Master Thesis

Collaborative Go

Alex Hugger

December 13, 2011

Supervisor: S. Welten

Prof. Dr. R. Wattenhofer
Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Abstract

The game of Go is getting more and more popular in Europe. Its simple rules and the huge complexity for computer programs make it very interesting for a study about collaborative gaming. The idea of collaborative Go is to play a simple two player game in a setup of teams behaving as one single player instead of one versus one.

During this thesis, we implemented a framework allowing to play collaborative Go. The framework is heavily based on the existing Go Text Protocol allowing an integration into already existing Go servers or a competition with existing Go programs. Multiple players are merged through the use of different decision engines into one single player. Two decision engines were implemented based on different interaction schemes. One is based on a voting process whereas the other one decides on the final move of a team by applying a Monte Carlo simulation.

The final experiments show that collaborative gaming is very interesting and attention attracting for the players. Especially the voting mechanism used in the collaborative player achieved a high user satisfaction. For an automated decision engine, we found out that it is very important to make the decision process as transparent as possible to all users, as this increases the level of the acceptance even when a bad decision was made.

Acknowledgments

During the accomplishment of this master thesis, a lot of people supported and guided me to a successful completion of the work. Therefore, I would like to thank the following people:

The people from the Distributed Computing Group for testing the software and providing feedback on the overall game play. Professor Roger Wattenhofer and Samuel Welten for allowing me to participate in such an interesting topic and providing me constantly with new ideas and problem solutions. Bea Meier, Marc Bruggmann, Christian Helbling and Marcel Hugger for all the perusal and the corrections of this paper. Last but not least I want to thank the Go Club Zurich, especially their president Lorenz Trippel, for the provided feedback about the client interface, their vision of collaborative Go and the extensive testing and evaluation sessions.

Zurich, December 13, 2011

Alex Hugger

Contents

1 Introduction **1**
- 1.1 Introduction to the Game of Go . . . 1
- 1.2 Computer Go . . . 3
- 1.3 Goal of this Thesis . . . 4
- 1.4 Outline . . . 4

2 Related Work **5**
- 2.1 Computer Go . . . 5
  - 2.1.1 Detailed Comparison to Chess . . . 6
  - 2.1.2 Early Computer Programs . . . 6
  - 2.1.3 Monte Carlo Methods in Computer Go . . . 7
- 2.2 Collaborative Gaming . . . 8

3 System **11**
- 3.1 Framework . . . 12
- 3.2 Go Text Protocol . . . 13
  - 3.2.1 Required Commands . . . 14
  - 3.2.2 Collaborative Go Extension . . . 14
- 3.3 Go Server . . . 16
- 3.4 Collaborative Go Player . . . 18
- 3.5 Collaboration Schemes . . . 19
  - 3.5.1 Voting Schema . . . 19
  - 3.5.2 Monte-Carlo Schema . . . 20
- 3.6 Client . . . 21
- 3.7 Web Framework . . . 21

4 Experiments **23**
- 4.1 Collected Data . . . 23
- 4.2 Collaboration Schemes . . . 25
  - 4.2.1 Monte Carlo Decision Schema . . . 26
  - 4.2.2 Voting Decision Schema . . . 26
- 4.3 Time Management . . . 27
- 4.4 Monte Carlo . . . 28

5 Conclusion **35**
- 5.1 Future Work . . . 36

A Glossary **39**

List of Figures

1.1 Capturing of Stones 2

3.1 Basic System Overview 11
3.2 Collaborative Player on KGS 13
3.3 Go Library UML 17
3.4 Voting Decision Engine 20
3.5 Monte Carlo Decision Engine 20
3.6 Graphical User Interface 21
3.7 Settings Form in the Web Server 22

4.1 Percentage of Actions performed in the Time Limit 27
4.2 Test Go Game 29
4.3 Problematic Situation for a Monte Carlo Engine 33

List of Tables

1.1 Go Ranking System 3

2.1 Comparison of Go and Chess 7
2.2 Rules for plain Monte Carlo Go 8
2.3 Rules for a successful Collaborative Game 9

3.2 Regenmove Specification 14
3.1 Required GTP commands 15
3.3 The Colgo GTP Extension 16

4.1 Logged Data during a Game 25
4.2 Setup for Monte Carlo Experiment 1 29
4.3 Result of the Monte Carlo Experiment 1 31
4.4 Result of the Monte Carlo Experiment 2 32
4.5 Simulation Values of the Played Moves from Black 32

1 Introduction

The game of Go is a very popular ancient board game. It origins in China and is over 2000 years old. The basic idea of the game is that two warlords try to conquer a shared area by controlling the opponent and capturing free space on the board. For understanding the attraction of the game of Go, one needs to know two important facts about it. First of all, the rules are easy to understand and can be taught to a complete beginner in less than five minutes. This allows beginners to participate in the game after a short instruction. The second factor is the wide range of different tactics during different states of the game[1], which results in very distinct games even when playing multiple times against the same opponent. Due to these factors, Go is one of the most played board games for two players in the east of Asia. But Go is not only in Asia widespread, in the German language area the game is getting more and more popular.

1.1 Introduction to the Game of Go

The game of Go is based on a few main rules. The rules are not standardized, which leads to different rule sets in different regions of the world. They differ mostly in areas that are not critical to the main game play. The following rules are the most important ones to know. Other more complex rules (like estimating the final score of a game) are not listed in this section. If the reader is interested in a more detailed explanation of the rules of Go, the corresponding Wikipedia page[2] gives a good overview.

- Go is a board game[3] played by two players using black and white stones.

[1] There are much more options than for example in a game of chess at any time in the game.
[2] Wikipedia - Rules of Go: http://en.wikipedia.org/wiki/Rules_of_Go
[3] The board is a square of a size between 9 and 19.

- Stones are placed alternating on the board intersections. Black always starts placing the first stone.
- When a stone or a group of stones loses the last liberty[4] to a stone of the opponent, the group is removed from the board.

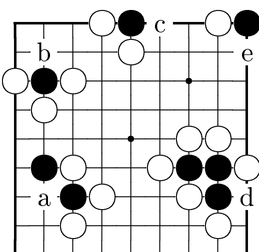


Figure 1.1: Illustration of positions where black stones can be captured.

- Suicide moves[5] are not allowed. When a stone placed in a suicide position removes the last liberty of at least one opponent stone, it is not a suicide move and is therefore legal.
- The game is finished when both players consecutively pass or one player resigns.
- The winner is the player with the higher score. The result is then expressed by the difference between both scores.

For allowing interesting games between different players, Go has a clever handicap system. Each player has a rank reflecting his Go playing strength. Table 1.1 shows all different types of ranks and the corresponding user stages. The transition between the different ranks is specified differently in each country, therefore there are no universal statements telling how to receive which rank. When the rating of the two players is not the same, the player with the worse ranking receives a certain amount of handicap stones on the board according to the difference of the two ratings. These stones are placed on the board before the game actually starts. The amount of stones is defined by the difference of the two handicaps. Of course the handicap stones are also adapted to the size of the board, because four handicap stones on a 9x9 board are worth much more than on a board with size 19x19. When both players have the same rank, no handicap stones are given to any player.

The starting player has a statistical advantage over the opponent for which the Go rules specify a correcting Komi value. The Komi value is normally a 5 point[6] advantage over the opponent. This advantage is included in the final calculation of the score and has no direct influence on the game play.

The calculation of the final score is highly depending on the used set of rules. The most common one is the territory scoring defined by the traditional Japanese rules. The main idea is to count the amount of empty spots that one player has surrounded minus the stones the opponent has captured. This gives a score value for each player.

---

[4] The liberty of a stone or a group of stones is defined as the number of empty spots next to the group.

[5] A move that removes the last liberty of a group of stones where the played move and the group of stones are from the same player.

[6] Especially in tournaments 5.5 is used as the Komi instead of 5. This removes the chance of having a draw at the end of the game.

Level	Range	Stage
Student	30k - 20k	Beginner
	20k - 10k	Casual player
	10k - 1k	Intermediate amateur
Master	1d - 7d	Advanced amateur
Professional	1p - 9p	Professional player

Table 1.1: The Go ranking system.

The final score is then expressed by stating the difference between the two values and the point of view [7].

The most commonly used Go terms in this thesis are listed and explained in the Glossary A.

