of plies search is reached.
2. A value is assigned to the leaf nodes. Wins for player A get $+\infty$, wins for player B $-\infty$ and the rest are evaluated by a heuristic.
3. The values are now back-propagated through the tree: Whenever it is player A's move to choose he chooses the move leading to the node with the higher value. Player A is *maximizing* his result. Whenever it is player B's move he chooses the move with minimal value for player A. Player B is *minimizing* player A's result.
4. Player A makes the move leading to the node with the highest value previously calculated.

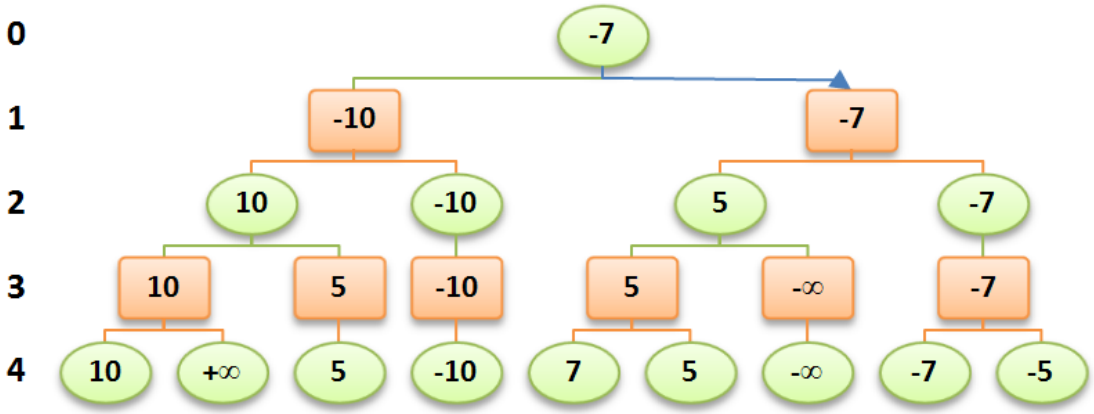


Figure 2.7.: An example of a simple Minimax search with 4-plies depth. Player A, green, chooses the highest value moves in row 0 and 2, whereas player B, orange, chooses the minimal value moves in row 1 and 3. The chosen move is marked blue.

Alpha-Beta Pruning

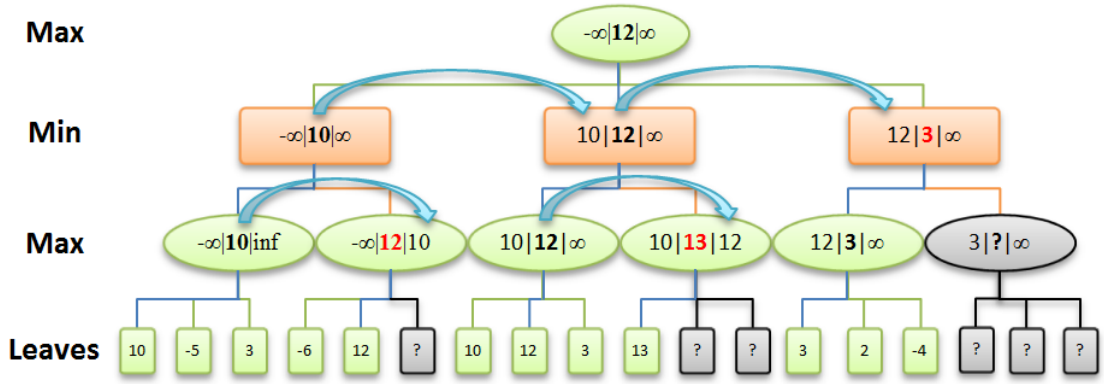


Figure 2.8.: An example of alpha-beta pruning. Game state nodes contain alpha, value and beta. Blue lines show which values or moves a player would choose in this situation. Blue arrows show how alpha and beta values are updated. Red values show values leading to alpha or beta cutting. Black nodes do not have to be searched thanks to alpha-beta pruning.
The maximizing player would choose the middle move first, the minimizing player then chooses the left move, because this returns a lower value of 12 instead of 13 for his opponent.

Minimax is a very expensive search algorithm because it needs expansion of the whole (sub)tree and has to look at every node in order to decide which move to take. Alpha-Beta Pruning is an extension optimizing Minimax. It was independently discovered several times and published in detail by Edwards and Hart in 1963 [45].

The basic idea is to avoid searching branches that will certainly never be played. To make the decision, that a branch will never be played, the results from previous branches can be used. To save this information two additional values are introduced. The highest value that player A is sure to achieve from a given node, α, and the smallest value that player B can force from a node, β. Alpha is initialized with negative infinity, beta with positive infinity. Based on these values two cut-off conditions are set. When the α-value of a node, where player A is allowed to choose the next move is higher or equal to the β-value of the parent node the branch does not need to be searched any more. This is due to the fact that player B would never choose the move leading to this node, when he can play a move that sets the maximum result of player A to β. When the β-value of a node, where player B is allowed to choose the next move, is smaller or equal to the α-value of the parent node the branch can also be cut. Player A would not choose a move that allows player B to force a lower result than player A is sure to get when taking a different move.

The savings from cutting branches depend on the ordering of the moves. When the best moves are searched first good α and β values allow the cutting of branches very quickly. In the worst case Alpha-Beta Pruning is not faster than Minimax, but in the

best case the branching factor can be reduced to its square root, allowing to search two times the depth than Minimax. In an average case the branching factor can be reduced by 25% percent [34].

Alpha-Beta Pruning has long been the prime algorithm used in game playing, especially chess. But it has one precondition that made it perform quite poorly in GGP. To start a search the leaf nodes have to be evaluated. In Chess and other games this is done based on heuristics composed through expert knowledge and experimentation. For other games like Go no heuristic is know or can be known in the case of games describes with a game description language. In these cases Alpha-Beta Pruning might cut off the best path if the evaluation of the result is wrong.

2.2.2. Monte Carlo Tree Search

The problem of estimating a value, which can not be calculated deterministically, is not unique to game playing. To solve this problem Monte Carlo methods have been used in many different fields of mathematics or physics since the 1940s [44]. The idea behind the Monte Carlo method is quite simple. With randomly created inputs a huge number of deterministic computations are performed and their results aggregated. The result of these computations will converge to the real result. When applied to games this means, that a large number of games are played randomly to the end and the results are then used to evaluate the starting state. A formula for the expected result is then $\delta = \frac{t}{n}$, where δ is the expected result, t is the sum of all results and n is the number of playouts[11] performed.

Monte Carlo methods had already been applied to games like Poker and Scrabble when the Monte Carlo Tree Search (MCTS) was proposed in 2006. MCTS was then able to decisively improve the AIs performance in the game of Go against human players [29, 21]. Since then it has been used to advance AIs for several other games as well [18, 31, 11]. MCTS is a four step algorithm adapting Monte Carlo methods to tree based game search algorithms.

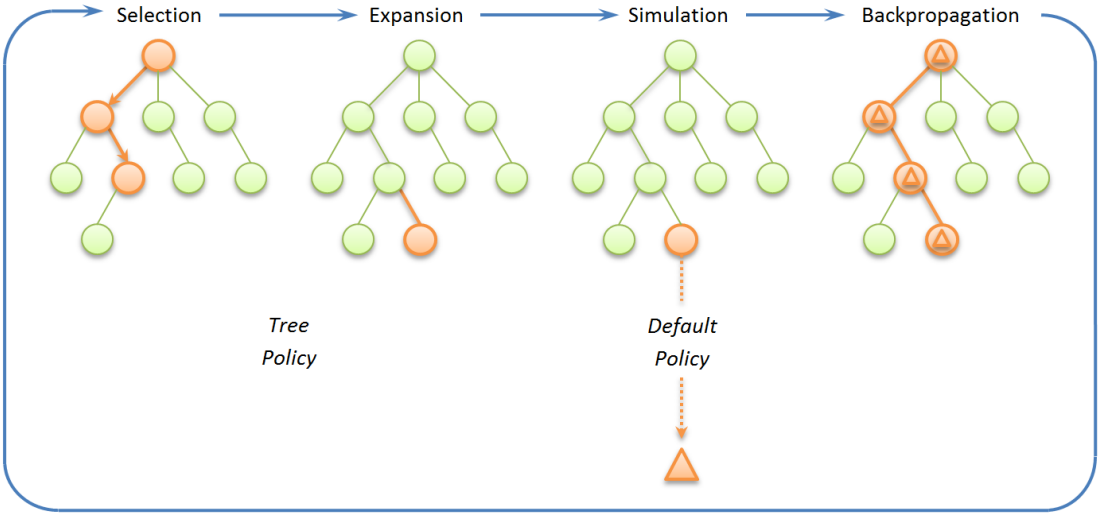


Figure 2.9.: The four steps of Monte Carlo Tree Search explained.
Illustration like Brown et al. [2]

[11] A playout is the process of playing a game from any game state to a terminal one.

The four steps are:

Selection Starting from the root node the game tree is traversed and one node selected for expansion. This is done by recursively comparing the estimated values of the current node, starting from the root, and its children and choosing the highest one. The value is the expected result when choosing the action leading to the node. It is calculated by using knowledge from previous playouts.

Expansion An unexpanded action is chosen from the selected node and a new node created for the resulting state.

Simulation From the newly created game state node a playout is started. The game is played using the *default policy*, explained below, until a terminal state is reached.

Backpropagation The value of the terminal state is propagated to all nodes on the game path. This result is used to update the statistics of the nodes and refine the estimate of the real value. In the simplest version the estimate would be the average of all values. How the estimates are calculated defines the selection strategy. Some strategies are explained below.

The first two steps are called the *tree policy*, the third step uses the *default policy*. The tree policy therefore defines how the game tree is traversed until a node that is still expandable, i. e. has actions not previously expanded and is not terminal, has been reached. The method whereby actions are chosen during playouts is determined by the default policy. The standard default policy randomly selects an action to play. This is what Monte Carlo simulations for games have been doing before.

MCTS's great improvement lies therefore in the introduction of the tree policy, which is informed by previous playouts. The tree policy has influence on two oppositional goals one wants to achieve while searching.

Exploration Exploration is the goal to look at every possible move and find the best one.

Exploitation Exploitation is the goal to concentrate on promising moves and improve them.

Upper Confidence Bound

The problem of trying to balance taking the current best possible action, exploitation, and trying out new options, exploration, is called the multi-armed bandit problem. It is named after traditional slot machines called one-armed bandits, because the problem resembles a slot machine with different levers that have different but yet unknown reward distributions. The goal of any strategy and the solution to this dilemma, would be to find the action with highest reward while achieving the lowest possible *regret* due to

exploration. Auer et. al. showed in 2002 that there exists a strategy that achieves logarithmic regret without any prior knowledge about the actions reward distributions [36]. They called this algorithm Upper Confidence Bound 1 (UCB1). Play the action j with the highest value in

$$x_j + \sqrt{\frac{2 \ln n}{n_j}}$$

where x_j is the average reward for an action j, n is the total number of plays done and n_j is the number of times action j has been taken. The value x_j is the exploitation part, getting larger the better the results from this action become. The equation $\sqrt{\frac{2 \ln n}{n_j}}$ is responsible for exploration. It grows with the number of plays that do not take this action, so that every action will eventually be chosen.

In 2006 Levente Kocsis and Csaba Szepesvári applied UCB1 to MCTS and called that UCB applied to Trees (UCT) [30]. They showed, that UCB1 is viable for tree based search and UCT is able to significantly improve search results compared to Alpha-Beta Pruning, plain Monte Carlo and other algorithms, in some domains.

There has been further research in what strategy is best suited for tree search and several improvements, mostly for the exploitation part, have been proposed [27, 16]. Table 2.3 shows some strategies important for this master's thesis.

Name	**Formula**
UCB1	$x_j + C\sqrt{\frac{D \ln n}{n_j}}$ with different constant values for C and D [36]
UCB-Tuned 1	$x_j + C\sqrt{V_j \frac{\ln n}{n_j}}$ with $V_j = \max(D, x_j(1 - x_j))$for small D [23]
UCB-Tuned 2	$x_j + C\sqrt{V_j \frac{\ln n}{n_j}} + \left(\frac{\ln n}{n_j}\right)$ [23]
SP-MCTS	$x_j + C\sqrt{\frac{D \ln n}{n_j}} + \sqrt{\frac{\sum r^2 - n_j x_j^2 + D}{n_j}}$, where $\sum r^2$ is the sum of the squared results previously achieved, corrected by the expected results. D is set high to stress the uncertainty of rarely explored moves. This Formula has been proposed for Single-Player Monte-Carlo Tree Search (SP-MCTS) [25, 26].

Table 2.3.: Several UCB strategies used in Monte Carlo Tree Search.

Besides various tree policies another focus of MCTS research have been parallelization schemes, to improve MCTS performance in competitions.

Parallelization Schemes

In recent years multi-core systems have become common, even in personal computers. Parallelization of algorithms has therefore become more of an issue. For MCTS several parallelization schemes have been proposed and tested [9, 17]. A short summary of them can be viewed in Figure 2.10.

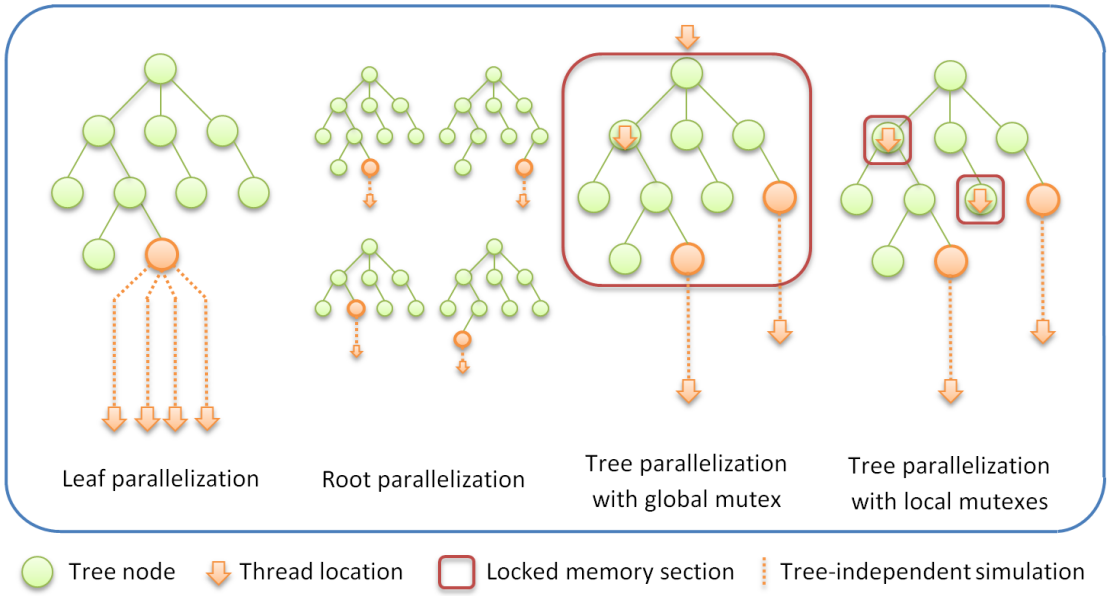


Figure 2.10.: MCTS Parallelization Schemes

In *leaf parallelization* the simulation step is parallelized and multiple simulations are run for the newly expanded node. The disadvantages of this approach are, that the selection, expansion and backpropagation phase are not parallelized and the search has to wait for all simulations to terminate before it can continue.

In the *root parallelization* scheme every process builds its own search tree and searches independently. This is especially suited if the different processes don't share memory, e. g. in a distributed network.

At last two *tree parallelization* schemes have been proposed. The less aggressive one uses a global mutex, to limit access to the tree, so that only one process at a time can select and expand a node. While the process is running its simulation, other processes are allowed to access the tree. This approach performs well, if the tree search costs little time, when compared with the simulations. Instead of a global mutex, local ones can be used. When every node can be locked independently, processes can access the tree at the same time. This creates overhead for mutexes and locking of every node. In both tree parallelization schemes a problem has to be solved: Different processes would select the same node to expand, but this might not be a good idea. For this scenario a *virtual loss* has been proposed by Chaslot et. al. in 2008 [22]. The UCB values of the nodes on the selection path are decreased until the simulation has finished, prompting other processes to take different routes and expand different nodes.

2.2.3. Nested Monte Carlo Search

Nested Monte Carlo Search (NMCS) has been proposed by Tristan Cazenave as an alternative to the memory consuming MCTS in 2009 [12]. The algorithm does not build up a tree, but improves upon simple Monte Carlo search from the initial state, when a nested search is made for every possible action. Their results are compared, the action with the best result is chosen and the path to the best result is saved. From the state reached by taking this action every possible action undergoes a nested search. Their results are compared as well. The action with the best result is chosen this time only, if it is better than the best result from all previous searches. If it is worse the next action is taken from the path to the best result. The next state is created by playing the action and this process continues until a terminal state is reached. The nested search itself is limited to a level depth after which the algorithm changes from a nested playout policy to one playout with random moves. A pseudo-code version of NMCS is shown in Algorithm 2.1.

Algorithm 2.1 Pseudo-code of Nested Monte Carlo Search

```
int nestedMCS(position, level){
    int bestScore = -1;
    while(state is not terminal){
        if(level == 1){
            // stop nested search and play to the end
            move = max(playGameForEveryPossibleMove(position));
        }else{
            move = max(nestedMCS(everyMove, level-1));
        }

        if(score of move > bestScore){
            bestScore = score of move;
            bestPath = path of move;
        }
        position = play(position, bestMove);
    }
    return score;
}
```

Nested Monte Carlo Search seems to be especially useful in single-player games and has been combined with MCTS to improve overall search results. The winner of the General Game Playing World Championship in 2009 and 2010 was a program called Ary[12]. In 2010 Ary split computing time between MCTS and NMCS and took the best move of both search runs [15]. NMCS therefore seems to be a useful strategy that can, due to its low memory usage, be run parallel to MCTS.

[12] For a list of all GGP World Champions since 2005 see: http://www.general-game-playing.de/activities.html

2.3. Software Engineering

In order to allow the reader a deeper understanding of implementation decisions that may or may not have had an effect on the experiments conducted, some parts of the implementation are explained in a section of each chapter. Because this is a master's thesis in computer science most techniques will be familiar to the reader. As this thesis does not propose new ideas to software engineering this chapter will only explain some methods used during implementation.

2.3.1. Design Pattern: Abstract Factory

Used in Monte Carlo Node Factory

An Abstract Factory provides an interface to create objects. Its intent is to *„provide an interface for creating families of related or dependent objects without specifying their concrete classes“* [38]. The object using the factory, therefore, does not have to know the specific class of the object that is created. He only accesses a predefined interface. As the factory class is abstract itself, different concrete factories can be given to the object to create new objects. This pattern is often used in graphical user interfaces, when the same abstract objects, e. g. window or button, are created and behave differently depending on the platform the program is currently running on. An Abstract Factory encapsulates the knowledge of how to create and the decision of what kind of object to create.

2.3.2. Design Pattern: Composite

Used in Expression (see 4.3)

The Composite Pattern is used, when objects are stored in a tree-like structure and the handling of tree-parts and the whole tree should be the same [53, 38]. For all tree object classes, the leaves and compositions, a common abstract interface is defined. This interface is then used to access objects in the tree and to perform operations on it. The operations defined in the common interface are either implemented in a standard version in the interface or by the leaves and compositions themselves. The Composite Pattern allows to add and change what kind of objects may be part of the tree, without changes to the methods using it.

2.3.3. Design Pattern: Proxy

Used in Attribute Reference (dynamic one)

The Proxy Pattern describes how an object is accessed. A proxy is an object that grants access to another object, acting as a placeholder for it [33, 38]. This is useful for example, when the real object is large and should only be loaded when needed or the object is in a remote place. The proxy has to conform to the same interface as the object

it represents, so that it can be used on the object's stead. It encapsulates the knowledge on how to directly access the object.

2.3.4. Design Pattern: Template Method

Used in Search Algorithm search()

A Template Method is a method that defines the structure of an algorithm by calling other methods [38]. These other methods do not have to be implemented by the class itself, but may rather be used to define access points for subclasses. A subclass can influence the algorithm by changing these methods, without changing the overall structure of the algorithm. The Template Method Pattern allows to encapsulate which parts of an algorithm should be changeable and which not and promotes code reuse in different algorithms through defining the common parts in their parent class.

2.3.5. Performance Analysis with VisualVM

While algorithms and data structures with good performance are important and have to be considered in the design stages of software development already, there always is a tradeoff between performance and readability and/or effort. Over-optimizing for performance during design and in early development phase may therefore be hurtful to the progress of the project as a whole. When it is not sure whether performance is a major issue for a module or class the general rule might be to create a solution, which is as fast as possible without having a major negative impact on effort or readability.

Especially in complex systems with unknown run-time properties, benchmarks can give valuable feedback on the performance and a performance analysis can help in finding bottlenecks. VisualVM[13] is a visual profiling tool for java programs. It consists of several scripts, accessing the java virtual machine while the program is running, and shows their results in a meaningful way. It can be used to analyze where the program spends its time, how much memory the objects consume and in which states the threads of the program are in. This information can then be used to find the most promising candidates to improve in re-factoring.

[13]The VisualVM tool can be downloaded for free at: http://visualvm.java.net/

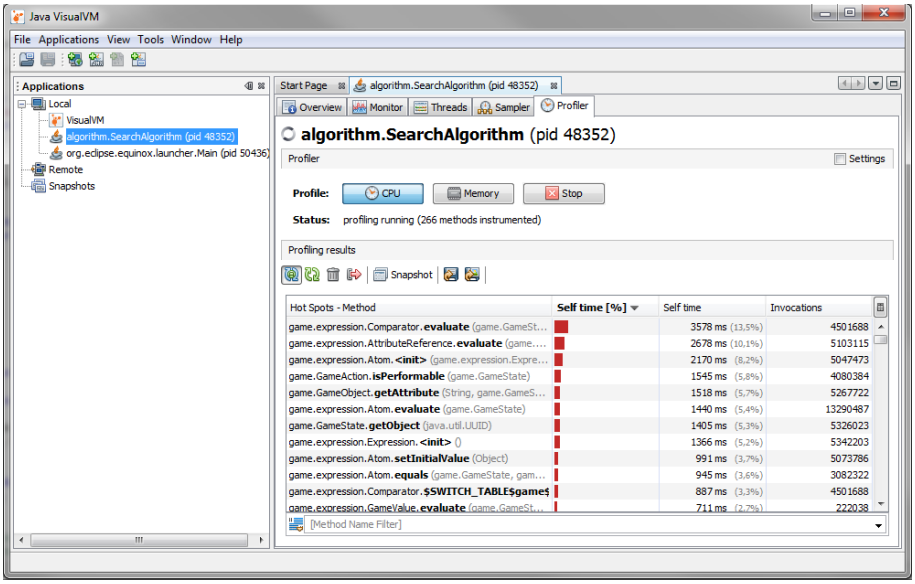


Figure 2.11.: A Java VisualVM CPU profiling example. The running applications are listed in the left column. On the right different profiling, sampling or monitoring options can be selected and their results viewed below.

3. Project Goals and Structure of the Software

As previously mentioned this master's thesis is about the prototype of a software tool meant to be applied in the development process of simulation style social games at Wooga. Because it is not for academic purposes only, the decisions made during design and implementation reflect the goals of this project. To make the reasoning process as transparent as possible this chapter discusses the most important goals for the prototype, derived in part from the use cases in Chapter 3.2. In Chapter 3.3 the overall structure of the prototype will be presented.

3.1. Goals of the Project

The tool will be *used by Product Managers* responsible for game balancing. They mostly *don not have a computer science background* or any deeper knowledge of programming languages. It is meant to be used in the *pre-release* phase of development, because at this point in time no real user data is available. After the game's release, statistical user data is used to improve the balancing of the social game.

3.1.1. Test Validity of this Approach

Creating an easy to use but still powerful tool with a game description language accompanying it is the task of this master's thesis. More important than actually implementing this tool is to evaluate if this approach is viable for game balancing. This evaluation can then be used to decide, whether or not to dedicate more resources to creating such a tool. The tool developed for this thesis is therefore a prototype to confirm or dismiss the approach.

