Adaptive Playout Policies for Monte-Carlo Go

Hendrik Baier

10.8.2010

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Osnabrück, August 2010
Hendrik Baier

Contents

List of Figures

Abstract

Go is a classical board game originating in China at least 2500 years ago. It is surrounded by a rich culture and tradition and famed for its profound depth and complexity arising from very simple rules. Go is deterministic, fully observable, state and action space are both discrete and finite; yet despite 40 years of efforts, writing a program to play Go on the level of human experts still stands as one of the grand challenges of AI. Techniques developed for such a program are likely to be applicable to many other game and non-game domains.
The traditional game programming technique of Alpha-Beta search with static evaluation functions does not apply well to Go: Its game tree is too large to be traversed even at shallow depths, and its positions are too dynamic to be effectively evaluated mid-game.
The dominant paradigm for computer Go players, Monte-Carlo Tree Search, uses statistical sampling to make move decisions. It gradually deepens a search tree in a best-first fashion by playing a large number of simulated games–playouts–and averaging their outcomes as an approximation to the value of a position. Playout moves in unknown situations are chosen according to quasirandom policies which favor good-looking moves without sacrificing sampling diversity.
The specific probabilistic policy used for selecting moves during playouts is of vital importance to the success of Monte-Carlo simulation in Go. So far however, such playout policies have generally been static, i.e. they have only exploited general expert knowledge or information learned offline from game databases or self-play. In this thesis, adaptive or dynamic playout policies are explored that take advantage of knowledge learned directly from previous playouts in the search. In this way, the sampling process improves itself even beyond the leaves of the search tree, leading to a significant increase in playing strength.

Acknowledgements

I first want to thank my thesis advisors Johannes Fürnkranz and Kai-Uwe Kühnberger who accepted my thesis proposal and made this work possible by offering their advice and support.
Furthermore, I would like to express my gratitude to Peter Drake for the considerable amount of time he invested in answering my early questions concerning many aspects of doing research on computer Go; for the interesting discussions and fruitful cooperation that followed; for reading my thesis and giving countless constructive suggestions to improve it; and of course for the continuing development of OREGO, the program used for all experiments in this thesis.
Moreover, I would like to thank Udo Waechter, Martin Schmidt, Friedhelm Hofmeyer, Dirk große Osterhues and Sang-Hyeun Park for their technical support and their efforts to solve all problems surrounding testing and evaluation.
I am grateful to Artus Rosenbusch for introducing me to the beautiful game of Go many years ago.
And finally, I would like to thank Corinna Zennig for her neverending support, her patience and her understanding whenever I could not stop thinking about Go; and for being the reason whenever I could.

Conventions

The following conventions are used throughout this thesis:

- Pseudo code is written in a typewriter-style font.

  ```
  print(``This is pseudo code'');
  ```

- Hat notation denotes estimates.

  $\hat{x}$ is an estimate of x.

- Small capital letters denote names of programs.

  This thesis is based on OREGO.

- Curly brackets denote nested if-then-else-statements. The cases are used top-to-bottom if the corresponding conditions apply.

  $$x \leftarrow \begin{cases} 3 & \text{if it is Wednesday} \\ 5 & \text{if it is June} \end{cases}$$

  On a Wednesday in June, the above statement assigns 3 to x. The corresponding pseudocode is

  ```
  if(date.day==wednesday) {
     x = 3;
  } else if(date.month==june) {
     x = 5;
  }
  ```

Chapter 1

Introduction

"Sooner or later, AI will overcome this scandalous weakness."

—John McCarthy

1.1 Artificial Intelligence and Games

Games have been a cornerstone of AI research ever since the early pioneering days. Only four years after the construction of ENIAC, the world's first general-purpose electronic computer, Claude Shannon published a paper on computer chess (Shannon, 1950). The checkers program written by Christopher Strachey at the University of Manchester in 1951 was one of the first successful AI programs. Classic board games like chess and checkers with their well-defined rules not only provide ideal abstractions from real-world situations, so-called "toy problems", and allow for straightforward comparison of algorithm performance. They also have an intuitive appeal to the general population, and their mastery is considered the epitome of intelligence and rational thought by many. Before the term *artificial intelligence* was even coined in 1956, researchers had been fascinated by the idea of challenging human intellectual supremacy by teaching a computer how to play.

With the optimism of the "golden years" of AI, Herbert A. Simon and Allen Newell predicted in 1958 that "within ten years a digital computer will be the world's chess champion" (Simon and Newell, 1958). It turned out to take three more decades to reach that level. But throughout these years and beyond, chess and other games have proved excellent test-beds for new ideas, architectures and algorithms, illustrating many important problems and leading to successful generalizations for work in other fields (Lucas, 2008; Lucas and Kendall, 2006). Chess became, with the words of the Russian mathematician Alexander Kronrod in 1965, the "*drosophila* of artificial intelligence" (McCarthy, 1990). Raj Reddy called the game an "AI problem par excellence" in his presidential address to AAAI in 1988, listing computer chess together with natural language, speech, vision, robotics and expert systems (Reddy, 1988).

Over the last decades, many of AI's most notable successes were connected to game-playing (Schaeffer and van den Herik, 2002). In 1994, the World Man-Machine Championship in checkers was won by the program CHINOOK (Schaeffer et al., 1992; Schaeffer, 1997). In 1997, the dedicated chess computer DEEP BLUE defeated reigning World Chess Champion Garry Kasparov under standard tournament conditions (Campbell et al., 2002; Hsu, 2002). A few months later, World Othello Champion Takeshi Murakami lost to the program LOGISTELLO (Buro, 1997). AI has also tackled *imperfect-information* games, which pose the additional problem of partially non-observable game states, and *stochastic* games, which include an element of chance. TD-GAMMON only barely lost to World Backgammon Champion Malcolm Davis in 1998, and programs have since grown stronger (Tesauro, 2002). Matt Ginsberg's GIB is playing bridge on master level (Ginsberg, 2001).

However, many challenges remain: In Poker, for example, programs have to deal with hidden information as well as opponent modelling for multiple players to understand basic "bluffing" tactics (Billings et al., 2002). New board games have been invented (Teytaud and Teytaud, 2009; Anshelevich, 2002; Lieberum, 2005)—some of them explicitly constructed to be difficult for traditional game-playing algorithms, encouraging new approaches e.g. for large action spaces (Fotland, 2004). Researchers have begun work on

General Game Playing (GGP)—the quest for a program that is able to play not only one specific game, but all games whose rules can be defined in a Game Description Language (Genesereth et al., 2005). Supported by a growing industry, computer games with continuous state and action spaces call for more intelligent and believable artificial opponents and allies (Laird and van Lent, 2001; Buro, 2004). Team games in real-time settings foster research into AI and robotics (Kitano et al., 1998). And in the classical games of Shogi and Go, human masters are still far stronger than any program written to date.

50 years after their publication, Arthur Samuel's words still hold true (Samuel, 1960):

> Programming computers to play games is but one stage in the development of an understanding of the methods which must be employed for the machine simulation of intellectual behavior. As we progress in this understanding it seems reasonable to assume that these newer techniques will be applied to real-life situations with increasing frequency, and the effort devoted to games (...) will decrease. Perhaps we have not yet reached this turning point, and we may still have much to learn from the study of games.

1.2 The Challenge of Go

As noted in Silver (2009), computer Go is in many ways the best case for AI. Other than in GGP, the rules of the game are known; compared to chess, they are simple; unlike those of backgammon, they are deterministic; other than in poker, the state is fully observable; different from many video games, state and action space are both discrete and finite; and other than in newly invented board games, strategy and tactics of Go are well-known and have been developed over centuries. Except for rare capture moves, board positions change incrementally, one stone at a time,

until the game terminates after a finite number of moves with a binary result.

The immense popularity and rich tradition of Go with millions of players and a thriving scene of professionals—in particular in Japan, China and Korea—has stimulated considerable research efforts in computer Go over the last decades, second only to computer chess. From 1985 to 2000, the Taiwanese Ing Foundation sponsored an annual computer Go event, offering a top prize of over one and a half million dollars for the first program to defeat a professional player. Yet while 10 years after DEEP BLUE's much-noticed success against Kasparov, chess is now played on super-human level by software on regular desktop computers (McClain, 2006), Go AI is still in its infancy, with the best programs only recently achieving low amateur master strength. The Ing Prize remained unclaimed.

The challenge of computer chess has been answered successfully by the combination of fine-tuned domain-specific position evaluation functions with exhaustive game tree search—made highly efficient through pruning techniques like the Alpha-Beta algorithm (Knuth and Moore, 1975) and flexible through selective search extensions like quiescence search (already predicted in Shannon (1950)). Since it typically involves the evaluation of millions of chess positions per second, some have described this approach as "brute-force"; and regarding chess as the "drosophila of AI", John McCarthy remarked disappointedly, "Computer chess has developed much as genetics might have if the geneticists had concentrated their efforts starting in 1910 on breeding racing *Drosophila*. We would have *some* science, but mainly we would have very fast fruit flies" (McCarthy, 1997).

However, computer chess methods turned out to be far from universally applicable even in the class of deterministic, two-player, zero-sum, perfect-information games. Go has so far resisted all known variants of brute-force search, not only due to the sheer size of its state space, but even more so due to the seemingly impregnable subtleties of position evaluation (see section 3.2). For many researchers, Go has therefore replaced chess as a new *drosophila*, a new *grand challenge* of AI (Schaeffer and van den Herik, 2002; Kitano et al., 1993; Cai and Wunsch, II, 2007). Go program-

ming has not yet become engineering, "Go sends investigators back to the basics—to the study of learning, of knowledge representation, of pattern recognition and strategic planning" (Mechner, 1998).

In the words of James Hendler (Hendler, 2006):

> Exploring the things that humans can do, but that we can't yet even imagine how to get computers to perform, is still the hallmark and challenge of AI. Pursuing it will keep the next 50 years of our field just as exciting as the past half century has been. In short, it's time to learn how to play Go.

1.3 Monte Carlo Tree Search

Monte Carlo simulation algorithms base their decisions not on exhaustive search of states, but on statistical sampling of *possible* states or future states (see section 2.4). They have seen their first successful game-related use in nondeterministic games for the sampling of random elements like dice or playing cards, and in partially observable games for the sampling of hidden information (Billings et al., 2002; Sheppard, 2002; Tesauro and Galperin, 1996). Applying simulation to a game without any hidden or random elements seems unintuitive; but in combination with selective tree search (Gelly et al., 2006; Coulom, 2006), this idea has increased the playing strength of top computer Go programs from that of a weaker club player to that of an amateur master.

After its introduction in 2006, *Monte Carlo Tree Search* (MCTS) has quickly become the dominating paradigm in computer Go. The algorithm searches for the best move in a given position by selectively growing a search tree, guided by repeatedly simulating complete games from the current position (see section 2.7 for a detailed description). The success of this approach lies in the ability of handling huge game trees through sampling; in the elegant management

of uncertainty through confidence intervals around move evaluations; in the asymmetric tree growth, deepening the search in the direction of the most promising moves; and in the *anytime property*, allowing the user to stop the algorithm at "any time" and retrieve the best move suggestion so far (Gelly, 2007).

However, MCTS has its shortcomings. Its simulated games (or "playouts") are based on simple probabilistic policies—since simulations need to be fast and diverse, move choices are usually relatively random. Integrating domain knowledge into the playouts has proven to be difficult without creating too much determinism and bias (Gelly and Silver, 2007; Silver and Tesauro, 2009). Thus, MCTS essentially answers the question: If I was to move here, would that improve my winning chances given relatively random play by both players?

On the one hand, this enables the algorithm to assess the overall strategic quality of a move without making assumptions about playing styles. On the other hand, it leads to relatively bad tactical play, because narrow sequences of required moves—often obvious to human players—are not correctly simulated (see section 4.1.2).

The primary question of this thesis is whether the problem of narrow sequences can be effectively solved by *dynamic playout policies*, i.e. by policies that are able to learn on-line from preceding playouts in the search process.

1.4 Overview

In the first part of this thesis, the literature is reviewed and the necessary background for this work is introduced:

- Chapter 2 describes the general framework of reinforcement learning and the place Monte Carlo Tree Search holds within it.
- Chapter 3 presents the application field of computer Go and reviews previous literature on simulation

policies for Monte Carlo methods in Go.

- Chapter 4 clarifies the research problem and states the goal of this work.

In the second part of the thesis, experiments on Monte Carlo Tree Search in Go are presented, divided into three groups:

- Chapter 5 deals with the first approach, the idea of collecting and using statistics about optimal move replies during search.
- Chapter 6 reports on the second approach, the idea of collecting and using single examples of successful move replies to any given opponent move, instead of trying to find the best one.
- Chapter 7 contains documentation of other experiments related to various aspects of Monte Carlo Tree Search.

The third part of the thesis, consisting of chapter 8, concludes by summarizing results and contributions, and looks ahead to further work.

Chapter 2

Reinforcement Learning

Reinforcement learning is the study of learning from interaction how to achieve a goal. It deals with the problem of learning optimal behavior, without being given descriptions or examples of such, solely from acting and observing the consequences of actions. The classical reinforcement learning task consists of an interactive loop between a learning *agent* and its *environment*: The agent repeatedly observes its situation—the *state* of the environment—, chooses an *action* to perform, and receives a response in form of a numerical *reward* signal, indicating success or failure. In most interesting cases, the agent's actions can also affect the next state. Trying to maximize its cumulative reward in the long run, the agent therefore has to learn by trial-and-error how his action choices influence not only immediate, but also delayed rewards.