1.2 Computer Go

Computer Go is interesting because of the simplicity of the problem and the fact that it is very difficult to implement a good computer player. As stated in Section 1.1, the game of Go consists of a few easy rules. Nevertheless, the computer programs playing Go have much more problems to beat their human opponents than for example chess programs. Currently the best computer program on the KGS Go Server[8] is 'zen19D' with a rating of 5d[9]. This means, that the best current program is able to compete with advanced amateur players. The problems arising when programming an artificial intelligent player for Go can be broken down to the following key points:

- **Size of the board**
  The normal Go board has a size of 19 which gives 361 possible locations for a stone on the board. During the game, almost all possible locations are legal moves. In the beginning, the number of possible moves can be reduced significantly, because of the symmetry of the board. But with the loss of the symmetry on the board, the options rise up again to 361 minus the stones already laying on the board. This is mainly a problem for all approaches that try to make use of a minmax search on the game tree.[10]
- **Nature of the game**
  In chess, the difficulty of the game is constantly decreased by removing figures from the board. In the end game, only few figures are left and all of them have a limited number of possible moves. In Go, there is the possibility that with each move the current situation gets more difficult.
- **Tactics**
  The game of Go is very complex regarding the effects of one stone to a situation on a completely different board location. For example capturing one

[7] When the player W scores 45.5 points and player B 40, the final result is W+5.5

[8] KGS Go Server: http://www.gokgs.com/

[9] The KGS Go Server periodically generates a list of computer programs playing on the online server. The whole list is available under: http://senseis.xmp.net/?KGSBotRatings

[10] So far, the biggest board that has been completely solved is a 5x5 resulting in a complete win by the computer program. More information can be found at: http://erikvanderwerf.tengen.nl/5x5/5x5solved.html

stone of the enemy can give the opponent the possibility to strengthen his stones at another location. Also some moves seem to be useless on the current board, but they can have a huge effect in a later game state.

1.3 Goal of this Thesis

Since the creation of a computer player is very difficult, we are interested in whether it is possible to create a good player by combining several novices to the game of Go into one single player. As it is not clear how a successful collaboration schema looks like, we need to have a system that allows us to perform the required experiments and to test different kinds of collaboration schemes.

The main goal of this thesis is therefore to provide a framework that allows to play games of Go in a new collaborative way. Until now there are only multi-user Go games, in which you compete with all of your opponents or do not share any knowledge between team members at all. The idea of the collaborative Go is to provide two teams of Go players with a new way of playing Go. Each team consists of multiple human players. Each one of them provides his team with a move suggestion. The team then can (based on different collaboration schemes) decide which one of the moves will be picked. The opponent only receives the information that the shown move has been selected by the players.

The framework is based on Java and the existing Go Text Protocol, which allows an easy integration to many online Go servers. The framework must provide a way to insert new types of collaboration schemes and an initial set of features usable for ensuring the correctness of the played game and analyzing Go situations. On top of that, it is required to have an easy to use client that allows unexperienced users to participate in a game of collaborative Go. In order to allow as much Go players as possible to participate in the collaborative Go project, we provide a web server that allows to start own collaborative Go games without the complex integration into an existing online Go server.

1.4 Outline

This thesis is organized as following: Chapter 2 covers related work and background information. The overall system including the used technologies, the different collaboration schemes, the GTP protocol and its extension are covered in Chapter 3. Chapter 4 describes what kind of data was gathered during the experiments. Chapters 4 and 5 are used to show the results gathered during this thesis and provide a small outlook on what can be done in the future.

2 Related Work

In this section, the focus lies on the related work done so far. First of all, it is important to know that computer Go is a very challenging current research topic and therefore consists of a huge amount of different work completed so far. Section 2.1 highlights some of the most important research in the past years that had an influence on how this thesis was conducted and what may had an effect on the decisions throughout this thesis. The presented work, mainly the section about the Monte Carlo approaches used in computer Go, influenced and assisted us to build an automatic decision engine. Without the presented work, such an engine would not exist because the underlying problem is too difficult and the creation of a computer program playing Go is not part of this work. Thanks to several conducted studies, we realized that the construction of an automated decision engine based on Monte Carlo is possible even in a short period of time.

Sadly, the field of collaborative gaming is not as popular and has therefore less work to influence this thesis. Nevertheless, there are multiple researches on the effects of collaborative gaming especially in MMORPG[1] and classical board games. Since we want to integrate the positive effects of collaborative gaming in our implementation of collaborative Go, we will take a small glimpse on what makes classical games more interesting.

2.1 Computer Go

Computer programs playing chess (or any other kind of board game) typically use a combination of tree searches and an evaluation function for the resulting state. Because Go is not only different to normal board games in many small features, but is also completely different in the nature of the game; instead of focusing on killing a fixed piece on the board (e.g. the king in chess), Go is often focusing on

[1] Massive multiplayer online role playing games

expanding into free territory of the board. This results in a lot of moves that effect the stones in an indirectly manner. This means, that the played stones often cut off opponent groups from each other instead of attacking one single stone by reducing its liberties.

2.1.1 Detailed Comparison to Chess

A very intuitive way to show why playing Go is a tricky task for computer programs is to compare it to the game of chess. Jay Burmeister wrote a technical report [3] about the game of Go where he takes a deeper look into the difficulty of the game itself. The Table 2.1 explains the main differences in detail according to his research. Due to these differences, the classical chess solutions, being a combination of a fixed depth tree search with an evaluation function on the resulting board, can not be used for playing Go.

Even a simple evaluation of the current board is a very tricky part in Go. A player needs to identify the structures on the board, decide whether they are alive or dead, remove the dead structures and then perform an area count to estimate the current value of this board position. Also, the player needs to think about which one of the empty areas may be big enough to contain a structure, which one is alive and if it is possible for the opponent to place such a structure in there. An even simpler evaluation, namely who wins the game when the current board state is final, has been shown to be polynomial-space hard [8].

2.1.2 Early Computer Programs

The first computer program playing Go was invented by Albert Zobrist in 1968 during the completion of his PhD [13]. Together with this program he invented the Zobrist hashing[2] which is nowadays used for detecting different positions. This hash function allows to evaluate each board state only once (for example in chess) or can be used to detect superkos[3]. The evolution of the computer programs was pushed by several annual competitions[4] which offered a quite large price money for the winner of the tournaments.

The initial attempt for building a Go player was to provide the computer program with Go knowledge. In Go, there are a lot of rules and guidelines on how to play in a given situation. Each played move has a corresponding answer and each existing structure has rules on how to deal with it. The programmers then implemented all of these guidelines [9]. When an analysis of a structure on the board matched a specified rule, the program played according to this rule. With a fast pattern recognition and a useful amount of rules, the bots played on a reasonable level. But this approach led to several issues. First of all, the program can never play better than the person who implemented it and the resulting program was specially vulnerable to human opponents. This is because the human detects missing rules or patterns of bad behavior and starts exploiting these. The result of this process is, that after a few games the computer program was beaten consistently by the human player.

[2] The details of this hash function can be found at: http://en.wikipedia.org/wiki/Zobrist_hashing

[3] The term superko stands for a repetition of an older board state over several moves. In short, the last move that creates a board state that is an exact copy of any previous board is illegal.

[4] One of the most famous ones, is the Ing Prize offered by a Taiwanese banker and Go player: http://en.wikipedia.org/wiki/Ing_Chang-ki

Feature	Chess	Go
Board size	8 x 8	19 x 19
# moves per game	Approx. 80	Approx. 300
Branching factor	Small (approx. 35)	Large (approx. 200)
End of game and scoring	Checkmate (simple definition - quick to identify)	Counting territory (consens by players - hard to identify)
Long range effects	Pieces can move long distances	Stones do not move, patterns of stones have long range effects (e.g. ladders)
State of board	Changes rapidly as pieces move	Mostly changes incrementally (except for captures)
Evaluation of board positions	Good correlation with number and quality of pieces on board	Poor correlation with number of stones on board or territory surrounded
Programming approaches used	Amenable to tree searches with good evaluation criteria	Too many branches for brute force search, pruning is difficult due to the lack of good evaluation measures
Human lookahead	Typically up to 10 moves	Even beginners read up to 60 moves
Horizon effect	Grandmaster level	Beginner level
Human grouping processes	Hierarchical grouping [4]	Stones belong to many groups simultaneously [10]
Handicap system	None	Good handicap system

Table 2.1: Comparison of the differences between Go and chess according to Jay Burmeister [3].

Furthermore, the fact that the amount of time and effort required to program a reasonable computer player is about five to ten person-years [9], led to think about new approaches in computer Go.

2.1.3 Monte Carlo Methods in Computer Go

One of the most discussed new approaches is the use of Monte Carlo methods for playing Go. An implementation without any Go knowledge was able to reach a 25k level on a 9x9 board [2]. The rules for playing were very simple and are listed in Table 2.2.

At first, it is unclear if a random game from a fixed starting point can provide some useful information. For explaining why it is not a bad idea to play random games, we provide you with an example. Let us assume that we have a board with a situation on it that is currently not finished. This means that a group of black stones on the board is neither alive nor dead. As a next step we assume that when black plays the right move, the group of black stones is alive. On the other hand, if black plays a different move and white has the possibility to play that move, the group has to be considered dead. The outcome of the counting function used for evaluation of the board therefore relies heavily on whether black plays this move or not. The average

I	Moves are performed randomly with probabilities assigned by the method of simulated annealing.
II	The value of a position in which the game is over is defined by counting.
III	To find the best move in a given position play the game to the very end as suggested by (I) and then evaluate as in (II); play many such random games, and the best move will be the one which does best on average.