3.1.2. Supporting Game Balancing

The purpose of this tool is to support the process of game balancing. Game balancing is a mainly mathematical task at the moment. The relationships between different values are expressed in formulas and graphs are charted to visualize them. Visualizations of the game mechanics in game loops are used to analyze the economy and find self-reinforcing loops or faucets and sinks of the games' currencies. Prototypes in the early development phase

and later Alpha and Beta versions can be used for user testing. Due to the low number of users this feedback will be qualitative rather than quantitative. With a description of the game and some configuration files the tool is supposed to return meaningful information to the Product Manager. The most general of which is how fast users of various play styles, e. g. power, mid-range and novice, will progress through the game. Additionally, information about the usage of different features and trends of some values could be displayed. This is schematically displayed in Figure 3.1.

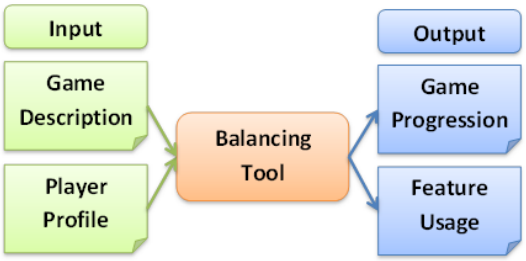


Figure 3.1.: A graphical representation of the input-output relation of the balancing tool. Input is a game description, with configuration files, and a player profile, describing a playing style.
The output is the best terminal state of a game and the path to this state, as well as some additional statistics on how different features of the game have been used.

3.1.3. Reusability of the Game Description Language

The tool is only useful, if the Game Description Language it includes is able to describe different games without needing any modification by a programmer. Generality in its limited field of social building simulation games is therefore a key requirement.

3.1.4. Ease of Use

One of the goals is to make using the tool as easy as possible. The intended user may not be able to write programs on his own. The syntax of game descriptions should be no more complex than SQL. After the game itself is described small changes to the configuration, e. g. reaching level two needs 100 instead of 50 experience points, must be simple and fast. The program has to be runnable with few parameters, allowing good results without sophisticated tuning by the user.

3.1.5. Performance of the Analysis

How the tool can be used depends to a great extent on the performance of the analysis. If it needs hours to get results, it would not be helpful in fine-tuning values iteratively. For a first steady result of the search a target was set at no more than 60 seconds. This would allow an easy perusal of huge changes and an opportunity to look deeper into promising

settings. For a more complete analysis of different user profiles, time is not a major issue. An in depth analysis may need 24 hours to produce results and will therefore only be used to confirm what has previously been iteratively created.

3.2. Use Cases

Defining how the software is going to be used is a vital part of the software development process. Use cases help discover the key players and their needs. Because this is an exploratory thesis, trying to show the possibility of using AI for game balancing, successfully describing and playing games is more important than the software to use it. The use cases for the tool are therefore on a high level, detailing what features the underlying software must allow but not specifying how it is going to be used in full detail.

Disclaimer: These use cases are not fully supported in the prototype of the balancing tool right now.

Create Game Description

Actor: Product Manager

1. Create game configuration (favorably with Excel)
2. Describe game mechanics
   a) Combine game configuration with game description
   b) Describe possible actions
   c) Describe initial condition of the game
   d) Describe terminal conditions
      e. g. 7 days passed, reached level 20, finished 5 sessions, ...

Change Game Configuration

Precondition: Game description ready (*Create Game Description*)
Actor: Product Manager

1. Change configuration values (favorably within Excel)
2. Calculate result and different metrics (if possible with one button-click)
3. *Analyze result* and *change game configuration* again if necessary

Analyze Result

Precondition: Result calculated
Actor: Product Manger

1. Compare values to previous results
2. *(optional)* Analyze different game features
   a) How did a value change over time?
      e. g. progress of credits, experience in chart
   b) Which actions did the player take and which not?
      e. g. he built seven schools, zero churches

3.3. Software Structure

The project is split into two distinct modules. The first module contains everything related to the game descriptions. This module is used by the second module containing the search algorithms. This division allows the usage of different search algorithms without the need to change any game description specific features.

Figure 3.2 and the description beneath give an overview of the tasks of the modules and connections between them, whereas the next two chapters explain their features and details.

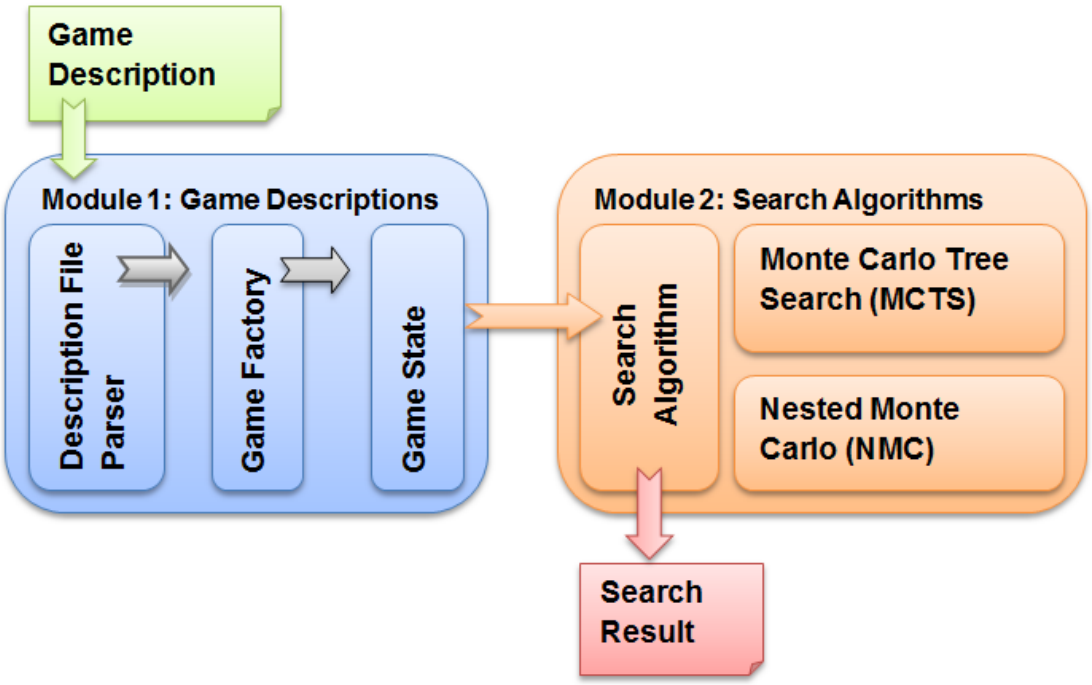


Figure 3.2.: Overview of the software structure: From game description to the search result

Module 1: Game Descriptions

The game description module consists of three parts. Its input is a game description file that is parsed by the *Description File Parser*. When the file has been parsed a *Game Factory* is created, with descriptions for all features of the game. From this factory the initial *Game State* is created. This state is used by the second module.

Module 2: Search Algorithms

The search algorithm module defines a search interface and currently has two different implementations for it. *Monte Carlo Tree Search* or *Nested Monte Carlo* are used to search from the initial game state to a terminal state. Both try to find the best possible result in a given amount of time. When the search has finished the result is returned.

4. Design of a new Game Description Language

In this chapter a new Game Description Language for social building simulation games is introduced and its features are explained.

The Overview in Chapter 4.1 states the problem addressed by the new game description language and shows why no currently existing game description language is suitable for the problem domain. Chapter 4.2 informs about the language features and its expressiveness, before the next chapter lays out some implementation details.

4.1. Overview

4.1.1. Goals and Requirements

Derived in part from the global project goals (see Chapter 3.1) some requirements for the game description language are set here. They have a major influence on the design of the new language.

Human Readable Because it was not clear at the beginning of this thesis, whether there would be a graphical user interface with which to create game descriptions, it was an important goal to create a human readable and writable description language.

Short Pragmatism acknowledges that the time it takes to describe a game will influence the likelihood of this tool being used. Short game descriptions, avoiding unnecessary repetitions, are therefore a goal in and of itself.

Easily Create New Content It has to require few or no changes within the game description to add new game content. In the optimal case an additional line in Excel translates to new content in the game.

Easily Change Content While it has to be be easy to create new content, changing the existing content has to be even simpler. Changing content is at the core of game balancing as described in the use case *„Change Game Configuration“* in Chapter 3.2.

Reusable This goal has two meanings. The description language has to be reusable, without changes to the language itself, for different games. That is simply what

a game description language is all about. Additionally, it is helpful to be able to reuse parts of game descriptions. As most social games today include some kind of leveling and energy mechanic, reusing them for different games makes game descriptions more understandable and faster to create.

Describe Social Building Simulation Games The new game description language has to describe social building simulation games. Their peculiarities are explained in the next section.

4.1.2. Social Building Simulation Games

As there is no common definition of social building simulation games (SBSG), this section will define the term and create a common understanding.

For a first game theoretic description the terminology from Chapter 2.1.1 is used to delimit the game family. Social building simulations are:

Infinite But Finite They start, when the player first logs in and gets his initial state, but are *not supposed to end*. The goal is to get ever bigger, climb up the ranking or stay on top, or just to play on. Players might stop playing, when there is no more new content for them to explore, but social games developers try to avoid that by constantly adding new content. For the purpose of game balancing we can define terminal conditions and create a *finite* game.

Discrete Players can only play one action at a time and the number of actions is limited. Actions are time based, but even time can be counted in discrete seconds.

Deterministic The result of player actions is often influenced to a small degree by chance. When a player does something there is always a small chance of him getting an unexpected reward. Some actions only yield the desired result with a likelihood, which is frequently unknown to the player. As non-determinism is not at the heart of these games they can be approximated as deterministic games for the purpose of simplicity. This simplification has of course an effect on the results for game balancing. The extreme cases of chance, very lucky or very unlucky players, will not be in the simulation, but when a game is played by millions of players very unlikely cases are bound to happen.

Single-player It might sound paradoxical, but to date social building simulations are mostly single-player games. They play like the single-player campaigns, known from traditional games, with a limited number of ways to interact with other players. There are two different kinds of interactions between players.

Visits Players are able to visit each others properties and perform some tasks therein. The visiting player gains some small rewards for his actions. The visited player profits as well, because the actions from the visiting player are

performed on his property as if he would have done them himself. Only in some battle games with a building simulation part may the effects of these actions be detrimental to the visited player.

Gifts And Requests Players can send each other different items, or request that items should be sent to them. The sending player either gets and looses nothing or even gains a reward as well. Gifting and requesting is therefore a win-win situation for both players. It is limited in most of the games, to avoid gifting fatigue and maintain the impression of scarcity of resources, moving players towards purchasing the needed resources.

Actions by other players can be seen as random events influenced by the player's total number of friends. The interaction with others initiated by the player himself can be modeled as actions with a non-deterministic result. As the other players game state is unimportant for most of the interactions and interactions are asynchronous, the games are modeled as single-player games.

Complete Information The player knows everything about the current game state, but there may be uncertainty regarding the future due to non-deterministic events.

Non-zero-sum Players do not compete for the same resources and these resources increase over time.

Symmetric Every player has the same possibilities and starts with identical game states. But the number of friends, their activity and the money the player spends on the game are different for each player. Therefore the rules are the same for every player and the game is symmetric, once the number of friends, their activity and the money available is defined.

Most notable Features

Social building simulation games have some notable features, which pose a challenge to game descriptions. The number of *objects* in the game is *dynamic* and its upper limit is unknown at the start. A *contract mechanic* plays an important role in keeping players coming back to a game. Contracts are actions selectable by the player that have delayed effects. The contract reward can only be selected and restarted after some time has passed. The main source of content are *missions*. They are used to tell the story, show different game features to the player and give him tasks and rewards for achieving them. A major limiting factor is the *energy system* that has been introduced to social games. The player needs energy to perform actions and this energy regenerates over time. This is a strong force in *limiting a players progression* through the game. Another important game mechanic used in these games is *experience*. Players gain experience points (XP) for actions and once the users XP has reached some threshold he gains a new *level*. Leveling-up is therefore a consequence to gaining XP and not to any specific action.

The new game description language is designed to describe finite, discrete, deterministic, non-zero-sum, symmetric, single-player games with complete information, that dynamically create objects, use a contract mechanic and missions as content, may limit player progression through energy and have an experience system.

4.1.3. Other Game Description Languages

Prior to creating a new game description language, the question of whether the currently available languages are capable of fulfilling the requirements mentioned in the previous chapters has to be answered. The two most likely candidates, the widely used and general GDL (or GDL II) and the newly created SGDL, are considered.

Using the Game Description Language

While the GDL (described earlier in Chapter 2.1.2) is supposed to be general in purpose it seems to be especially suited to describe board or card games. There is no class system, which would allow the possibility to easily reuse some functionality, as is used most of the time when creating computer games. This would make it very difficult to add new content without major changes to the game description itself. The GDL is also not well prepared to handle newly created objects in the game. In the GDL all effects of an action have to be explicitly stated for every action. This would make a leveling up system as a side effect to gaining XP very difficult, as it would have to be the effect of every action that increases XP. It might be possible to describe social building simulation games with the GDL, but these descriptions would neither be short nor understandable nor easy to change. GDL and its expansion GDL II are therefore not suitable for describing SBSGs.

Using the Strategy Game Description Language

The biggest hurdle in using the SGDL is its status as a work in progress and that therefore, no implementation or full documentation has yet[1] been published. It is designed to describe all kinds of computer strategy games and might therefore be well suited to allow classes, dynamic object creation and maybe even leveling-up through its world's meta mechanics. Due to this lack of information, SGDL can't be used yet, but because it has tackled some of the same problems, some of its ideas are used as the basis for the new game description language.

4.2. The Wooga Game Description Language

The Wooga Game Description Language (WGDL) is designed to allow the description of social building simulation games as defined above. In WGDL, like in SGDL, the main

[1] As of July 2012 no comprehensive documentation of SGDL is available.

components of a game are objects (*called Game Objects*) and actions (*called Game Actions*). The objects contain the state of the game and the actions are the game mechanics, changing the games state. A game is described through descriptions of its objects, its possible actions, an initial game state and one or more terminal game states. Playing the game involves choosing one of its actions, being available at that point in the game. The action may then change the current game state and the game ends when a terminal state is reached.

4.2.1. Language Features

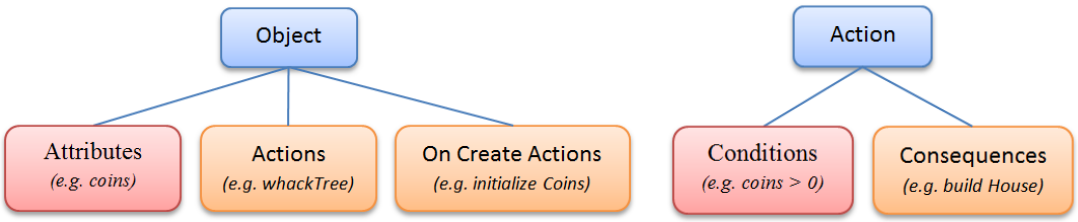
Next, the language features, available to describe games in WGDL, are summarized. The syntax of the feature follows below, because the representation can be looked at and changed separately from the semantics of the description language. The listing starts with the outermost feature, encapsulating all the others, the *Game State*. The other features are then also sorted from higher to lower level ones where such an order exists.

Game State

The container for a game, containing all game mechanics and the current state, is called the *Game State*. It consists of all currently existing objects, all currently available actions, independent of their legality in this state, a set of rules, that are actions that enforce game mechanics and are executed after every player action. The *Game State* also keeps track of time, gaming sessions and the history of actions, leading from the initial state to now. As a special global object, used to store counters, it provides a *Game Value Store*.

Game Object

Game Objects are a central part of every game. They consist of *Game Attributes* containing values, *Game Actions* that are available for this object or that are executed on its creation. Examples in games are buildings and units. To facilitate reusability and dynamic object creation, objects belong to a *Game Object Description* defining its initial values and capabilities. A game objects composition is shown in Figure 4.1a.



(a) An Object (b) An Action

Figure 4.1.: The left figure shows that an object consists of attributes, actions and on-create-actions which are executed when the object is first created.
In the right figure an action with its conditions and consequences is displayed.

Game Object Descriptions and Game Classes

A *Game Object Description* defines initial values of objects and belongs to one *Game Class*. A *Game Class* defines *Game Attributes* and their standard values for objects of that class. Classes inherit attributes and actions from their parent class, if they have any. This allows to encapsulate functionality or state that is shared by objects in a class and expand upon this base class with different child classes. The relationship between classes, object descriptions and objects is shown in Figure 4.2.

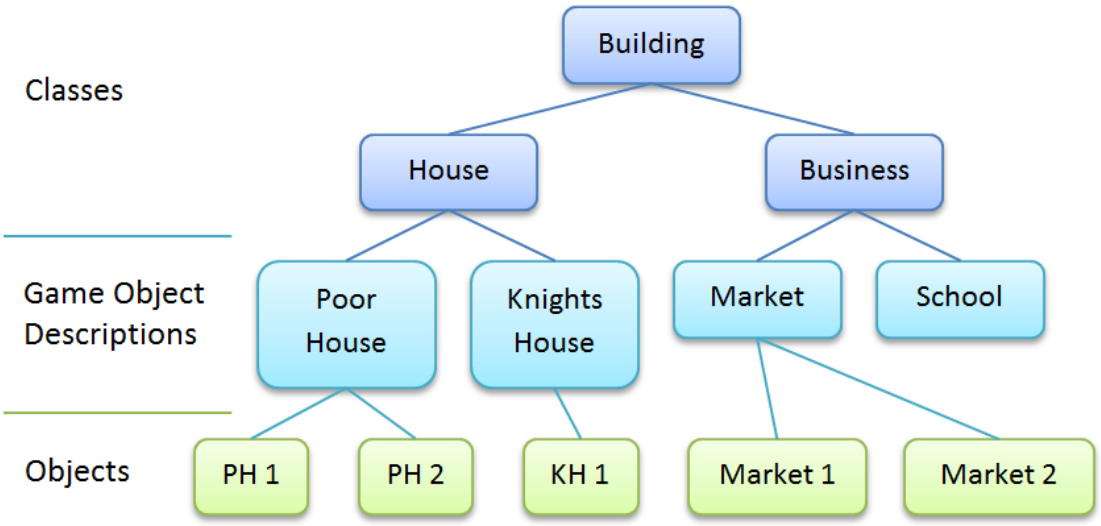


Figure 4.2.: The class Building is parent class of House and Business. Two different house types and two different business type have been described. Of these descriptions different amounts of object instances, in green, exist in a game state.

Game Attribute

All state defining values are contained in the attributes of the objects. Values are identified by name which is unique per object. Attributes may have individually changeable

values for every object or they can be shared among all objects of the same object description. These values are stored in an *Expression*.

Game Action

Game Actions are the link with which players interact with the game. Any action consists of conditions and consequences. All conditions have to be true for the action to be playable in this state. Once the action is played its consequences are applied one after another. All actions that the player may choose to play belong to an object and they may be referring to another object as well. Actions in the game are executed without any parameters, because this allows a universal interface for action selection. An action can be seen in Figure 4.1b and an example of actions referring to other objects is shown in Figure 4.3.

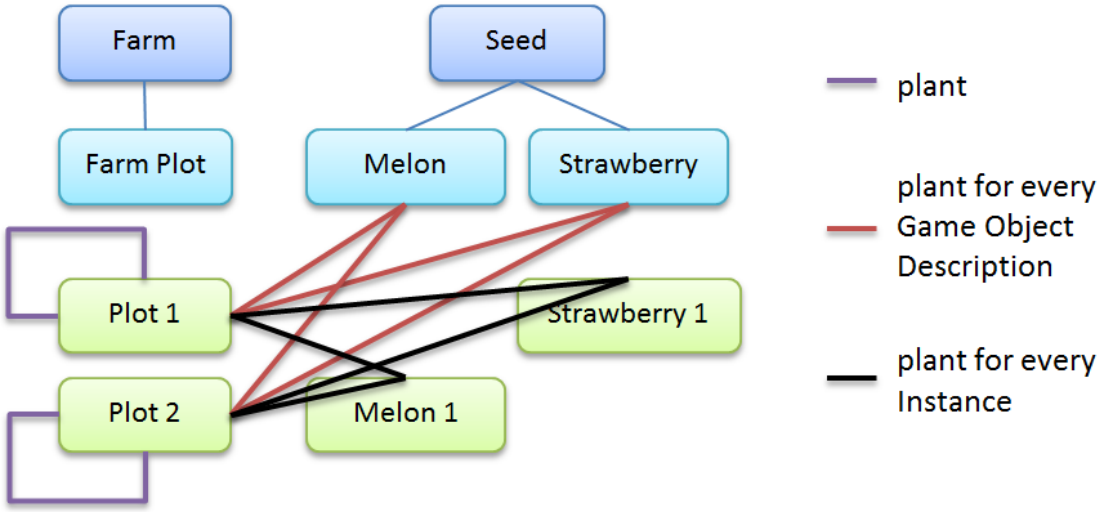


Figure 4.3.: The dark blue rectangles symbolize classes, the light blue object descriptions and the green object instances. This figure illustrates the difference between three actions.
The purple action is defined for class farm and every plot object has one plant action associated with it (symbolized with purple line to itself).
The red action is defined for class plot but associated with game object descriptions of class seed. Every plot object has one action for each light blue seed object description.
The black action is also defined for class but associated with instances of class seed. Therefore every plot object has one action for every green instance of seed. This could be more than one per game object description.
This system ensures that actions do not need arguments, because for every possible combination an individual action is created.

Rule

Game Actions can be used as rules. A rule is an action that can't be chosen by the player but is applied after every move, if its conditions are met. Rules are amongst other things

used to level up the player, when his experience points have reached the necessary value.

Condition

Conditions are logical terms returning a boolean value, i. e. true or false, when they are evaluated. They are used to restrict the applicability of actions. The logical terms are stored as an *Expression*.

Expression

Expressions are built in a tree structure, where every expression has a meaning by itself and up to two child nodes, a left and a right. There are different kinds of Expressions: *Comparators* are used to compare values and return a boolean result and *Operators* are used to combine and change values. Examples are addition ($+$) or assignment ($=$). *Atoms* are the leaves of the tree that contain a value. The available Atom types are *Long*, *Double*, *Boolean* and *String*. Special *Expressions* are *Object Reference* or *Attribute Reference*, used to point to other objects or their attributes and *Game Value*, used solely to access the global *Game Value Store*. All expressions types along with their use cases are listed in Table 4.1.

Expression Types	**Used for**
Atom	storing the basic values: *Boolean, Double, Long and String*
Comparator	comparing their left and right side: $==$, $!=$, $<$, $>$, $<=$, $>=$ or implementing boolean logic: *and, or, xor*
Operator	mathematical operations and assignment: $=$, $+$, $-$, $\times$, $\div$
Attribute Reference	accessing a game objects attribute during a game
Game Object Reference	accessing a game object during a game
Game Value	accessing the values stored in the Game Value Store

Table 4.1.: The Expression types explained

Consequence

Consequences are used to define effects of Actions. They are named to allow their reuse in different actions. *Direct Consequences* change an attributes value with an *Expression*. *Conditional Consequences* apply another consequence when their condition is met. This corresponds to an if clause in programming. *Timed Consequences* apply another consequence when some time has passed. This is used to create timers, e. g. this action can be executed again after 5 minutes. *Create Consequences* are used for dynamic object creation, creating a new object when they are applied. *Create And Assign Consequences* also create a new object and additionally assign a reference to it, that can later be accessed. *Game Consequences* alter the global *Game Value Store*. All consequence types are listed in Table 4.2.

Consequence Type	Used for
direct	changing the attributes of game objects
conditional	applying consequences only when their condition is met
timed	applying consequences when some time has passed
create	creating new game objects
create and assign	creating new game objects and assigning a reference to access it
game	changing the global Game Value Store

Table 4.2.: The Consequence types explained

Game Value Store

Many missions in social games are of the type *„collect this thing five times“*, or *„build that thing five times“*, or more general *„do this action five times“*. There are different options on how to track progress and completion of these missions:
An observer pattern could be used, where missions would register to be informed about any actions performed, that increase the progress towards their goals. Because this approach is difficult to formalize concisely for game descriptions and it would complicate the flow of information, another approach was used. Mission relevant actions are tracked by counters and their data is then used by the mission goals. Counters can already be described with the available game features, but they have to be manually created for every action and are are quite inflexible. The *Game Value Store* is therefore designed to allow more sophisticated counters.