This simple problem formulation, more precisely specified in section 2.2, captures the elementary aspects of learning from interaction with an environment. Its study has led to a variety of algorithms with strong theoretical underpinnings and notable practical successes: From animal behavior (Sutton and Barto, 1990) to helicopter control (Abbeel et al., 2007), from marketing (Abe et al., 2004) to vehicle routing (Proper and Tadepalli, 2006), from trading (Nevmyvaka et al., 2006) to spoken dialogue systems (Singh et al., 2002), from brain modelling (Schulz et al., 1997) to robot soccer (Stone and Sutton, 2001). This chapter introduces

some of the basic concepts of reinforcement learning, with emphasis on the class of Monte Carlo methods which have revolutionized computer Go in recent years. This forms the foundation for the presentation of Monte Carlo Tree Search.

2.1 Exploration and Exploitation

An agent faced with the reinforcement learning problem has to learn from its own experience, without explicit guidance or supervision. One typical challenge of this task is the tradeoff between exploration and exploitation. *Exploration* means the trial of new behaviors, the choice of actions that have not been tried before, in order to determine their effects and returns. *Exploitation*, on the other hand, denotes the choice of actions that are known to be successful, the application of learnt knowledge, in order to generate the maximal reward.

In stochastic environments, where the reward to a given action in a given situation is a random variable, many samples may be needed to estimate the expected rewards of the agent's options. Finding the optimal balance between exploration and exploitation, determining when to trust in acquired knowledge and when to seek for more information or certainty, is a non-trivial task.

2.1.1 The Multi-Armed Bandit Problem

The simplest setting in which this task can be studied is the case of the one-state environment. In the co-called *multi-armed bandit* problem (Robbins, 1952), an analogy is drawn to slot machines, casino gambling machines also known as "one-armed bandits" because they are operated by a single lever. The multi-armed bandit problem confronts the gambler with a number of arms instead, each of which provides a reward drawn from its own probability distribution. The gambler, initially without knowledge about the expected values of the arms, tries to maximize his total reward by repeatedly trying arms, updating his estimates of the re-

ward distributions, and gradually focusing on the most successful arms. Exploitation in this scenario corresponds to choosing the arm with currently highest estimated value; exploration corresponds to choosing one of the seemingly suboptimal arms in order to improve its value estimate, which may lead to greater accumulated reward in the long run.

Formally, the multi-armed bandit problem is defined by a finite set of arms or actions $A = \{1, \ldots, a\}$, each arm $a \in A$ corresponding to an independent random variable X_a with unknown distribution and unknown expectation μ_a. At each time step $t \in \{1, 2, \ldots\}$, the gambler algorithm chooses the next arm a_t to play depending on the past sequence of selected arms and obtained rewards, and the bandit returns a reward r_t as a realization of X_{a_t}. Let $T_a(n)$ be the number of times arm a has been played during the first n time steps. Then the objective of the gambler is to minimize the *regret* defined by

$$\mu^* n - \sum_{a=1}^{A} \mu_a E\left[T_a(n)\right] \tag{2.1}$$

where $\mu^* = \max_{1 \leq i \leq A} \mu_i$ and E denotes expectation.

Let $Q^*(a)$ be the true (unknown) value of arm a, and $\hat{Q}_t(a)$ the estimated value after t time steps, typically the average of the rewards received from choosing this arm up to time step $t - 1$. One simple way of balancing exploration and exploitation is ϵ-*greedy* action selection: This gambler algorithm is defined by choosing with probability $1-\epsilon$, for small $\epsilon \in \mathbb{R}$, the *greedy* action, i.e. the arm with currently highest estimated value (a_t^* such that $\hat{Q}_{t-1}(a_t^*) = \max_a \hat{Q}_{t-1}(a)$). With probability ϵ, a random action is chosen, such that infinite exploration of all actions is guaranteed and all $\hat{Q}_t(a)$ converge to $Q^*(a)$.

Instead of choosing randomly among all actions when exploring, *softmax* action selection rules give selection probabilities to all actions as a graded function of their currently estimated value. The most common softmax method uses a *Gibbs*, or *Boltzmann*, distribution, choosing action a on time

step t with probability

$$\frac{e^{\hat{Q}_{t-1}(a)/T}}{\sum_i e^{\hat{Q}_{t-1}(i)/T}} \tag{2.2}$$

where T is a *temperature* parameter, regulating the tradeoff between exploration and exploitation.

More sophisticated and efficient algorithms have been developed. Lai and Robbins (1985) showed that the best regret obtainable grows logarithmically with the number of time steps t; Auer et al. (2002) achieved logarithmical regret not only in the limit, but uniformly over time. Their algorithm, in its simplest form, is defined by first trying each action once to get an initial estimate, and then selecting the action a that maximizes

$$\hat{Q}(a) + \sqrt{\frac{2\ln(n)}{n_a}} \tag{2.3}$$

where n_a is the number of times a was chosen so far, and n is the total number of trials so far. Action values are exploited by the first, and explored by the second summand, representing a confidence interval for the expected reward. This is called the UCB formula (for *Upper Confidence Bound*), variations of which were influential in Monte Carlo Tree Search in Go (see section 3.3).

2.2 Markov Decision Processes

In the full reinforcement learning problem, the agent has to learn how to act in more than one situation, and explores the consequences of its actions both with regard to immediate rewards received, and to changing states of the environment. Most reinforcement learning research uses the mathematical framework of *Markov decision processes* (MDPs) to formalize this problem (Puterman, 1994).

A Markov decision process is defined as a 4-tuple $(S, A, P_\cdot(\cdot,\cdot), R_\cdot(\cdot,\cdot))$, where:

- S is the set of *states*—in the case of Go, the set of game positions;
- A is the set of *actions*—in our case, the set of moves[1];
- $P_a(s,s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that choosing action a in state s at time t will lead to state s' at time $t+1$ (the *transition function*)—in Go, the transition function can be used to model both the rules of the game and the unknown behaviour of the opponent player, who constitutes part of the environment of the playing agent. If it is only used to model the rules, transitions become deterministic[2], but the reinforcement learning formulation then has to be extended to account for the opposing goals of two agents (see section 2.7);
- $R_a(s,s')$ is the direct reward given to the agent after choosing action a in state s and transitioning to state s' (the *reward function*)—for the task of learning Go, a useful reward function could be

$$R_a(s,s') = \begin{cases} 1 & \text{if, according to the rules of Go, position } s' \text{ is a winning terminal position} \\ -1 & \text{if, according to the rules of Go, position } s' \text{ is a lost terminal position} \\ 0 & \text{otherwise}[3] \end{cases}$$

Model-based reinforcement learning methods assume that the transition and reward functions of the environment are

[1] The set of available moves depends on the current position $s \in S$; where necessary, it is referred to as $A(s)$.

[2] In the deterministic case, the simplified notation $P_a(s) = s'$ can be used for the only state s' with $P_a(s,s') = 1$.

[3] Draws are very rare in Go, but before the introduction of fractional *komi* values like 5.5 points (see section 3.1.2), so-called *jigo* results were possible with the same number of points for Black and White at the end of the game. Depending on the ruleset, this could mean a win for White, or *hikiwake* (draw).

known to the learning agent. *Model-free* methods, such as the Monte Carlo techniques described later, require only experience and no prior knowledge of the environment's dynamics.

The defining property of MDPs is the Markov property, or "independence of path" property—the fact that the agent can gather all relevant information for predicting the future and deciding about its actions from the current state signal alone. Future states and rewards only depend on the current state and action, and not on the history of states, actions and rewards that have led up to it:

$$\begin{aligned} &Pr(s_{t+1}, r_{t+1} | s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_t, a_t, r_t) \\ &= Pr(s_{t+1}, r_{t+1} | s_t, a_t) \end{aligned} \tag{2.4}$$

Finally, the behavior of the agent that is subject to learning and optimization can be described by the notion of a *policy*, a mapping π from states of the environment to probabilities of selecting each possible action when in those states.

$$\pi(s, a) = Pr(a_t = a | s_t = s)^4 \tag{2.5}$$

In this work, we are dealing with an *episodic* task: The agent's experience can naturally be divided into independent sequences of interactions leading from a *start state* to one of a number of *terminal states*. In our case, these episodes are individual games leading from the empty board to a win or loss. The goal of the agent in an episodic task is finding a policy that, at any point in time t, maximizes the expected total accumulated reward (or *return* R) collected during the rest of an episode:

$$R_t = r_{t+1} + r_{t+2} + \ldots + r_T \tag{2.6}$$

where T is the final time step of the episode.

[4] For deterministic policies, one can simplify by writing $\pi(s) = a$ for the only action a with $\pi(s, a) = 1$.

In our formalization of the Go learning task, this corresponds to finding a policy that reaches a winning terminal state in order to receive the reward of 1, or as an approximation: a policy that maximizes the probability of reaching a winning state of the game. In the next section, a general approach to solving this task is described.

2.3 Solving MDPs

2.3.1 Value Functions

While rewards for reaching a particular state are given immediately by the environment (according to the reward function), a more informative quantity for the agent is the total amount of reward, the *return*, it can expect to receive starting from a given state over the entire rest of the episode. This quantity, predicting the long-term worth of a state to the agent, is not directly given by the environment—it has to be estimated from experience, and can subsequently be used to design better policies. In many successful reinforcement learning algorithms, it is represented in form of a *value function*, mapping each state (or state-action pair) to an estimate of how desirable it is to be in that state (or how desirable it is to perform that action when in that state).

Since the expected future rewards depend on the actions the agent is going to take in the future, value functions are defined with respect to a specific policy. The *state-value function* (or just *value function*) V^π is the expected return when starting in a given state and following policy π thereafter. The *action-value function* Q^π is the expected return when selecting a given action in a given state and following policy π from there:

$$V^\pi(s) = E_\pi \left[R_t | s_t = s \right] \tag{2.7a}$$

$$Q^\pi(s, a) = E_\pi \left[R_t | s_t = s, a_t = a \right] \tag{2.7b}$$

where E_π denotes expectation under policy π.

Success in solving a reinforcement learning problem amounts to finding a policy that guarantees high return. Using value functions, one can define a partial ordering over policies, such that one policy π is equal to or better than another policy π' if and only if the values of all states under π are greater than or equal to their values under π':

$$\pi \geq \pi' \iff \forall s \in S.\ V^{\pi}(s) \geq V^{\pi'}(s) \tag{2.8}$$

For every MDP, there is at least one policy that achieves the largest possible return from all states, and is therefore better than or equal to all other policies. This is called an *optimal policy* π^*. All optimal policies of an MDP share a unique *optimal value function* V^* and a unique *optimal action-value function* Q^*, defined by

$$\forall s \in S.\ V^*(s) = \max_{\pi} V^{\pi}(s) \tag{2.9a}$$

$$\forall s \in S. \forall a \in A(s).\ Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{2.9b}$$

It is the agent's goal to learn an optimal policy—or to approximate one as closely as possible, if constraints on computation time or memory make it impossible to determine or represent optimal actions for all states. This is the case for Go with an estimated number of 10^{170} legal positions (see section 3.2).

2.3.2 Generalized Policy Iteration

Policy evaluation is the process of computing the state-value function V^{π} for a given policy π, i.e. answering the question what return can be expected from all states when following that policy. *Policy improvement* is the process of improving a policy, given its state-value function—for example by setting all of the policy's action choices to the actions that maximize return according to current value estimates. This policy change in turn affects state values, which can be computed again by policy evaluation; and so, value-based

reinforcement learning algorithms can enter an iterative cycle of improvement, alternatingly making the value function consistent with the current policy (policy evaluation) and making the policy greedy with respect to the current value function (policy improvement).

In fact, both interacting processes do not have to finish before the other starts: They can be interleaved at a fine-grained level, by shifting the value function only a little bit into the direction of the current policy's value function (policy evaluation), and optimizing the policy only partly on the basis of the current value function (policy improvement). "As long as both processes continue to update all states, the ultimate result is typically the same—convergence to the optimal value function and an optimal policy" (Sutton and Barto, 1998).

Abstracting from the specific methods used for evaluation and improvement, and independent from the precise way of integrating both processes, this general idea is called *generalized policy iteration* (GPI). It works analogously for state-action value functions instead of state-value functions. GPI represents the core mechanism of most reinforcement learning techniques, including Monte Carlo methods as presented in the next section.

2.4 Learning from Experience: Monte Carlo Methods

The term *Monte Carlo* (MC) methods generally refers to any algorithm utilizing repeated random sampling in its computations. In reinforcement learning, it is used to denote a class of simple, model-free evaluation algorithms specifically tailored to episodic tasks.

As explained in section 2.3.1, the value of a state is the expected cumulative future reward starting from that state. At the end of each episode, episodic tasks provide well-defined sample returns for all states visited in that episode. The return of a given state can therefore be estimated by averaging the returns received after visits to that state in

a number of complete episodes[5]. According to the law of large numbers, averages converge to expected values as samples increase, and thus Monte Carlo estimates converge without bias to the true value function as the agent experiences more and more episodes. Again, the same principle applies to the estimation of state-action value functions[6]. This averaging of complete sample returns is the basic idea of Monte Carlo methods.

Combined with a greedy improvement strategy, a complete learning algorithm according to the GPI schema can be designed:

- Policy evaluation: After each episode of experience, the state-action value estimate for each visited state-action pair (s, a) is updated with the return following the visit. The computation of the average can be implemented incrementally with the backup assignments

$$n_{(s,a)} \longleftarrow n_{(s,a)} + 1 \tag{2.10a}$$

$$\hat{Q}^{\pi}(s,a) \longleftarrow \hat{Q}^{\pi}(s,a) + \frac{r - \hat{Q}^{\pi}(s,a)}{n_{(s,a)}} \tag{2.10b}$$

  where $n_{(s,a)}$ counts the total number of times action a has been chosen from state s in all episodes so far, and r is the return received after visiting state-action pair (s, a) in the episode at hand—in our application case, -1 for a lost game or 1 for a game that was eventually won.