Table 2.2: The rule set used in plain Monte Carlo Go by Bernd Bruegmann [2].

result of the outcome then differs by the value of this black group. These differences in the outcome of a game allow to detect the good moves on the board. Monte Carlo Evaluation was conducted in many different ways, even without any adaption of the probability a move is selected [1]. Despite the fact that this approach leads to relatively good results, considering the small amount of Go knowledge required to build such a program, the final solution behaved badly especially in situations where the order of the moves was important. Monte Carlo may be able to detect multiple good next moves, but the order in which these moves are required to be played is undetectable. Since our decision engine has only to select a single move out of a list of reasonable proposals from the players and therefore not to worry about a bad move ordering, we decided to build our engine on top of a Monte Carlo decision engine presented in [1].

Current Go programs build on top of multiple different approaches from many different fields. These fields contain methods from temporal difference learning [11] up to upper bound confidence for tree searchers [6]. Of course, also the classical methods are still used and required for having a good computer player. The current top computer players (like Zen19D[5]) are in almost all cases closed source and provide no information which approaches are used in them.

2.2 Collaborative Gaming

The huge success of online multiplayer games have once again shown that playing with friends can be interesting not only when competing with each other. Because the MMOG[6] are very complicated to analyze, most studies have been conducted on classical board games. These board games are much clearer considering the structure and the action of the games. If a computer game is required for a study, "Age of Empires" is a very popular candidate because it is well known and easy to play. When analyzing games, there are two very often used terms, namely competitive and cooperative. Cooperative games require a team play of all members to successfully achieve the goal of the game, whereas in competitive games, the players are competing with each other. In cooperative games, each player can have his own goal and payoff. This is the difference between cooperative and collaborative gaming. In collaborative gaming, each player has the exact same goal and payoff. If the team wins, everyone wins and vice versa. According to a recent research [12] the following rules in Table 2.3 are required for a satisfying collaborative game. Of course, not all

[5] Zen19D Bot on KGS: http://senseis.xmp.net/?Zen
[6] Massive Multiplayer Online Game

of them can be perfectly integrated in a simple game like Go but we will try to stay as close to these rules as possible.

1	To highlight problems of competitiveness, a collaborative game should introduce a tension between perceived individual utility and team utility.
2	To further highlight problems of competitiveness, individual players should be allowed to make decisions and take actions without the consent of the team.
3	Players must be able to trace payoffs back to their decisions.
4	To encourage team members to make selfless decisions, a collaborative game should bestow different abilities or responsibilities upon the players.

Table 2.3: The rules for a successful collaborative game according to [12].

3
System

For playing collaborative Go, a new system must be built because the existing implementations of Go servers only allow playing two regular engines based on GTP[1] against each other. For our purpose, a more complex system according to the Figure 3.1 is required.

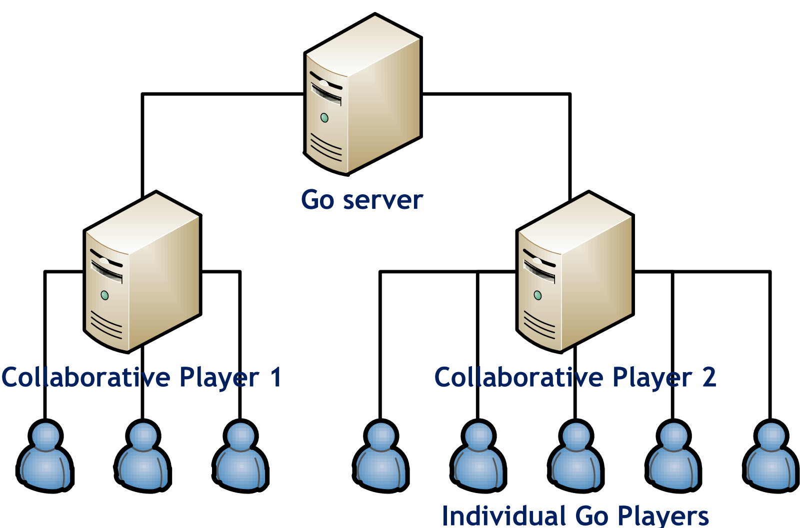


Figure 3.1: The required system using one Go server and two collaborative players with different users. The black lines stand for a GTP communication link.

[1] Go Text Protocol - More information about this standardized protocol for playing Go can be found in Section 3.2.

The overall system consists of several independent software programs that communicate with each other over different channels. This chapter explains the detailed construction of each part and the way they interact with each other. The different collaboration schemes are explained in detail including all required GTP extensions used on the communication channels and the control flow between the different participants of the decision process. All parts of the system are implemented in Java except for the web framework which is built on a LAMP[2] stack.

3.1 Framework

The term framework stands for the overall system including external software used for integrating the implemented system with other servers like the KGS online server. The Go server, the collaborative player and the client exchange all necessary data over the Go Text Protocol (GTP). This gives the possibility to completely separate the different programs from each other. It allows to easily manage the failure of a client without any interference on the Go server or the other clients. Also, it provides the collaborative Go player with the possibility to include any Go playing software and play software assisted games.

The Go server is the control instance of any game. It serves as an access point for both collaborative players (or any other GTP capable Go program) and is responsible for managing the correct sequence of actions and operations. The collaborative Go player behaves as a normal Go program from the perspective of the Go server. Its main responsibility is to process, merge and interact according to the type of collaboration schema with all registered users.

Because the collaborative Go player implements the standard GTP interface, it can be easily connected to the KGS online server. There it is possible to participate as a computer program in unranked matches against users from all over the world. Figure 3.2 clarifies this setup. Because the players participating in the collaborative player change all the time, playing ranked games would make no sense as the real rank of the collaborative player will change with players joining and leaving the collaborative player.

To increase the fun factor and evaluate whether a communication channel can augment the overall team skills, we decided to integrate a very small chat server into the collaborative player. The chat allows communication between all team members. Neither private chat messages to a single player nor communication to the other team are currently allowed.

[2] A Linux system running an Apache web server working with MYSQL and PHP.

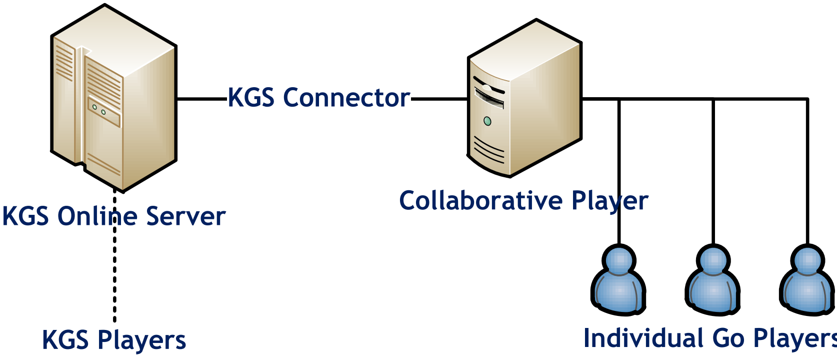


Figure 3.2: The setup connecting a collaborative player to the KGS online server as a computer player with the KGS Connector.

3.2 Go Text Protocol

The Go Text Protocol (GTP) is an attempt to standardize the communication messages required for playing Go over different communication channels. The current version is 1.0. For version 1.0, a clear definition of the protocol is missing. There is only the existing implementation in GNU Go 3.0.0[3]. The only available document describing the complete protocol is the draft for version 2.0[4]. Since there is no more discussion on the mailing list[5] and GNU Go 3.8.0 is implementing the draft of the 2.0 version, any implementation of GTP in this software is based on the current draft [5].

The GTP is an asymmetric protocol using a master and a slave. In a normal setup, the master is the host of the game and the playing Go program serves as the slave. Every communication origins from the master. The slave has to answer every request with a response. When multiple requests arrive at the same time, the slave has to answer all of them in the order of arrival.

A command has typically the following syntax

```
[id] command_name arguments \n
```

and is responded with either a success

```
=[id] response \n\n
```

or an error message

```
?[id] error_message \n\n
```

The id is an optional number identifier. If it is provided in the command, the response must start with the same identifier as in the corresponding command. If the id was not sent with the command, it may be omitted when sending the response.

3 GNU Go: http://www.gnu.org/s/gnugo/
4 Go Text Protocol: http://www.lysator.liu.se/~gunnar/gtp/
5 GTP mailing list: http://lists.lysator.liu.se/mailman/listinfo/gtp

The following Section explains the required commands in detail. The GTP specification draft defines a lot more commands, especially for the tournament subset. If a Go program wants to play tournaments, it is required that it knows about the placement of handicap stones (`fixed_handicap`, `place_free_handicap`, `set_free_handicap`) and the supported time command (`time_settings`). As these features are not required in our system, they were omitted.

3.2.1 Required Commands

The commands listed in Table 3.1 must be implemented by any system that supports the GTP. These commands provide a Go playing system with the basic actions that can be performed when playing Go. Of course, playing Go is possible with a smaller set of commands. The implemented game server described in Section 3.3 works perfectly fine even when only the four basic commands `boardsize`, `play`, `genmove` and `quit` are understood by the client.

3.2.2 Collaborative Go Extension

For playing collaborative Go, the existing Go Text Protocol needs to be extended. Of course, the amount of required extensions used while playing is heavily based on the type of collaboration scheme. The one command required in any type of collaborative setup is the `reg_genmove` command specified in Table 3.2. This command is part of the regression subset defined by the GTP version 2.