It is a tree based structure, storing number values by name. Values of different depths can be accessed by their name and mathematical operations on sub-trees can be implemented. Currently a sum operation is used to sum up the values of all child nodes. The usage of the *Game Value Store* is explained in Figure 4.4.

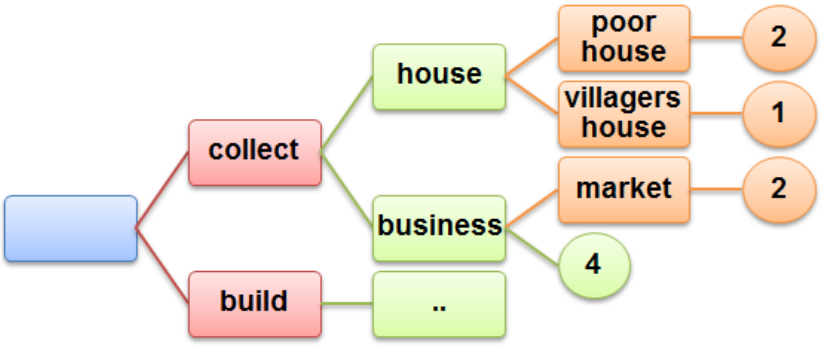


Figure 4.4.: An example of a Game Value Store. Every rectangle is a node, accessible through its name. It contains sub-nodes and a value, shown in a circle, of its own. Values for collect, house and build are not displayed. Some usage examples:
store.getValue(collect.house.poorhouse) = 2
store.getValue(collect.house.sum) = 3

Helper Features

The following features build upon the features already explained and provide additional information important for playing the game. They mark the start and end of the game and influence the way it is played.

Initial Game State The initial game state is defined by all the objects present at the time. In WGDL objects can be declared to be in the initial game state.

Terminal Game States/Goals Terminal game states define, when the game is over. In WGDL they consist of a condition, which has to be met for the game to finish. It is allowed to set more than one goal. Goals are then pursuit in order until the last goal is fulfilled.

Action Groups Actions that may be chosen as a move by the player are categorized in action groups, which have a name and a likelihood. When selecting random actions to play, a group is selected based on its probability and, of this group, one action is then chosen at random. This concept is explained in detail in Section 5.2.4.

4.2.2. Syntax Example Applications

Beside the question of what features have to be available to successfully describe social building simulation games, a decision on how to syntactically represent these features in the WGDL had to be made. The most logical choice was to build upon GDL, which is based on Datalog and uses KIF, a knowledge interchange format. While being very powerful and expressive, KIF is designed to allow machine readable knowledge interchange between computers, and therefore not designed to be easily human readable.

Another alternative is to use an XML notation, as XML is already widely used for configuration files and other web related content. There are also many XML-parsers available that would make implementing a game description parser much easier. But again XML is hard to read.

Until a graphical user interface, to describe games, is created and the game description language no longer exposed to the describing user, a human readable format has been chosen. This provided a challenge in writing a description file parser, which is able to parse these files.

In order to facilitate understanding the concepts and features mentioned above, this chapter contains some examples of WGDL's current syntax along with a definition in Extended Backus-Naur Form (EBNF). EBNF is a metasyntax designed to express context-free grammars. A full and continuous EBNF-definition of WGDL can be found in Appendix B.

Classes and Objects

The first example shows how game classes, objects and attributes are defined.

```
Syntax:
/* Classes have a name and may have another class as its parent. */
Class = "class", name, [":", className];

/* Object Descriptions are named and belong to a class. */
Object = "object", name, "is", className;

/* Class attributes are the same across all instances of one description.
    They may not be modified during the game.
This corresponds to constants in traditional programing languages. */
Attribute = "attribute", (("class", className | className), name,
    AttributeTypeWithValue | objectName, name, value);

AttributeTypeWithValue = Type, Value;
Type = "long" | "double" | "boolean" | "string" | "object:", className;
Value = String;
```

Algorithm 4.1 shows how two different kinds of buildings can be described. They use two kinds of contract mechanics. From one building type, the *house*, coins can be collected in regular intervals. In *businesses*, the second kind of building type, coins can also be collected, but the business has to be supplied with food, before the interval towards recollection is started. The game mechanics to use the information stored in these objects is not described in the example.

Algorithm 4.1 Class Examples
Building is the parent class for *house* and *business*, which inherit the attributes defined for *building*. For businesses two additional attributes (*isFed* and *foodCost*) are defined. Below that three objects of class *house* and *business* are described, their attributes initialized to some values, where the standard, set in the classes, is not supposed to be used.

```
class building

// buildings have a coin value and a recollection time that is the same
    for every object of that class
attribute class building coinValue long 0
attribute class building recollectAfterSeconds long 0

// every building object may be either collectable or not
attribute building isCollectable boolean true

// there are two types of buildings, houses and businesses
class house : building
class business : building

// businesses have additional attributes
attribute business isFed boolean true
attribute business foodCost long 0

// there are three building objects, two houses and one business
object poor_house is house
attribute poor_house coinValue 50
attribute poor_house recollectAfterSeconds 3600

object villagers_house is house
attribute villagers_house coinValue 15
attribute villagers_house recollectAfterSeconds 300

object market is business
attribute market coinValue 40
attribute market recollectAfterSeconds 46
attribute market foodCost 20
```

Actions and More

The second example shows actions with their conditions and consequences.

```
Syntax:
/* Actions can belong to a class or not.
If they do not belong to a class they are game rules the player con not
    select himself.
The third kind of actions is create actions, which are executed once a new
    object is created.
Actions can refer to other classes with the keywords forEveryInstance and
    forEveryGOD (GOD = Game Object Description). Then they either are
    created for every instance of the other class or for every object
    description. */
Action = "action", (name | className, name, [("forEveryInstance" |
    "forEveryGOD"), className] || "create:", className, name);

/* Conditions are directly associated with their action and have a boolean
    expression that tests whether the action is performable. */
Condition = "condition", actionName, Expression;

/* Consequences are first created and later assigned to one or more
    actions.
Different consequence types need different parameters. */
Consequence = "consequence", name, (("direct" | "create"), Expression |
    "createAndAssign", Expression, Expression | ("conditional" | "timed"),
    Expression, consequenceName | "game", "increase", Expression);

ConsequenceAssignment = "hasConsequence", actionName, consequenceName;
```

The example in Algorithm 4.2 shows two actions with their conditions and consequences. With *whackTree* the player can increase his wood stock but has to spend energy on the task. The second action *plant* is available for every farm in the game. It is used to plant different types of seeds, resulting in different cost, food returns and time until it can be harvested.

Algorithm 4.2 Actions Examples

The action *whackTree* can only be applied when the player has energy left. If applied one wood is added to the wood stock in the global object and one energy removed from the player.

The action *plant* is only available when the player has enough money, is in a high enough level and the farm is free. As a consequence the seeds produced food is stored in the farm, the seed has to be payed, the farm is marked as occupied and a counter is started to mark the farm as harvestable after the grow time defined in the seed object has passed.

```
// The whackTree action has one condition and two consequences
action globalClass whackTree
condition whackTree global.energy > long 0
consequence addWood direct global.wood = global.wood + long 1
consequence removeOneEnergy direct global.energy = global.energy - long 1
hasConsequence whackTree addWood
hasConsequence whackTree removeOneEnergy

// for every game object descripton of a seed, i.e. every seed type, a
    plantSeed action is created for every farm
action farm plantSeed forEveryGOD seed
condition plantSeed global.coins >= seed.coinCost
condition plantSeed seed.unlockLevel <= global.level
// this refers to the farm this action belongs to
condition plantSeed this.isFree
consequence addFood direct this.food = seed.foodProduced
consequence paySeed direct global.coins = global.coins - seed.coinCost
consequence farmIsOccupied direct this.isFree = boolean false
consequence farmMayBeHarvested direct this.isHarvestable = boolean true
// after the time defined in seed.growTime has passed the
    farmMayBeHarvested-consequence is applied
consequence farmMayBeHarvestedInTime timed seed.growTime farmMayBeHarvested
hasConsequence plantSeed addFood
hasConsequence plantSeed paySeed
hasConsequence plantSeed farmIsOccupied
hasConsequence plantSeed farmMayBeHarvestedInTime
```

Accessing Attributes and Objects

With WGDL references to objects can be kept as attributes and accessed during the game. The example Algorithm 4.3 shows how this is used in the game description.

```
Syntax:
/* To access attributes of objects the object is referred to.
This points to the object the action belongs to, objectName directly
    accesses a global object and className is used when an action is
    defined for all instances of a class or for multiple game object
    descriptions.
Attribute access can be stacked multiple levels. */
AttributeAccess = ("this" | objectName | className), ".", attributeName,
    {".". attributeName};
```

Algorithm 4.3 Accessing Attributes and Objects Example:
The *tree* has a *fruit* attached to it, that can be collected by using the *collectFood* action of the *tree*.

```
// objects of class tree have a fruit object associated with them
class tree
attribute tree fruit object:fruit null

class fruit
attribute fruit foodValue long 5
attribute fruit isCollectable boolean true

action tree collectFood
// tests if the associated fruit's status is collectable
condition collectFood this.fruit.isCollectable
// adds the food value of the food to the global food store
consequence addFood direct global.food = global.food + this.fruit.foodValue
hasConsequence collectFood addFood
// resets the food value of the trees food to zero
consequence setFoodValueToZero this.fruit.foodValue = long 0
hasConsequence collectFood setFoodValueToZero
```

Creating Objects

The next example shows the dynamic features of WGDL. There are consequences to create objects during a game and actions that are performed whenever an object is created. See Algorithm 4.4 for the example.

```
Syntax:
/* An action which is only applied when a new object is created. */
OnCreateAction = "action", "create:", className, name;

/* Two consequence types deal with creating objects.
Create consequence just creates a new object whereas the create and assign
    consequence saves a reference to the new object in an attribute. */
CreateConsequence = "consequence", name, ("create", Expression |
    "createAndAssign", Expression, Expression);
```

Algorithm 4.4 Creating Objects Example:
The build action is available for every building description that has been put in the game description. It needs coins to create a new building. When the *createBuilding* action is applied a new object is created from the description in building. The *counterUpdate* action is called whenever a building is created. Here it just increases the a counter in the *GameValueStore*. The counter is named *„building, built, objectName“*.

```
// one build action for every game object description of a building, or
    its subclasses business and house, is created.
action globalClass build forEveryGOD building
condition build global.coins > building.coinsCost

// The class name building is replaced by the name of the game object
    description this action is supposed to create.
consequence createBuilding create building
hasConsequence build createBuilding

// Whenever a building is created the counter is increased
action create:building counterUpdate
consequence increaseBuildingCreatedCounter game increase String building,
    String built, this.name
hasConsequence counterUpdate increaseBuildingCreatedCounter
```

Start and End

In addition to the game mechanics initial and terminal states have to be defined to create a finite game. The usage is shown in the example Algorithm 4.5.

```
Syntax:
/* Initials are objects that exist in the initial game state. Multiple
    instances of the same object description may be created. */
Initial = "initial", objectName

/* Goals are terminal conditions for the game or subgames. The algorithm
    starts with goal one and finishs with goal maxNumber. */
Goal = "goal", PositiveNaturalNumber, BooleanExpression
```

Algorithm 4.5 Start and End Example:
Poor_ house is defined as an object of the class *house*. In the initial game state the player will have two poor houses.
The game has two goals. First the player has to reach more than 1000 coins and then he has to gain another 1000 for a total of more than 2000 coins to end the game.

```
class house
attribute house name String none
object poor_house is house
attribute poor_house name poor_house

// two poor houses are created for the initial game state
initial poor_house
initial poor_house

// two subgoals have to be reached before the game is terminated
goal 1 global.coins > 1000
goal 2 global.coins > 2000
```

4.3. Implementation

While implementing the first WGDL system several design decisions had to be made. To improve the understanding of the implications of using WGDL to describe games, some important implementation features are described here.
The whole project has been implemented in Java, because the tool is supposed to be used on several platforms, the author is familiar with the language and Java provides the capability of writing the core system in the same language as the graphical user interface.

The game description module consists of several parts.

The Description The parser responsible to parse the game descriptions and the classes for descriptions of objects, actions and consequences are part of the description.

The Expressions It consists of the expressions, operators, comparators, references and atoms. They hold or provide access to values during a game.

The Game The main and high level features, like the game state, actions and objects belong to the game part.

Description File Parser

The Description File Parser has the task to create a Game Factory from a game description file. This Game Factory can then be used to create the initial game state and new objects throughout the game. The descriptions of different game features, e. g. conditions, consequences, actions, are read in sequentially and their representations in the system created based on a set of conditions, defined directly in the file parser. For a pseudo-code example see Algorithm 4.6. While parsing is a straightforward task for most features, the

parsing of expressions is more complex. In expressions brackets have to be observed and operators and comparators have to be applied in order.

$long\,2 + long\,5 * long\,7$ evaluates to $long\,37$ while $(long\,2 + long\,5) * long\,7$ evaluates to $long\,49$

Nested brackets are allowed as well and mathematical and logical operations are supported.

The order of operators and comparators is defined, from lowest to highest priority, as follows: $=$, $==$, $!=$, $\&\&$, $||$, xor, $>$, $\geq$, $<$, $\leq$, $+$, $-$, $\times$, $\div$

Algorithm 4.6 The file parsing algorithm in the Description File Parser.
First the file is read into memory, then the game features are created sequentially and the resulting Game Factory, initialized with these features, is returned.

```
GameFactory parseFile(){
        createNewGameFactory();
        readInFile();

        // create the game features sequentially
        createClasses();
        createAttributesForClasses();
        createObjects();
        createAttributesForObjects();
        createActionGroups();
        createActions();
        readInGameValues();
        createConditions();
        createConsequences();
        addConsequencesToActions();
        addActionsToGroups();
        createInitials();
        createGoals();

        return GameFactory;
}
```

Everything is an Expression

As previously mentioned an expression may consist of objects with different expression types. All objects of these types conform to the same interface as the expression itself. From an outside perspective there is therefore no difference between a part of an expression and a whole expression. This design pattern is known as Composite (see Chapter 2.3.2).

The common expression interface is minimal. All expressions allow access to a left and a right child, creating a tree structure of expressions. Expressions can be cloned,

hashed and compared for equality. And most importantly for playing games, they can be evaluated, returning an Atom which itself is an expression again. Some examples are listed in Figure 4.5.

Expressions are meant to be used during game play, storing and changing the games state. But attribute and object references have to be part of expressions before they are bound to in game instances of the attributes and objects they point to. To tackle this problem, description objects for these references have been introduced, which do conform to the expression interface as well, allowing them to be part of expressions. Before starting a game these descriptions have to be resolved to real instances.

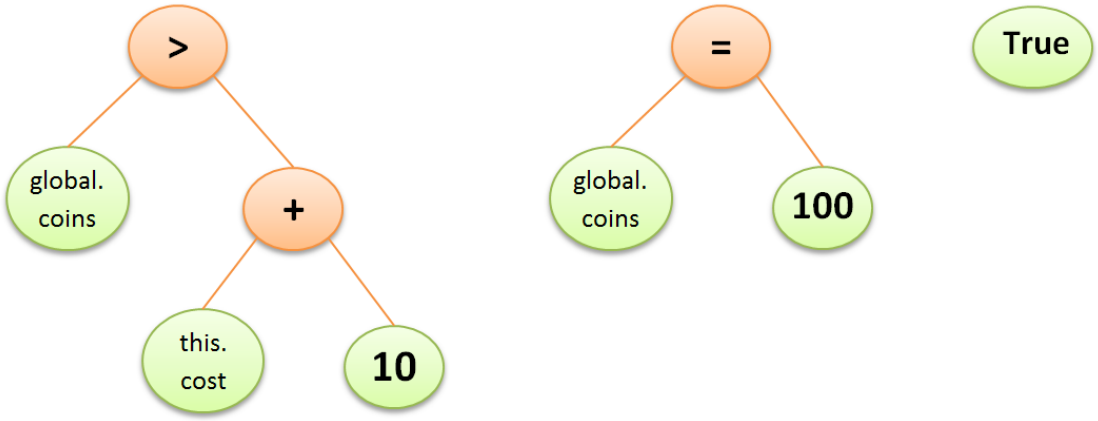


Figure 4.5.: This figure shows three expressions in a tree representation.
The left one uses a comparator to compare the attribute value of global.coins to the sum of this.cost and ten. This expression returns true or false once evaluated.
The second expression assigns the value 100 to the attribute global coins.
The third is an atom with the boolean value true.

From Description to Instance

In the Game Factory, descriptions of actions and objects are stored. While creating the initial game state and whenever a new game object is created, these descriptions are used as a blue-print for the object. As both game objects and game actions may contain expressions, in their attributes or their consequences and conditions, description expressions have to be resolved to in-game expressions. In these resolving step descriptions, like *this*, *object of class x* or the name of a *global object*, are replaced with the in-game name of the object they refer to. Attribute and game object references then act like proxies (explained in Chapter 2.3.3) throughout the game. When these expressions are evaluated, they access the other object and return the referenced value.

4.4. Results and Limitations

The WGDL in its current version is able to describe *finite, discrete, deterministic, non-zero-sum, symmetric, single-player games* with *complete information*. As shown in Chapter 4.1.2 this allows the description of simplified SBSGs. Before the limitations of WGDL are shown the results are compared against the goals and requirements defined in Chapter 4.1.1.

Results

Every goal is reviewed separately.

Human Readable The syntax is human readable, as it was supposed to be. But due to the large size of a game description the overview is easily lost. This could be improved by introducing a library system, allowing the inclusion of different files into one game description. Different parts of a game could then be more easily separated and the structure would improve understandability.

Short Some steps have been undertaken to limit game description size. Consequences can be reused and an inheritance system for classes avoids description duplication. But conditions can not yet be reused. The reuse of consequences comes at the cost of linking existing consequences to the actions and the initialization of attributes takes up quite a lot of space. It has to be further explored how shortness of descriptions can be improved without negative impact on readability. A logical consequence would be to introduce brackets or indentations to describe relations between game features.

Easily Create New Content As long as additional content can not be imported from Excel or an XML file, but has to be manually added, this goal can not be considered achieved. But the class and object description system creates the foundation, making this a mere technical task.

Easily Change Content This goal is tightly coupled with the one above. Changing content in Excel is not yet supported, but the changing of values in the game description itself is straight forward.

Reusable As WGDL is able to describe SBSGs the first part of this goal has been reached. The reusability of parts of game descriptions is more complex. Without a library system descriptions have to be copied and pasted to reuse them.

Describe Social Building Simulation Games As described at the beginning of this chapter, this goal is considered achieved with the limitations mentioned in the next chapter.

Limitations

While the overall goal of describing SBSGs has been achieved the expressiveness of WGDL has several limitations worth mentioning.

The biggest limitation to the use of WGDL is its focus on single-player games. This makes the comparison with other description languages more difficult and limits the number of games that can be described with WGDL. Because multi-player has been considered to be of low importance for SBSGs this will not be an issue when describing games.

Another limitation is that WGDL is at the moment of this writing not suited to describe non-deterministic effects. The influence of chance is different from game to game, but in general not that strong. For the purpose of simplification the effects of chance have to be modeled with the expected value, rather than real chance values. Actions returning a bonus of ten coins with the likelihood of 20 percent must therefore be describes as actions returning a bonus of two coins every time they are applied. When one wants to analyze lucky or unlucky players and their progress, the game description has therefore to be changed, reflecting higher or lower expectancies.

Figure 4.6.: Part of the game map in Magic Land. The tooltip shows that the decoration sunflower adds a bonus of 1% to the coins returned from the tavern next to it. The total coin bonus for the tavern from all decorations add up to 45%.

Because WGDL uses only comparatively simple data structures, where no kind of list or array is allowed, it is not suited to describe maps. When a map is only used to limit the number of objects in the game it can easily be replaced with a counter. But in addition to the limitation of space maps are also used to create area effects. Area effects are changes to one or more objects based on their location. In Magic Land a Wooga SBSG title described in appendix A, these area effects are used to give bonuses to the return of buildings and farm plots[2]. Decorations can be bought, increasing the return in their

[2]A screenshot of a part of the map in Magic Land can be seen in Figure 4.6.

vicinity by some percentage points. The effect of these area effects on the balancing can therefore not be analyzed with WGDL and if they are useful to the player, a simulated player who is not able to use them has a disadvantage. Especially in later stages these area effects become quite dominant in Magic Land, because it is possible to increase the return of a single building and therefore the return of using one energy point by some hundred percentage points.

To limit the complexity of game descriptions and because it has not proved to be essential in describing SBSGs, WGDL does not allow for loops or recursion. These features could be added, if they are required in the future.

The same holds true for the deletion of objects. Currently objects can never be deleted, but this could be implemented by defining a delete consequence similar to the create consequences currently available.

Conclusion

In this chapter WGDL, a language for describing SBSGs, has been introduced. It is based on the GDL and SGDL, improving on GDL with its human readability, dynamic object creation and the possibility to more easily add new content and improving on SGDL with its focus on single-player social games. It supports the description of contract mechanics, game rules and provides all the tools necessary for a mission system. With the Wooga Game Description Language the foundation for General Game Playing in Social Building Simulation Games has been set.

The next chapter explores how well Monte Carlo search algorithms are suited to play these games and gain valuable insights for game balancing.

5. Playing with Artificial Intelligence

In this chapter some modifications to the Monte Carlo Tree Search are discussed. The goal of these modifications is to more successfully play games described with the WGDL.

5.1. Overview

5.1.1. Requirements and Goals

The goal of every General Game Playing Agent is to play as well as possible. To be helpful for game balancing the algorithm and its results have to meet some additional requirements. The standard MCTS algorithm has been modified to meet these requirements and where several options already existed the ones being most promising to further the goals the most have been chosen.

Be Fast The performance of the analysis has been defined as one of the main project goals. A fast analysis allows fast feedback to configuration changes and this enables iterative configuration development.

Be Human Like As this tool is meant to predict human behavior and progress in the game, the AIs play should be as human-like as possible.

Be As Good As Humans This is closely linked to the last goal. In order for the results to be useful, the AI has to perform as well or better than humans. If this is not the case, it can not be used to predict how fast players progress or may only be helpful as a lower bound. But the goal is to predict the progress for users of various play styles.

Player Profiles These various play styles have to be supported in some kind of player profile. How long and often players play each day, as well as what game features they concentrate on, should be configurable.

5.1.2. Analyzing a Social Game

To be able to successfully play and analyze social games their peculiarities have to be known and addressed. Therefore some key features influencing the search algorithm are listed here.

As social games are designed to be played indefinitely a game will have many moves. In Magic Land, the game which is used as the basis for this thesis, there can be around one hundred actions, e. g. collect coins from a building, in the first session. Every other session then has around 30 to 60 actions. As the average player plays around three to four sessions a day, this adds up to approximately one thousand actions in the first week. In comparison, an average chess game ends after 80 moves. The search therefore has to be prepared for considerable longer games.

In the prototype's description of Magic Land the player has to choose between roughly 20 actions on every move. This increases over time as money and player level, as a measure for experience, allow the purchase of more buildings and decorations and, therefore, more existing buildings to interact with.

It is also worth noticing, that there will not be disproportionately better moves that would make games considerable shorter. This is different to chess and other two player games, where there might be wins in one move and game length may vary a lot. When user progression is measured in experience points the player's decision can only affect this in a small way. Most actions reward the player with one experience point.

But players can destroy what they have built in Magic Land and many other games. This decision is rarely a wise one. It might only make sense to destroy buildings when there is no more space or the building will never be used again but returns some coins upon being destroyed. So there clearly are very bad moves in social games.

Another noticeable feature of SBSGs are investments. Building an expensive building or buying anything expensive might seriously decrease the players coins which might be part of the final score evaluation. But these buildings increase his earning capacity and will return more than the investment, sometimes only after being used more than a hundred times. This way seemingly bad decisions might turn out to be good ones after playing many moves.