- Policy improvement: Before the start of the next episode, the agent's policy is made greedy with respect to the new state-action value function. In the

[5] In this work, only first visits to a given state in any episode are considered (*first-visit MC method*). The *ko* rule of Go forbids game loops, such that no state can be revisited in the same game (see section 3.1.2).

[6] If no model is available, it is more useful to estimate state-action values rather than state values, since they allow the determination of best actions during policy improvement without explicit lookahead. This is why in the following, state-action value functions are used. State values are used analogously in section 2.7.2.

case of our Go learning task, each position could simply be mapped deterministically to the maximally valued move from this position:

$$\pi(s) = \underset{a \in A(s)}{\operatorname{argmax}} \hat{Q}^{\pi}(s, a) \tag{2.11}$$

However, extra measures need to be taken here to ensure exploration, since sample episodes of deterministic policies only contain information about the value of a single action from each state. In order to allow for estimating other actions, one can either select every state-action pair with nonzero probability as the start of an episode, following a deterministic policy thereafter; or one can modify the policy to have nonzero probabilities of selecting all actions in all states.

Monte Carlo methods have two notable drawbacks: First, they only learn on an episode-to-episode basis, not during episodes. Estimates are only updated from the results of finished episodes, not for example from the estimates of successor states (*bootstrapping*). Second, since the return received from an entire episode depends on a sequence of many action choices and transitions, Monte Carlo value estimates have very high variance.

The advantages of Monte Carlo approaches are threefold: First, they require no prior understanding of the environment's dynamics, and are able to learn directly from experience instead. Second, they naturally focus learning on the states and actions that are actually relevant to the agent's policy—often a very small subset of the whole state and action spaces. In Go, for example, the number of board positions that are likely to appear in a game between competent players is much smaller than the number of all legal positions. Third, as detailed in the next section, Monte Carlo methods can work with *simulated* experience in the same way they handle *real* experience.

2.5 Learning from Simulated Experience: Planning

The field of reinforcement learning encompasses *learning* as well as *planning* methods[7]. *Learning* is the process of producing or improving an agent policy given experience in the environment, for example by using Monte Carlo methods as described in the last section. *Planning* is the process of producing or improving an agent policy given a model of the environment, i.e. solely through internal computation, without any direct interaction with the environment.

The term *model* can be used for any knowledge the agent has about the workings of the environment. A model predicts the next state (and reward[8]) given the current state and a possible action the agent could take. If the environment's dynamics are not deterministic—as in Go, where the opponent's behaviour cannot be foreseen precisely—this prediction can in principle have one of two forms: A *distribution model* returns a complete probability distribution over possible next states, as given by (an approximation to) the transition function; and a *generative model* or *sample model* returns *one* possible next state, sampled from (an approximation to) the transition function. In many domains, it is much easier to construct an adequate sample model to *simulate* an environment than to explicitly determine its complete underlying distributions.

Given a sample model, an agent can select hypothetical actions, sample their hypothetical consequences, and collect hypothetical experience about their values. Whereas the real environment generates real experience, a sample model of the environment generates simulated experience. This experience can be used for improving a policy to maximize expected return—with the same algorithms that can be applied to real experience, e.g. Monte Carlo methods as in section 2.4. If the model approximates reality closely enough, the policy learnt through internal policy iteration

[7] The name "reinforcement learning" has historical roots. In this respect, it is somewhat misleading.

[8] In Go, the reward distribution is known, so only state predictions are of interest.

will succeed when applied in the real environment of the agent. This is the principle of Monte Carlo planning.

In Go programs, the sample model is typically constructed by modelling the opponent's behaviour with a policy similar to that of the agent, but with the opposite goal[9]. As mentioned before, this can be understood as an extension of the reinforcement learning problem to two agents, taking turn in executing their actions. Alternatively, the opponent could be interpreted as an adaptive part of the environment.

The advantage of Monte Carlo planning for a computer Go player is that with a simplified model, thousands of games can be simulated in seconds, giving the algorithm some statistical confidence in its value estimates and policy. Obtaining such a large number of games against human players would be infeasible. On the other hand, another problem persists: Not only is a given position of Go forbidden to appear twice in the same game—it is also extremely unlikely to return in any future game, considering the huge game tree. In addition, it is extremely difficult to successfully generalize from Go positions and acquire more compact value function representations (see section 3.2). Hence, it seems futile to store planning results.

2.6 Planning for the Current State: Search

Search is the process of producing or improving an agent policy solely for the *current* state. Other than general planning algorithms which aim at finding optimal actions for all states in the state space, or at least for states likely to be visited by the current policy as in Monte Carlo planning, search is only concerned with the computation of the agent's optimal *next* action.

Search algorithms typically work by unfolding a *search tree*

[9] This method is based on the assumption of rational play, its own policy being the best approximation to rationality the agent can provide. Explicit opponent modelling has not yet been researched in Monte Carlo Go.

from the current state (called the *root state*), where each branch corresponds to one possible transition of the environment, and each node corresponds to one possible future state (or state-action pair). An approximate value function is then applied to the leaf nodes, and interior nodes are evaluated by recursively backing up these values, until the available actions at the agent's current state can be estimated[10]. Finally, the best next action is selected to be executed in the environment, and all search results are forgotten.

Other than the learning and planning methods covered so far, search usually does not change any general policy or value function of the agent; the approximate value function needed for the leaves of the tree is often hand-crafted and remains static throughout the agent's lifetime. Instead, search focuses all computational and memory resources on the computation of a partial policy for the current situation and the immediate future of the agent. Interleaving search and action on-line, the search algorithm itself can be considered a sophisticated policy. This makes search a highly efficient technique in comparison to general planning, and for large domains like Go the only viable option.

2.6.1 Sampling-Based Search

Full-width search refers to a class of search algorithms that take into account *all* possible continuations from the root state up to a certain depth, accounting for all possible next actions of the agent and all possible next states of the environment in the tree. Evaluations of interior nodes are computed by *full backups* from the values of all child nodes. Variants used in game tree search include the well-known *Alpha-Beta algorithm* (Knuth and Moore, 1975), a depth-first search with minimax value backup and pruning of provably suboptimal moves. It has often been extended to a variable-depth search, selectively deepening and exploring

[10] If the leaves reach the end of the episode, true values can be returned and an optimal policy can be derived. Generally, this is not the case, and *bootstrapping* is necessary: the derivation of estimates (at the root) from other estimates (at the leaves).

the more promising continuations of play (see e.g. Campbell et al. (2002)).

In domains with large *branching factors*, i.e. when the number of possible actions in a position and/or the number of possible next states for a state-action pair is high on average, full-width search can be ineffective due to the exponential growth of the tree with the branching factor. *Sample-based search* solves this problem by not considering all possible successor states, but sampling only a certain number of them according to a generative model as described in section 2.5. Evaluations of interior nodes are determined by *sample backups* from the value of a single sampled successor. An example of this approach is the *Sparse Sampling Planner* introduced in Kearns et al. (1999), where at each node up to a given tree depth only a predetermined number of children nodes is expanded.

Kearns' algorithm explores the state space uniformly, with constant width and depth regardless of the values returned. Various improvements of his work consider exploration policies as discussed in section 2.1.1 instead, gradually shifting the focus of the search to higher-valued actions. In Chang et al. (2005), for example, the action choice at every interior node of the tree is treated as a multi-armed bandit problem, and Auer's UCB method is used for adaptive sampling. In Péret and Garcia (2004), a Boltzmann distribution is applied to the same task.

For the application scenario of Go, only one problem remains: In the search techniques described so far, approximate value functions are necessary to evaluate the tree's terminal nodes. However, it is extremely difficult to construct reliable and fast evaluation functions for Go (see section 3.2). A possible solution lies in the Monte Carlo principle: evaluation of states and actions through repeated random sampling of episodes.

2.7 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a best-first, sampling-based search algorithm with Monte-Carlo evaluation. Independently developed in similar form by Coulom (2006) and Kocsis and Szepesvári (2006), MCTS is the central algorithm in modern computer Go, and the framework of this thesis. Beyond Go, variants have been applied successfully in many other games such as Amazons (Lorentz, 2008), Lines of Action (Winands and Björnsson, 2009), Settlers of Catan (Szita et al., 2009), Solitaire (Bjarnason et al., 2009), Poker (den Broeck et al., 2009), real-time strategy games (Balla and Fern, 2009), and General Game Playing (Sharma et al., 2008). MCTS is also increasingly used for high-dimensional control in non-game domains, e.g. in de Mesmay et al. (2009) or Rolet et al. (2009).

In this section, Monte Carlo Tree Search is first described informally; then framed as a reinforcement learning algorithm and put in relation to the GPI schema; and finally presented as pseudocode.

2.7.1 Informal Description

For each move decision of the Go-playing agent in a real game, the MCTS algorithm grows a search tree to determine the best move. From the current position, this tree is selectively deepened into the direction of the most promising moves, which are chosen according to the success of simulated games starting with these moves.

The tree initially contains only the root node s_r, representing the current game position. Each further node added to the tree stands for one of its possible successor positions s, and contains at least the *visit count* and the *win count* of this position. The visit count is the number of times this node has been sampled in the search process so far, and the win count is the number of times a simulated game passing through this node has been won by the player who reached it.[11]. The *win rate*, win count divided by visit count, is the

[11] Values can either be represented from the point of view of alternat-

current value estimate $\hat{V}^{\pi}(s)$ of position s.

MCTS works by repeating the following four-phase loop until computation time runs out. It can be interrupted after any number of iterations to return the current result. Each complete loop represents one simulated game.

- Phase one: *selection phase*. The tree is traversed from the root s_r to one of the leaf nodes s_l. At every step, a selection policy is utilized to choose the move to sample from this position. The selection policy has to balance exploitation of positions with high value estimates and exploration of positions with uncertain value estimates. Kocsis and Szepesvári (2006) showed that although reward distributions in game trees are not stationary as in the classic multi-armed bandit problem, it is possible to apply the UCB multi-armed bandit strategy in every node, and achieve convergence to the optimal policy. This variant of MCTS is called UCT. Other, domain-specific approaches are described in section 3.3.2.

- Phase two: *expansion phase*. One or more successors of s_l are appended to the tree. A common expansion policy in Go is the addition of one newly sampled position per episode. It is also possible to add several positions from the episode, or to expand only positions that have been visited several times.

- Phase three: *simulation phase*[12]. According to a simu-

ing players, which allows for maximization at all levels of the tree, or from the point of view of the same player throughout the tree, which has to be combined with alternating maximization and minimization.

[12]No consistent nomenclature for the four phases has emerged in the literature yet, in particular for the "selection" and "simulation" phases. In fact, moves are "selected" throughout both phases, and both phases together constitute one "simulated" game. Also, the term "playout" (or "rollout") is used extensively in the literature, referring to either the simulation phase or selection and simulation phases together. Furthermore, the term "sampling" has been applied to the selection of moves in both phases, or in the simulation phase only.

In this thesis, "selection", "sampling" and "simulation" are used in their general meaning with respect to all phases of MCTS. When a distinction between phases is necessary, "selection phase" and "simulation phase" are used. "Playout" is used synonymously with "simulation phase".

lation policy, moves are played in self-play until the game is finished. While uniformly random move choices are sufficient to achieve convergence in the long run, convergence speed can be improved by using more sophisticated playout policies (see section 3.4). However, construction of a strong yet unbiased policy is a difficult task. Addressing this problem is the aim of this thesis.

- Phase four: *backpropagation phase*. After the end of the simulation has been reached and the winner of the simulated game has been determined, the result is backpropagated to all nodes traversed during the playout. Visit counts of these nodes are incremented, and win counts are incremented for all nodes reached by the winning player. Value estimates of all involved positions are thus updated, such that the next simulation can explore improved move choices.

In the selection phase at the beginning of each simulated game, move decisions are made according to knowledge collected on-line in the search tree. Beyond the leaves of the tree, no knowledge about the value of individual positions is available, so the playout policy is used as a rough approximation of reasonable play. The agent's play becomes stronger as more games are played out, the tree grows, and move estimates improve. Under certain conditions on the distribution of rewards, MCTS converges to the optimal policy (Kocsis and Szepesvári, 2006).

When time runs out or the loop has been repeated sufficiently often, the program returns the best move at the root as the search result[13]. Generally, the move with highest visit count is chosen.

2.7.2 MCTS and Reinforcement Learning

As introduced in section 2.2, the task of learning Go can be described as a reinforcement learning problem by viewing Go positions as states, Go moves as actions, Go rules

[13] Because the rules of Go are known and deterministic, estimation of position and move values is equivalent here.

as defining the transition function, and results of Go games as rewards for the agent. In this light, Monte Carlo Tree Search is a value-based reinforcement learning algorithm, which in combination with a generative model becomes an on-line planner.

The tree $T \subset S$ is a subset of the state space, focusing on the current state and possible successors. After n simulated episodes of experience—n games—it contains n states, for which distinct estimates of V^π are maintained. For states outside of the tree, values are not explictly estimated, and moves are chosen randomly or according to a playout policy.