Name	`reg_genmove`
Input	Color (String)
Output	Vertex (String)
Remark	This command is almost the same as the `genmove` command. The only difference is, that the genmove command plays the selected move on the board, while the `reg_genmove` only proposes this move.

Table 3.2: The `reg_genmove` command specification in detail.

For collaboration schemes that include a voting mechanism, some new GTP commands are required. Together with the ones used for initializing and controlling the GUI client, these commands are packed in a GTP extension that we named Colgo. Table 3.3 lists all protocol commands used in the extension. For a participation in a collaborative Go game, only few of them (`colgo-my_name`, `colgo-vote`) must be supported. The other commands are only used for controlling the behavior of the user interface and have no effect on the actual game play.

The chat protocol used between the client and the collaborative player is not part of the GTP. This is because a chat protocol is typically not asymmetric at all. Any client must be able to send the information he wants at any time. Because of the nature of the chat, it is also important to immediately deliver the received message to all chat participants. When the chat protocol would be integrated into the GTP, one could have the problem that he first needs to answer the received vote command before replying to a chat message. As the chat message may be important during the process of voting, this behavior would not be acceptable.

Name	`protocol_version`
Input	-
Output	Version number (Integer)
Remark	Most of the current implementations return "2".
Name	`name`
Input	-
Output	Name (String)
Remark	The name of the software without any version number.
Name	`version`
Input	-
Output	Version (String)
Remark	Version number of the software. ("3.8.1")
Name	`known_command`
Input	Command name (String)
Output	Is the command known? (Boolean)
Remark	There is no distinction between unimplemented and unknown.
Name	`list_commands`
Input	-
Output	List of commands (List[String])
Remark	All implemented commands are returned (including private extensions).
Name	`quit`
Input	-
Output	-
Remark	The response to this command must be sent before closing the connection.
Name	`boardsize`
Input	Size (Integer)
Output	-
Remark	One has to clear the board manually after resizing it.
Name	`clear_board`
Input	-
Output	-
Remark	Clears the board to the initial state.
Name	`komi`
Input	Komi (Float)
Output	-
Remark	All values for the komi are accepted.
Name	`play`
Input	Move (Color and vertex of the move)
Output	-
Remark	Example for a move string: "white h5" or "B F4"
Name	`genmove`
Input	Color (String)
Output	Vertex (String)
Remark	"pass" and "resign" are also valid vertices.

Table 3.1: List of all required commands a valid GTP implementation has to provide.

Name	`colog-player_name`
Input	Name (String)
Output	-
Remark	Inform the player about his team name.
Name	`colgo-my_name`
Input	-
Output	Name (String)
Remark	Inform the collaborative player about the user name.
Name	`colgo-opp_name`
Input	Name (String)
Output	-
Remark	Inform the player about the team name of the opponents.
Name	`colgo-gui_state`
Input	Message (String)
Output	-
Remark	Inform the user about the current state of the game and the required actions he has to perform.
Name	`colgo-level`
Input	-
Output	User rank (String)
Remark	Inform the collaborative player about the user ranking.
Name	`colgo-user_list`
Input	Names of all current team members (List[String])
Output	-
Remark	Inform the user about all users currently participating in his team.
Name	`colgo-chat`
Input	Port (Integer)
Output	-
Remark	Inform the user about the port where the chat messages must be delivered to.
Name	`colgo-vote`
Input	All proposed vertices (List[String])
Output	Selected vertex (String)

Table 3.3: List of all GTP commands provided by the Colgo GTP Extension.

3.3 Go Server

The Go server is the game controlling part of the framework. Its main purpose is to ensure the correct game flow. Therefore, the main part of it is the game logic. Also, it serves as a Go library, which provides different functionalities to the collaborative player and the GUI client. The last component of it is a logging framework which allows to store the played game in different locations using different formats.

Due to the fact that currently no open source Java library for the game of Go exists and the ones used in most of the Go playing programs cannot be extracted and used in an easy way, the first task of the server is to provide the game logic in a clear and structured way. The design of this part is very close to the physical representation of board and stones, which makes it much easier to use the provided classes than in existing implementations.

Figure 3.3 shows the most important points of the architecture of the Go library. As this library will be used in the decision engines of the collaborative Go player, it contains more features that may be helpful when implementing new collaboration schemes, which are not shown in the diagram (for example listing all neighbors of a stone or simulating a move on the current board). Some features have not been implemented because of the difficulty of the underlying problem. For example, the scoring function is not able to distinct between stones that are alive or dead. This simplifies the scoring to a simple area count on the board. For getting correct scores it is therefore required to play until both players agree that the current state of the board is final[6].

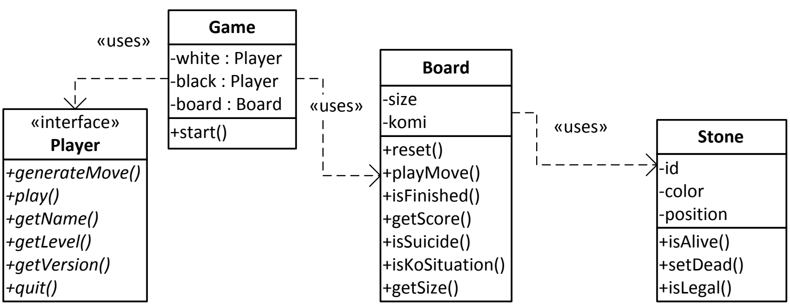


Figure 3.3: The main parts of the integrated Go library displayed in UML.

Also, a missing feature in the Go server is the whole time management. This is because there are several different time settings used in Go and due to the complexity of some popular setups. In Go tournaments the rules often specify a time setting like "60' + 30" byoyomi". This means that the first hour of the game one can play as slow as one wants to. After this hour every move has to be completed in under 30 seconds. It is also very popular to specify a certain amount of byoyomi times. Lets say a user gets five 1' byoyomi times. This means that when the user plays his stone in under 1' he keeps his total amount of byoyomi times. If it takes him longer to play the number is reduced. When a player has no more byoyomi times left, it is considered as a resign and the player loses the game. Further information about time management in Go can be found at [7].

The Go server also provides the implementation of the GTP and several different types of players. In general there are three different types of players. The first is the normal network player. This player is used for including players over a communication link based on a host name and a port number. The underlying logic is almost the same as the local player. The local player takes a command which will be executed and the standard input and output are mapped to the controller instead of the network sockets. The last type of player is a SGF player. SGF[7] is a widespread format for storing all different types of board games. This type of player is mainly for debugging and testing of the Go library, but it can also be used to replay games

6 Final means, that all stones lying on the board are alive. This decision has to be made by the two players by playing each unclear structure until there is no more discussion about the state.

7 More information about the Smart Game Format and the detailed specifications can be found at: http://www.red-bean.com/sgf

from a game log and run different types of analyses[8] on them.

The logging part of the server is quite useful for many different types of applications. For example every played game is available as a SGF file. This allows to replay, discuss and analyze each played game[9]. This is especially in a collaborative setup very interesting, as the players may discuss the bad moves after the game and learn from their mistakes. Of course, the voting information is not stored in the SGF log of the game, as this process happens completely in the collaborative player. For that reason, the game is logged into a SQL database allowing to couple the logs of the collaborative players together with the plain game log of the Go server.

3.4 Collaborative Go Player

The collaborative player is the point where information and proposals from all clients get together and are processed according to the different collaboration schemes specified in Section 3.5. The software provides two different interfaces depending on the point of view. From the Go server perspective, the collaborative player looks like a regular Go playing software supporting the minimal requirements from the GTP specification. From the other side, it looks like a Go server with an integrated chat server, which understands all commands from our extension set of the GTP.

This separation from the Go server and the fact that a collaborative player can be run completely independent from any other part of the system, gives a huge advantage when it comes to computer supported playing[10]. Let us assume that we use a very complex and computational heavy way to decide which move will be played or proposed to the user. Thanks to the independence of the collaborative player we can run it on a powerful machine, where only our decision process is consuming resources and there is no need to share resources with the Go server or any other part of the system.

Computer supported playing can then be achieved by inverting the two phases of the current decision process. One could evaluate each possible move on the board and provide the user with the top five moves according to this evaluation. The user then selects a move from the suggestions instead of suggesting a move to the collaborative player. The computer assisted playing would therefore result in only a change of the underlying collaboration schema.

The main requirement of the collaborative player is to provide an easy way to exchange different types of collaboration schemes. This can be achieved by providing an interface for all schemes and a very simple way to integrate the new schema into the current software. Extending the feature set of the player can be done by adapting only the main class. No other changes at any location in the code (except when the GTP extension needs to be adapted) are necessary.

Another requirement is to ensure that the speed of the game is in a reasonable range. As the byoyomi time management is very difficult to support in a collab-

[8] For example, it can be interesting to compare every move of two good players to the output of a computer Go program. This allows to identify where a computer program may still have some difficulties.

[9] There are a lot of tools named SGF Editors that allow to modify, comment and mark the information stored in the SGF. A good list of available products is available under: http://senseis.xmp.net/?SGFEditor

[10] This would help users to get a feeling for the game of Go especially when some arbitrary (and not always good) moves are mixed under the suggestions.

orative setup with multiple users, the collaborative player is using a much simpler time management with fixed time slots. This means that the software is enforcing a time limit of 30 seconds per action. After that, the user is considered not responding and the software discards the possible input from a not responding user.