5.1.3. Setting Goals

Every search needs a goal it is looking for. As social games can't be won, this goal or terminal condition has to be set artificially. The goal is therefore set according to the needs of the user of the tool.

In Chapter 1.2.3 the core goals for balancing social games have been defined. They are:

Progression It is important to estimate the players progression, to set rewards where they create the most motivation or in order to know how much content has to be produced. One question could be: Where is the player, playing four five minute long sessions per day, after one week? Time or played sessions will therefore be a standard search parameter. The question could also be asked the other way around: How long does the player have to play, to achieve level twenty?

Balance Game Features Another important question is, how well the different game

features are balanced against each other. Human players will optimize their game play habits using only the features, that help them the most. These features are called strong, whereas clearly worse features are called, weak. To analyze which features are strong or weak the actions the player took to reach his goal have to be included in the search output.

Summarized, the product manager wants to know how far a player can progress in a given amount of time and which actions he will choose to reach this goal. The search algorithm's task is to produce a result that helps in answering this question.

5.2. Customizing the Playing Algorithm

Over the last few years MCTS has been an active field of study and many different expansions have been proposed. The most promising of these have been adapted for this thesis. This section will describe which methods are used and why they have been chosen. In the next chapter the run-time performance and their impact will be evaluated. For an easier understanding the methods have been split according to the step of the algorithm they influence the most.

5.2.1. General

The methods listed here affect the algorithm as a whole and can not be allocated to one specific step.

Winning and Loosing

At the end of each simulation the game state has to be evaluated. For SBSGs a win or loose evaluation is not very helpful. This is because a game is never really lost, but people differ in their progress.

The evaluation function is influenced by the terminal condition itself. If the terminal condition is a time, e. g. one day, that time period is a measure of player progression which would be interesting from a balancing perspective. Progress might be measured in experience points, coins earned, buildings bought, missions finished or anything else. If the terminal condition is a resource, e. g. reach 1000 coins, it might be of interest how fast this can be done. The result should then be higher, the less time is elapsed.

As MCTS aims to maximize the result and to gain balancing insights, the evaluation function has to return higher values for better game states according to specifically asked questions. The standard question this tool is supposed to answer is, how far can a player progress in a given amount of time. The terminal condition is therefore time based and the evaluation function has to reward progress. This is done by a configurable result calculator, which calculates the result with a given formula. Different game attributes

can be used as input and their impact on the result is set in percentage points. An example might be:

$$result = 50\,\% \times xp + 30\,\% \times coins + 20\,\% \times missionsFinished$$

Most MCTS policies are optimized for result values between zero and one. One could change this, but would have to recalibrate the values for different orders of magnitude in the result. The decision was therefore made to normalize all results to a value between zero and one. This is also important for every part of the formula. Else a higher value would have greater influence on the end result and the defined percentages would be overruled. The influence of every factor is determined by the limits used to normalize, wider limits decreasing the influence of a value, and by the percentage points the normalized result is weighted with.

The result calculator can now also be used to make the AIs behavior more human like and to define the concentration of a player on a specific game feature.

Divide and Conquer

One of the exceptional qualities of SBSGs are their enormous game play length as was discussed in Chapter 5.1.2. Playing long games creates a huge tree space, which is exponential to the length of a game. This prolongs the simulations because they have to play to the end. Selecting a node in a deep tree is also more expensive. Long games are therefore bad for the performance and make finding good results much less likely. To tackle this problem subgoals have been introduced to WGDL, effectively lowering the length of each search to reach a subgoal. Playing the game is then no longer done in one search run, but rather one MCTS is performed for every subgoal, using the result of the previous search as the input for the next run. The concept is depicted in Figure 5.1. It has some drawbacks worth mentioning:

Miss Best Result When a previous MCTS run for a subgoal does not return the best possible game state, the next run can not find the overall best possible result. It has effectively been cut from the search space. MCTS's promise to always find the best possible result, given enough time, is therefore no longer true for later search steps. When there is enough time within every step this problem does not exist.

Investments Are Difficult A subgoal is a point in the game when states are evaluated and a decision has to be made to determine which state is good and which is bad. It is sometimes quite difficult to decide if some investment made by the player will be useful in the long run. Maybe he bought an expensive item or object, which will grant him huge returns in the future and improve the result that is achieved after the last subgoal has been reached. Using subgoals to search might therefore encourage short term gains rather then long term planning. This could be counteracted by

introducing a value for economic power, i. e. the ability to generate coins, experience points and to finish missions, into the result formula.

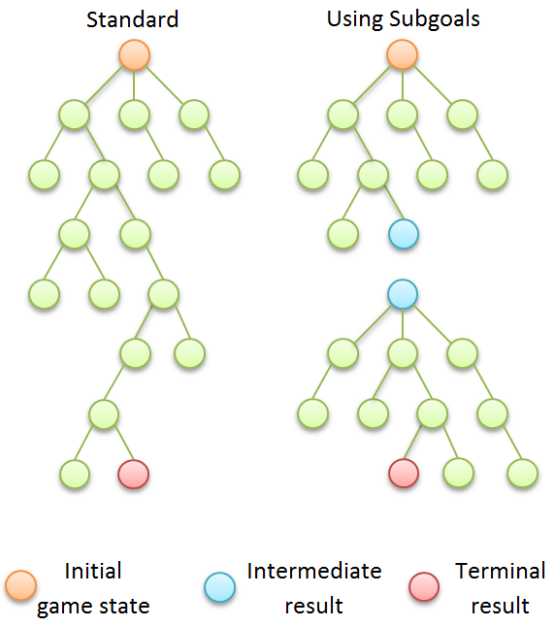


Figure 5.1.: This figure shows how the divide and conquer strategy of using subgoals changes MCTS by creating a new search tree from an intermediate result. The depth of every search tree is then less than with the standard MCTS approach.

Next the modifications to the four steps of MCTS, node selection, action expansion, game simulation and result value backpropagation are presented.

5.2.2. Selection

In the selection step a node of the existing tree, that needs further exploration, is selected. Different selection policies try to balance exploration and exploitation. They mostly differ only in the formula used to calculate a node's rank. Because of that it is easy to implement different algorithms and compare their results to find the one most suited for this problem.

Single Player Selection Algorithms

For comparison purposes different tree selection policies, all explained in Chapter 2.2.2, have been implemented. Standard UCB, UCB-Tuned 1, UCB-Tuned 2, SP-MCTS and UCB for SP, which is a new proposal for single player games. In single player games, results from games don't depend on good or bad play from the opponent, as there is none. This means that a good result can't be counteracted with a better move by the opponent, which has not yet been discovered. It therefore seems prudent to give the good results more weight on the selection. With UCB for SP a new formula, using the squared

results, is introduced to achieve this goal. It builds upon the UCB-Tuned 1 algorithm, which has been the most promising during development and changes the winrate to partly use the best result instead of the average. The formula is therefore:

$$y_j + C\sqrt{V_j \frac{\ln n}{n_j}}$$

with $V_j = \max(D, x_j(1 - x_j))$ and

$$y_j = \frac{t_j}{n_j}(1 - w) + w * b_j$$

with $t_j$being the total value of all results for this action, b_j being the best result for this action and w being a constant between one and zero. While the exploration part to the right remains the same as in UCB Tuned 1, the exploration part (y_j) balances the average result against the best result with constant w.

Non-Tree Tree Policy

The standard MCTS approach is to start at the root node and select it for expansion, if it is not yet fully expanded. If it is fully expanded the child node with the highest value from an UCB formula is chosen and the test for full expansion applied to this node. This process repeats until a node is selected for expansion or all nodes are fully expanded. Due to this exploration the current best path is not always chosen chosen but another path is explored. If a good result is then found from this path, the backpropagation ensures that the whole path is more likely to be exploited in the future. But even if the simulation found a new best path the changes to the tree structure would be quite slow. In an attempt to improve the reaction time on emerging new best paths, i. e. finding a surprisingly good result in an otherwise badly evaluated part of the tree, a non-tree tree policy has been introduced as an alternative. Instead of a tree search for the best unexpanded node, all nodes are considered with their UCB value. The policy is depicted in Figure 5.2. As all nodes, instead of only the nodes on the path and all the children on this path that must be considered, this approach takes considerably longer. For the first runs it may however be viable because the advantage of choosing the current best node instead of choosing the current best path outweighs the performance issues. The evaluation in Chapter 6.2.2 proves that this approach is successful.

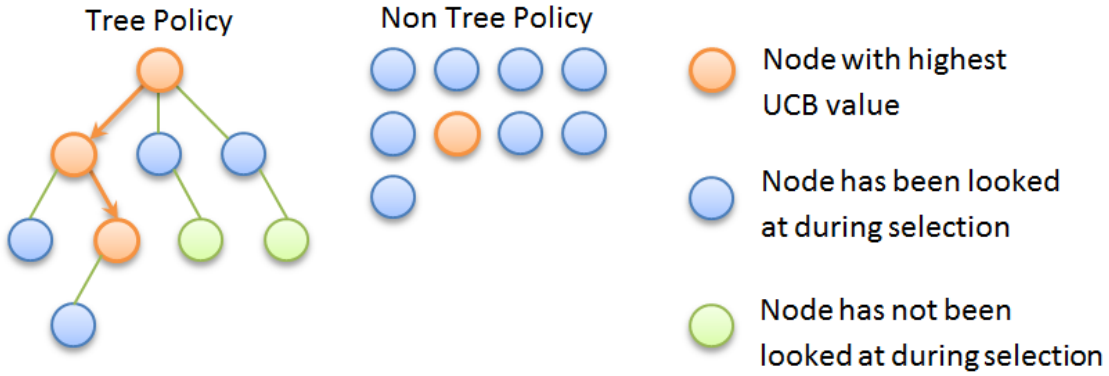


Figure 5.2.: A comparison between two selection policies.
The tree selection policy starts add the root node, looks at all children and continues this process on the node with the highest UCB value. Several nodes will not be looked at during one selection process.
In the non tree selection policy all nodes are looked at, no matter their position in the tree.

5.2.3. Expansion

In the expansion step one or more different moves are selected for expansion from the node returned in the previous step. For every expanded action a new node is created and added to the tree. To improve the performance and the quality of the results some adjustments have been made.

Many Paths, One Goal

Although games are represented in a game tree, this representation is somewhat flawed. It does not account for different paths leading to the same game situation. This can just be ignored, but using this information could be beneficial to the search. Combining different paths that lead to the same game state can save memory, when one node is used to represent the game state that is reached in two ways. This can also be used to evaluate this game state with the results of the random playouts from both paths. To avoid the introduction of a graph rather then a tree, the algorithm tests if the new game state is already in the tree. If a similar game state is already is found no new node is created and no playout performed. This avoids a considerable number of multiple game states, resulting in more efficient memory usage.

Start Small, Increase Steadily

Due to the tree policy, selecting every node on the path, which is not already fully expanded, the tree widens very early. If the branching factor is not very small the wide tree prevents the tree expansion into deeper regions. As a result a wide and shallow search tree is built. But as the player in a single player game wants to search one time

before his first move and then play to the end, without further searching, the whole path to the terminal state should be in the game tree. One way to improve the depth of a search is to artificially limit the branching factor, as is shown in Figure 6.8. Out of all possible actions at one point only a small amount may be expanded. When a sufficiently deep game tree is built with this limited branching factor, the limit may be gradually increased. This technique is called *progressive widening*. The prototype has the option to start with a branching factor limit, which is expanded whenever the tree is fully searched with this limit.

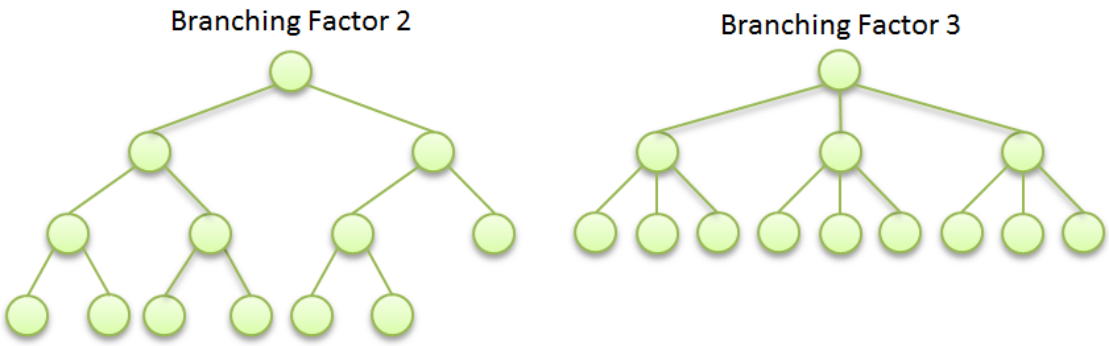


Figure 5.3.: The two trees both consist of thirteen nodes, but due to its smaller branching factor the left one is one level deeper.

Narrow Trees Grow Higher

When playing games with high branching factor (bf) for a short time, progressive widening may not be enough to allow tree growth to the terminal node. While it would be possible to start with a bf of one, the resulting path would hardly be a search tree. Then the bf would have to be increased to two. But even with this small value the number of nodes in depth n could be up to 2^n. The tree could therefore have *1048576* nodes in depth *20*. When the program is only searching for a short time and expanding the tree evenly this would produce low and wide trees. To improve growth towards the height this thesis proposes an additional limit to the number of nodes on one level. When this limit is reached, nodes on the level below may no longer be expanded. This leads to more narrow and high trees as illustrated in Figure 5.4.

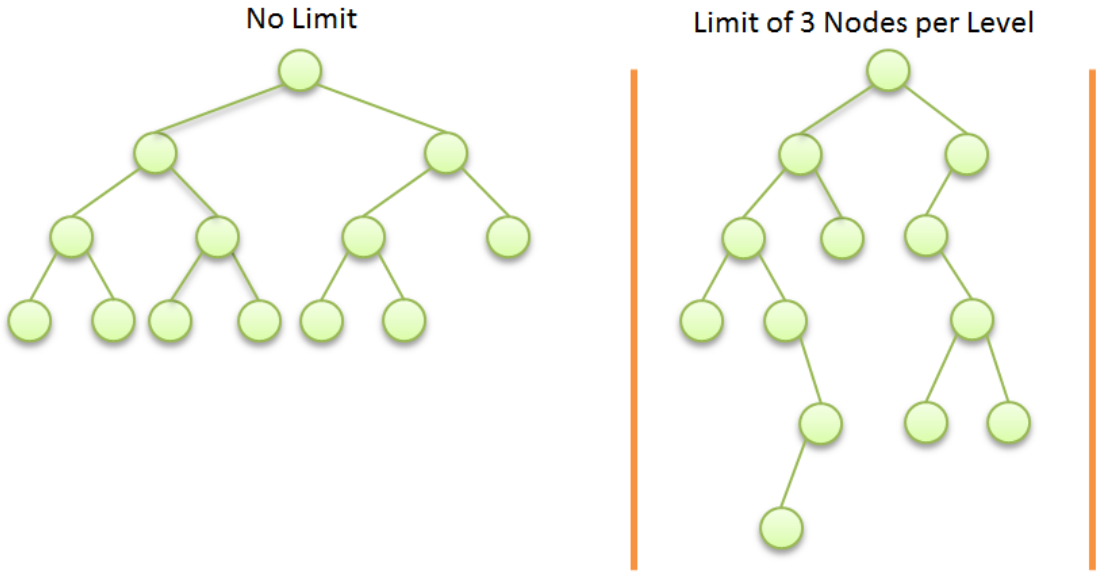


Figure 5.4.: Both trees have a branching factor limit of two, but the right one is additionally limited to three nodes per level. With the same amount of nodes the tree is two levels deeper.

The limit is set dynamically based on the total number of nodes the algorithm expects to be in the tree when the search has finished and the length of the games.

$$limit = \frac{estimatedNodes}{gameLength} \times x$$

The length of a game is taken from the current best result and the factor x may be used to soften or harden this rule.

Another approach would have been to use different branching factors for nodes on odd and even levels. This has not been implemented and tested.

Do Not Forget The Best Path

As previously explained, AIs playing single player games have a great advantage over AIs playing multi-player games. Once a single player AI has found a good path to the terminal state there is no opponent that can prevent the AI from taking this path. This leads to the comfortable position of every result achieved via one path being possible, without any outside help like a badly playing opponent. It is therefore only natural to save the best path found so far, even if not all the nodes on this way are yet in the tree. This grants the additional bonus of a fast reply from the algorithm. If the algorithm is asked to return the sequence of moves it wants to play, before the search tree reaches a terminal state, it normally can not replay any path to the end. When the best path is saved, this is no longer true. The prototype has the option to save the current best path, by adding all intermediate states with nodes to the tree.

5.2.4. Simulation

In the simulation step, the game is played from the newly expanded game state to the end. Which moves are chosen until the end of the game is decided by the simulation policy. The most basic simulation policy is to always choose a random move.

Action Groups

Randomly choosing between all moves has some serious disadvantages. The number of moves of one kind define how likely the player is to choose a move of this kind. If, for example, there are forty different things to buy and only ten other actions, like collecting coins from buildings, the likelihood of choosing a buy action is 80 %. If there are some bad moves, or moves the player should only do in some exceptional cases, this may become a major problem. In SBSGs buying a new building is one such expensive action. When the likelihood of choosing a bad or expensive move is too high the chance of finding a good path fast is very low.

To solve these problems the concept of action groups has been introduced to WGDL. See Figure 5.5 for an illustration. All actions are sorted into groups and these groups each have a likelihood to be chosen. Inside the action group the actions are then chosen randomly with equal likelihood. Thus the number of actions in one group has no effect on the likelihoods of actions in other groups being chosen. This concept is also helpful in creating player profiles. This has been identified as a goal in Chapter 5.1.1. By changing the likelihoods of the action groups preferences of players in choosing their actions can be modeled. The drawback of this approach is however, that it introduces human knowledge about the different actions into the game. If none is available all actions have to belong to the same group, creating the same result as if there was only random action picking.

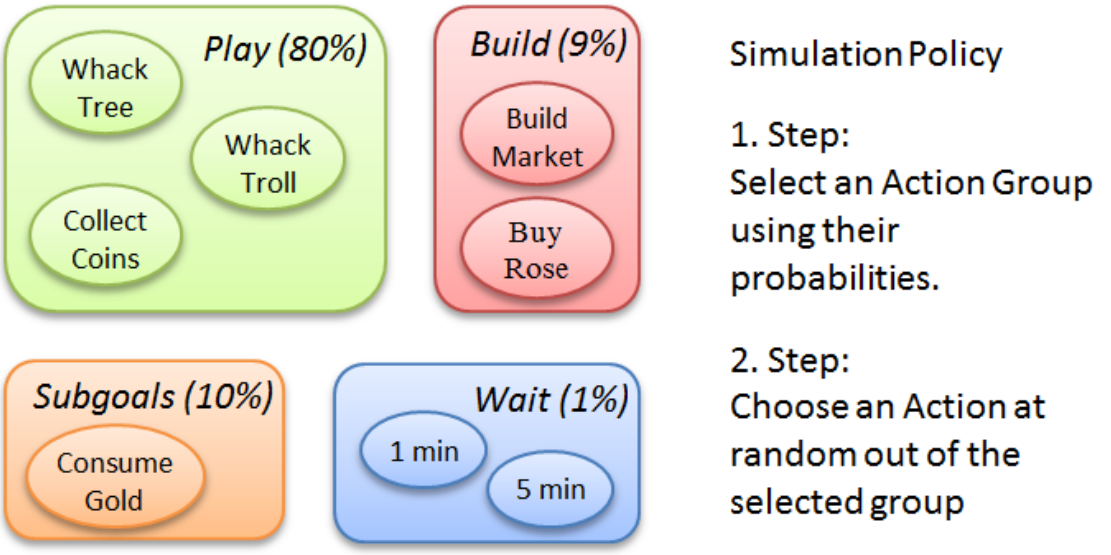


Figure 5.5.: This illustration shows the four action groups used in the Magic Land game description with some example actions in them.

Forfeiting Hopeless Games

Most of the processing time of the algorithm is spent during the playouts. Avoiding unnecessary steps can therefore be helpful in increasing performance. To abort games, that can not return a good result, before a terminal state has been reached, a method for evaluating the the current state must be implemented. Evaluating a game state via a heuristic is not possible in GGP and this is why MCTS is used for this kind of problem. The only time a game state is evaluated is when it is terminal. Then the scoring function returns a result, that has been normalized to be between zero and one. In some cases the potential of a game state to return a certain result can be analyzed. If the result is calculated by the amount of game time needed to reach a goal, and needing less time is better, the time already spent in this game state is the best result that can possibly be reached. When the value(s) used to calculate the score are continuously increasing or continuously decreasing the best possible result of a game state can be calculated. During the simulation this value can be tracked and the simulation stopped, when the value reaches a lower bound. When a game's best possible result, multiplied with a constant set to two in the prototype, is lower than the best result currently found, the game is considered lost, stopped and evaluated to zero. This evaluation to zero, even if the real result might still have been 0.2, is an aggressive approach undervaluing bad moves.

5.2.5. Backpropagation

During the backpropagation step the result obtained from the simulation is propagated to all nodes on the path to the newly created node. To this straightforward step no

significant modifications have been made.

5.2.6. Parallelization and Consecutiveness

One of the major goals of the search was to return results very fast. Parallelization is a comparatively easy way to improve performance.

Root Parallelization and Tree Parallelization

For testing purposes and to accommodate the best results in different computing environments, two parallelization schemes have been implemented in the prototype. They were both already explained in Figure 2.10. In the Combined Search Algorithm, multiple Monte Carlo instances can be run in parallel. This approach is called *root parallelization*. In addition to that, every Monte Carlo instance can also have multiple worker threads. These threads access the tree, using a global mutex and run their independent simulations. This is called *tree parallelization*.

Both approaches combined allow to focus on a single tree, if time is of the essence, but when the locking approach no longer scales to new threads or multiple attempts are more important, several search trees can be built independently.

Divide the Time

Especially due to progressive widening, which limits the branching factor by not expanding all possible actions, some search attempts may be doomed to a bad result quite early. If the first actions chosen prohibit a good result even a long search might not be able to overcome this. When using short search times, progressive widening does not happen very often, because the tree does not get fully expanded. To approach this problem the idea was born to increase the number of searches in the given time, rather then search for a longer time. So instead of using the sixty seconds dedicated to a search run, two search runs of thirty seconds or three of twenty seconds could be performed. This approach of consecutive search runs has been implemented in the prototype as well. The hope is to decrease deviation in the results producing more stable ones.

5.3. Implementation

The search algorithm's implementation consist of two major parts. They are the algorithm itself and the definition of the tree structure, which is used during the search.

As the Monte Carlo Search Tree algorithm has been heavily researched over the last years, many modifications have been proposed. Most of the time it can not be determined how well they perform in a specific domain they have not yet been tested for. To get the best possible result for searching SBSGs, different modifications had to be tested against each other. To support this, the algorithm has been implemented in an easily configurable way.

With *Search Algorithm*, an interface to access the search algorithm has been created. As the main part, a MCTS algorithm has been implemented and for testing purposes an NMCS algorithm has been created too. A meta algorithm called *Combined Algorithm* has been implemented as well, combining multiple MCTS and NMCS in one algorithm.

The Search Method

The different search algorithms can all be started with the same search method already defined in *Search Algorithm*. It is marked with the key word, „final“, to disallow overriding it. It defines the basic setup for all search algorithms, how they start and what is done with their result. But it does not know how to search for the best way to play the game. For this, the search method then calls a template method (see Chapter 2.3.4), which has to be implemented by the subclasses.

In fact there are two template methods, called *searchTimes* and *searchMilliseconds*. They differ in their termination condition. *SearchTimes* stops after a given amount of search runs has been performed, whereas *searchMilliseconds* stops after a given amount of time has passed.

Detecting Multiple Game States

As described in Chapter 5.2.3 sometimes different paths lead to the same game state. To avoid creating a graph rather then a tree, these different paths are not joined together to reach the same node which holds their common game state. But as multiple game states may fill up the search tree, a way to detect and avoid them had to be introduced. Because game states get quite big a full comparison is not efficient.