From the perspective of generalized policy iteration, the two interacting processes within MCTS are:

- Policy evaluation: After each episode of experience, the value estimate of each visited state $s \in T$ is updated with the return from that episode.

$$n_s \longleftarrow n_s + 1 \tag{2.12a}$$

$$\hat{V}^\pi(s) \longleftarrow \hat{V}^\pi(s) + \frac{r - \hat{V}^\pi(s)}{n_s} \tag{2.12b}$$

  where n_s is the number of times state s has been traversed in all episodes so far, and r is the return received at the end of the current episode.

- Policy improvement: During each episode, the agent's policy adapts to the current value estimates. In the case the UCB policy is used in the selection phase, and a uniformly random policy in the simulation phase:

$$\pi(s) = \begin{cases} \underset{a \in A(s)}{\operatorname{argmax}} \left(\hat{V}^\pi\big(P_a(s)\big) + \sqrt{\frac{2 \ln(n_s)}{n_{P_a(s)}}} \right) & \text{if } s \in T \\ \operatorname{random}(s) & \text{otherwise} \end{cases}$$

  where $P_a(s)$ is the position reached from position s with move a, and $\operatorname{random}(s)$ chooses one of the actions available in s with uniform probability.

Many implementations focus on the estimation of state-action values instead of state values. The core ideas of the algorithm remain unchanged. In this case, the policy evaluation step for each visited state-action pair (s, a) is:

$$n_{s,a} \longleftarrow n_{s,a} + 1 \tag{2.13a}$$

$$\hat{Q}^{\pi}(s,a) \longleftarrow \hat{Q}^{\pi}(s,a) + \frac{r - \hat{Q}^{\pi}(s,a)}{n_{s,a}} \tag{2.13b}$$

where $n_{s,a}$ is the total number of times action a has been chosen from state s; and the policy improvement step is

$$\pi(s) = \begin{cases} \underset{a \in A(s)}{\operatorname{argmax}} \left(\hat{Q}^{\pi}(s,a) + \sqrt{\frac{2 \ln(n_s)}{n_{s,a}}} \right) & \text{if } s \in T \\ \operatorname{random}(s) & \text{otherwise} \end{cases}$$

if UCB is used for the selection phase and random actions are chosen in the simulation phase.

2.7.3 Pseudocode

(Adapted from Chaslot et al. (2007).)

```
while(hasTime) {
   currentNode ← rootNode
   while (currentNode ∈ T ) {
      lastNode ← currentNode
      currentNode ← Select(currentNode)
   }
   lastNode ← Expand(lastNode)
   R ← PlaySimulatedGame(lastNode)
   while(currentNode ∈ T ) {
      currentNode.Backpropagate(R)
      currentNode.visitCount ← currentNode.visitCount + 1
      currentNode ← currentNode.parent
   }
}
bestMove = Argmax(N ∈ Children(rootNode)) N.visitCount
```

Chapter 3

Computer Go

This chapter discusses related work in the application field of Go by first introducing the game and early attempts at programming Go AI, then describing the rise of Monte Carlo methods in computer Go, and finally examining previous approaches for simulation policies in Monte Carlo Tree Search.

The first section gives an outline of the history, rules, and basic concepts of Go, as well as the methods for rating playing strength.

3.1 The Game of Go

3.1.1 History

The origins of the game of Go[1] are lost in myth. Legends attribute its invention to the Chinese emperors Yao (2337-2258 BC) or Shun (2255-2205 BC) as a means of teaching discipline, concentration and balance to their sons. Other sources speak of a genesis "more than 2500" or "more than 3000" years ago, connecting Go to rituals of divination and

[1]The game has spread to the West mainly from Japan and has therefore come to be known under its Japanese name *Go* (or *Igo*). It is called *Weiqi* in Chinese and *Baduk* in Korean.

astrology, or to the pieces of stone Chinese generals used to plan out their campaigns.

The earliest surviving references to Go as a game can be found in the *Chronicle of Zuo*, the earliest Chinese historical annal, in the books of Mencius, and in the *Analects* of Confucius, the greatest work of ancient Chinese philosophy. At the time these books were written—in the 3rd-4th century BC—Go was popular amongst the Chinese aristocracy; along with playing the zither *guqin*, calligraphy and painting, it was considered one of the four cultivated arts. The oldest surviving Go board can be dated to the Western Han Dynasty (206 BC - 24 AD), while the oldest surviving text devoted specifically to Go (the *Essence of Go* by Ban Gu) originates in the 1st century AD. The rules of Go have remained essentially unchanged since then.

In Japan, Go has been known at least since the early 7th century AD. One hundred years later, the Japanese ambassador at the Chinese capital of Ch'ang-an, Kibi no Makibi (695-775 AD), further popularized Go with the Japanese aristocracy and Imperial Court. The game appears in various classical works of Japanese literature, including *The Tale of Genji* (early 11th century), sometimes considered the world's first novel. In 1612, four hereditary Go academies were founded by the Tokugawa Shogunate: The houses of Honinbo, Hayashi, Inoue and Yasui. The support of professional Go players by the state and the continuing competition of the four houses over the years of the Edo period (1603-1868) led to the most significant development of the game's theory and player skill.

In Europe, Go has been described for the first time in the essay *De Circumveniendi Ludo Chinensium* (*About the Chinese encircling game*) by Thomas Hyde in 1694. However, it remained relatively unknown, and available descriptions were mostly incomplete. The German engineer Oscar Korschelt (1853-1940), who had lived in Japan from 1878 to 1886, published a book on Go at the end of the 19th century, introducing the game to Germany and Austria. Edward Lasker (1885-1981), one of the leading chess players of the time, learned about it in Berlin in 1905 and later co-founded the New York Go Club, beginning to spread the game in the US. In the second half of the 20th century, the centuries-

old Japanese domination in Go was challenged by China and Korea, and more and more international tournaments were established. The International Go Federation today has members in 71 countries all over the world. The game is believed to be played by 25-100 million people according to various estimates.

3.1.2 Rules

Various rulesets for Go exist. Their most significant differences relate to the methods of ending and scoring a game, the handling of game loops arising from repetition of earlier positions, and the status of certain rare positions. However, in practice different rulesets rarely lead to different game results, and consequences for game strategy are mostly negligible. The character of the game remains unchanged.

For beginning players, clarity and comprehensibility are most important; for computers, consistency and ease of computation are paramount. In order not to obscure the simplicity and beauty of the basic rules, the Short Rules as given in Jasiek (2007) are quoted here[2]. In-depth information and discussion on subtle rule nuances can be found in Jasiek (2010) and Sensei's (2010).

> The game is played on a grid board. Typically it has 19×19 intersections, but 9×9 are also fine[3]. Two players compete. The first player uses black stones, the other white.
>
> The players alternate. A player may play or pass. Playing is putting one's own stone on an empty intersection and removing any surrounded opposing stones. To avoid cycles, a play may not recreate any prior configuration of all stones on the board[4].

[2] They are identical to the rules implemented in the program OREGO, which is the basis of all algorithms described in this thesis.

[3] 9×9 and 13×13 boards are in use for educational purposes and for quick, informal games. 19×19 is the regular board size treated by Go literature and used in tournaments.

[4] The anti-cycle rule is called *ko* rule in Go, a Japanese word meaning "eternity" as well as "threat". *Ko fights* are an interesting strategic con-

Two successive passes end the game. Then the player with more intersections wins. Intersections are his if only his stones occupy or surround them[5].

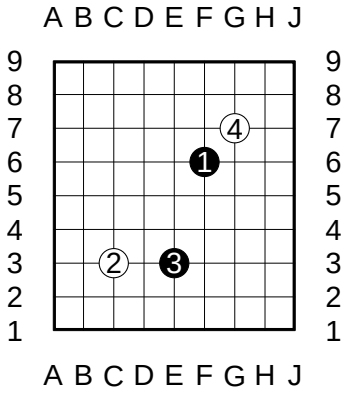

Figure 3.1: A game position after the first four moves.

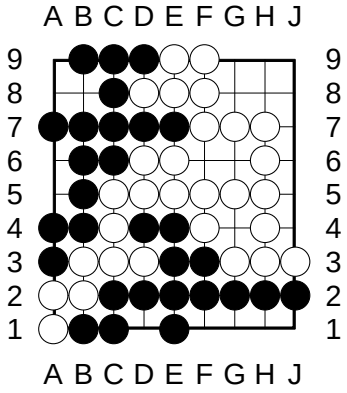

Figure 3.2: A game position after the game has ended.

sequence of it. — Depending on the ruleset, a "configuration" may or may not include the player to move (*situational* or *positional superko*). In some rulesets, it is forbidden to recreate a position with two successive plays (simple *ko*), whereas recreations after more than two moves lead to the end of the game "without result".

[5]This is true for *area scoring* as used in "Chinese rules", which are employed by most computer programs. With *territory scoring* as used in "Japanese rules", the ruleset known to most Western players, intersections occupied by own stones are not added to the score, while any removed stones of the opponent are added. The difference usually amounts to no more than one point.

All rulesets offer a possibility for human players to end the game as soon as there is agreement about the final outcome. Thus, it can be avoided to unnecessarily prolong a game. For computers, on the other hand, it is often difficult to judge who is ahead in a given position, so games between computers are usually played out to the end.

3.1.3 Basic Concepts

The black and white pieces that are played on empty intersections of the board are called *stones*. A set of adjacent, connected stones of the same color is called a *string*, *chain* or *block* (see figure 3.3). Empty intersections adjacent to a string are called *liberties* of the string. If a string has only one liberty left, it is said to be in *atari*; when it is surrounded, its liberties are reduced to zero and it is *captured* by removing it from the board (see figures 3.4, 3.5 and 3.6). In some rulesets, captured stones of the opponent are kept as *prisoners* and influence the scoring. *Suicide* is the result of taking the last liberty of your own string; it is allowed or forbidden depending on the ruleset. In any case, removal of the opponent's stones precedes removal of own stones (see figures 3.7 and 3.8).

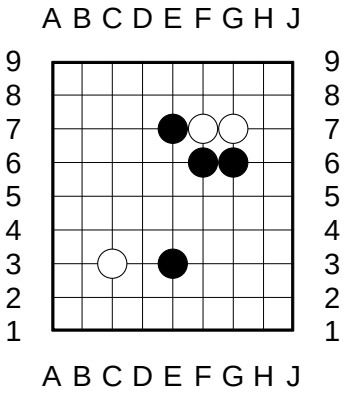

Figure 3.3: A position with five strings: Two white ones and three black ones.

It is possible to construct a string (or set of strings) that is immune to capture. For example, suppose a string sur-

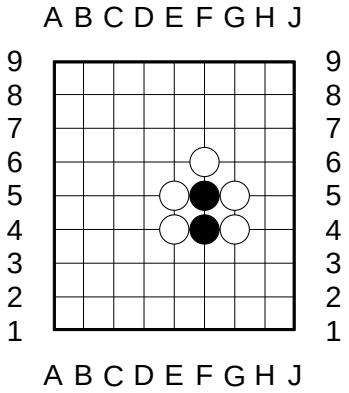

Figure 3.4: The black string is in atari: It has only one liberty (f3).

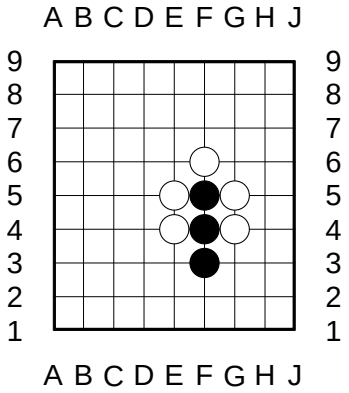

Figure 3.5: If it is Black's turn in the situation in 3.4, he can play f3; now his string has three liberties (e3, f2, g3) and therefore escapes capture.

rounds two separate empty intersections, called *eyes*, as in figure 3.9. The opponent would have to play on both of these intersections to capture the string, but either one of these moves would be suicidal. Strings that cannot be captured are called *alive*; strings that cannot be made alive are *dead*. A *group* is a loosely connected set of stones and/or strings that act as a functional unit in the game and are considered effectively connected (see figure 3.10). An essential part of Go skill is the correct judgement of the life and death status of a group.

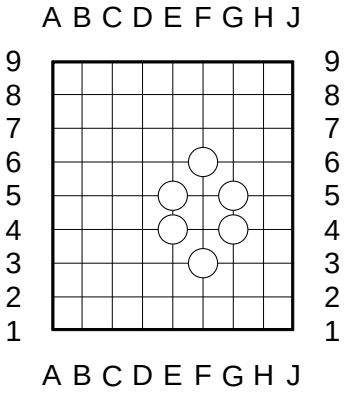

Figure 3.6: If it is White's turn in the situation in 3.4, he can play f3; now the black string is completely surrounded and is therefore removed from the board.

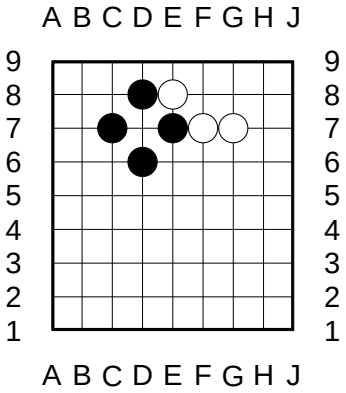

Figure 3.7: In this position, White is not allowed to play d7, as his stone would have no liberties there and would be instantly captured (suicide).