3.5 Collaboration Schemes

The collaboration schemes are the heart of the collaborative player. They contain the information according to which the next move is selected. The current approach is to allow each user to provide a suggestion of the next move. After that, the decision engine selects one move out of the suggestions. This process can be with the support of the users (for example by voting) or the engine can decide completely by itself.

All decision engines must implement the DecisionMaker interface. This interface consists of only two operations. One to add a move as a proposal (`addMove`) and one to get the final move from the decision engine (`getMove`). If a decision engine wants to perform any operation including a user interaction, other Java classes may need to be changed too. The following two schemas show the different approaches that can be used in a decision engine to select a move. Of course they serve only as an example of what can be achieved and are not necessarily the best and the most interesting approaches to the problem.

3.5.1 Voting Schema

With the voting decision maker, a two phase decision engine was implemented. The first phase consists of a request for move proposals to each user. All suggestions are then collected and merged together in the decision engine. In a second phase, the decision engine informs all users about all moves that were proposed. To reduce influences from the user level, the moves are displayed anonymously with only one additional information, telling which move was proposed how many times. Because of the anonymity of the suggestions the user cannot just pick the move proposed from the best player in the team, but he has to select the best move according to his knowledge. If for some reasons different moves receive the same amount of votes, the system decides arbitrarily which move will be played. The Figure 3.4 explains how the control flow looks like. The clients are displayed in blue and the collaborative player in gray.

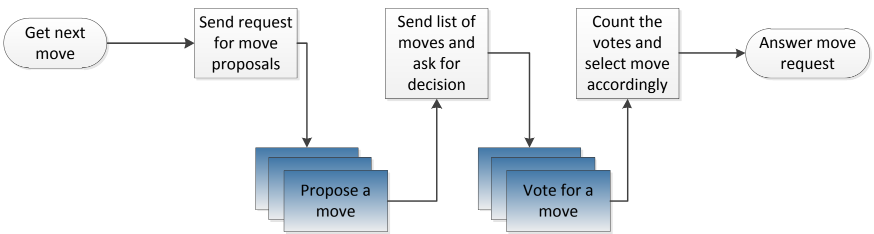


Figure 3.4: The control flow of the Voting Decision Engine.

3.5.2 Monte-Carlo Schema

As stated in Chapter 2, Monte Carlo is an important approach for current Go problems. To verify if this approach is also satisfying for a normal user when it comes down to selecting the best move from a list of suggestions, the second decision engine is based on a Monte Carlo approach. The first phase of the collaboration process is exactly the same as before. This means that each user is again queried for his proposal. After that, Monte Carlo methods are used to evaluate the estimated value of each move. The move with the highest value will then be selected by the engine.

Figure 3.5 explains the overall approach of the decision engine. The first part of the evaluation is to create clones of the board for each proposed move. The proposed move will then be played on one of the clones. This new board is then inserted into a simulator. This simulator plays a fixed number of pseudo random games to the end. In detail, each simulation process performs random moves up to a level, where no more legal moves are available. After that, a scoring function according to the used rule set in the game is used as a score estimator. This score is then interpreted as a value of the board before the simulation process. All score values of the simulated boards are then merged into a single value of the initial board state by using the arithmetic average. This score value is calculated for each proposed move. The decision engine then picks the one move that did result in the highest score value.

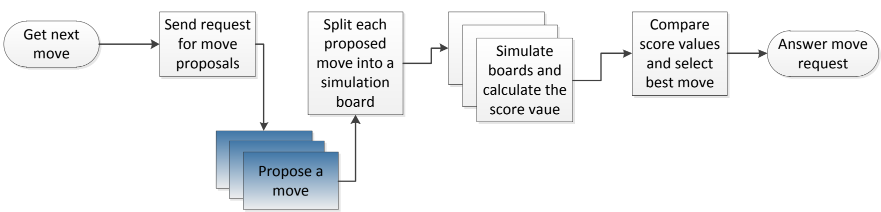


Figure 3.5: The control flow of the Monte Carlo Decision Engine.

3.6 Client

The client is the only part of the system that interacts with the user directly. The setup dialog is mainly used to enter the access code which is required to connect to the collaborative player. The user name is to identify a team member in the chat and in the member list. This has no effect on the game play, it just serves the need of the users to be able to distinct their team members clearly from each other or even matching them to a known person and therefore increase the fun factor of the game. The question about the user ranking is currently not used at all, as it might influence the user behavior during the game. Nevertheless, it is meant to allow a detection of distinct user behavior between different skilled players. Figure 3.6 shows the client software during an actual game.

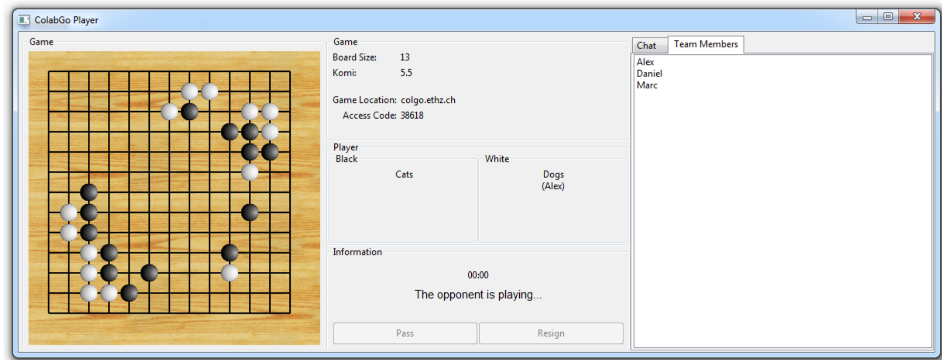


Figure 3.6: The graphical user interface provided to the player.

The whole client is based on the SWT toolkit[11] developed by eclipse. One of the main challenges during the implementation of the client was to create one single, executable JAR-File that runs on Linux, Windows and OS X. This is solved by including all possible SWT libraries (Linux, Windows, OS X each with 32 and 64 bit) and including the correct library after the system start by using reflection.

3.7 Web Framework

The web framework is designed for user groups that do not have the possibility to run the game server and the collaborative players on their own machine or just want to play a short game of collaborative Go. The main task is therefore to allow the participation in games of collaborative Go without worrying about any kind of software or setup configurations. This should lower the initial work to an absolute minimum and therefore increase the overall interest in the game. The only thing the web server does, is taking the desired settings of the groups provided in the web form and run the required parts of the software with these settings. Figure 3.7 shows the minimal requirements of the settings for playing a game of collaborative Go.

[11] SWT stands for the standard widget toolkit and is released by the eclipse foundation under the Eclipse Public License. More information can be found at: http://eclipse.org/swt/

Figure 3.7: The user interface for launching a new game of collaborative Go.

The framework behind the form is mainly responsible for managing the available ports for gaming. Other than that it just executes the JAR files of the game server or the collaborative player with the correct input parameters. Of course, the web framework takes also part of removing old processes. Especially because the launch of a game cannot be stopped after clicking on the "Start the game" Button. One has either to play the game to the end or wait for the web framework to kill the process after a certain time.

4 Experiments

Throughout the development process and the testing of the framework, we conducted several experiments. The experiments can be split up into two classes. The first one was a real user experiment with several experienced[1] and some intermediate players for gathering personal feedback. These tests allowed us to integrate the huge know-how from the players into the final software.

The second tests were more on a technical basis. An evaluation of the usefulness of the Monte Carlo Decision Engine was conducted and is explained in a graphical way in Section 4.4. One of the goals of these tests is to explain why the Monte Carlo approach works and why there is still a need for better ways to play computer Go. Other tests were conducted by using the gathered data during test runs in the Go Club Zurich. These tests are mainly used to underlay the personal feedback provided by the players. In Section 4.3, we took a look at the timing constraints of our system and how they were accepted by the players used to the classical Go time management.

In very few words, the overall feedback was very positive and especially the experienced players had a huge set of new ideas and features that may be integrated into the system.

4.1 Collected Data

During a game of Go with our software, each action performed by a player[2] is logged into a SQL-Database. As an action, we consider a user behavior which includes an integration of the other participants through a GTP call or a result of another complex

[1] An experienced Go player plays between three and five times a week. Also, he should be playing longer then three years. The ranking of the player should be in the low Dan level.

[2] Here, a player stands for a single user participating in a collaborative player and the collaborative player itself.

operation like a Monte Carlo simulation. Table 4.1 explains the logged data in detail. The logging features of the game server are required for analyzing and replaying the game. In the collaborative player, the collected data serves for evaluating the different collaboration schemes and taking a deeper look into the behavior of the users.