This problem can be solved with the introduction of a hash function, converting the state into one single value, which is the same, when the state is the same. To create this game state hash, all changeable parts, e. g. object attributes, have to be included. These hashes are then stored in a hash map that grants constant access times with the hash as a key, and are currently compared on a per level basis. Only game states reached with the same amount of actions in a different order are therefore compared.

Creating Nodes

The tree nodes are used to store search related information about the game state they represent. How often has it been visited, what results have been obtained from playing through this nodes game state, how many children does it have and what is its parent. This information may then be used to calculate the upper confidence bound, a value representing a combination of the result, that is expected to be obtained from playing the action leading to this game state and an exploitation value that favors unexplored nodes to broaden the search. There are many different formulas for calculating this value.

To allow the usage of different calculation methods that can be accessed with the same interface, therefore making the rest of the algorithm ignorant of what kind of formula is used, the abstract factory pattern has been used. For every formula a specific node factory has been written, implementing the abstract factory interface. The algorithm itself uses the abstract factories interface to create nodes. By changing the factory object the algorithm creates different kinds of nodes, using different upper confidence bound calculations.

Improving Node Size

Early tests showed, that the search tree took up too much memory. Because every single node was too big, the maximum amount of nodes and runs was very low. An analysis of the memory consumption with VirtualVM showed, that the biggest object inside the tree node was the game state. To solve this problem the game state was no longer stored in the node. When expanding a node and playing from the state that has been reached before the game state is now recalculated from the initial game state following the actions on the path through the tree. This decision has decreased search speed and memory consumption, thus allowing for larger trees but needing more time to build them. Tree nodes without stored game states are two to three orders of magnitude smaller than nodes with stored game states.

Store Upper Confidence Bound Values

As the upper confidence bound values of a node, calculated with UCB, SP-MCTS or any other UCB formula shown in Chapter 2.2.2, may depend on its parent and visits to itself, the value may change whenever the parent node or the node itself is visited. This dynamic value could therefore just be calculated whenever it is requested during the selection phase. But most formulas use square roots, logarithms or other computationally expensive methods. To increase the performance, the value is therefore not computed on request but stored and recalculated whenever it might change, due to visits to the parent node or the node itself. This approach can not be used when a random value is added to avoid tie situations, which has been proposed and used in implementations before.

These approaches could be combined by dynamically adding a random value to the precalculated upper confidence bound value.

5.4. Results and Limitations

The modifications described in this chapter are supposed to improve the usefulness of MCTS in the problem domain. Before the performance of these modifications will be evaluated in the next chapter - measured in search speed and the quality of the results they produce - the results and limitations of MCTS with these settings and changes are weighed.

Results

The results are again mainly measured by how well the goals set in Chapter 5.1.1 have been met.

Be Fast The evaluation in the next chapter will show that MCTS in this implementation performs fast enough to be used iteratively.

Be Human Like What kind of actions the player takes, and therefore whether he plays human-like or not depends mainly on the result formula that is set. It was possible to steer the prototype towards finishing missions, what is believed to be human-like playing behavior. But analysis of real user data has also shown that there is not one human-like playing style, but that humans play games rather differently. Without a baseline defining human game play this goal can not accurately be evaluated.

Be As Good As Humans When the measure for playing as well as humans is progression with experience, the AI is able to play well as as humans and better than most for short game length. For long games it is not yet on level with human players. A detailed analysis of the AIs performance, regarding its results, can be found in the next chapter.

Player Profiles With different session durations and pauses between these sessions, specific player profiles can be modeled. The playing styles, meaning what kind of features a player concentrates on, can be described by changing the likeliness of action groups and more strongly by configuring the result formula to reward whatever the player should concentrate on.

Limitations

To improve the performance of MCTS some concepts have been introduced that are domain specific and may even violate GGP principles.

One such modification is the inclusion of action groups. While combining different actions in one group and increasing or decreasing their likelihood to be chosen increases the algorithms performance, it also introduces human knowledge. These assumptions about what actions are helpful to the player and how often they should be tried out might not be available or worse, even be wrong. Action groups do change the probability of individual actions. But because all of them will eventually be chosen, this does not change MCTS fundamentally. All actions combinations are still tried out and the best result will be found, given infinite time and memory.

Another strong force in guiding the MCTS and effectively changing the end result is the result formula. While it is necessary to define how to evaluate a terminal state returning a score, sophisticated result formulas can exceed this goal by rewarding actions the user deems beneficial to achieve a good result. For example the formula could incorporate coins, food and finished missions, in addition to the experience points which would be enough to track progression on one axis. The main disadvantage may be that this feature might allow the user to incorporate wrong assumptions about user behavior, skewing the results.

The third limitation worth mentioning is the introduction of subgoals to divide the game into several smaller games, effectively decreasing the tree space one search has to cope with. This approach makes result formulas that reward more than plain progression mandatory, because these formulas now have to evaluate game states that are not terminal when looking at the whole game. If the division of a game is possible and appropriate evaluation functions can be found subgoals are worth the effort as the next chapter will show.

Conclusion

This chapter has shown that MCTS is suitable to play social games, given some modifications and enhancements. With the result formula a new feature to calculate a games score has been introduced. To successfully play SBSGs the game length can be cut down into several sessions, the path to the best result can be added to the tree and tree height growth can be forced by artificially limiting the branching factor and the maximum tree width. Human behavior and different play styles can be modeled, thereby losing some generality. The next chapter will evaluate the performance and the results, using different modifications and settings.

6. Experiments and Evaluation

In this chapter the search algorithms are evaluated in different environments and with different settings. The measuring results can be used to propose which settings should be used to achieve the best and/or fastest results. They will also show, if the approach of playing social games with MCTS is viable for game balancing and where further improvements should be considered.

6.1. The Environment

For testing purposes two games have been described. With Magic Land (ML) the prototype can be tested on a real social game, while Simple Coins may help to evaluate the algorithms performance in a fully calculable test environment. All test runs have been made on standard personal computer hardware, that is at approximately the power game balancers have at their disposal.

6.1.1. The Games

With Magic Land, Wooga's most popular energy based SBSG has been used as a test case. But the number of actions possible and their combinations and timed effects make it nearly impossible to calculate, how well the game can be played. If this would not be the case a balancing tool would not be needed. The results in Magic Land are therefore compared against data gathered from real users. To compare search results against theoretical maximums a second game description, with actions that clearly differ in their value but can be calculated has been created.

Magic Land

Wooga's Magic Land (ML) is, mechanics wise, a typical example of modern social games. The game play is paced with an energy system, the player can start contracts on farm plots and buildings and a soft as well as a hard currency is used. The soft currency (coins) is used to buy buildings, contracts and decorations, while the hard currency (diamonds) may be used to speed up the game by fulfilling tasks, unlocking items prematurely or buying special items. The story is told through missions, which are the main drivers for players actions. A more detailed description can be found in appendix A.

Because the prototypes performance on ML has been tested against real user data it is especially important to clarify where game description and the actual game are the

same and where not. Some simplifications for the ML description had to be made due to the lack of expressive power of WGDL, others have been made to ease description or even improve playing performance.

Comparing Magic Land Against the Description **Features, that are in the game description:**

- Buildings: Businesses and Houses
- Farms: With different contracts
- Some Decorations
- Non Player Characters (npcs): Trolls and Unicorns
- Missions: Several mission lines, 22 missions in all
- Coins, food, stones, trees and basic materials
- XP and leveling up

Features, that are not in the game description or different to the game:

- Area effects based on map positions and other map consequences, like limiting the number of buildings
- Hard currency (diamonds)
- Most advanced materials, which can be obtained from buildings and are only needed in later missions
- Friends, gifts and requests
- Chance events are either left out (random dropping of energy) or implemented with their expectancy (return of basic materials from houses)
- More missions missing
- Food cap that can be raised by buying special buildings
- Whacking npcs, stones and trees does normally take several actions but is modeled as one action consuming several energy points and returning several materials

The ML description currently consists of 598 lines describing game mechanics and 579 lines describing content. The descriptions of classes and actions with their conditions and consequences are counted as game mechanics, while object descriptions, their attribute initializations and the terminal conditions are counted as content. A more detailed listing of the line distribution is available in table 6.1.

Game Mechanics		Game Content	
class	24	object	127
attribute	111	attribute	431
action	50	initial	17
condition	69	goal	4
consequence	153		
hasConsequence	162		
isInGroup	25		

Table 6.1.: This table lists the number of lines dedicated to different game features in the game description.

Simple Coins

In Simple Coins (SC) the player has to choose between twenty-one different actions, twenty of which return between one and twenty coins while the twenty-oneth ends the session. The coin getting actions need one energy and the end session action refills the energy to its initial value of 30.

The best result of one session is obtained by choosing the best action, get twenty coins, every time and ending the session only after the energy is depleted. This would result in a best result for one session of $30 \times 20 = 600$ coins, while the worst result would be zero, when the first action chosen is to end the session prematurely. This game does not use any timed effects, game rules or other social game specific features, but its optimal results can easily be calculated. Its game description can be viewed in Appendix D.

6.1.2. The Benchmarking System

All benchmarks mentioned in this chapter were performed on an up-to-date personal computer with the following specifications:

CPU Intel Core i5-3450 - 4 × 3,100 MHz

Memory 8 GB DDR3 RAM, 1,333MHz, CL9

Hard Disk Samsung 830 Solid State Drive, 128 GB

Operating System Microsoft Windows 7 Professional, 64 Bit

Editor Eclipse IDE Indigo

Java Version JRE 7, JDK 1.7.0_03

6.2. Experiments

The different settings and modifications are evaluated separately. They are assessed based on their speed, measured in runs per second, and the quality of the results they produce. Speed is only a secondary metric, based on the assumption that faster search speed will provide better results. The quality of the results is measured in the absolute value of the results, higher being better.

All the modifications are explained shortly but can be found in full detail and same order in the previous chapter.

6.2.1. General

Result Formula

The result formula used to evaluate terminal game state with a value between zero and one is the main force determining how the AI plays a game. It is therefore important to understand how different result formulas influence the result of a search.

To map all parts of the formula linearly to a value between zero and one a minimum and maximum value is set for every part. The value can then be calculated with the expression $\frac{realValue-min}{max-min}$, if a bigger real value is better, or $\frac{realValue-min}{max-min} \times (-1) + 1$, if a smaller value is better. The minimum and maximum value have to be manually set and may be estimated or tuned wit a trial and error approach. The first experiment (E1) tests if it is important to set these values as narrow as possible.

Experiment 1: Result Formula Limits In E1 four different result formulas are compared against each other. The minimum values are always the same, zero, but the maximum values are doubled for every new formula, making the limits less and less accurate. The standard formula n is $\frac{xp-0}{500-0} \times 0.5 + \frac{missionsFinished-0}{22-0} \times 0.3 + \frac{coins-0}{15000-0} \times 0.2$. The maximum value for xp is 500, 22 for finished missions and 15000 for the number of coins collected. The formulas are tested on ML playing the first four sessions, one session at a time.

The results in Figure 6.1 show, that matching result formula limits are of importance to the end result. The best fitting result formula n returns the best result, and results get worse the more widely the limits are off the mark. The data supports the claim that n is better than $2n$ with 95% confidence and better than $4n$ and $8n$ with 99,5% confidence[1].

[1] The significance levels of the experiments are calculated by comparing the two samples using the Student's t-test with a one-tailed distribution. This approach is used by Wooga to evaluate A/B tests of new game features.

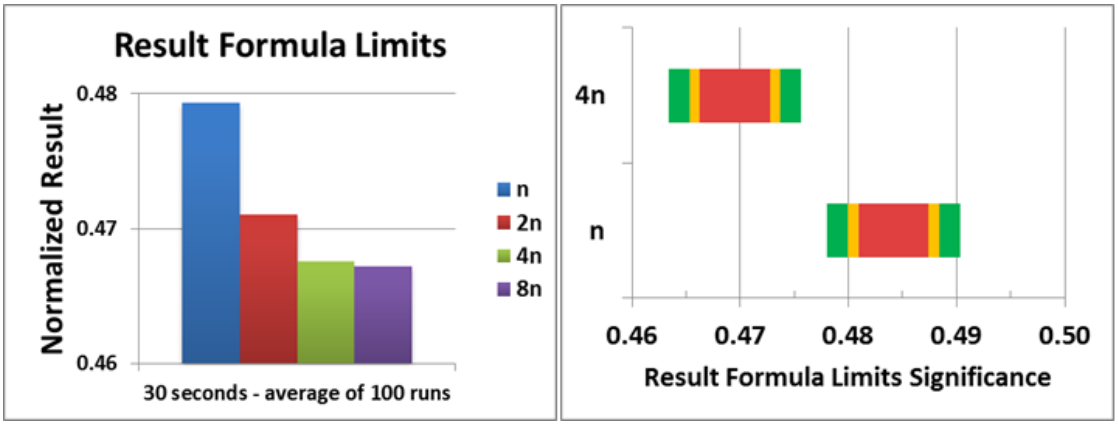


Figure 6.1.: The left chart shows the average result achieved using the four different result formulas. To allow comparison all results have been recalculated from the xp, missions and coins values using the standard result formula. To make differences more visible the lower limit of the graph has been set to 0.46 instead of zero.
Averages:
$n = 0.479328206$
$2n = 0.471085758$
$4n = 0.467560861$
$8n = 0.467253782$

The right chart shows the distribution around the means for n and $4n$. The significance of the result is the highest value of the bars of one color which do not overlap. Green is 99% significance, yellow 95% and red 90%. In this case both bars do not overlap and therefore n is better than 4n with a significance level of at least 99%.

It is now clear that the result limits do matter. Currently the same formula is used for all subgoals and has therefore to be initialized with values viable for all subgoals. These results suggest, that results could be improved further by using better fitted result formulas for each subgoal individually. So the xp limit could be set to 150 for the first session (first subgoal), be increased to 200 for the next and so on.

Experiment 2: Factors in Result Formulas Experiment 2 (E2) tests how the composition of result formulas affects the game result. Result formulas can be comprised of different game values, which may be limited individually and their influence weighed against each other. To measure the influence of weighing factors, two different weighing schemes are compared. In xpMain the main focus, 50%, is on gathering experience point, while in coinsMain, collecting coins is the main target, 50% as well. To evaluate the importance of combining different values in one result formula the combined formulas compete with three single value ones. XpOnly, missionsOnly and coinsOnly only take the corresponding value into account, completely ignoring everything else.

The results are shown in Figure 6.2. When comparing xpMain against coinsMain their

different focuses are clearly visible. XpMain returns the most XP while coinsMain returns nearly as much coins as coinsOnly. Although the weight and limit for missions is the same in xpMain and coinsMain, xpMain performs a lot better on this. This can be explained by the nature of the missions. Completing missions often costs coins, e. g. buying decorations or flat-out consuming coins, and is sometimes rewarded with XP.

More can be learned from looking at the single value formulas. XpOnly performs worse than both combined formulas on its prime focus. MissionsOnly performs a lot worse than xpMain on its focus, missions, as well. This shows, that concentrating on one value only may not always be the best strategy, when one is trying to optimize this value. The most likely explanation for these results is, that both strategies waste their coins early on, and this affects their ability to improve their results by finishing missions. Their exceptionally low values in coins supports this thesis. CoinsOnly differs from the other single value formulas in that it actually succeeds in optimizing its value more than the combined formulas do.

This experiment has shown, that different result formulas lead to different playing styles and that a wise combination of values can, but must not, be more effective in optimizing even a single value.

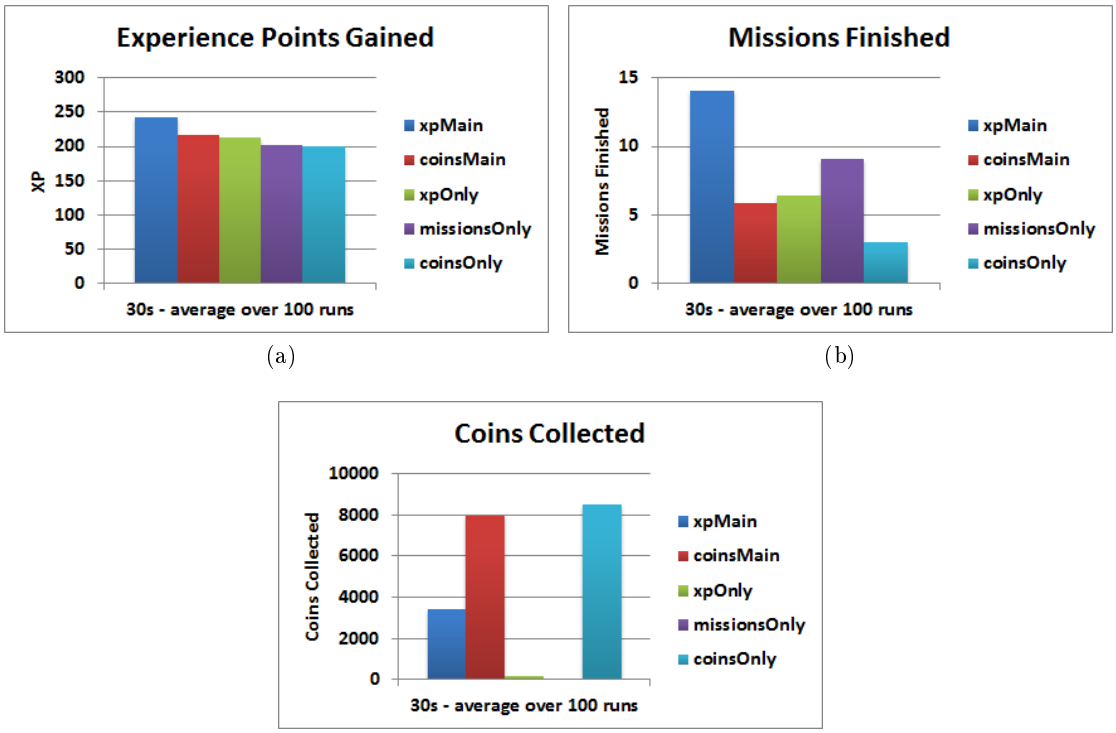


Figure 6.2.: These graphs show the average values for xp, finished missions and coins, using five different result formulas:
xpMain = 50% xp + 30% missions + 20% coins
coinsMain = 20% xp + 30% missions + 50% coins
xpOnly = 100% xp
missionsOnly = 100% missions
coinsOnly = 100% coins

Divide and Conquer

One of the main modifications, while adjusting MCTS to succeed in playing SBSGs, is the introduction of multiple subgoals to limit the depth of a search tree. The effects of this decision are tested in Experiment 3.

Experiment 3: Using Subgoals to Improve Results Three different settings are tested against each other. Completing the four sessions directly (1×4 sessions), with two goals (2×2 sessions) and with four goals (4×1 session).

The results of E3 are abundantly clear. Using one subgoal for every session, rather then playing four sessions in one run, improves the results considerably. The average result increases from one goal, ≈ 0.35, over using two goals, ≈ 0.44, to ≈ 0.48, when using four goals. The results from two goals are 25% better than from one goal, and four goals increase by another 8% making it 35% higher in total.

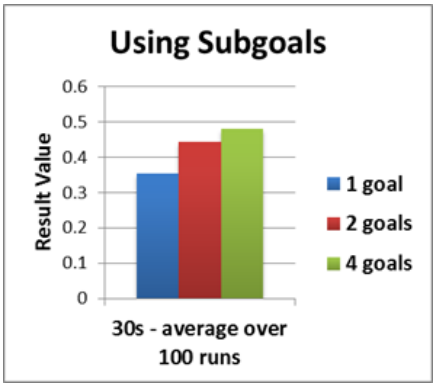


Figure 6.3.: This graph shows the average result for playing the first four sessions of ML using one goal, two goals or four goals. Using more (sub)goals clearly improves performance.

Using subgoals seems to be a huge improvement, despite the disadvantages anticipated. These were mentioned in Chapter 5.2.1. One issue was, that subgoals would inhibit investments. This might not be a problem, because investments might not be that important in ML or because only using one goal does not find good investments as well. The second expected issue was that cutting the search tree would prevent the ai to find the best solution later on. But the AI is currently to far away from finding the optimal path, at least under the tested time constraints, that this would become a problem.

6.2.2. Selection

To the selection step two modifications have been proposed. With UCB for SP a new single player UCB algorithm has been proposed. The non-tree policy, which selects the most promising node without searching the tree, is a radically different approach to node selection. Both algorithms will be experimentally tested.

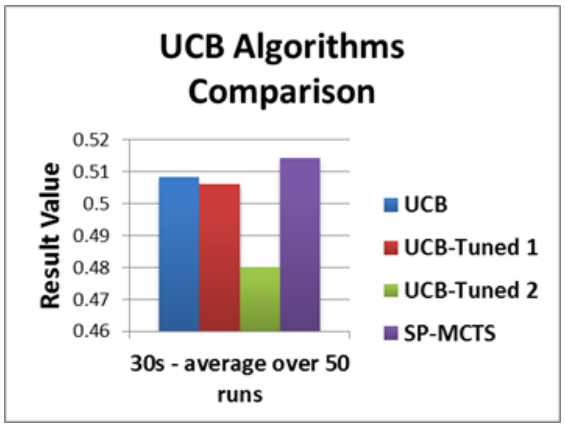
UCB algorithms

Before UCB for SP is evaluated, the other UCB algorithms are calibrated[2], by testing the results for a variety of different variable settings, and compared against each other.

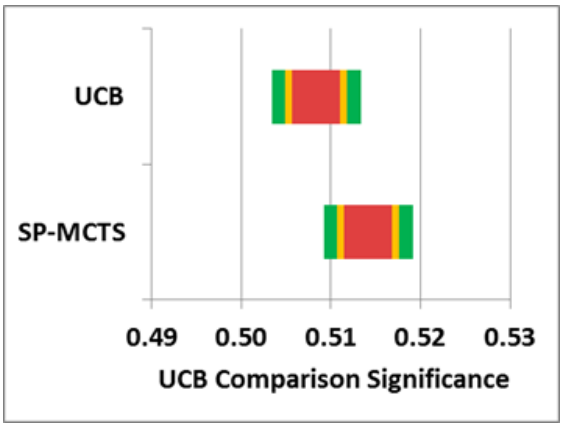
Experiment 4: Comparing UCB Algorithms In Experiment 4 each UCB algorithm is tested with its best settings against the others, to see how well they perform. The results show, that UCB and UCB Tuned 1 perform alike, UCB Tuned 2 clearly worse and SP-MCTS, the singe player variant, seems to be slightly better. All algorithms being at least 0.0149 results points better than UCB Tuned 2 is statistically significant with a confidence of at least 99.5%. SP-MCTS being better than UCB and UCB Tuned 1 is

[2]The calibration results can be viewed in Chapter C of the Appendix.

only true with 90%, respectively 95%, confidence. The improvement is only marginal, if any at all. These results can be viewed in Figure 6.4.



(a) The results of known UCB algorithms



(b) Significance comparison between the best two algorithms

Figure 6.4.: Figure 6.4a shows the average results for four different UCB algorithms. UCB Tuned 2 is clearly inferior to the other three. SP-MCTS might be slightly better than the middle group of UCB and UCB Tuned 1.
$UCB = 0.5083371$
$UCBTuned1 = 0.5061597$
$UCBTuned2 = 0.4802266$
$SP - MCTS = 0.5141475$

Figure 6.4b shows a comparison of the results from UCB and SP-MCTS. While the results of SP-MCTS seem slightly better, the distributions represented by bars of different color, overlap partly. Only the red bars do not overlap. Therefore the significance of SP-MCTS being better than UCB is only 90%. Yellow bars represent 95% and green ones 99%.

Experiment 5: How Well Performs UCB for SP? The newly proposed single player UCB algorithm called UCB for SP is built upon UCB Tuned 1. It replaces the average result of all search runs through a node, with the best result. This replacement can be linearly regulated from zero, only average result, to 1, only best result. With a best result weight of zero UCB for SP is equal to UCB Tuned. In E5 several best result weights are tested to see if UCB for SP can increase its results upon UCB Tuned 1.

The Results in the Chart 6.5 show a negative trend, when increasing the best result weight. The peak at 0.2 is not statistically significant, but the results deteriorate strongly above 0.6. E5 shows, that UCB for SP is, at least in this scenario, no improvement upon UCB Tuned 1.