By making the first move of the game, Black gets a certain advantage over White. This can be compensated for by *komi*, a predetermined number of points added to the white score after the game has ended. Komi for 19×19 boards usually varies between 5 and 8 points. Furthermore, Go allows for interesting and challenging matches between players of different strength by providing an effective handicap mechanism: Players can agree to place a number of stones of the weaker player's color (black) on the board before the

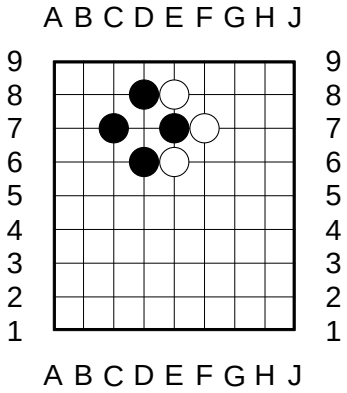

Figure 3.8: In this position, White is allowed to play d7, since after the removal of the opponent's stone e7 the new white stone d7 would have one liberty (no suicide).

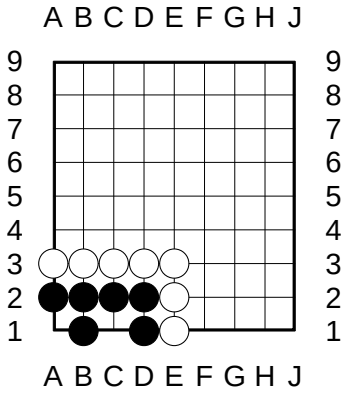

Figure 3.9: The black string is alive. White cannot play at either a1 or c1, as both moves would be suicide. So the black string's liberties cannot be taken away.

game begins, to compensate for the skill difference[6]. Figure 3.11 shows the position before White's first move in a 19×19 game with four handicap stones.

[6] In this case, White has the first move, and no komi is used. Giving Black the first move and using no komi is equivalent to one handicap stone.

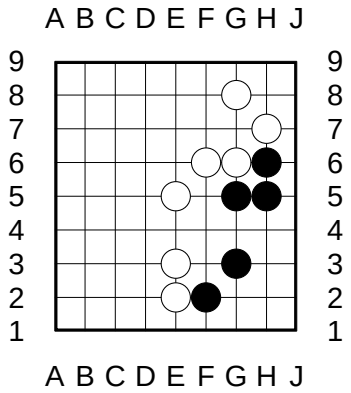

Figure 3.10: The black stones form a group. They are not connected yet, but work together to secure the bottom right corner.

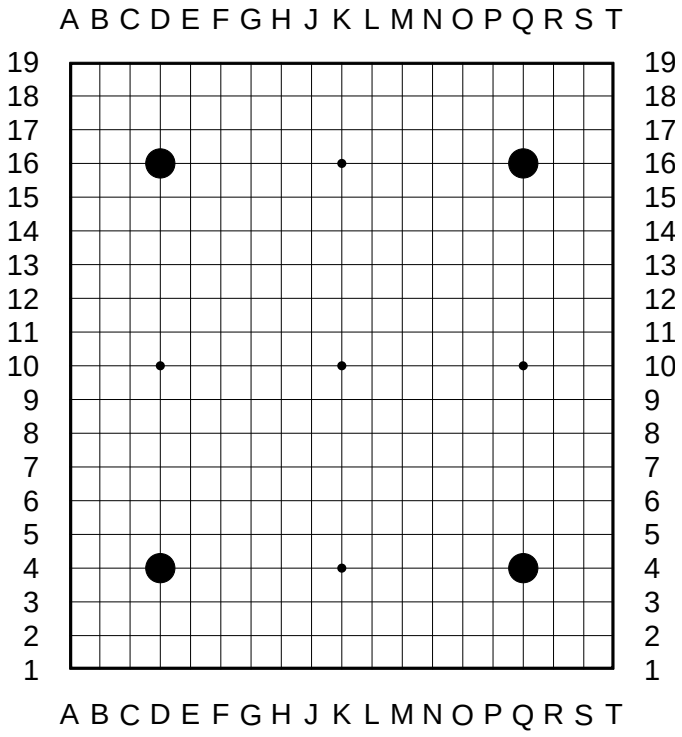

Figure 3.11: The placement of four handicap stones.

3.1.4 Ratings

Go players are rated on a scale divided into *kyu* (beginner), *dan* (amateur master) and *professional dan* ranks[7]. Beginners who have just learned the rules of the game are usually rated around 25 kyu to 30 kyu; from there, they advance downwards to 1 kyu (abbreviated 1k). If they progress beyond that level, they can advance upwards from 1 dan to 7 dan (abbreviated 1d to 7d). To go beyond the 7th amateur dan, players have to acquire professional status through one of the major Go organizations in Japan, China, Korea or Taiwan. Professional rankings are 1 dan through 9 dan (abbreviated 1p through 9p).

For amateurs, ranks can be determined by the amount of handicap stones needed to allow an even game between two players: A difference of one rank corresponds roughly to one handicap stone. For professionals, ranks are awarded on the basis of tournament performance. The difference between two professional dan grades is only about one third or one fourth of a stone.

With statistical methods and the help of computers, it is nowadays possible to compute more precise player ratings on the basis of individual game results, assuming that the performance of a given player in a given game is a random variable from a certain distribution class. With the *Elo scale*, originally introduced in chess and used for example on the Computer Go Server (CGOS) for computer players, a difference of 100 points between two players predicts that their win rate will be about 64:36; a difference of 200 points corresponds to a win rate of about 76:24, etc. (Dailey, 2010)

In computer go, effectively evaluating the playing strength of a program in reasonable time is an unsolved problem (see Müller (1991) for some early considerations). Because different players have different strengths and weaknesses—this holds for programs as well as for humans—the only way of establishing a truly reliable rat-

[7] Kyu and dan ranks have been invented for Go in the Edo period, but have later also been applied to martial arts like judo and karate, and even to ikebana and tea ceremony. The well-known black belts of martial arts indicate dan ranks.

ing is by playing a large number of games against a variety of opponents. At least against computer opponents, this is possible e.g. on the Computer Go Server, but it is prohibitively time-consuming for development and optimization purposes. A compromise used for most current Monte-Carlo based programs and adopted for this thesis is testing against only one computer opponent: GNUGo (Foundation, 2009). Since GNUGo is a knowledge-based program with an entirely different architecture from MC programs (see section 3.2), the risk of overfitting is somewhat lowered, and GNUGo plays comparatively fast.

Still, it is difficult to obtain statistically significant results for a minor algorithm improvement without playing on the order of one thousand games, each taking several minutes up to several hours depending on the time settings of the MC program. Some researchers (Takeuchi et al., 2008; Gelly and Silver, 2007) have therefore taken the approach of supervised learning: By using a set of test positions from professional games and comparing the estimation of a position's value by the program (interpreted as a winning probability) with the real outcome of the game on record. However, to the author's knowledge there are no rigorous evaluations of this measure's correlation to playing strength[8].

The next section turns to the field of AI and gives an overview of the first four decades of computer Go.

[8] It is not used in this thesis for three reasons: First, the values computed by Monte Carlo do not have to be close to the "real" winning probabilities against a strong player—they should only be a monotone increasing function of this probability. An algorithm computing values of 0.6 for a winning move and 0.5 for a losing move will fare just as well in actual play as an algorithm giving values of 0.9 and 0.8 to these moves; but the error compared to the value on record, 1.0 for the winning move, is different. Second, it has been reported that training on professional games leads to weaknesses against amateur players, because refutations of weak moves cannot be learned (Stern et al., 2006). Positions occurring in high-level games are not representative of all human Go knowledge, as the consequences of suboptimal play are not demonstrated. Training on amateur games however increases noise. Third, training on professional games obviously does not enable an algorithm to surpass human capabilities in the long run.

3.2 Traditional Computer Go

As noted in section 1.2, the classic approach of combining game tree search with Alpha-Beta pruning and a static evaluation function, highly successful in other games like chess (Campbell et al., 2002), checkers (Schaeffer et al., 1992) and draughts, connect-four (Baier, 2006), Nine Men's Morris, and Othello (Buro, 1998), fails in Go. All of these games share important properties with Go: They are two-player games; they are perfect-information games, as no element of the game state is hidden from the players; they are zero-sum games, as one player's loss is the other player's gain; they are deterministic games, as there is no element of chance involved. Still, Go cannot be played effectively using the same techniques, presumably for two reasons.

First, Alpha-Beta and similar exhaustive search algorithms cannot deal with the branching factor of Go, i.e. the number of legal moves from an average game position. While the branching factor of chess, for example, is around 35, an average Go position offers circa 300 possible moves, rendering a traversal at even shallow depths virtually impossible. Relatedly, the total number of legal Go positions is estimated to be around 10^{170}, while there are less than 10^{50} chess positions[9] (Allis, 1994; Tromp and Farnebäck, 2006). Neither highly efficient pruning techniques like Alpha-Beta nor constant improvements in computer hardware can alleviate the exponential growth of the game tree.

On the other hand, Go on a 9×9 board has a comparable branching factor to chess, and is just as problematic for Alpha-Beta (Müller, 2002). The second and probably more important distinction between Go and other board games is that despite decades of research efforts, no one has yet succeeded in creating a fast, accurate evaluation function for Go. While in chess, for example, the probability of winning from a given position correlates relatively well with the number and quality of remaining pieces for both play-

[9] These figures ignore history attributes. Technically, a chess position includes the information of how many moves have passed since the last capture or pawn movement (for the *50 moves rule*), and a Go position includes the information about all previous stone arrangements in the game (for the *ko rule*).

ers, Go positions are difficult to decompose into computationally cheap and meaningful features (Burmeister and Wiles, 1995). Stones can interact over large distances on the board and exert influence on events 50 or 100 moves apart. In order to evaluate a position, the life and death status of groups, the number and size of *ko* threats and other factors have to be taken into account, which themselves require search and interact in subtle ways that are difficult to model. Unfortunately, Alpha-Beta depends strongly on accurate evaluation functions—search results can be sensitive to just a single altered leaf evaluation amongst millions.

3.2.1 Beginnings of Computer Go

The first Go program was written in 1960 (Lefkovitz, 1960), the first computer Go paper appeared in 1963 (Remus, 1962), and the first win against a human beginner—according to some sources (Burmeister and Wiles, 1997), the first complete game ever played by a program—was achieved in 1968 (Zobrist, 1970). Like other early programs, Zobrist's algorithm was based on an influence function, approximating the effect of stones on potential territory by computing an influence field around every stone that decreases with distance. The next step in computer Go was the attempt to subdivide the board into zones or subgames, in order to simplify reasoning about them, and to use abstract board representations e.g. on the level of groups (Reitman and Wilcox, 1979). Another generation of Go programs was characterized by introducing patterns to suggest moves for typical, reoccurring local situations (Boon, 1990).

In the 1980s and 90s, the strongest programs integrated all of these ideas and many more, experimenting with various tactical and strategic board representations, local and global search techniques, evaluation functions based on functions approximators like artificial neural nets, and heuristic move generators derived from human knowledge. Programmers tried to model vague human concepts like connectivity, safety and strength of groups, territory and influence trade-offs, eyespace or shape, and transform them into computational ones. Rule-based expert systems generated moves with corresponding urgency values; goal-

oriented search with specific evaluation functions was used to determine the tactical status of strings; pattern and rule libraries were created by hand or with machine learning approaches like temporal-difference learning or explanation-based learning. Algorithms for particular classes of fights, endgames or *ko* situations were divised. Expanding or reducing territories, attacking or defending groups and other subgoals of the game were often handled by a number of independent submodules, whose results (or lack of results) then had to be integrated on a global level. Self-play, supervised learning and evolutionary methods have been applied to improve game strength. Bouzy and Cazenave (2001); Müller (2002); Burmeister and Wiles (1997) provide a survey of the field.

3.2.2 Subproblems of Go

In parallel to working on complete Go-playing programs, researchers turned to aspects of the game that seemed amenable to separation and closer analysis. The essential problem of life and death has been approached by a large number of search algorithms and static classificators (Benson, 1976; Wolf, 1994; Chen and Chen, 1999; Kishimoto and Müller, 2005). Methods for recognizing safe territory have been developed (Müller, 1997; Niu and Müller, 2006b,a). Search for local goals has been refined (Ramon and Croonenborghs, 2004; Yoshizoe et al., 2007). Connecting (Cazenave and Helmstetter, 2005b) and separating stones (Cazenave, 2006), as well as the problem of *seki* (mutual life) have been addressed (Niu et al., 2006). Go on various small board sizes was solved (van der Werf et al., 2003; van der Werf and Winands, 2009).

Since Go is a very visual game in which patterns and shapes play a large role, the theory of mathematical morphology—used as a tool by the image processing community for the processing of geometrical structures—has been applied to Go for territory and influence computations (Bouzy, 2003). And because Go situations can often be divided into independent or almost independent subgames which are seperately solved by human players, combinatorial game theory (CGT) has been used for the late endgame and a number of

other situations, with the goal of formalizing Go as a sum of games (Berlekamp and Wolfe, 1994; Müller, 2003).

3.2.3 Problems of Traditional Computer Go

Although excellent results have been obtained in some subproblems, the integration of specific problem solving tools into complete game-playing programs has proven very difficult (Bouzy and Cazenave, 2001). Life and death solvers typically only work for completely surrounded groups; combinatorial game theory can still only be applied to a small fraction of Go positions; the assumptions and preconditions of many such methods are rarely met in actual gameplay, where groups and territories usually have open boundaries, and subgames interact. In addition, a program consisting of many interdependent parts is hard to balance, maintain and extend: Every added piece of knowledge, intended to improve play in one area, can have unforeseen and unintended consequences in other areas.