Action	Start of a new game
Logged by	Go server
Table	game
Type	insert
Data	*gameId* - Identifier of the game *startTime* - Time stamp
Action	Game is finished
Logged by	Go server
Table	game
Type	update
Data	*endTime* - Time stamp
Action	Move from a player
Logged by	Go server
Table	move
Type	insert
Data	*gameId* - Identifier of the game *color* - Color of the played stone *position* - Position of the player stone *timeStamp* - Time stamp
Action	Player joins a game
Logged by	Collaborative player
Table	userJoin
Type	insert
Data	*gameId* - Identifier of the game *name* - User name *userLevel* - Ranking of the user *team* - Team name *timeStamp* - Time stamp
Action	Player leaves the game
Logged by	Collaborative player
Table	userLeave
Type	insert
Data	*gameId* - Identifier of the game *name* - User name *team* - Team name *timeStamp* - Time stamp
Action	A request for the next move is sent to the players
Logged by	Collaborative player
Table	genmove
Type	insert
Data	*gameId* - Identifier of the game *player* - Name of the collaborative player *timeStamp* - Time stamp

Continued on the next page

Action	Answer of the move request is received
Logged by	Collaborative player
Table	moveResponse
Type	insert
Data	*gameId* - Identifier of the game *player* - Name of the team *moveId* - Number of the move *position* - Position of the player stone *timeStamp* - Time stamp
Action	Request for vote is sent to the players
Logged by	Collaborative player
Table	vote
Type	insert
Data	*gameId* - Identifier of the game *player* - Name of the team *moveId* - Number of the move *timeStamp* - Time stamp
Action	Response of a voting request is received
Logged by	Collaborative player
Table	voteResponse
Type	insert
Data	*gameId* - Identifier of the game *player* - Name of the team *moveId* - Number of the move *position* - Position of the player stone *timeStamp* - Time stamp
Action	Monte Carlo engine has selected a move
Logged by	Collaborative player
Table	decision
Type	insert
Data	*gameId* - Identifier of the game *player* - Name of the team *moveId* - Number of the move *position* - Position of the player stone *timeStamp* - Time stamp

Table 4.1: The data which gets captured during a game of collaborative Go.

4.2 Collaboration Schemes

The first question we needed an answer to, was whether the two implemented collaboration schemes are interesting from a user perspective. This is not an obvious question because there are no games involving players with such collaborative schemes. Additionally, this question is crucial to the success of the software and the idea of collaborative gaming, because games are mostly played because of the fun they provide. If the collaboration process is boring or disappointing the players, then the game will never become popular.

This experiment was conducted with two very distinct user groups. One user group consisted of six experienced people of the Go Club Zurich and the other of complete novices to the game of Go. The idea behind these two groups was, that the feedback provided from the players is valid for all kinds of Go players regardless of how long they have been playing Go. The Monte Carlo decision schema was only tested on the experienced Go players, because we are mainly interested in the fact whether the moves selected by the engine are reasonable[3] or not.

For both user groups, the chat functionality was the feature that lead to the biggest discussions. Both groups wanted to send chat messages to the opponent because this would increase the fun of the game especially when all participants know each other. The experienced group also mentioned that the chat should also provide the players with the possibility to discuss and analyze the game after it is finished.

4.2.1 Monte Carlo Decision Schema

The Monte Carlo decision was found very fair, because both teams had the same evaluation function for their proposed moves. All six players stated that the evaluation lead to some strange moves. Especially the moves located near to the already played stones seem to be preferred by the evaluation. The main conclusion was that when it is automatically decided which move is picked, the team needs to be informed why this move was picked. This can be done by providing all values of the Monte Carlo evaluation to all players. Also it would be nice to see what moves were proposed by the other players, even when there is no possibility to influence the decision engine. Furthermore, the engine must be very careful when a user suggested a pass move. This is because a pass is only good when all groups on the board are alive or dead. In all other situations the play of a pass is a huge disadvantage. It was suggested that a pass or resign move is only selected when all provided move suggestions are pass or resign moves.

4.2.2 Voting Decision Schema

The voting decision was found very interesting and catching for both the good and the bad players. This is, because both can learn a lot about the game and the handling of the different situations without taking full responsibility when something goes wrong. Especially at the group of the good players, it showed how different certain situations can be handled and that it is not always clear which move is the best one.

During the test run of the experienced group, one team was locally separated and only connected trough the Internet, the other one was sitting right next to each other but without seeing on the neighbors screen. The members stated that for having success as a team, sometimes the communication channels are required. The separated group made therefore heavy use of the chat function whereas the other group mainly communicated through voice. One tester also requested the option that the chat can be completely disabled during the game and gets active only after the end of the game. This may generate situations - especially when there are

[3] It is for non-experienced players very difficult to decide whether a played move was the best of all the options or not. This is because a rule of thumb says, that one should always play the move that conquers the biggest size. For deciding how many stones one can achieve by a given move, a lot of game experience and know-how is required.

huge experience differences in the team - in which it is unclear on how to perform and the effect of going one or the other path would be clearer. One huge disadvantage without the chat function is, that the team has to consider a simpler strategy because it is not possible to agree on the next twenty moves. In complex scenarios, one single bad move can lead to a whole different situation and therefore the focus needs to be much closer to the current situation taking one move after the other.

4.3 Time Management

For enforcing the going-on of the game, a time management has to ensure that the players perform the corresponding actions in a certain period of time. Therefore we had to implement a system easily understandable to all kinds of players that also fulfills the requirements of experienced Go players. The time management implemented in our system is different to what has been used so far in Go playing tournaments. Because of that, we were interested in whether our setup also matches the needs of the different kinds of players. The longest possible period one team has to wait for the action of the other team is one minute. This can happen when at least one player misses the time limit in the move suggestion as well as in the voting phase of the collaborative process. For testing our setup, we asked both test groups the same questions regarding the time limits. Both considered the waiting period of at most one minute as reasonable. Especially since it is the case that when the action was performed faster from all players no one has to wait for the end of the time limit. In order to check whether the provided answers match our collected data, we looked at the number of exceeded time limits per team during a game. The Figure 4.1 shows the results of the experienced players. We do not show the results of the beginner group, as this group never exceeded one single time limit. This is mainly because beginners think less about their actions and play normally guided by their instinct, which leads to a very short response time for each action.

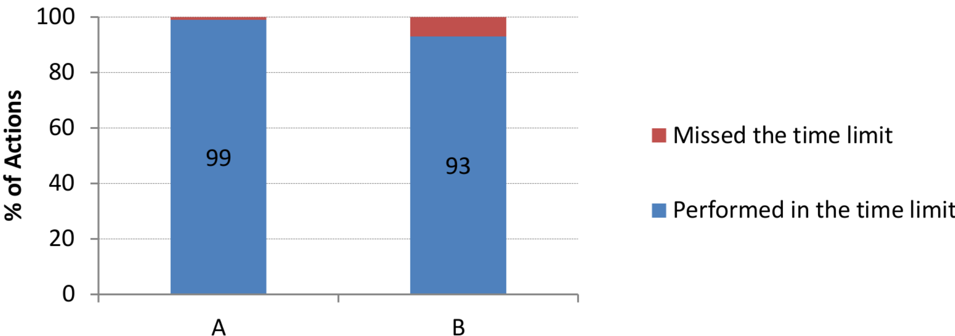


Figure 4.1: The percentage of actions performed in the fixed time limit of 30 seconds. Team A communicated only through the chat of the GUI whereas team B sat on the same table and was able to discuss all actions.

The experienced players mentioned that in the regular case a time of 30 seconds is long enough for suggesting a move or making a selection. But during a game of Go, there are always some moves which may have a huge impact on the game and where a good player likes to think more about the current situation. For supporting such a game play the byoyomi time management would be perfect. Also, a com-

pletely different approach based on something close to the fisher time[4] could be a possible solution. This would mean that you can increase the time available for the next action by performing the current action faster then the given time limit. Of course it is unclear whether this approach is really working with a large amount of users. An interesting fact is that the missed time limits from team B are mostly due to a heavy discussion about the current situation and therefore not realizing that an action needs to be performed. This was later in the game solved by enabling the system speakers of the devices running the client software. The system beep then reminded the players that an action needs to be performed.

As a final result regarding the time management, one can say that the existing approach works for all kinds of players independent of their skill level. The experienced players prefer - as expected - the more complex time managements they know from various online servers and tournaments. The beginners don't care about time management at all.

4.4 Monte Carlo

An interesting question is, whether the Monte Carlo engine selects a useful move. For testing whether the selection process is of any use, we decided to create a test in which we replayed a game of Go between two high ranked KGS players. The game we took is available on the KGS online server. Two 7d players[5] played against each other. For simplifying the test to the reader, we will only look at the black players moves and actions. The figure 4.2 shows the final board of the game.

[4] Fisher time is a time management named after its creator Bobby Fischer. The basic idea is that you gain with every action performed a small amount of time. If you perform actions very fast the amount of time available will steadily grow whereas a slow player constantly looses time. A detailed explanation can be found at: http://senseis.xmp.net/?FischerTiming

[5] A ranking of 7d stands for a strong and experienced player. As this is the KGS ranking, the real level may be around 5d. This is because the algorithm of KGS constantly produces levels that are a bit better than the real one we are used to in Switzerland.