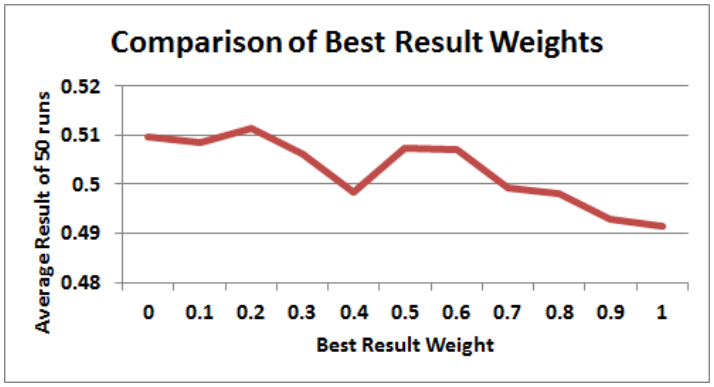


Figure 6.5.: Development of average results for different best result weights

Non Tree Policy

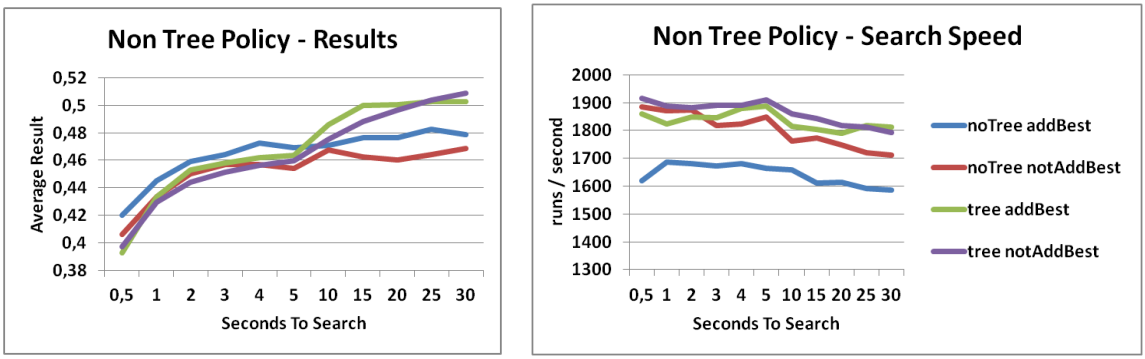
In order to explore promising nodes faster, although they are not on the most promising path, a non tree selection policy has been proposed in this thesis. Instead of traversing the tree from the root, searching for an unexpanded node, all nodes are included in the search.

Experiment 6: Testing the Non Tree Policy The non tree policy is tested on two different settings. One time the option to add the path to every newly found best game state is activated, the next time it is not. This option is evaluated further down, under the heading „Do Not Forget The Best Path“. E6 tests the performance for many different search times, because the expectation is, that non tree search is good for short search times, but will decrease when the tree grows large and its selection process is a lot slower than the tree approach.

The results from E6 confirm these expectations. The non tree search policy, in combination with adding the path to the best node, returns the best results up to circa five seconds search time. With longer search times it is clearly outperformed by the tree search policy. The non tree policy without adding the path to the best node can not improve on the tree policy for short search times and also performs worse above circa five seconds search time.

When looking at the search speed, measured in MCTS search runs per second, the non tree policy is slower as has been expected, but the drop is not as high as expected. This is most likely due to the search speed being low for ML which results in a relatively small number of nodes to search. From the comparison between the non tree policy with and without adding the best path it seems like the non tree policy spends a lot of time on adding newly found best paths. The results show, that this time is well spent.

Summing up it can be said, that the non tree policy performs better for low search times or maybe the first 10,000 nodes. After this time the algorithm should however switch to a tree policy, thereby combing the best of both.



(a) Result comparison between tree and no tree policy

(b) Speed Comparison of the same

Figure 6.6.: The speed and results shown for different search times. Beware, the horizontal axis does not scale linearly!
Significance of non tree policy with adding the best path, noTree addBest, being better than using the tree policy and adding the best path, tree addBest:
0.5 seconds $>=$ *99.5%*
1 seconds $>=$ *95%*
2 seconds $>=$ *90%*
3 seconds $>=$ *90%*
4 seconds $>=$ *99%*
5 seconds $>=$ *90%*
Non tree policy performs better for search times of five or less seconds.

6.2.3. Expansion

Many Paths, One Goal

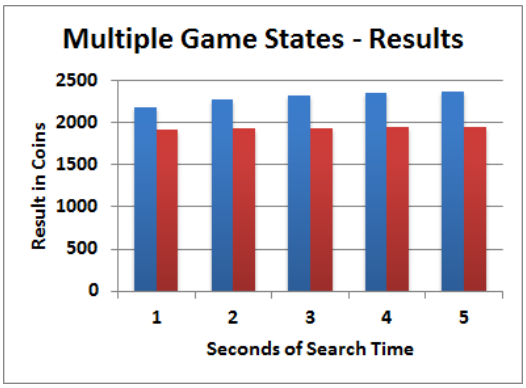
In a game tree different path can lead to the same game situation later on. This does not fit well with a tree representation. To avoid multiple nodes representing the same game state a search for game states on the same level of the tree can be performed. When a node for the new game state already exists it is not added to the tree and the search run is aborted, the search restarting by selecting a new node.

Experiment 7: Searching Multiple Game States In E7 the effects of this search are tested by turning it on and off with all other settings being stable. When comparing the settings with thirty second searches on ML no differences in the results, speed and tree size could be detected. The number of multiple game states found was to low to have any impact. To see if this is ML specific and might be different for bigger search trees or games with less different actions E7 is performed on Simple Coins.

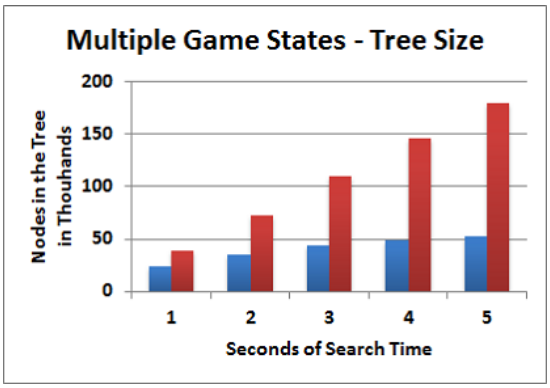
When looking at the results from E7, in Figure 6.7, a positive effect of the search is obvious. With search the results improve from 2188 to 2369 when using five instead of one second to search. When not using multiple game state search results only improve

from 1921 to 1956. Results are therefore overall higher and improve faster with more time given. With search enabled the theoretical maximum for Simple Coins with four sessions, which is 2400, is nearly reached. A look at the tree sizes can help explain these differences in results. The tree growth when search for multiple game states is slowed down considerably, thereby avoiding a lot of unnecessary nodes. The comparison of search speeds is no longer unexpected. Search speed with searching for multiple game states is faster, because although the search run is aborted when a multiple game state is found, this still counts towards the search runs performed. The time saved for playing out the game to estimate the value of the new node can then be used for a new search run. Search speed alone does not account for the better results. If this were the case searching two seconds without multiple game state search would have to return better results than searching one second with searching for them. The absolute number of search runs is higher for the first one, but the latter one returns the better results.

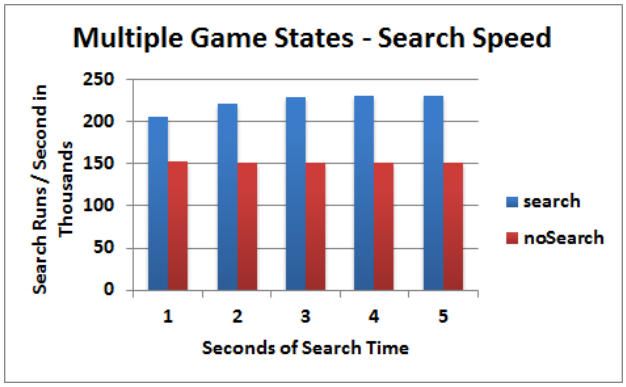
The multiple game state search seems to work for games with comparatively few different actions. The implementation currently does not recognize the same action on different objects to lead to an equal game state, also this might sometimes be the case. Allowing this without finding false positives would require a deeper analysis of paths to a game state. Maybe this is a worthwhile topic to optimize, because the search certainly helps in the more synthetic Simple Coins.



(a) The average results when (not) using multiple game state search



(b) Development of tree size



(c) Search speed comparison

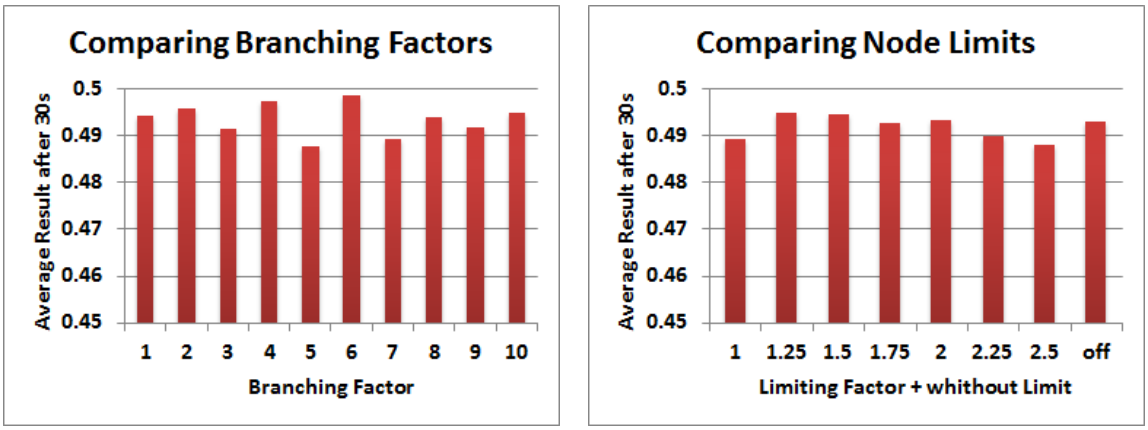
Figure 6.7.: These figures show the advantage of activating the search for multiple game states when playing Simple Coins for four sessions.

Start Small, Increase Steadily

A technique called progressive widening has also been implemented. It limits the branching factor of the tree to enforce deep search trees instead of wide and shallow ones. Does this approach improve the results as it has done for other implementations?

Experiment 8: Testing the Branching Factor In E8 branching factors ranging from 1 to 10 were tested on ML with 30 seconds search time. The results fluctuate a little bit, but there is no clearly better or worse setting and no trend whatsoever. The same is true for four sessions of Simple Coins with 0.5 to 3 seconds search time. The ML results can be viewed in Figure 5.3.

The branching factor has no noticeable impact on the overall performance of the algorithm.



(a) Results for different branching factors

(b) Results for different node limits

Figure 6.8.: Both figures do not show the full bar length in the vertical axis to allow the reader to see the small fluctuations.
In Figure 6.8a no discernible trend can be seen. The branching factor seems not to affect the results.
In Figure 6.8b several limiting factors for the level based node limit are tested. They do not improve or deteriorate the results.

Narrow Trees Grow Higher

To force tree growth towards deeper levels a limit for the number of nodes per tree level has been proposed in this thesis.

Experiment 9: Testing the Node Limit per Level In E9 the node limit is tested with several options. The lower the limit factor, the less nodes are allowed per level. A small example can explain this further:
In the example a best result has been found after 10 moves and the algorithm has one second time to search, with a search speed of 1000 search runs per second.

If divided evenly this would allow for 100 nodes on every tree level. This number is then multiplied with the limiting factor.
The results in Figure 5.3 show that different limiting factors as well as turning the node limit off do not have a measurable impact on the results. It therefore seems prudent to remove this feature with the overhead it produces.

Do not Forget the Best Path

There are two different ways to deal with a playout finding a new best game state. The result can just be backpropagated from the new node and the best game state saved, to return it if no better is found in the future. In single player games forgetting a once found best path is of no use. Another option is to not only backpropagate the result, but to add all game states on the path to the best game state to the tree. The effects of using

this setting are explored in Experiment 10.

Experiment 10: Adding the Best Game State's Path to the Tree How does adding the best path to the tree perform for different search times? This question is addressed in E10.

The results in Figure 6.9 show that adding the best path consistently improves the average results when the search time is a second or more. The average result differences and the significance of the results vary a little bit, but they are consistent enough to state that adding the best path is an improvement. An explanation for why not adding is most likely better when the search time is only half a second could be, that the time spent for adding several nodes to the tree is not well spent at the start. In addition to that most improvements on the best result, leading to new paths to add, are quite early in the search process.

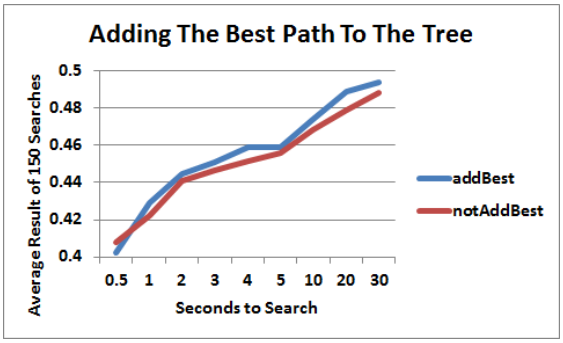


Figure 6.9.: Adding the best path to the tree instead of only saving the best game state performs better for search times of one second and above.
Significances of the statetments:
addBest worse than notAddbest:
0.5 seconds time >= 90%

addBest better than notAddbest:
1 seconds time >= 99%
2 seconds time >= 60%
3 seconds time >= 90%
4 seconds time >= 99,5%
5 seconds time >= 80%
10 seconds time >= 90%
20 seconds time >= 99.5%
30 seconds time >= 99,5%

6.2.4. Simulation

After it became abundantly clear during development, that a simple random playout approach would not return satisfactory results, modifications to the simulation strategy had to be made. They are tested in this chapter.

Action Groups

Action groups have been invented to transfer human knowledge about actions and their usage frequency into the search. This approach does conflict with general game playing but it might be a valid one for the purpose of balancing.

Experiment 11: Using Action Groups In E11 the effects of using action groups with different likelihoods are studied. In all experiments up until now the actions were categorized in four groups. They are:

Build Actions for building businesses and houses as well as buying decorations. Standard likelihood is 9%.

FinishSubgoal Actions with the soul purpose is to finish a part of a mission. E. g. consume an amount of gold as a payment. Standard likelihood is 10%.

Wait Actions that do nothing except to increase the game time. Standard likelihood is 1%.

Play All other actions in the game, like interacting with trees, stones and buildings. Standard likelihood is 80%.

To test action groups the standard likelihoods are changed. In missions (for finishSubgoal), building and waiting the likelihood of either group is raised by 10% which is deducted from the play group. In playing 5% are deducted from build and finishSubgoal and added to the playing group. In groups equal, all the groups have an equal likelihood of 25%. To test the effect of using no action groups there is one setting, called „1 group“, where all actions belong to the same group with likelihood of 100%. This setting corresponds to a total random simulation policy.

The results of the standard, missions and playing configuration are very similar. Increasing the likelihood to wait and thereby finish the session early decreases the average result a little bit. Even more hurting is to increase the building, respectively spending, probability. The comparatively bad results from using one group show, that using action groups provided huge benefits. In groupsEqual the less populated categories of waiting and building are weighted equal, leading to higher likelihoods for these actions. The results are even worse than with one action group. This shows that action groups likelihoods need to be set carefully to improve rather than deteriorate the performance.

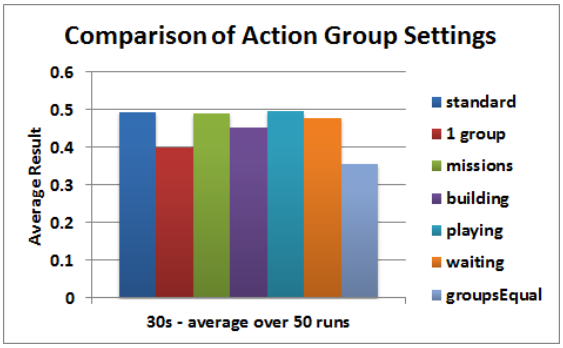


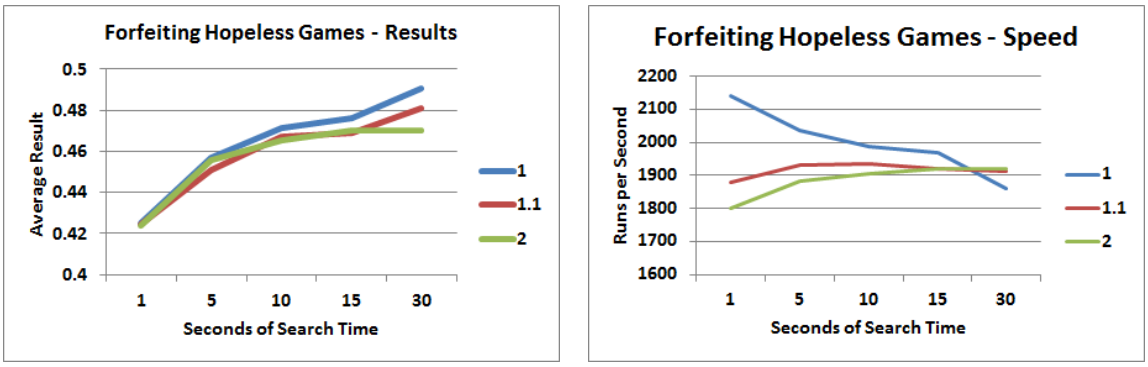
Figure 6.10.: The comparison of different action groups settings shows that using action groups is able to improve results considerably. Standard setting is $\approx 23\%$ better than 1 group setting. Different settings also show that fine-tuning the groups probabilities is needed to get the best possible results.

Forfeiting Hopeless Games

Once a terminal state has been while playing the game a base line for the expected results exists. With every new best game state the expectations increase. With forfeiting games the ideas was to end a playout before a terminal state is reached if a new best game state is can no longer be reached. The upside of this approach is that time can be saved on unpromising playout, but aborting a playout leads to a result of zero being backpropagated. This will undervalue game states a lot.

Experiment 12: Should Hopeless Games Be Aborted? The best possible result from a game currently played is multiplied with a value that is used to define how strict the algorithm is. With a setting of one every game that can no longer reach a new best game state is aborted. This is lessened for values higher than 1 because they increase the best possible result, before comparing to the currently best value. For E12 eleven different settings have been tested. From the hardest, 1, to the softest, 2, in steps of 0.1.

Figure 6.11 shows the average results for three settings. The results from 1.1 to 1.8 are nearly equal, as well as 1.9 and 2. They are therefore represented by one value for reasons of clarity. Using a strict factor of 1 outperforms a less strict factor of 1.1 and 2, especially after 10 or more seconds. When looking at the search speed the factor of 1 is actually faster than the other settings, but this turns around when searching for 30 seconds. The difference in results after 30 seconds has therefore to be due to evaluating hopeless games to zero, rather than just faster search speed.



(a) Result Comparison (b) Speed Comparison

Figure 6.11.: The result comparison shows the strict forfeiting setting of 1 consistently returning the best results, while the speed comparison shows that up to 30 seconds this might be at least in part due to its superior search speed.

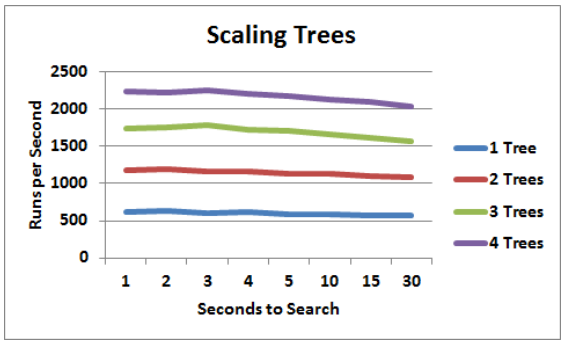
6.2.5. Parallelization and Consecutiveness

Root Parallelization and Tree Parallelization

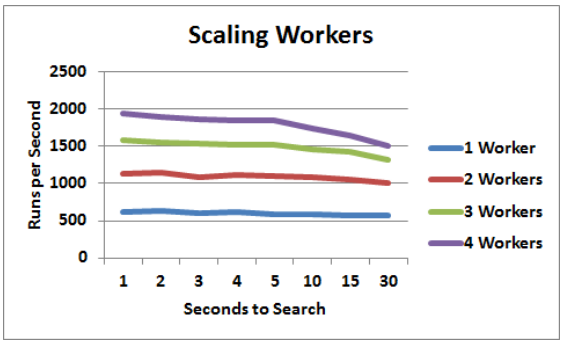
In the Combined Search Algorithm two different parallelization schemes have been implemented. Creating several independent trees, root parallelization (rp), and using several workers on a single tree, by locking access to the tree before accessing it, called tree parallelization (tp). The next experiments compare these approaches.

Experiment 13: How Do the Parallelization Schemes Scale? First the two approaches are compared in how they scale, when the number of threads used for the program is increased. On the test system up to four threads can be processed in parallel.

The results in Figure 6.12 show that both approaches can profit from more cores, but the root parallelization approach scales better. For rp with five seconds search time a second thread adds 558 runs per second to the 579 from 1 thread. The third adds another 568 and the fourth only 468. This last drop might be due to the fact, that the algorithm gets slower for bigger trees or that there are IDE and system threads conflicting with the search when all cores are meant to be used. For tp the second thread add 527, the third 428 and the fourth only 329. With four threads rp is about 18% faster than tp when searching for five seconds. This gap increases with longer search times, because tp slows down faster. This can be explained by the growing search tree resulting in longer locking selection and backpropagation operations with tp.



(a) Root Parallelization (rp)



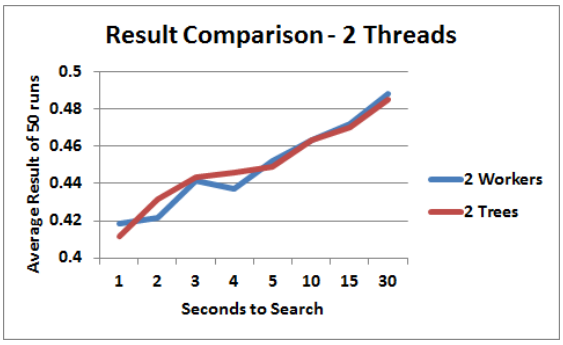
(b) Tree Parallelization (tp)

Figure 6.12.

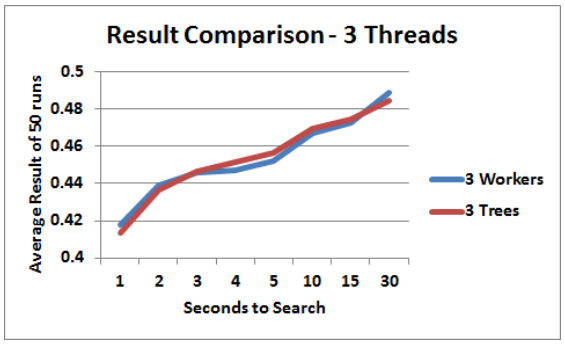
Experiment 14: How Do the Parallelization Schemes Perform? In E14 the results of both schemes are compared, to see how and if the differences in speed translate to differences in results.

The comparison graphs in Figure 6.13 show that there are no significant differences in results between the two approaches for up to four threads and 60 seconds search time. For search times above 60 seconds (only shown for 4 Threads) the tree parallelization approaches returns better results.

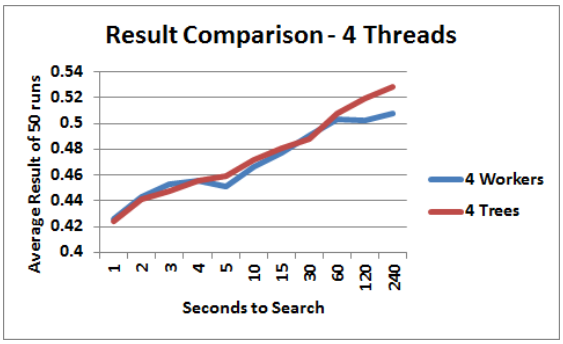
For lower search times the speed advantage of rp over tp is to small or compensated by the advantage of building up one tree with the knowledge of multiple threads.