Traditional computer Go suffered from the classic knowledge acquisition bottleneck: The playing strength of programs often depended on the strength of their programmers (Burmeister and Wiles, 1997), whose attempts at understanding their own intuitions through introspection were of limited success. Rule bases were riddled with exceptions and exceptions to exceptions. Because the vague concepts humans use to describe Go can be formalized in many, idiosyncratic ways, there was little agreement in the computer Go community about models and algorithms, and thus progress was slowed.

3.2.4 Strength of Traditional Computer Go

Establishing ratings of knowledge-based computer programs through games against humans is a problematic task: Humans are quick at adapting to their opponent and exploiting its weaknesses, while most traditional Go programs were not able to learn effectively. Consequently, they would fall again and again for the same tactics.

One of the strongest programs of the 1990s, HANDTALK, was officially awarded a 3 kyu diploma by the *Nihon Kiin*, the largest Japanese Go organization. Another program, GO++, was estimated to be around 7 kyu. However, these ratings were very optimistic, and because playing strength is a brittle quality often determined by the weakest move in a game, they could rarely sustain a level above 10-15 kyu. Experienced and observant players could easily find their weak spots and consequently achieve wins like that of Martin Müller (5d) against the program MANY FACES OF GO in 1998—despite the extraordinary number of 29 handicap stones (Müller, 2002).

In the next section, the second major phase of computer Go is described—the era of Monte Carlo Tree Search. This introduces the background for my work on MCTS simulation policies in Go.

3.3 Monte Carlo Go

3.3.1 Beginnings of Monte Carlo Go

As mentioned in section 1.3, Monte Carlo methods—relying on random sampling and averaging returns—have traditionally been used in games where hidden information or an element of chance make probabilistic estimates a natural choice. Successful examples include Poker (Billings et al., 2002), Backgammon (Tesauro and Galperin, 1996), Scrabble (Sheppard, 2002), Tarok (Luštrek et al., 2003), and Bridge (Ginsberg, 2001).

The first application of Monte Carlo to deterministic, perfect-information games has been described in Abramson (1990). The proposed *expected-outcome model* uses the expected value of a game's result from a certain position on, estimated by random sampling, as an evaluation function for leaf nodes of a traditional Alpha-Beta search tree. Other than the common handcrafted, ad-hoc evaluation functions, expected-outcome provides an "elegant, crisply defined, easily estimable, and above all, domain indepen-

dent" model of position evaluators according to the author. Its performance is usually inferior in comparison to fine-tuned evaluators that incorporate complex expert knowledge; in domains where it is hard to construct an efficient static evaluator, however, expected-outcome is a worthwile alternative.

The first Go program employing simulated games was presented in Brügmann (1993): GOBBLE. GOBBLE estimated the value of a given move using the *All-moves-as-first heuristic* (AMAF), averaging the results of all simulated games in which the move had been played at *any* time, not only as first move. In terms of reinforcement learning, the values of actions were learned independently of state. Then, the order of all moves in the next playout was determined by their values and a randomization factor based on simulated annealing. GOBBLE's success in finding reasonable moves without incorporating any domain knowledge was remarkable, but as no tree structure beyond the first ply[10] was used, correct sequences of moves could not be found by this technique.

10 years later, Monte Carlo Go was taken up again by the research community in Bouzy and Helmstetter (2003), when improvement of the traditional Go program INDIGO stagnated due to the knowledge acquisition bottleneck. The authors used basic one-ply tree search with Monte Carlo evaluation of leaf nodes, and experimented unsuccessfully with the extension to two-ply minimax search. In order to speed up and focus the search, pruning techniques were introduced to exclude seemingly suboptimal moves from the search process.

In further work, striving to combine the strengths of existing Go knowledge and the Monte Carlo approach, a knowledge-based move generator was proposed to preselect moves for subsequent Monte Carlo evaluation (Bouzy, 2005a). In order to make deeper tree searches possible despite the large branching factor of Go, a minimax algo-

[10]The term *ply*, coined in Samuel (1959) and commonly used in research on two-player games, refers to one move by one of the players. It was introduced to avoid misunderstandings due to the different meanings of the word *move*, referring to one ply in Go, but to two successive plies of both players in chess.

rithm with iterative deepening was developed that pruned move candidates at every level of the tree (Bouzy, 2004). For the framework of global tree search with Monte Carlo evaluation, experiments with different pruning techniques were conducted (Bouzy, 2006). Other researchers combined classical, goal-based searches with the new statistical approaches, by using sampling to determine the subgame that correlates best with winning the game, and local searches to choose the appropriate move for the subgame (Cazenave and Helmstetter, 2005a).

In 2006, Coulom (2006) recognized the main problem of permanently pruning moves: Due to the uncertainty associated with statistical move estimates, good moves are too frequently excluded from the search process. Building on search algorithms from the field of Markov decision processes, Coulom developed a way of achieving highly selective tree search without permanent exclusion of moves. Thus, tree search could be combined with Monte Carlo evaluation while guaranteeing asymptotic convergence to the optimal move. A similar algorithm was designed by Kocsis and Szepesvári (2006) and introduced to Go in Gelly et al. (2006): The *UCT* algorithm as described in section 2.7.2.

Monte Carlo Tree Search has initiated a revolution in computer Go in recent years. It uses a simple framework and, in its basic form, little to no domain-specific knowledge, which makes it easier to develop than traditional programs (and encourages interested researchers to join the field). It maintains a global view of the game, without depending on error-prone subdivision of the board or pruning of moves. Its playing level increases with additional computation time, and it is able to present a "best move so far" at any point—other than knowledge-based programs which usually only compute one single solution in a fixed period of time. With respect to Alpha-Beta, Monte Carlo algorithms have the advantages of completely dispensing with the evaluation function, efficiently focusing the search, and robustly handling uncertainty.

3.3.2 Approaches to Monte Carlo Tree Search

The four phases of Monte Carlo Tree Search, as detailed in section 2.7.1, are *selection* (move choice within the tree), *expansion* (tree growth), *simulation* (move choice beyond the tree) and *backpropagation* (updating of information). Simulation strategies, the focus of this work, are reviewed in detail in section 3.4. This section deals with the approaches to expansion, backpropagation and selection that have been used in the literature on MCTS in Go.

Expansion Phase

Expansion deals with the questions: How many nodes should be appended to the tree after one simulated game, and under what conditions should nodes be appended? Since the first papers on MCTS in Go were published, all strong algorithms have opted to add at most one node per simulation. Usually, nodes are expanded when they have been visited a predetermined number of times—at the second time in OREGO, the program used in the work at hand. Some algorithms promote nodes to *internal nodes* after they have proven their worth by a larger number of visits; in internal nodes, more expensive heuristics are applied to guide search (Chaslot et al., 2007). Examples follow in this section.

Backpropagation Phase

Backpropagation deals with the question: Which pieces of information should be extracted from finished playouts and averaged in visited tree nodes in order to improve their value estimates? It has been found (Coulom, 2006) that a simple binary result—win or loss—works better than the backpropagation of more precise point results like "lost by 2 points", "won by 7 points" etc. When such win margins are maximized, Monte Carlo tree search tends to greediness when in the lead, instead of playing safe and stable. With binary results, Monte Carlo acts more appropriately

to its current estimated chance of winning: When behind, it takes risks, when in the lead, it plays solidly. As a result, MCTS programs often win by the smallest margin possible, 0.5 points, while they tend to lose overwhelmingly[11].

Selection Phase

Selection deals with the question: Which positions in the tree should be chosen for further sampling, and how is the value of a move defined? The tradeoff between exploration and exploitation has to be considered here, as well as the possibility of generalization in learning and the influence of offline knowledge.

In Coulom (2006), the proportion of wins to visits—the winrate—was used as the value of a given move. For exploration, each move choice in the tree was made according to a probability distribution over all legal moves, similar to the Boltzmann distribution (see section 2.1.1).

Gelly et al. (2006) introduced the UCT algorithm to Go, with an improved UCB bandit for exploration. In addition to the winrate, the concept of *first play urgency* was introduced, which allows UCB to focus on very successful moves without having to try all legal moves first. Also, experiments with copying value information from "grandfather" nodes were made, with the assumption that many move values will not change drastically within two plies.

Coquelin and Munos (2007) presented a new exploratory policy developed specifically for tree search. Another multi-armed bandit formula was introduced in Stogin et al. (2009). In most modern programs, such explicit exploration policies have been found superfluous, since the *RAVE* generalization technique (explained below) provides sufficient noise (Chaslot et al., 2009).

In Chaslot et al. (2007), domain knowledge was used for the first time as *prior* information in the tree nodes. Pat-

[11] Some researchers have reported a small gain when the binary result is minimally modified according to the point result. I could not replicate this behaviour.

tern values, values for capturing and escaping capture and values for proximity to previous moves artificially modified the winrate in inner nodes of the tree, awarding them for example an extra 50 virtual wins (*heuristic bias*). This has the effect of guiding search as long as few "real" samples are available, but decreasing in importance for well-explored moves. This heuristic information was also used to temporarily prune moves that seem inferior, but without pruning them permanently.

Coulom (2007) used heuristic knowledge in a similar fashion to bias search and to "soft-prune" moves, but expanded it to a variety of features whose values were learned automatically with a variant of the Elo model. Features included *Monte Carlo features*, like the attribute of an intersection of belonging to the opponent in $x\%$ of completed simulations. Learning of patterns for the heuristic bias was also explored in Hoock and Teytaud (2010), utilizing a genetic programming approach.

A feature-based value function acquired offline by reinforcement learning methods was used as prior information in Gelly and Silver (2007). Additionally, this paper introduced the most successful generalization technique for MCTS in Go yet: *RAVE* (*Rapid Action Value Estimation*). In basic MCTS, only the success of simulations with move a made from the current board s can influence the value estimate for choosing a from s. With RAVE, the success of all simulations with move a made from *any subsequent position to* s is included in the estimate.

Similar to Brügmann's AMAF, where move values were computed independent of position, the RAVE technique defines a neighborhood on positions that share win and visit counts of their moves. Any position that follows the current board in a given simulation counts as a neighbor during backpropagation of the return. Helmbold and Parker-Wood (2009) assessed the performance of several RAVE variants, with different neighborhood definitions of varying coarseness.

RAVE estimates can be collected much faster than basic MCTS estimates—if position s occurs 300 moves before the end of a simulation, it will receive only one MCTS up-

date, but 150 RAVE updates in the backpropagation step. However, RAVE winrates are also considerably noisier than MCTS data, since they essentially estimate the value of a move independent of move order in the simulation at hand. They are discounted accordingly, losing in weight as more MCTS updates are collected for the move at hand, and effectively bridge the gap between heuristic estimates and MCTS estimates.

Because moves with bad value estimates in the current position can still appear later on in a playout, RAVE values are continually updated for all moves. They have even proven a suprisingly effective exploration policy. In many modern programs, RAVE has therefore replaced UCT-like confidence intervals (Chaslot et al., 2009; Lee et al., 2009). Berthier et al. (2010) addresses the problem of restoring consistency—the property of converging to the optimal move in the limit—for algorithms with such modified exploration techniques. A typical policy is:

$$\pi(s) = \begin{cases} \underset{a \in A(s)}{\operatorname{argmax}} \left(\hat{Q}^{\pi}(s,a) \cdot (1 - c_{s,a}) + \hat{Q}^{\pi}_{RAVE}(s,a) \cdot c_{s,a} \right) & \text{if } s \in T \\ \operatorname{random}(s) & \text{otherwise} \end{cases} \tag{3.1}$$

where $\hat{Q}^{\pi}_{RAVE}(s,a)$ is the Rapid Action Value Estimate of choosing a in s, and $c_{s,a}$ is the discounting coefficient. Let $n_{s,a}$ be the total number of times move a has been played from position s, and $n^{RAVE}_{s,a}$ the total number of times a has been played in a simulation after s; then the discounting coefficient of e.g. OREGO is:

$$c_{s,a} = \frac{n^{RAVE}_{s,a}}{n^{RAVE}_{s,a} + n_{s,a} + n^{RAVE}_{s,a} \cdot n_{s,a} \cdot x} \tag{3.2}$$

with a weighting factor $x \in \mathbb{R}$.

Drake and Uurtamo (2007b) examined whether Go knowledge is more effectively applied as heuristic bias, influenc-

ing value estimates of tree nodes, or as an informed playout policy. They found that stronger playout policies have a greater effect, while also being more time-consuming.

In Pellegrino et al. (2009), it was found that UCT can be improved by an extension to the value formula for an individual move a. In addition to winrate and confidence interval, a term for the covariance of playing move a and winning a simulation was added. However, the results could not be extended to Go engines with RAVE-type exploration.

3.3.3 Strength of Monte Carlo Go

One additional advantage of Monte Carlo Tree Search that was not mentioned so far is its relatively easy parallelization (Cazenave and Jouandeau, 2007; Chaslot et al., 2008; Enzenberger and Müller, 2009). Tournament and show matches have been played on massively parallel hardware. However, although MCTS scales well with the number of cores, the additional computational resources make the limitation of brute force only more apparent: With increasing numbers of simulations per move, programs experience diminishing improvements in playing strength. Although this has not been researched thoroughly, it is suggested by practical results, e.g. in the scaling studies of Coulom (2006). It hence appears that selection and simulation strategies of MCTS algorithms still have to be improved considerably to play on a par with human masters.

From August 26 to October 4, 2008, the MCTS program MoGo played a number of matches against human players in an event held at National University of Taiwan, including the professional player Jun-Xun Zhou 9p. According to the player's comments in subsequent interviews, MoGo had reached a level of roughly 1p on the 9×9 board, and 2-3d on the 19×19 board. This is an impressive and encouraging achievement.

In the next section, previous work on policies for the simulation phase of MCTS is detailed. Improving these policies is the aim of this thesis.