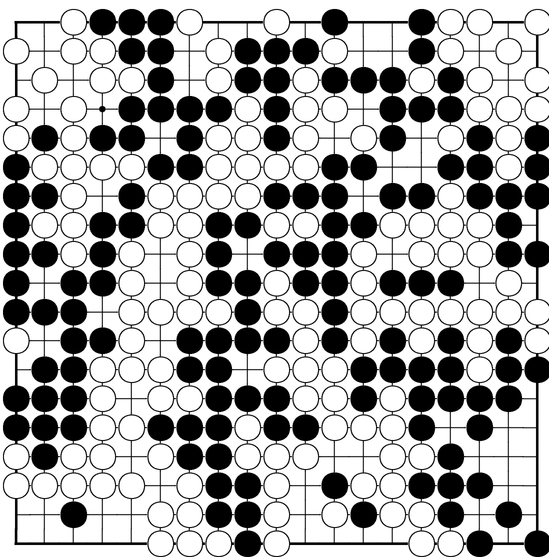


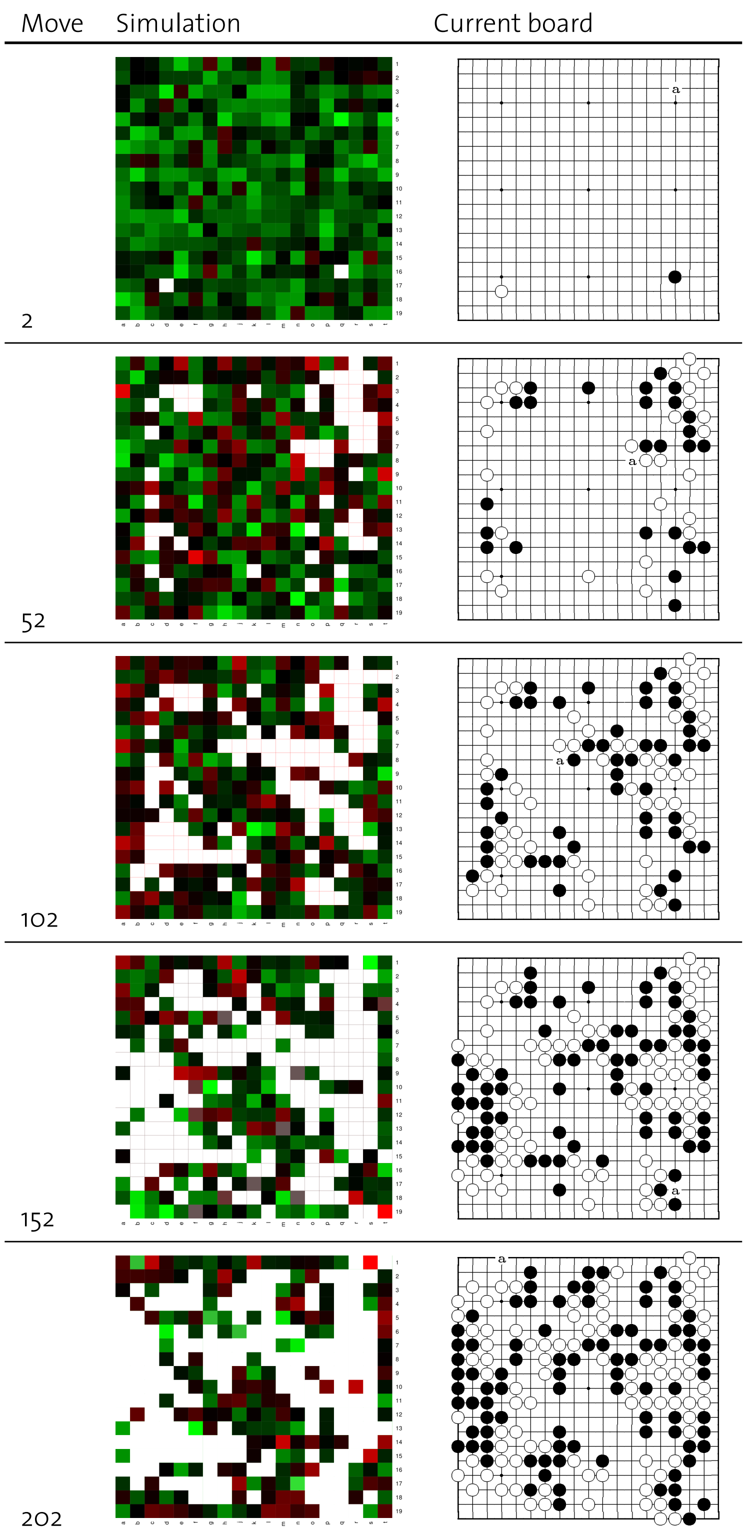
Figure 4.2: Final state of the Go game between likam2 (white) and twoeye (black), both ranked 7d, used for testing the Monte Carlo engine.

The experiment was based on several snapshots we took during this game. Basically, we stopped after a certain amount of moves and then looked at the current board state. First, we extracted all possible moves[6] and then we simulated all moves with our Monte Carlo engine. This process was repeated with several different setups of the simulator. The first results showed in Table 4.3 were created with the following setup displayed in Table 4.2. The heat maps on the left show the values for positions in the corresponding color. For a deeper understanding of the process, it is important that it is clear that the resulting score value of the simulation may be way off compared to the real result of the game. This can be seen in the simulation of the move 302, where most results are colored in red which stands for a loss around -30 points. In the end, black wins the game with 5.5 points despite of the fact that according to our simulation the result should be around W+30. This is because the simulation does not have a good evaluation function that can decide whether a group is alive or dead. For the decision which move is the best one, only the differences between all the moves are relevant. Therefore, it is possible to select a move even without knowing how much it is actually worth. The important fact is, that it is worth more than all other moves.

Game	likam2 (w) vs. twoeye (b) - 04.05.2011 on gokgs.com
Final score	B+5.5
Simulated moves	2, 52, 102, 152, 202, 252, 302
Number of simulations	100

Table 4.2: The simulation setup of the first game.

[6] As a possible move we considered all moves that are legal on the current board, except the ones that filled own eyes.



Continued on the next page

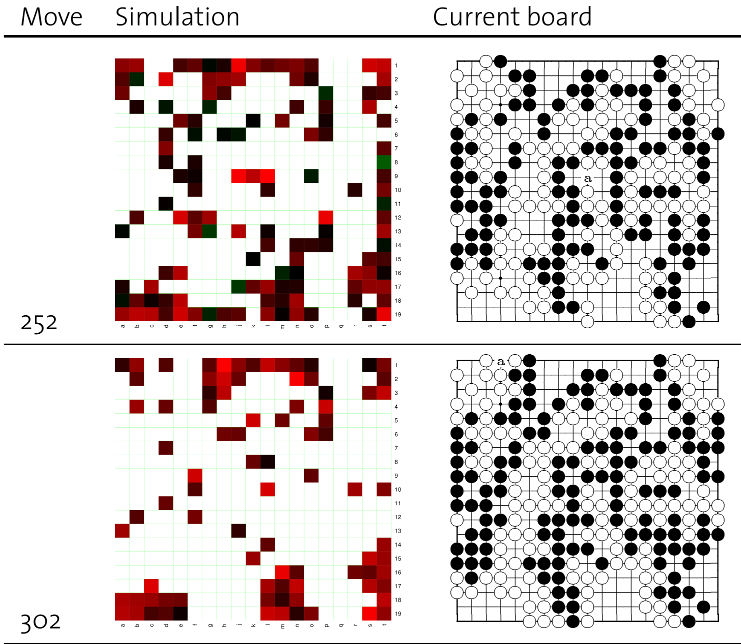


Table 4.3: The results of the Monte Carlo simulations. The white spots in the heat maps indicate spots where due to the illegal position no scores are available. Green represents a high simulation value and red stands for a very low one. The positions marked by *a* are the locations where black played the next move.

When looking at the heat maps, one is able to spot positions on the board where black can gain multiple points. Let us take a deeper look into the simulation of the 252nd move. The Monte Carlo simulation detects on T8 a good move with a value much higher than all other options. When we now take a look at the board state fifty moves later, we detect that the area around T8 was played and conquered by the black player. Also we can see that the actual best move proposed by T8 was never played. Instead, black played all moves around this spot. T7 would be a much better move because it connects the two groups of black stones lying next to each other. Nevertheless, the Monte Carlo method was able to correctly detect a hot spot on the board.

As we are highly interested in whether this bad behavior is a problem of the Monte Carlo approach or if the number of simulations was just to small to correctly detect and evaluate the situation, we conducted a second experiment with almost the same setup. We will simply increase the number of game simulations from 100 to 1000 games per move. The rest of the setup will stay the same as before. The Table 4.4 shows a reduced set of the results from the second experiment. As we can see, not much has changed. The differences in the color range are now much smaller. This is due to the fact that the results are much smoother because of the increased number of simulations. When we now look at the situation around T8 we can detect some new results. First of all, T8 is no longer the best available move. The values of the moves from T7 to T9 are very close together. The two moves that attract attention are T5 and T10. T5 is very bad, because after a black stone was placed at T5 white can play T7 and kill the two black stones in between. T10 is considered a good move to the simulator as it appears to block the white player from accessing the T row. Increasing the number of simulations has shown to decrease the standard deviation. During the first experiment, the standard deviation was around 40 points. When

increasing the amount of simulations to 1000 games, this value drops to 6 points.