(a) Using 2 Threads



(b) Using 3 Threads



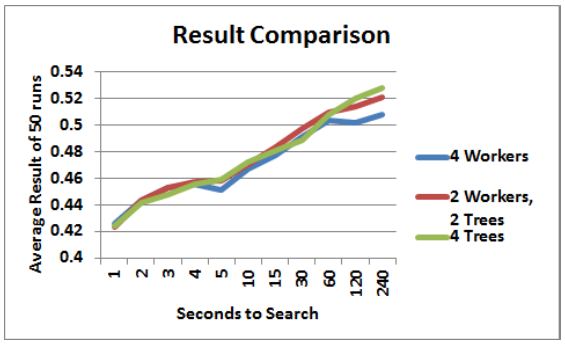
(c) Using 4 Threads

Figure 6.13.: Caution: Improvement is not linear over search time. It does only look this way due to the non linear horizontal axis.

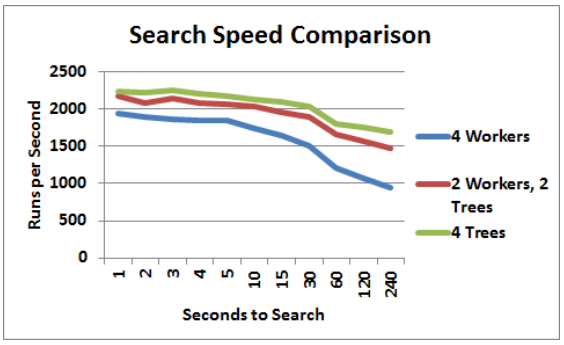
Experiment 15: Combining Parallelization Schemes Up to now the parallelization schemes were compared against each other. In E15 both schemes are combined to find out which combination of trees and workers returns the best results. The only sensible combination using four threads is building up two search trees with two workers each. This setting has therefore to compete against only scaling the number of trees and only scaling the number of workers.

Figure 6.14 shows the results and a speed comparison between all three settings. The speed comparison is now also continued to search times up to 240 seconds. While all three settings continuously decrease in search speed the drop-off when using 4 workers is considerably higher. The combined setting can compete with rp but is still outperformed at 120 and 240 seconds search time.

These results suggest that search speed is of more importance to the result than creating bigger and thus more well informed search trees. For projecting the performance in more parallelized multi-core systems this is a favorable conclusion, because the number of trees can easily be increased without negative impact on the search speed.



(a)



(b)

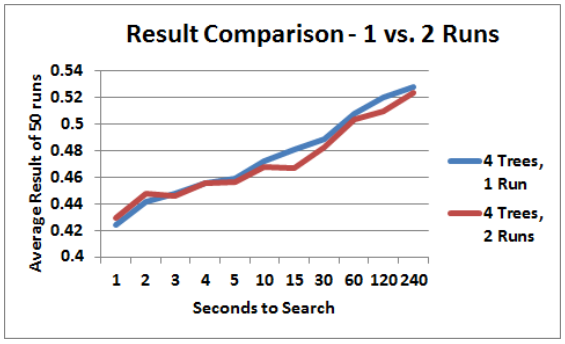
Figure 6.14.: The result comparison shows results of all three parallelization settings being similar up to search times of sixty seconds. Then the improvement of 4 Workers can no longer keep pace with 2 Workers, 2 Trees and especially 4 Trees, which returns the best results for long search times.
The search speed comparison suggests a relationship between speed and results. The 4 Workers setting always being slowest and decreasing faster than the two other settings.

Divide the Time

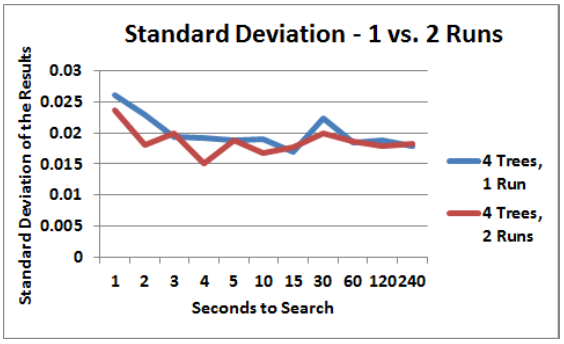
A new parallelization approach introduced in this thesis is to split the time available to search and run multiple shorter search runs.

Experiment 16: Multiple Consecutive Search Runs In E16 the setting of one single run is tested against two consecutive runs for search times from one second to four minutes.

The results in Figure 6.15a show that consecutive search runs have negative impact on the quality of the results, when the search time exceeds five seconds.. This could be acceptable if the goal to decrease the deviation of the results would have been achieved. But Figure 6.15b shows that lower deviation is not consistently achieved. Using multiple consecutive search runs is therefore considered to be a failure.



(a)



(b)

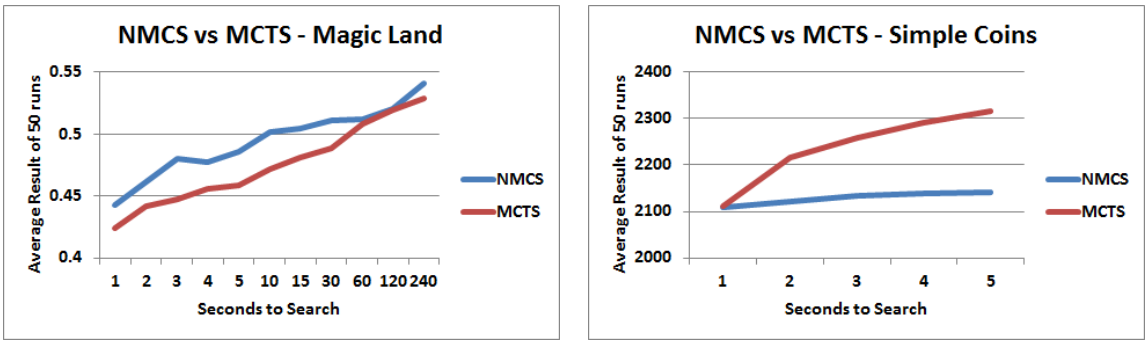
Figure 6.15.

6.2.6. Testing Nested Monte Carlo Search

As a second algorithm NMCS has been implemented as well. It can be used separately and in conjunction with MCTS searches. Earlier research showed that NMCS is well suited for single-player games [12], what made it an especially promising candidate.

Experiment 17: Comparing NMCS and MCTS Results For E17 both algorithms have been executed four times in parallel using the combined search algorithm implemented in the prototype. Because both algorithms use every core of the benchmark system this is a fair comparison of their power.

The results in Figure 6.16a are very clear. For search times lower than sixty seconds four NMCS algorithms outperform four MCTS algorithms considerably when playing Magic Land. For larger search times MCTS seems close this gap although it is not sure if this would continue for search times above four minutes.



(a) Magic Lang comparison (b) Simple Coins comparison

Figure 6.16.: These figures show, that NMCS outperforms MCTS in Magic Land for search times below one minute. In Simple Coins MCTS performs better and improves faster than NMCS.

As this result shows that all improvements to MCTS were not able to improve results over a fairly standard NMCS search this needs explaining. Two possible explanations are listed here:

- The most successful modifications, Action Groups and Subgoals, are used by NMCS and MCTS alike. The advancements to MCTS for social games are transformable to advancements to NMCS.
- Searching Magic Land is very slow. Therefore there is no huge tree after less than one minute, the time being again divided between four sessions. Without a huge tree to store information on good paths raw search speed is more important. This claim is supported by the results for searching Simple Coins in Figure 6.16b, where search speeds are orders of magnitude faster and trees therefore a lot bigger.

All the settings have now been tested and their influence on the result evaluated. Because this thesis want to answer the question whether it is possible to gain insights for game balancing from automatically playing these games, the prototype has to compete against human players in the next chapter.

6.3. Evaluation

6.3.1. Comparison with Human Players

How good are Human Magic Land Players

In order to evaluate the performance of the prototype its results have to be compared to real players. The target is to be better than most, about 90%, of the real players.

For the analysis of real user data fifteen days of game logs have been analyzed. In this time span around 750,000 new users started playing. The session log files contain the duration of a session, its date and time, and the experience points of the user at the end of every session. Tracking missions per player is to problematic, so the comparison has to be based on experience points. Several problems surfaced while analyzing the user data:

Log Data Does Not Match Game Play The session logs used for the comparison ended a session, whenever 60 seconds without any action had passed. This rule is very strict, because these timeouts happen a lot and players might continue shortly after. The session count is therefore to high for at least some players. This problem has been tackled for by comparing duration of game play rather than number of sessions or by combining sessions with lower than thirty minutes pause in between to one session. Another approach is to decrease the number of sessions the prototype may play when competing against real players by a factor of two. This factor has been computed by the difference of sessions logged with 60 seconds timeout against sessions logged from newly loading the website.

Human Players Behave Unsteady The prototype was built with the presumption, that player styles can be described by the number of sessions per day and the length of their sessions. A view on real user data has shown, that hardly any players behave regularly in this sense. The number of sessions they play varies daily, many players do not play every day, and sessions last from some seconds to over an hour in extreme cases. Fortunately the data sample was large enough to allow to find at least some players with regular playing behavior.

Short Term Comparison

For the short term comparison four sessions are played in one day. This comparison is just, because there are enough missions to play four sessions. The differences between ML and real Magic Land are therefore smaller than for the long term comparison.

The real user data is collected from all new players who play four sessions on their first day, playing between 1065 and 1935 seconds in total and have at least 100 XP. The duration limits have been set to ensure that they had about as much time as the prototype, the XP limit ensures that the players actually played the game. Data has also been analyzed for every player reaching the fourth session or playing three sessions per day. The results are very similar to the ones from the 4 sessions per day, regarding median but they have more extremes due to missing limits.

Results for the balancing tools prototype are obtained for playing four sessions on one day, each session lasting for 300 seconds to a total of 1200 seconds. The progress distribution of human ML players is shown in Figure 6.17. With a median XP value 242 after 60 seconds search the prototype beats 92% of all players. This result is directly to the right of the third spike in the human players chart. When factoring in, that human players were allowed to play longer than the prototype, but some of the players may not be in their fourth gaming session due to logging differences, the prototypes performance is at least in the region of human players.

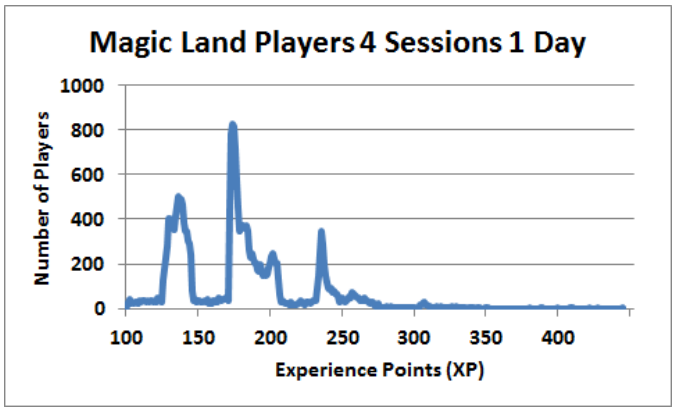


Figure 6.17.: Human player data for few gaming sessions.

Long Term Comparison

For the long term comparison different scenarios were used. The human players were selected by the number of sessions they played per day. Another condition was that they finish a session roughly (+/- 500 seconds) at the duration specified. From all users fulfilling these requirements the median days needed to play that far as well as the median session durations and the median sessions count were calculated. The session duration the prototype had to play to compete with them was then calculated by dividing the total duration by the median sessions count. The prototype had the median sessions count as his number of sessions or, in the stricter of the two settings, only half of this, with doubled session time, to compensate for logging errors. These sessions then were equally divided between the median days a human player took to reach the duration.

An example:

Users with 15-25 sessions per day that reached the total play time of 12500 seconds took 4 days in the median, had a median session duration of 196 seconds and a median session count of 64.

The prototype had therefore 64 sessions in four days, that makes 16 sessions a day. Each session has $\frac{12500}{64} \approx 195$ seconds. In the stricter setting the prototype has only 32 session with a session length of 390 seconds.

Duration	**Logged Sessions per Day**	**Days**	**Prototype Sessions per Day**	**Median XP**	**Prototype XP**
10000 s	4-8	6	3 | 6	1148	706 | 1206
10000 s	8-12	4	4 | 8	1012 90% = 1432	710 | 1383
12500 s	15-25	4	8 | 16	1073 90% = 1554	1318 | 1578

Table 6.2.: This table shows a comparison of the prototype with human players. The duration users compared to have played is shown in the first column. The second states how many sessions they had to play per day to be in the dataset. The second is the median number of days the human players took to play the duration. For the prototype a strict and a losse setting of days are calculated. In „Median XP“ the human results of the median, and where needed the 90th percentile are shown. The prototypes results are shown again for the strict and loose setting.

In Table 6.2 three long term comparisons are shown. The first scenario with few sessions (4-8) over 6 days is devastating for the prototype. With the strict, and most likely more accurate, setting it achieves a score around 706 XP whereas the median result of human players is 1148 XP. With the loose setting allowing for more sessions the prototype overtakes the median by a small margin.

In the second and third scenario the prototype manages to beat around 90% of the human players, when the loose setting is applied. In the second scenario the strict version is again dismal, while the strict setting in the third scenario is a little below the region of beating 75%.

Remark on the Comparison When looking at the comparison data this still shows huge variation. The comparison would have been fairer if more strict rules could have been applied. But the data set was to small to allow sufficiently significant numbers with stronger limits. The median session duration was calculated from the number of sessions of a player and the total duration when the XP were measured. This can skew the results because long and short sessions of one player are mixed together.

Another problem is that there is not enough content in the ML game descriptions. Missions do not last for as long as they should and the rewards of the missions a player

normally finishes when reaching the levels they have after the duration of 10000 or 12500 seconds add up to around 200 XP. This is a handicap for the prototype.

All together the data set and the game description were not good, or large, enough to allow for a definite assessment of the prototypes performance. As there is at least doubt about the performance compared to humans it is not yet ready for use in game development.

6.3.2. Performance Analysis

In this section the tool VisualVM is used the analyze where the prototype spends his calculation time while searching with MCTS. The findings can then be used to optimize implementations in the future. The sampling result can be seen in Figure 6.18.

Hot Spots - Method	Self time [%]	Self time	Self time (CPU)
game.expression.Comparator.**evaluate** ()		116200 ms (32.7%)	116200 ms
game.expression.AttributeReference.**evaluate** ()		96717 ms (27.2%)	96717 ms
game.expression.GODAttributeReference.**evaluate** ()		25132 ms (7.1%)	25132 ms
game.expression.GameValue.**evaluate** ()		12615 ms (3.5%)	12615 ms
game.GameState.**getGameObject** ()		11129 ms (3.1%)	11129 ms
game.GameState.**selectRandomPerformableActionFromGroup** ()		10605 ms (3%)	10605 ms
game.GameValueStore.**getValue** ()		9908 ms (2.8%)	9908 ms
game.expression.Atom.**equals** ()		5547 ms (1.6%)	5547 ms
game.GameAction.**isPerformable** ()		4806 ms (1.4%)	4806 ms
game.expression.Atom.**clone** ()		4617 ms (1.3%)	4617 ms
game.GameAction.**<init>** ()		5415 ms (1.3%)	4535 ms
game.expression.Expression.**hashCode** ()		4363 ms (1.2%)	4363 ms
game.GameObject.**addAttribute** ()		3984 ms (1.1%)	3984 ms
game.GameFactory.**getGameObjectDescription** ()		3854 ms (1.1%)	3854 ms
game.GameState.**findActionsInGroup** ()		3678 ms (1%)	3678 ms
game.expression.Operator.**evaluate** ()		2964 ms (0.8%)	2964 ms
game.GameObject.**<init>** ()		2875 ms (0.7%)	2554 ms
game.GameAttribute.**clone** ()		2490 ms (0.7%)	2490 ms
game.GameState.**cloneGAList** ()		2466 ms (0.7%)	2466 ms

Figure 6.18.: VisualVM sampling results for a MCTS search with the prototype.

By far the most time is spent for evaluating comparators (ca. 32%), an expression type mainly used in actions conditions, and attribute references (ca. 27%). Attribute references are used to access attribute values in expressions as well. Other expression types like game object description reference (ca. 7%) and game value (ca. 3.5%), to access the game value store, follow.

Approximately another 3% are spent for selecting a random action.

Over 70% of the computing time is spent dealing with Expressions. This is were most performance can be won in the future. This could be for example by caching values instead of evaluating them all over again.

7. Summary and Outlook

7.1. Summary

With the Wooga Game Description Language (WGDL) this thesis has introduced a new game description language for social building simulation games. Its feature set is designed to allow the description of game and contract mechanics, object creation and missions systems while enabling users to easily add new content through its class system.

Although WGDL currently misses a chance and a map system it is suitable to describe the aforementioned games. This has been demonstrated with an implementation of Wooga's social game Magic Land.

To play games described in WGDL a prototype of a balancing tool has been implemented in the course of this thesis. It uses Monte Carlo Tree Search (MCTS) to find ever better solutions. To optimize the results several new modifications to MCTS have been implemented and evaluated. The most successful of these were the introduction of action groups categorized by human knowledge to guide game play towards favorable moves, increasing the results by nearly 25%, and splitting games into several sub games, which increases the results by 35%. A necessary prerequisite was to introduce result formulas evaluating the results and stressing the importance of several important game values, like experience points, coins and finished missions.

Small result improvements have been obtained by adding all nodes on the way to a best game state to the tree, by forfeiting games when they can no longer lead to a new best game state and by using a non tree policy for short search times.

Unsuccessful were the attempts to introduce a new UCB formula called UCB for SP and a limit for nodes on every level of the tree. Both approaches have not been able to improve search results.

The performance of the prototype was improved by implementing several parallelization schemes, most of which have previously been implemented in other projects.

The comparison between results from MCTS and NMCS suggest, that the low search speed in ML prevents MCTS from profiting from its built up search tree. Speed improvements are a main concern for future research.

A comparison of the results with real user data of Magic Land players shows that it is better than more than 90% of the players for the short time of four sessions but its long time performance can not yet keep up with human players. This limits the applicability of this approach for game balancing but the performance is overall promising enough to

encourage further research in this area.

7.2. List Of Own Contributions

In this chapter the new contributions of this thesis to the research field are listed briefly. This thesis has built and expanded upon on research in two different areas. Several known and new enhancements have been evaluated. The field of game description languages and the field of playing (general) games with Monte Carlo search methods.

7.2.1. Contributions to the Field of Game Description Languages

- The creation of a new game description language called Wooga Game Description Language (WGDL) with the following new features:
  - timed consequences allow for contract mechanics
  - dynamic object creation
  - game mechanics through rules
  - class and object-description system to support the description of content
  - support for missions through advanced counters in the game value store
- WGDL is the first language designed and able to describe social building simulation games

7.2.2. Modifications to the Monte Carlo Tree Search

To adapt MCTS for games described in WGDL several modifications have been proposed and tested in this thesis. Some were very successful, others did not provide any benefit.

- Result formulas are introduced to evaluate terminal game states as well as lead the search towards optimal solutions
- Dividing the game and search in several smaller tasks through subgoals has been especially successful
- Modifications of the selection step:
  - A new single player selection policy called UCB for SP
  - A new non tree policy to improve results in short searches
- Modifications of the expansion step:
  - Limiting the nodes per level to increase tree depth
  - Adding the path to the best game state to the tree

- Enhancement of the simulation policy:
  - Grouping actions and assigning the groups a probability to make good moves more likely
- Parallelization schemes:
  - Consecutive searches allow multiple short search runs instead of one long run

7.3. Outlook

The summary has shown, that the approach described in this thesis has great potential even if it has not yet been fully realized. This section will show some points worth of being pursued in future research.

Improve Action Selection Improving the simulation step is one of the best ways to improve search results for MCTS. Currently the actions are grouped manually. This improves the search results in ML by nearly 25%. The selection of actions based on their frequency in the path to good results or even based on an analysis of their consequences might be worth further research. Is it possible to analyze the game mechanics from the result formula backwards? In ML this could look like this: Good results need high experience points, experience points are added by almost every play action and sometimes a lot more by finishing missions. Therefore it might be a wise idea to finish missions. To finish missions the player has to do.... This process mimics human behavior.

Performance Improvements To improve the performance of the prototype two approaches seem obvious. The scaling could be further improved by optimizing locking and parallel execution. A distributed search with multiple systems has been proposed for MCTS as well. When looking at where the search currently spends most of its processing time the execution and creation of expressions stands out. A caching mechanism for condition results and more lightweight expressions that are shared between objects promise huge performance improvements.

Analyze Human Game Play In spite of many different analysis methods there is still no clear picture how human players play social building simulation games. This makes assessing the value of insights from the tool very difficult. Are there discernible patterns in playing behavior and can different playing styles be distinguished? A game description with more content that is even more similar to the real game could help this. Implementing an automatic system to transfer game configurations from Excel to game descriptions would ease this a lot.

Other Use Cases The methods tested in this thesis could be applied to other fields than game balancing. One prominent idea was to use it as a cheat detection tool for mobile games. In these mobile building simulation games the user can play extensive sessions offline and then send his game state to the server to save it. While the backend server validates every action performed in a SBSG and forces the client to reset if an action is out of order, resetting half an hour of offline play because one command is not correctly recognized when the game state should be saved is difficult to convey to the player. If the search tool would be able to play better than most human players its results of playing could be compared to the players supposed game states. If the player changes his coins to a million but the tool only achieves 1000 coins in the same time this could be detected and the game state revoked.

The application of GGP for game development hast just started and there is plenty of room for improvement.

A. Magic Land

Wooga's SBSG Magic Land (ML) has been launched on Facebook in 2011. In ML the player assumes the character of a prince or princess and builds up his own kingdom. All actions of the player are guided towards increasing the wealth and size of his kingdom.

There are five resources and many more materials in the game. The resources are:

Diamonds This is the hard currency, i. e. it has to be bought or is only very seldom gifted to the user, of the game. It can be used to finish missions, speed up tasks, unlock new items and buy special items.

Coins The soft currency of the game can be collected through normal game play and is used to buy contracts, buildings and decorations.

Food Businesses, a special type of building, need food before coins can be collected from them. There is only limited space for this resource, but it can be increased by buying special buildings.

Energy Nearly every action costs the player one energy. Every five minutes one energy is refilled. This limits the amount of actions the player can perform. There is an upper limit to the amount of energy that can be saved. This limit goes up to 40.

Experience Points (XP) Every action that takes energy also returns one XP. Finishing missions is sometimes rewarded with a considerable amount XP as well. Whenever the XP values climbs over a threshold the player reaches a new level. This refills his energy, might increase the energy limit, and is sometimes rewarded with coins and a diamond as well. XP is the measure for progress in the game. With higher level new game features are unlocked.

The materials are wood, stone and many items that can be gathered from non person characters (npcs) and buildings, like swords and books. Materials are needed to finish mission goals and for constructing buildings. They are stored in an inventory.

Figure A.1.: A closeup look on Magic Land. On the top the different resources are shown. The friends are listed at the bottom. In the picture is a female chracter as well as the main castle, a tavern, a villagers and a knights house and some decorations. This is a very early kingdom.
Copyright Wooga GmbH

What does the player actually do in Magic Land? He does ...

Fight NPCs Trolls, gnomes, giants and toads as well as unicorns, the nice npcs, can be whacked. They return special materials once beaten and another npc shows up after some time.

Build Buildings A town is only as good as its buildings. Businesses and Houses can be constructed by the player. They can be used to obtain materials and coins. To return anything they have to be connected with the main castle or another special building via a street. This is where the map comes into play.

Remove Trees, Grass And Stones Trees, grass and stones can be removed by the player. From trees wood and from stones stone is obtained.

Decorate The Town Decorations can be bought for coins to increase the beauty of the town, subjectively, and to gain bonuses to the returns of farms and buildings.

Plant Crops On his farm plots, a special building type, a player can plant seeds which grow to a harvestable state in a specified amount of time. The more expensive and long growing the plant, the more food the player gets from harvesting them.