3.4 Playout Strategies in Monte Carlo Go

In principle, consistency of Monte Carlo Tree Search—that is, convergence to the optimal policy in the limit—can be guaranteed even for truly random playouts, with moves beyond the tree chosen according to a uniform probability distribution[12] (Kocsis and Szepesvári, 2006). However, convergence can be greatly sped up through the use of more informed simulation policies, called *quasi-random* in computer Go. Go knowledge has been used in Monte Carlo playouts even before the introduction of tree search (Bouzy, 2005a), and the strength of the playout policy has been shown to generally correlate well with the strength of the whole MCTS algorithm (Pellegrino and Drake, 2010).

The ongoing efforts to increase the accuracy of playout policies in order to boost MCTS performance are based on a convincing intuition: If vanilla MCTS measures a move's probability of winning *given random play afterwards*, then MCTS with informed playouts estimates a move's probability of winning *given reasonable play*. Furthermore, stronger playouts should need fewer samples to find near-optimal regions of the search space; if we were in possession of a perfect playout policy, a single sample game would tell us the optimal next move.

On the other hand, beginning with Bouzy and Chaslot (2006), it has been repeatedly found that increasing the strength of a playout policy—measured by its strength as a standalone player—is not a sufficient condition for stronger MCTS. Surprisingly, stronger playouts can lead to weaker Monte Carlo Search and vice versa (see also Gelly and Silver (2007) for empirical results).

The reasons for this are at least two-fold: First, the stronger a policy becomes, the more it exploits Go knowledge and the less it explores the space of all possible games. As Pellegrino and Drake (2010) remarks, MCTS with a too deterministic policy does not measure the probability of win-

[12] The usual exception are moves which would fill one of your own eyes—these are rarely useful, and since they could lead to the death of a living group, termination of the playouts in reasonable time could not be guaranteed. See also 7.2.

ning, but only the probability of winning *given a particular play style*. Second, as suggested in Gelly and Silver (2007) and theoretically formulated in Silver and Tesauro (2009), each piece of knowledge incorporated into a playout policy runs the risk of introducing a certain *bias* to the distribution of samples. A policy that is adept at defending groups, for example, but slightly less skilled in attacking them, will lead to gross overestimations of the safety of groups. In this sense, strong MCTS requires a balanced spread of simulations.

In addition, there is a simple tradeoff between strength and speed of computation: Increasing strength leads to more meaningful samples, while increasing speed leads to a greater number of samples and greater statistical confidence in the results.

The next section presents some of the published approaches to the simulation phase of Monte Carlo Tree Search in Go. Section 3.4.1 deals with static policies, not depending on the results of earlier playouts in the search; their move suggestions depend solely on features of the given position. Section 3.4.2 covers first attempts at dynamic policies which improve through online learning.

3.4.1 Static Playout Policies

In the first paper applying MCTS to Go (Coulom, 2006), playout moves of the program CRAZY STONE were chosen according to their *urgencies*. Urgency was a numerical value set to 1 for all moves except for capturing moves and escape moves (saving a string from capture by the opponent), which were strongly favored proportionally to the number of stones they would capture or save. The idea appeared first in Bouzy (2005a). Additionally, various rules were applied to recognize "useless" moves, i.e. captures or escapes that would backfire immediately. The numerical values were arbitrarily chosen and no tuning was performed.

The application of pattern matching in playouts had already been suggested by Brügmann (1993). In Gelly et al.

(2006), describing the application of UCT to Go in the MoGo program, 3×3 intersection patterns inspired by Bouzy (2005a) were introduced to MCTS. Other than previous programs, MoGo tried to match its patterns only around the last-played move on the board instead of globally; this led to the playing of natural-looking local move sequences.

The basic MoGo policy π_{MoGo}, which has also become the default playout policy of Orego, works as follows: 1. If the last move of the opponent put any stones in atari, it tries to save them by capturing or escaping. 2. Otherwise it looks for matches of a set of 13 common patterns[13] around the last-played move. Each pattern is centered around a move candidate and specifies the values of the eight surrounding intersections together with the information of whether the candidate should be played or not. These patterns have been hand-coded to represent Go principles like cutting the opponent's stones, connecting own stones, and playing *hane*[14]. In this way, knowledge about commonly worthwile local tactics is introduced to the playouts. 3. If no pattern matches, it looks for capturing moves anywhere on the board. 4. If no string can be captured, it plays randomly on a legal intersection.

As Silver and Tesauro (2009) note, this small set of simple rules and small patterns has remained the basis of most successful handcrafted playout policies ever since, and "adding further Go knowledge without breaking MoGo's 'magic formula' has proven to be surprisingly difficult". However, in the time leading up to the publication of Chaslot et al. (2009), at least three more heuristics were added, played in this order between the escape- and pattern-heuristics described above: 1. The *nakade* heuristic, trying to play in the center of a three-intersection eye of the opponent (destroying its potential to become two eyes); 2. the "fill board" heuristic, trying to play on one of a small number of randomly chosen locations on the board if they are empty and surrounded only by empty intersections; 3. the "approach move" heuristic, essentially adding excep-

[13] They contain don't-cares which can be resolved into black, white or empty, so the number of distinct matching 3×3 areas is actually higher.

[14] "reaching around" an opponent's stone

tions to the pattern matching rule.

MANGO, the program of Chaslot et al. (2007), used move urgencies as the sum of two values: A capture-escape value similar to the one proposed in Bouzy (2005a) and the value for the surrounding 3×3 pattern learned in Bouzy and Chaslot (2006). Additionally, a proximity factor was used to strongly favor moves adjacent to the last-played move, achieving a similar effect to local pattern matching in MOGO.

More handcrafted heuristics were presented in Cazenave (2007)—a better definition of *eye*[15] and tactics for strings with two liberties—as well as Drake and Uurtamo (2007b)—including "fighting" heuristics for strings with low liberty count.

Bouzy and Chaslot (2006) tried for the first time to *automatically* optimize urgencies for individual 3×3 patterns by reinforcement learning methods; for 9×9 Go, a small improvement was achieved, but it did not translate to the larger 19×19 board. The overhead in computation time was not specified, and it is not clear whether the observed increase in the average point outcome of games actually corresponds to a statistically significant increase in the win-rate.

Another partly successful reinforcement learning approach was tried in Gelly and Silver (2007). Here, the evaluation function learned by reinforcement learning methods in Silver et al. (2007) turned out to make a strong standalone player, but a weak MCTS policy. It was however successfully integrated as a heuristic to bias values of freshly created tree nodes, and the apparent paradox of the failure as Monte Carlo policy ultimately led to the definition of playout balance in Silver and Tesauro (2009).

In Coulom (2007), a probability distribution over legal moves was learned on the basis of considering each move as a "team" of matching features, and learning the values of individual features from game databases according to

[15] with the goal of excluding fewer good moves by never playing in your own eyes

the Bradley-Terry model (Hunter, 2004). The Bradley-Terry model underlies the Elo rating as an estimation of player strength; it has been generalized to estimate the strength of individual players from the results of team games. Here, players were identified with features, while the occurrence of a move in a professional game was considered a win of this move's feature team over all other feature teams on the board. A subset of the learned feature weights, including those for 3×3 patterns, capture, escape, self-atari, and adjacency to the last-played move, was employed as playout policy in the MCTS framework of CRAZY STONE, and achieved a considerable improvement in playing strength.

Drake and Uurtamo (2007a) considered another way of combining various heuristics into a playout policy: In order to avoid the costly computation of all heuristics, a *roulette wheel* scheme was employed in which at most one heuristic is used to choose any given move, determined by random selection according to learned probabilities.

Finally, the problem of playout *imbalance* was examined in Silver and Tesauro (2009), and a a softmax policy parameterized with the weights of location-dependent and location-independent patterns was trained to minimize bias through gradient descent reinforcement learning methods. The results were promising, although experiments have so far only been conducted on 5×5 and 6×6 boards, and in the context of naive one-ply Monte-Carlo search instead of MCTS.

3.4.2 Dynamic Playout Policies

All policies in section 3.4.1 have been handcoded or learned offline. In this section, a brief sketch is given of past work on simulation strategies that use information acquired during search.

Judging a move's potential value partly from its earlier performance during search—the *history heuristic*—is a well-known idea in the field of heuristic search (Schaeffer, 1989). Brügmann's earliest experiments with Monte Carlo Go, based on simulated annealing, already featured this aspect,

albeit not in the sense of a playout policy: The order of all moves in the next playout was determined before playing, giving each move a priority depending on the past success of playouts containing it (see section 3.3). The idea was taken up in Bouzy and Helmstetter (2003) and converted into a playout policy that chooses in every position the move with highest e^{Kv}, where v is the move's current evaluation (winning rate of playouts containing it) and K the inverse of the *temperature* used in simulated annealing.

For Monte Carlo Tree Search, Drake and Uurtamo (2007a) used a variant of the history heuristic focused on the search tree. Instead of estimating the value of a move by the win-rate of all playouts containing it, it was here estimated by the frequency of being chosen as best move by the tree selection strategy. Also, a *second-order* history heuristic was introduced, which kept track not only of frequently chosen moves, but of frequently chosen answers to any given move of the opponent. When a group is attacked, for example, it might be necessary to connect it to another group; this move, however, is only useful when actually provoked and does not need to be prioritized otherwise. When the second-order history heuristic is used, the most frequently chosen move response to the last-played move of the opponent is returned.

In Bouzy (2005b), the authors considered the fact that past playouts contain more information for a player than the final outcome "win" or "loss". The final owner of each intersection of the board is also available—and this information can be averaged to determine which parts of the board are more or less likely to be controlled by the player. The urgency to play on an intersection can then be adjusted to be higher for fiercely contested intersections than for areas whose fate is relatively clear.

This idea was transferred to Monte Carlo Tree Search by Coulom (2007), where the probability of point ownership was used as one positional feature for computing the move probability as described in section 3.4.1. However, the feature was considered too computationally expensive to include it in the subset that was implemented as a MCTS playout policy.

The concept of *criticality*, introduced in Coulom (2009), is a further refined variant of Monte Carlo features: Instead of measuring how likely a given intersection is to belong to the player, criticality looks at past playouts to measure the covariance of owning a given intersection and winning the playout. This way, one can distinguish between areas which have unclear point ownership because they will be divided in a not yet determined way between the two players—giving no one an advantage in the end—and areas which have unclear point ownership because they will fall to a not yet determined player, deciding the game. To my knowledge, no empirical results have yet been published.

Drake (2010), finally, is the paper whose early versions laid the foundation for the work described in chapter 6. The main idea of the *last-good-reply policy* is closely related to the second-order history heuristic described above: It is the idea of remembering your successful answers to the opponent's moves in earlier playouts, and repeating them whenever the opponent makes the same moves again. For example, if connecting my groups successfully defended against an attack and allowed me to win a playout, then the same attack should also be answered by connecting in later playouts.

However, there are two important differences between the two heuristics: First, the last-good-reply policy learns replies from entire playouts, not only from the tree, which allows quicker learning of more replies with higher exploratory spread. And second, the last-good-reply policy only saves the *last* reply that lead to a won playout—no frequency, no winrate, or any other statistics on the quality of the move answer. When the last-good-reply player gets to choose a move, the last-played move of the opponent (say e4) is looked up in a table, and either the last successful answer to it is returned (say f5), or the move decision is delegated to the uniformly random policy (or, in OREGO, to the classic MOGO policy as described in section 3.4.1)[16].

[16] Rimmel and Teytaud (2010) is a recent publication that was brought to my attention shortly before finishing this thesis. While dealing with the game of Havannah and not Go, its idea of *contextual Monte Carlo* is also similar to that of the last-good-reply policy: For every pair of moves, an average score is being maintained indicating how well the player fared in playouts where he played both moves. When the policy

gets to choose a move, the last-played move by the *same* player is looked up, and the move that completes the pair with the currently best score is returned. The use of learning follow-ups to your own moves—instead of replies to your opponent's moves—in Go is touched upon in section 6.3.3.

Chapter 4

Problem Statement

4.1 Current Deficiencies of MCTS

In the last chapters, Monte Carlo Tree Search has been introduced with an emphasis on its advantages in comparison to other search- or knowledge-based approaches to computer Go. This chapter briefly outlines unsolved problems in Monte Carlo Go, followed by a statement of the purpose of this work.

4.1.1 Handicaps

Move choices of Monte Carlo algorithms are based on statistical estimates of winning probabilities. For this reason, they rely on the estimates of good and bad moves being significantly *different*. If a Monte Carlo program is extremely behind or extremely ahead in a game, winning chances of all available moves are so concentrated near 0% or 100% that it becomes difficult to distinguish good from bad moves. To the program, it seems as if it would lose or win anyway, and move choices often start to appear somewhat random to the observer.

In an even game without handicap stones, this is less of a problem, since e.g. having a very large advantage in an

even game is likely the result of being a much stronger player. In this case, the assumption of winning regardless of precise execution tends to be true. In a handicap game however, when Black starts with a number of stones already on the board, the opposite is the case: Black takes the handicap because it is the weaker player. The impression of excellent winning chances from the start is wrong here—in fact, handicap is meant to provide for equal chances. Black needs to play very carefully and solidly in order to give White, the stronger player, fewer opportunities to catch up and overcome the handicap. Careless play resulting from the belief in a sure win is likely to be punished. An analog problem exists for a Monte Carlo program playing as White and giving handicap.