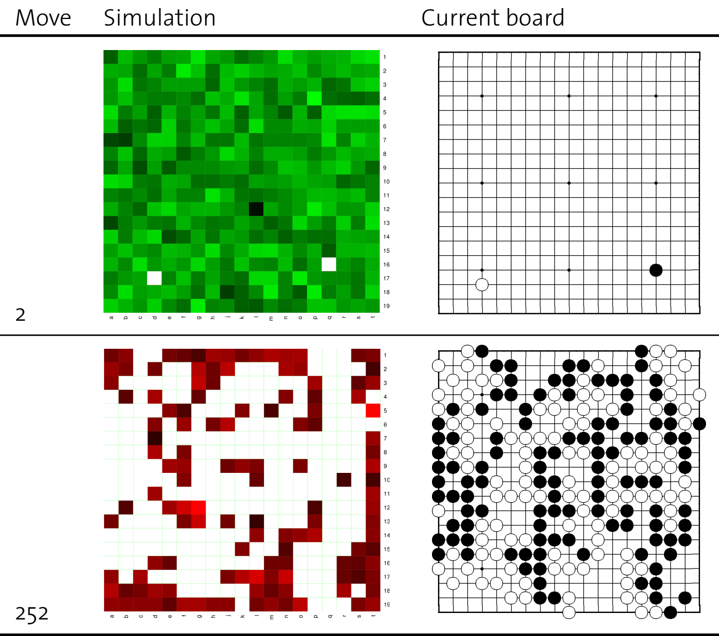


Table 4.4: The results of the Monte Carlo simulations. Only the second and the move 252 are displayed. By increasing the number of simulations the results get smoother and more accurate.

Table 4.5 shows the simulated values of the moves the black player actually played and compares them to the overall values of the simulation. An important fact to know is that a Go player not always plays the best move first. Sometimes moves that preserve the order of playing are inserted. These moves are called to have "sente" which will force the opponent to answer to the played move and therefore preserve the previous situation on other board locations. During the endgame, these moves are much more important than in the beginning of the game. When looking at the detailed scores, we detect that (except for the move 202) the simulation agreed that the selected move is better than a random move. Our Monte Carlo decision engine therefore gets better results from a team than an arbitrary decision engine.

Move	Played position	Value	Worst	Best	Average	# of better moves (Total moves)
2	Q3	5.9	2.5	6.7	5.2	95 (346)
52	N8	7.4	0.7	8	5.9	76 (301)
102	H8	15.0	-0.2	15.3	8.3	7 (256)
152	Q17	4.4	-9.2	9.5	3.2	84 (213)
202	D1	-13.4	-26.3	-1.3	-9.5	97 (181)
252	K8	-29.4	-47.0	-2.6	-28.3	62 (129)
302	D1	-29.9	-56.4	-4.5	-36.3	15 (85)

Table 4.5: The simulated values of the played moves by black and some statistics on the calculated scores of the simulations for each board.

Throughout these experiments, we learned that Monte Carlo is indeed a reasonable way for implementing an automated decision engine. This is especially because the engine does not have to choose a move from all possibilities but receives a small set of moves and can pick the best out of them. The Figure 4.3 shows a situation that is very bad for all kinds of Monte Carlo engines, because the order in which the moves are played is crucial to the success. But when the player suggests the move, it is very easy for the engine to detect that the suggested move is indeed a good one.

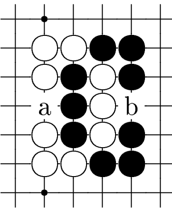


Figure 4.3: Situation, where it is difficult for a Monte Carlo engine to select the correct move. This is because the value of the black move at *a* and the white move at *b* is mainly based on which move was played first.

5 Conclusion

The main goal of this thesis was to build a system that allows to evaluate different collaboration schemes by playing the game of Go. Due to the lack of an existing Go library, the first step was to build a reusable Java library with the complete game logic required for gaming. Throughout the elaboration of the system, we experienced that an easy to use and open source library supporting the basic features of the game of Go is a need of the current Go community. Of course, implementing the game from scratch was not part of the initial time table and therefore caused a reduction of the time that was spent developing and testing alternate collaboration schemes. Nevertheless, the implemented system fulfilled all requirements we had in our initial draft. Especially the separation of all system parts (server, collaborative player and client) allowed us to be very flexible on what can be achieved with our software.

As a second step, we took a deeper look into the use of Monte Carlo methods for playing Go. The conducted experiments provide an understanding on why Monte Carlo is such a wide spread topic in computer Go. Also the fact, that thanks to the Go library, a simple program running some simulations on a board can be implemented very easily, allows interested Go players without a computer science background to understand and evaluate this approach by themselves.

Since most of the Go players are interested in the current research in computer Go, we learned that once the players do know about a new system, they are quite interested to use it. A huge drawback is the current lack of popularity in Switzerland and the rest of Europe. Because of that, it was quite a challenge to advertise the implemented system and motivate the players to participate in the collaborative Go project. Especially since for an interesting collaborative Go game at least six players are required.

5.1 Future Work

Since one of the most mentioned requirements of the local Go scene was an open source library implementing the logic and some basic functionalities of the game of Go, the integrated Go library could be extended and completed by some interesting features like a score estimator or a life/death decision tool.

For increasing the popularity of the collaborative Go, some new features especially on the client side will be required. For some users, it is important to play the game without downloading a software. They would like to play the game directly through the browser. Furthermore, an integration into a social platform like Facebook or Google+ would increase the popularity by allowing known user groups competing with each other. The other approach for increasing the popularity is to provide a mobile solution which runs on current mobile devices with iOS or Android. Because these applications are mostly installed from the corresponding market places, a fancy user interface would be required in order to get popular. A high ranking in the markets could automatically attract new players even when they are not integrated in a local Go community that already knows about the collaborative Go project.

The last (and from a technical perspective the most interesting) extension of the current work will definitively be some new collaboration schemes. These collaboration schemes could integrate some different approaches like marking hot spots on the board or highlighting groups with low liberties. An extended version of the current collaborative schemes may involve schemes that adapt the task a user needs to perform in each move according to his skill level. Also an interesting point would be, whether it is possible to create a collaboration schema in which the players perform better as a team than the best player by himself.

Bibliography

[1] B. Bouzy and B. Helmstetter. Monte-Carlo Go Developments. In *ACG. Volume 263 of IFIP*, pages 159 – 174. Kluwer Academic, 2003.

[2] B. Bruegmann. Monte carlo go, 1993.

[3] J. Burmeister and J. Wiles. CS-TR-339 Computer Go. Technical report, Department of Computer Science, The University of Queensland, 1995.

[4] W. G. Chase and S. A. Herbert. Perception in chess. *Cognitive Psychology*, 4:55–81, 1973.

[5] G. Farnebaeck. Specification of the Go Text Protocol, version 2, draft 2. http://www.lysator.liu.se/~gunnar/gtp/, 2002.

[6] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo go. Technical report, 2006.

[7] R. Hunter. Byoyomi explained. *British Go Journal*, 106:43 – 44, 1997.

[8] D. Lichtenstein and M. Sipser. Go is polynominal-space hard. *Journal of the ACM*, 27:393–401, 1980.

[9] M. Mueller. Computer Go. *Artificial Intelligence*, 134, 2002.

[10] J. S. Reitman. Deducing memory structures from inter-reponse times. *Cognitive Psychology*, 3:336 – 356, 1976.

[11] N. N. Schraudolph, P. Dayan, and T. J. Sejnowski. Temporal Difference Learning of Position Evaluation in the Game of Go. In *Advances in Neural Information Processing Systems 6*, pages 817–824. Morgan Kaufmann, 1994.

[12] J. P. Zagal, J. Rick, and I. Hsi. Collaborative games: lessons learned from board games. *Simul. Gaming*, 37:24–40, 2006.

[13] A. Zobrist. *Feature extraction and representation for pattern recognition and the game of go*. PhD thesis, The University of Wisconsin - Madison, 1970.

Glossary

The following list explains the most common Go terms used in this thesis.

block	A block of stones are connected stones of the same color.
board	The board on which the game is played. Normally a size of 19x19 is used, but 9x9 and 13x13 are also very popular.
byoyomi	A very popular time management system.
connection	Two stones are connected when they share at least one side. Stones that are diagonally adjacent to each other are not connected.
dead	A group of stones that cannot survive an attack by the opponent.
eye	Empty spot on the board surrounded by stones of only one color.
gote	A move has gote when the move was forced by a sente move.
group	A group are connected stones of the same color.
ko	A repetition of an old board state by a cycle of two moves.
komi	A fixed number of points (normally 5.5 or 6.5) to the player not starting the game, because of the first players advantage.

liberty A free spot next to a stone or a group. The liberty of each stone in a group is the sum of the liberties for all individual stones.

life A group of stones with at least two eyes or a group that can survive any attack by the opponent.

sente A move has sente when it forces the opponent to respond.

stone Marker of one team in either white or black color.

superko A repetition of an old board state by a cycle of more than two moves.