Finish Missions Several stories about the kingdom and its characters are told in missions, that consist of up to three tasks the player has to finish before the mission is completed. Missions grant big rewards of XP and coins, but can be very time and coins expensive to solve.

Interact With His Friends The friends kingdoms can be visited and small favors can be done. A gifting and requesting system allows the player to send gifts to his friends and ask for special items or help in completing a task.

Expand His Kingdom Every so often a new part of the map is revealed and covered in fog. The fog has to be removed by the player, before he can use the land to expand his town.

There is more to do in Magic Land, but the concept should be clear now.

B. Wooga Game Description Language Definition

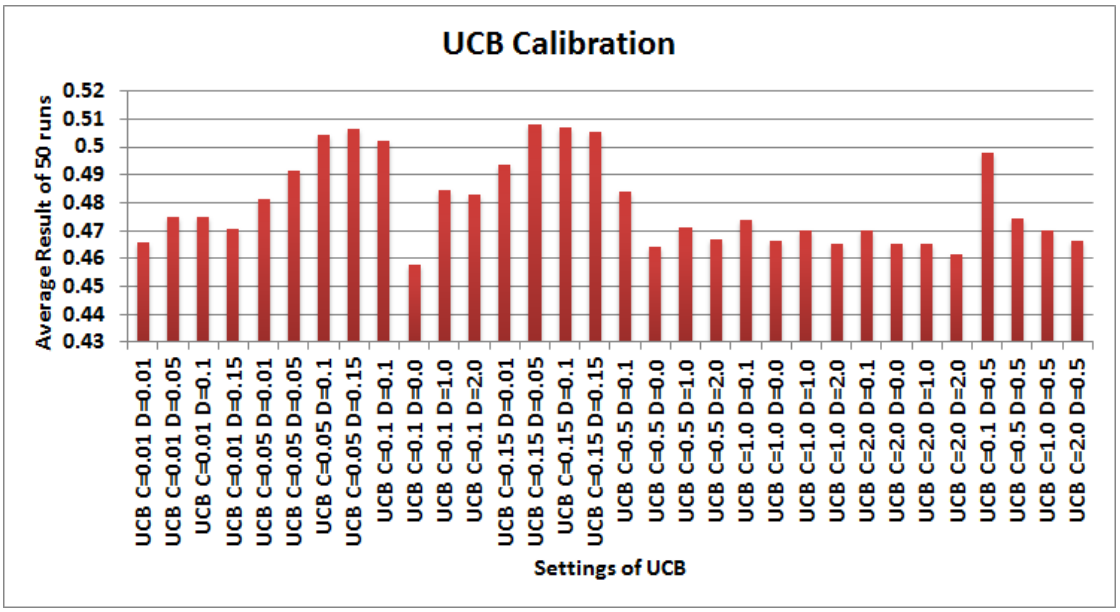
```
GameDescription = Goal, newLine, Initial, newLine, {(Goal | Initial |
    Class | Object | Attribute | Action | Condition | Consequence |
    ConsequenceAssignment | GameValueDefinition | ActionGroup |
    ActionGroupAssignment) newLine};
newLine = "\n";

// Classes and Objects
Class = "class", name, [":", className]; Object = "object", name, "is",
    className;

// Attributes belong to objects or classes
Attribute = "attribute", (("class", className | className), name,
    AttributeTypeWithValue | objectName, name, value);

// Attributes/Atoms have different types
AttributeTypeWithValue = Type, Value; Type = "long" | "double" | "boolean"
    | "string" | "object:", className; Value = String;

objectName = name;
className = name; name = String;

// Actions are the moves a player can take
// Actions belonging to no class (associated className) are rules, the
    player can NOT apply
// Rules are applied automatically once their condition is met or when an
    object is created ("create:" Rules)
Action = "action", (name | className, name, [("forEveryInstance" |
    "forEveryGOD"), className] || "create:", className, name);

// Actions have conditions which have to be met for it to be applicable
Condition = "condition", actionName, Expression;

// Consequences store the effects of an action
Consequence = "consequence", name, (("direct" | "create"), Expression |
    "createAndAssign", Expression, Expression | ("conditional" | "timed"),
    Expression, consequenceName | "game", "increase", Expression);

// Assignments define which action has wich consequence
```

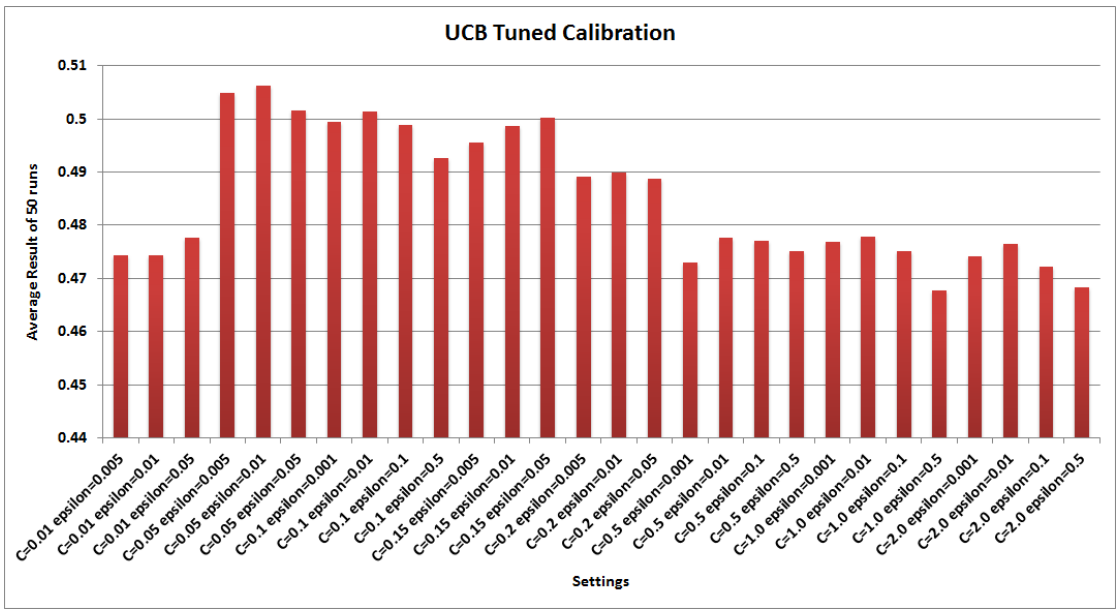
```
ConsequenceAssignment = "hasConsequence", actionName, consequenceName;

actionName = name;
consequenceName = name;

// Object Attributes can be accessed in different ways
// 1. the object of the class this belongs to = this
// 2. a named global objects
// 3. object of a specific class
AttributeAccess = ("this" | objectName | className), ".", attributeName,
    {".". attributeName};

// Initials are object that exist in the initial game state
// multiple creation of the same object is allowed
Initial = "initial", objectName

// goals are terminal conditions for the game or subgames. Starts from
    goal 1 to goal maxNumber.
Goal = "goal", PositiveNaturalNumber, BooleanExpression

// Expressions are tree based and used to change values, compare them and
    access attributes.
Expression = Atom | AttributeReference | ObjectReference |
    BooleanExpression | MathExpression | GameValue;

// Atoms are the basic unit
Atom = Type, Value;

// Already defined earlier
AttributeReference = AttributeAccess;

// objects are referenced this way. No whitespace between : and its name
ObjectReference = "object:", name;

BooleanExpression = Boolean | (Expression, [BooleanOperator, Expression]);
    Boolean = "TRUE" | "FALSE"; BooleanOperator = "==" | "!=" | "<" | ">"
    | "<=" | ">=" | "and" | "or" | "xor";

MathExpression = Expression, MathOperator, Expression; MathOperator = "="
    | "+" | "-" | "*" | "/";

// Accessing Game Value Store // Game values are defined and than accessed
    by their name in conditions
GameValue = "game.", name; GameValueDefinition = "game", name, Expression,
    {",", Expression};

// Action Groups categorize actions
ActionGroup = "actionGroup", name, PositiveNumber;
ActionGroupAssignment = "isInGroup", groupName, actionName; groupName =
```

```
    name;

// positive and negative numbers, doubles
Number = [-], PositiveNumber; PositiveNumber = Digit, {Digit}, ["."
    Digit], {Digit};

// all characters are allowed except whiteSpaces
Character = "A" | "a" | "B" | "b" | ... | "z" | ? specialCharacters ?;
Digit = "0" | PositiveDigit;
PositiveDigit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
    String = (Number | Character), {Number | Character};
    PositiveNaturalNumber = PositiveDigit, {PositiveDigit};
```

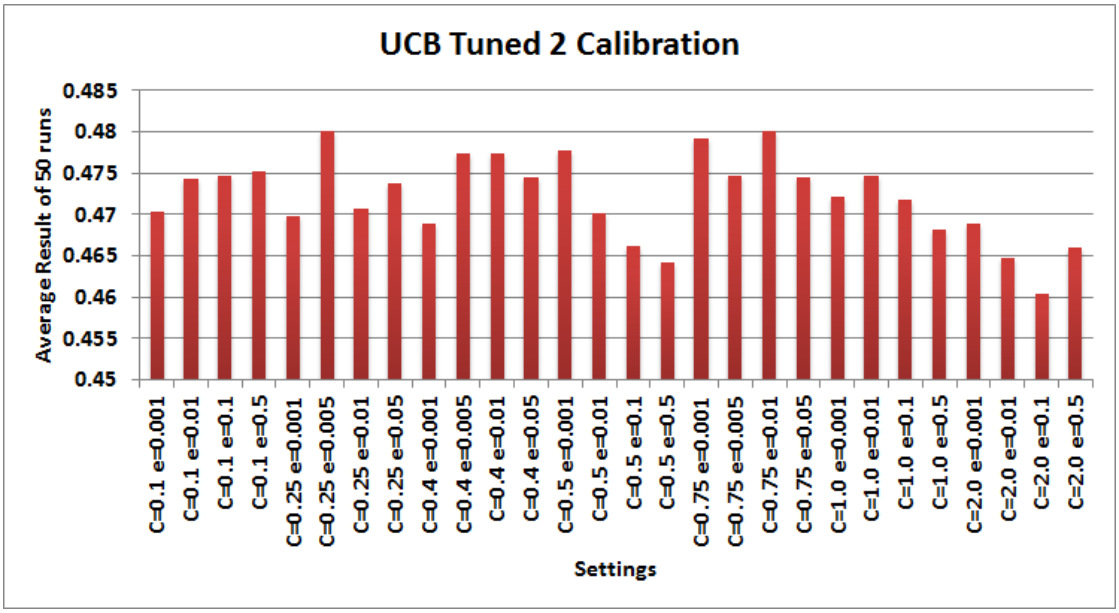
C. Experimental Results

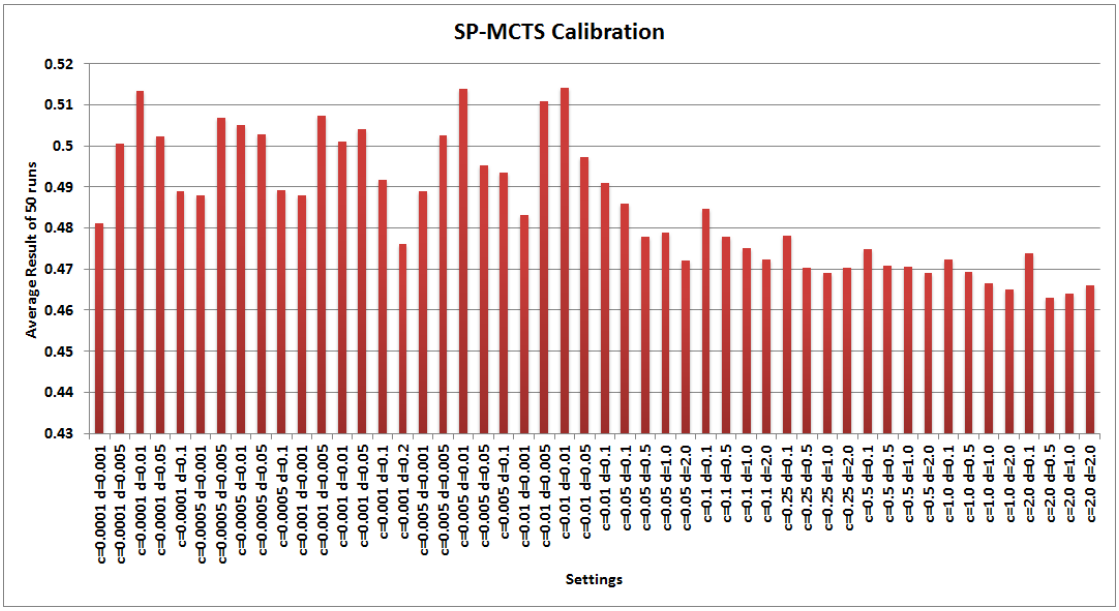
Calibration Results of UCB Formulas





C. Experimental Results





D. Game Description Example: Simple Coins

```
// Define the global class to hold the players important values
class globalClass
attribute globalClass time long 0
attribute globalClass sessionsFinished long 0 attribute
globalClass coins long 0 attribute
globalClass energy long 30

// create a game object description from that class
object global is globalClass
// create one object of type global
initial global

// The coin generators return coins when their getCoins action is applied
class coinGenerator
attribute coinGenerator coinValue long 0

object coin1 is coinGenerator
attribute coin1 coinValue 1
initial coin1

object coin2 is coinGenerator
attribute coin2 coinValue 2
initial coin2

// ... up to twenty, this content step has to be automated in the future!
object coin20 is coinGenerator
attribute coin20 coinValue 20
initial coin20

// increase the global coins value and decrease the energy
action coinGenerator getCoins
condition getCoins global.energy > long 0
consequence addCoins direct global.coins = global.coins + this.coinValue
consequence removeEnergy direct global.energy = global.energy - long 1
hasConsequence getCoins addCoins
hasConsequence getCoins removeEnergy

// fnish the session and refill the energy
```

D. Game Description Example: Simple Coins

```
action globalClass endSession
consequence refillEnergy direct global.energy = long 30
consequence changeTime direct global.sessionsFinished =
    global.sessionsFinished + long 1
hasConsequence endSession refillEnergy
hasConsequence endSession changeTime

// two action groups
actionGroup wait 0.1
actionGroup getMoney 0.9

isInGroup getMoney getCoins
isInGroup wait endSession

// four sessions each its own subgoal
goal 1 global.sessionsFinished == long 1
goal 2 global.sessionsFinished == long 2
goal 3 global.sessionsFinished == long 3
goal 4 global.sessionsFinished == long 4
```

E. Attached DVD - Content

The DVD attached to the back of this thesis contains the following content:

Source Code The java source code of the prototype implemented for and tested in this thesis.

Code Documentation The code documentation has been generated using Javadoc. A documentation of the public interfaces, as well as a documentation of all interfaces, including private and protected ones is available in HTML. The prototype is not extensively documented, but packages and classes are. The important search classes are covered in more detail.

Executable An executable java archive has been created with some predefined settings. It can be used to test the prototype on the readers own PC. Currently no arguments are supported, so the predefined settings have to be used. It has been created using maven.

Game Descriptions All game descriptions used for evaluation, especially Magic Land with several different goals, are supplied.

Experiment Data The data generated while performing the experiments to evaluate the prototype is contained in separate folders on the DVD.

Thesis This thesis is available as a PDF.

Pictures The pictures and figures used in this thesis are on the DVD as well.

Bibliography

[1] Mahlmann T., *„The Strategy Games Description Language (SGDL)“*, Center For Computer Games Research ITU Copenhagen, April 2012

[2] Brown C. et al., *„A Survey of Monte Carlo Tree Search Methods“*, IEEE Transactions on Computational Intelligence and AI in Games, Vol. 4, No. 1, March 2012

[3] Mahlmann T., Togelius J., Yannakakis G. N., *„Modelling and evaluation of complex scenarios with the Strategy Game Description Language“*, IEEE Conference on Computational Intelligence and Games (CIG), 2011

[4] Mahlmann T., Togelius J., Yannakakis G. N., *„Towards Procedural Strategy Game Generation: Evolving Complementary Unit Types“*, IT University of Copenhagen, 2011

[5] Jensen B., Nielson J., *„Artificial Agents for the Strategy Game Description Language“*, Study, IT University of Copenhagen, August 2011

[6] Budde L., Huth N., *„Soziale Netzwerke: Eine repräsentative Untersuchung zur Nutzung sozialer Netzwerke im Internet“*, Representative Study, Bitkom, p. 6ff., 2011

[7] Thielscher M., *„A General Game Description Language for Incomplete Information Games“*, School of Computer Science and Engineering, University of New South Wales, 2010

[8] Smith J., Hudson C. *„Inside Virtual Goods: The Future of Social Gaming 2011“*, Report, Version 2.0, Inside Network Inc., p. 14f., 2010

[9] Bourki A. et al., *„Scalability and Parallelization of Monte-Carlo Tree Search“*, The International Conference on Computers and Games, 2010

[10] Browne C., F. Maire, *„Evolutionary game design“*, IEEE Transactions on Computational Intelligence and AI in Games, 2(1), p. 1-16, 2010

[11] Winands M., Björnsson Y., *„Evaluation Function Based Monte-Carlo LOA“*, Advances in Computer Games, Lecture Notes in Computer Science, Volume 6048, ISBN 978-3-642-12992-6, Springer-Verlag Berlin Heidelberg, p. 33ff, 2010

[12] Cazenave T., *„Nested Monte-Carlo Search“*, International Joint Conference on Artificial Intelligence 2009, p. 456-461, Pasadena, July 2009

[13] Kulick J., *„World Description Language - A logical Language for agent-based Systems and Games“*, Bachelor's Thesis, Freie Universität Berlin, 2009

[14] Kulick J., Block M., Rojas R., *„General Game Playing mit stochastischen Spielen“*, Technical Report, Freie Universität Berlin, 2009

[15] Jean Méhat and Tristan Cazenave, *„Combining UCT and Nested Monte-Carlo Search for Single-Player General Game Playing“*, IEEE Transactions on Computational Intelligence and AI in Games, 2009

[16] Björnsson Y., Finnsson H., *„Cadia Player: A Simulation-Based General Game Player“*, IEEE Transactions on Computational Intelligence and AI in Games, Vol. 1, No. 1, 2009

[17] Cazenave T., *„Parallel Nested Monte-Carlo Search“*, Proceedings of the 2009 IEEE International Parallel & Distributed Processing Symposium, 2009

[18] Teytaud F., Teytaud O., *„Creating an Upper-Confidence-Tree program for Havannah“*, Advances in Computer Games Conference 12, Pamplona, 2009

[19] Chaslot G. et al., *„Meta Monte-Carlo Tree Search for Automatic Opening Book Generation“*, Proceedings of the IJCAI'09 Workshop on General Intelligence in Game Playing Agents. p. 7-12, 2009

[20] Love N. et al., *„General Game Playing: Game Description Language Specification“*, Standord Logic Group, Computer Science Department, Stanford University, Technischer Report LG-2006-01, 2008

[21] Chaslot G. et al., *„Monte-Carlo Tree Search: A New Framework for Game AI“*, Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, pages 216-217, Stanford Univ., California, 2008

[22] Chaslot G., Winands M., van den Herik H. J., *„Parallel Monte-Carlo Tree Search”*, in Proc. Comput. and Games, LNCS 5131, Beijing, China, p. 60–71, 2008

[23] Chaslot G. et. al., *„Combining expert, offline, transient and online knowledge in Monte-Carlo exploration“*, 2008

[24] C. Browne, *“Automatic generation and evaluation of recombination games,”* Ph.D. dissertation, Queensland University of Technology, 2008.

[25] Schadd M. et. al. *„Addressing NP-Complete Puzzles with Monte-Carlo Methods*“, Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning, Volume 9, p. 55-61, Brighton, UK, 2008

[26] Schadd M. et al., *„Single-Player Monte-Carlo Tree Search“*, Computers and Games: 6th International Conference, Beijing, China, 2008

[27] Coquelin P., Munos R., *„Bandit Algorithms for Tree Search“*, Uncertainty in Artificial Intelligence, Vancourver, Canada, 2007

[28] Kaiser D., *„The Structure of Games“*, Florida International University Electronic Theses and Dissertations, 2007

[29] Coulom R., *„Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search”,* Proc. 5th Int. Conf. Comput. and Games, Turin, Italy, p. 72–83, 2006

[30] Kocsis L., Szepesvári C., *„Bandit based Monte-Carlo Planning“*, Computer and Automation Research Institute of the Hungarian Academy of Sciences, Budapest, 2006

[31] Schäfer J., *„Monte Carlo Simulation im Skat“*, Otto-von-Guericke-University Magdeburg, 2005

[32] Osborne M. J., Rubinstein A., *„A Course in Game Theory“*, Massachusetts Institute of Technology, ISBN 0-262-65040-1, 2004

[33] Freeman E. et al., *„Head First Design Patterns“*, ISBN 0-596-00712-4, O'Reilly Media, 2004

[34] Russell S. J, Norvig P., *„Artificial Intelligence: A Modern Approach“*, 2. Edition, Upper Saddle River, New Jersey, ISBN 0-13-790395-2, 2003

[35] Campbell M., Hoane A. J., Hsu F., *„Deep Blue“*, Artificial Intelligence, Edition 134, p. 59, 2002

[36] Auer P. et al., *„Finite-time Analysis of the Multiarmed Bandit Problem“*, Machine Learning, Kluwer Academic Publishers, The Netherlands, 2002

[37] Romein J. W., Bal H. E., Grune D., *„The Multigame Reference Manual“*, Faculty of Sciences Dept. of Mathematics and Computer Science Free University Amsterdam, The Netherlands, 2000

[38] Gamma E. et al., *„Design Patterns - Elements of Reusable Object-Oriented Software*“, professional computing series, ISBN 0-201-63361-2, Addison-Wesley, 1995

[39] Allis L., *„Searching for Solutions in Games and Artificial Intelligence“*, Thesis, University of Maastricht, 1994

[40] Pell B., *„A Strategic Metagame Player for General Chess-Like Games“*, AAAI Technical Report FS-93-02, 1993

[41] Pell B., *„Strategy Generation and Evaluation for Meta-Game Playing“*, Dissertation, University of Cambridge, 1993

[42] Pell B., *„Metagame: A New Challenge for Games and Learning“*, Heuristic Programming in Artificial Intelligence 3 – The Third Computer Olympiad, 1992

[43] Pitrat J., „Realization of a general game-playing program“, IFIP Congress (2), p. 1570-1574, 1968

[44] Hammersley J. M., Handscomb D.C., *„Monte Carlo Methods“*, Methuen & Co Ltd., London, 1964

[45] Edwards D. J. , Hart T. P. , *„The alpha-beta heuristic“*, Massachusetts Institute of Technology, 1963

[46] Newell A., Shaw G., Simon H., *„Chess-Playing Programs and the Problem of Complexity“*, IBM Journal of Research and Development, p. 320-335, 1958

[47] von Neumann J., Morgenstern O., *„Theory of games and economic behavior“*, Princeton University Press, p 46f, 1953

[48] Shannon C. E., *„Programming a Computer for Playing Chess“*, Philosophical Magazine, Series 7, Volume 41, No. 314, 1950

[49] Morrison C., *„CityVille Edges Past 100 Million MAU — Over Half Are International Users“*, http://www.insidesocialgames.com/2011/01/12/cityville-edges-past-100-million-mau-over-half-are-international-users/, 2011, visited at 31.07.2012

[50] Schreiber I., *„Game Balance Concepts“*, Series of 10 Articles/Lectures, http://game-balanceconcepts.wordpress.com/2010/07/07/level-1-intro-to-game-balance/, visited at 01.08.2012

[51] McCarthy J., *„What is Artificial Intelligence¿ ‘*, Computer Science Department, Stanford University, Article, 2007, http://www-formal.stanford.edu/jmc/whatisai/node1.html, visited at 31.07.2012

[52] Schaefer S., *„Tic-Tac-Toe (Naughts and Crosses, Cheese and Crackers, etc.)“*, http://www.mathrec.org/old/2002jan/solutions.html, 2002, visited at 31.7.2012

[53] Geary D., *„A look at the Composite design pattern“*, http://www.javaworld.com/javaworld/jw-09-2002/jw-0913-designpatterns.html, visited at 01.08.2012