Existing approaches to solving this problem mainly revolve around the concept of *dynamic komi*: When Black is given handicap stones, the outcomes of simulated games are changed by adding a number of points to White's score. As a result, a simulation only counts as won if Black wins by a large margin. This margin is gradually reduced throughout the game, so that Black aims eventually aims for a win in real, unmodified Go. The higher demands on winning the game for Black are intended to counterbalance the handicap, and achieve a simulation winrate of around 50%. However, the technique has been criticized for tricking Black into taking unnecessarily large risks.

At some point in the future, opponent modelling in Monte Carlo playouts may be a viable approach to skewed win distributions. Apart from a first experiment described in section 7.4, the problem is not adressed further in this work.

4.1.2 Narrow Sequences

The probably most important current issue in computer Go is the failure of MCTS in most *semeai*[1], as well as many life and death problems and other situations that require *precise move sequences* to be handled correctly. The reason for

[1] *Semeai* are mutual capturing races between two enclosed, adjacent groups of opposing colors, when both groups can only live by capturing the other group. Fighting semeai is a major part of Go tactics.

this tactical weakness lies in the exploratory randomness of simulation policies.

Imagine a fight where the attacking player has a choice between several threatening moves, but the defending player needs to choose a single specific reply to each of these moves to successfully defend. Human players might easily see these possible defenses and correctly consider the attack futile. The MCTS algorithm however will vastly overestimate its chance of succeeding, because it is more likely to randomly encounter a good attacking move than a good defending move.

To make things worse, even after the game tree has been expanded enough to find the correct answers to the attacks, the problem persists by being continually pushed beyond the search horizon. Other than humans who can successfully generalize from a local solution once it is solved, MCTS has to find the answers independently in every branch of the tree. The search will favor any branch where distractive and ultimately meaningless moves prevent the tree from examining the attack more closely, because these branches generate higher reward than the branches where the futility of attack has already been found. This is can be considered a new and subtle variation of the *horizon problem* known from Alpha-Beta search.

The converse problem arises for the defender, whose simulations return pessimistic results: MCTS will quickly focus on the first branch where the successful move answers are found. Other branches may contain even better play, but are ignored for a long time due to the skewed statistics.

For this reason, the first assumption of this work is that the horizon problem in Monte Carlo Tree Search should be solved by creating stronger playout policies. If the distribution of moves in the random simulations could be improved, narrow sequences would be played out correctly in all branches, and statistical sampling could be used to find strategically strong moves even in situations with close tactical fights.

Due to the tradeoff between knowledge and exploration however, as well as the tradeoff between knowledge and

speed mentioned in the previous chapter, there are certain limits to the depth of tactical knowledge that a playout policy can be equipped with. With the additional risk of biasing simulations—more and more difficult to avoid with increasingly sophisticated policies—the ad-hoc development of strong playouts has become a "dark art" (Lee et al., 2009).

Therefore, it is the second assumption of this thesis that future gains in playout strength will be mainly achieved with *dynamic* strategies—with policies that autonomously acquire knowledge during search, adapting themselves to the current problem.

4.2 Goal of this Work

As discussed in section 3.3, learning in Monte Carlo Go has until now been generally restricted to the tree, where value estimates for positions in the agent's immediate future are constructed from sample returns and improved with transient RAVE estimates and heuristic knowledge. In the simulation phase on the other hand, as shown in section 3.4, it is still common to rely solely on static knowledge. Playout policies are generally handcrafted or acquired offline through machine learning techniques. The inability to improve simulations while searching leads to systematically biased estimates in many critical situations, as explained in section 4.1.2.

Simulated games provide more information than their binary result: The complete sequence of moves leading to the result, even if relatively myopic and random as a result of simple and exploratory policies, can potentially be input to a learning process. The RAVE technique serves as a convincing demonstration. It is the goal of this thesis to utilize this information throughout entire simulated games, not only in the selection phase, but in particular in the simulation phase.

4.2.1 Preliminary Considerations

In the first phase of this work, several objectives for research outcomes were formulated.

- As soon as solutions to subgames or subproblems on the board are found—such as the correct defense against an attack move, or a successful way of invading the opponent's territory—it would be desirable to be able to *reuse* these solutions throughout the rest of the search process.
- In this way, a solved situation could reduce the statistical variance of simulation outcomes, improving the signal-to-noise ratio. However, solutions should always stay "soft", i.e. the search algorithm must not apply them unconditionally without allowing for exceptions or uncertainty, similar to how heuristic estimates or "soft pruning" can be eventually obsoleted by sample returns (see section 3.3.2).
- The basic MCTS algorithm estimates values of future moves without generalization over states. To understand narrow sequences as addressed in section 4.1.2, the search process should be able to acquire knowledge about successful moves, sequences of moves or whole subtrees of the game tree independent of state. That way, they could be executed at any suitable time during a simulation, in both selection and playout phases[2].
- In recognizing and reusing partial strategies, the search algorithm should adapt to the preconditions of these strategies, i.e. it should learn the contexts in which the strategies can be successfully applied. In other words, complete independence of state should be automatically replaced by an appropriate generalization over states.
- As a long-term goal, it would be ideal if knowledge about partial strategies could be gathered in such

[2]The RAVE heuristic is a successful example of learning state-independent move values. Similarly, it would be desirable to infer the state-independent values of entire partial strategies.

a way that statistical consistency guarantees can be made, and convergence to the optimal policy in the limit remains undisturbed (Kocsis and Szepesvári, 2006; Berthier et al., 2010).

4.2.2 Move Answers

It is a well-known property of Go that many moves have correct local replies: They must be answered in a specific way, regardless of the global situation on the rest of the board[3]. In figure 4.1, for example, if White plays at c1, Black has to play at b1 in order to save his group of stones from being captured. The same holds for white g9 and black h9 in the mirrored situation at the top.

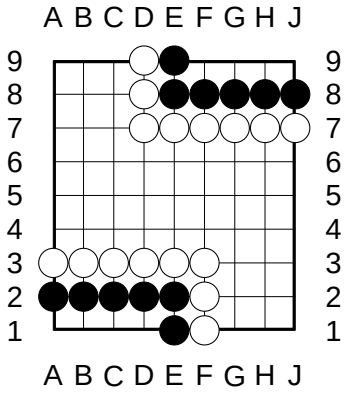

Figure 4.1: Local move answers.

Humans understand the (often partial) independence of such local situations and exploit it by performing local instead of global searches for optimal moves. Basic Monte Carlo Tree Search, however, does not share any information between locally identical board positions: Even after finding the correct answer black b1 to white c1, any distracting moves such as white g9, black h9 change the global

[3]The preference of local over global optimization is natural in many domains: The optimal action after turning on my computer, for example, is usually typing in my password. This is independent of other factors influencing my behavior like the time of day, the weather or even the program I intend to use.

position and force MCTS to re-explore the situation at the bottom from scratch[4].

In AI research on chess (cf. Campbell et al. (2002)), the term *forced moves* has been coined for such move answers. In the framework of Alpha-Beta search, they are usually accounted for by increasing the search depth in all branches where forced moves occur (because they reduce search width). As for previous work on Go, Cazenave (1998) proposed a system for automatically generating logic programs that define forced moves. Although Brügmann (1993) already suggested maintaining "not just the average value of each move but the average value after a particular other move was played first", no publications on the subject with respect to Monte Carlo Go seem to exist.

The main idea underlying this thesis is the exploitation of locally optimal move replies for the goal of creating adaptive playout policies. The value of a move depending on the *previous move* (or moves) in the simulation is estimated and used for action selection. In this way, generalization between states is based on their appearance in move sequences. Instead of applying only static knowledge during playouts, new policies are created that are informed by the contexts of positions, and by knowledge acquired on-line about successful actions in these contexts.

4.3 Experimental Framework

All work for this thesis is based on version 6.10 of the OREGO Go program, provided under the GNU General Public License by Peter Drake at Lewis & Clark College, Portland, Oregon (Drake et al., 2009). OREGO features an MCTS player with RAVE-based selection policy (section 3.3.2) and MOGO-like simulation policy (section 3.4.1). This basic algorithm achieves a rating of 11 kyu on the KGS Go

[4]The justification being that in a slightly modified situation, my milk may be boiling over, making running to the kitchen a superior action choice over typing my password. As mentioned in section 3.2, correct judgment on the independence of subgames has been a challenge in computer Go since the beginning.

Server and served as the baseline for performance comparisons in the following chapters. RAVE was used in all conditions.

Unless stated otherwise, experiments were conducted by playing a set of games against version 3.8 of GNUGO (Foundation, 2009). GNUGO was set to its default strength level of 10. Area counting ("Chinese rules"), positional superko and a komi of 7.5 were used. OREGO and GNUGO played equal numbers of games as Black and White.

In order to leave optimization and fine-tuning of the strength-speed tradeoff to further work, most experiments were not time-controlled, but used the same number of simulations per move decision in all conditions.

The statistical significance of all results is assessed by two-tailed z-tests for two proportions. Unfortunately, playing a sufficient number of games to show significant differences between conditions is a computationally expensive process. All experiments with inconclusive results are ongoing.

Chapter 5

Adaptive Playouts via Best Replies

This chapter describes the first approach to using move replies in playout policies—the use of a move value estimate that is conditioned on the previous moves in the game.

5.1 Conditional Value Estimates

In Monte Carlo Tree Search, distinct values of states or state-action pairs in the immediate future of the agent are estimated (section 2.7.2). The case of state-action value estimation is repeated here for convenience. After every episode of experience $s_1, a_1, s_2, a_2, \ldots, s_T, r$, the estimate of each state-action pair (s_t, a_t) in the search tree is updated using the return from that episode:

$$n_{s_t,a_t} \longleftarrow n_{s_t,a_t} + 1 \tag{5.1a}$$

$$\hat{Q}^\pi(s_t, a_t) \longleftarrow \hat{Q}^\pi(s_t, a_t) + \frac{1}{n_{s_t,a_t}}\big(r - \hat{Q}^\pi(s_t, a_t)\big) \tag{5.1b}$$

In order to determine optimal move replies for the simulation phase, a separate set of value estimates can be main-

tained: The expected returns of each possible move when played as a reply to each possible move of the opponent. Let $P = \{\text{White}, \text{Black}\}$ be the set of players, and $p_t \in P$ the player to move at timestep t. At the end of an episode, the following additional update step is performed for each action $a_t, t \geq 2$:

$$n_{a_{t-1},a_t,p_t} \longleftarrow n_{a_{t-1},a_t,p_t} + 1 \tag{5.2a}$$

$$\hat{V}^{\pi}_{reply}(a_{t-1}, a_t, p_t) \longleftarrow \hat{V}^{\pi}_{reply}(a_{t-1}, a_t, p_t) + \frac{r - \hat{V}^{\pi}_{reply}(a_{t-1}, a_t, p_t)}{n_{a_{t-1},a_t,p_t}} \tag{5.2b}$$

where n_{a_{t-1},a_t,p_t} is the number of times move a_t has been chosen by player p_t directly after move a_{t-1} had been played by the opponent, and $V^{\pi}_{reply} : S \times S \times P \mapsto \mathbb{R}$ is defined by:

$$V^{\pi}_{reply}(a, b, p) = E_{\pi}\left[R_t | a_{t-1} = a, a_t = b, p_t = p\right] \tag{5.3}$$

This way, every move (except the first) is thought of as a reply to the previous move, and its value as reply is estimated independently of state.

The corresponding policy is improved by

$$\pi(s_t) = \begin{cases} \underset{a \in A(s_t)}{\operatorname{argmax}} \left(\hat{Q}^{\pi}(s_t, a) \cdot \left(1 - c_{s_t,a}\right) + \hat{Q}^{\pi}_{RAVE}(s_t, a) \cdot c_{s_t,a} \right) & \text{if } s_t \in T \\ \underset{a \in A(s_t)}{\operatorname{argmax}} \left(\hat{V}^{\pi}_{reply}(a_{t-1}, a, p_t) \right) & \text{otherwise} \end{cases} \tag{5.4}$$

As suggested in Brügmann (1993), "this could be called the first order approach, and there clearly is a generalization to n-th order". More informative than the value of move a_t as a successor to move a_{t-1} ($V^{\pi}_{reply}(a_{t-1}, a_t, p_t)$) is

the value of move a_t as a reply to moves a_{t-1} and a_{t-2} ($V^{\pi}_{reply-2}(a_{t-2}, a_{t-1}, a_t, p_t)$). Conditioning on increasingly longer contexts should result in fewer samples, but also increasingly precise value approximations, since larger parts of the local[1] context are considered. Generalization only conflates moves in the more distant past.

However, longer contexts pose a computational challenge in terms of time and space complexity. Concerning space, maintaining value estimates for all black and white moves conditioned on all possible three-move histories already requires almost 34 billion floating-point variables, which is infeasible for current working memory constraints. "In the limit of large n the complete game information has been stored and the method becomes equivalent to an exhaustive tree search" (Brügmann, 1993). Concerning time, considering all possible history lengths of all moves in a simulation is quadratic in the number of moves.

5.2 Move Answer Tree

Two solutions to the complexity problem have been considered in this work: First, the restriction of the maximal history length to a predetermined value; second, the use of a data structure that adapts to the histories actually appearing in simulations.

Only if move a_t frequently appears in playouts, it is promising to estimate the value of its successor a_{t+1} conditioned on a_t. And only if move a_{t+1} is repeatedly played as a reply to move a_t, it is worthwile to take both a_t and a_{t+1} into account as context for the value estimation of a subsequent move a_{t+2}. By selectively growing a *move answer tree* into the direction of actually occurring answers, the playout policy should be able to focus learning on the most relevant move sequences for the current search.

This leads to an algorithm maintaining conditional move

[1] In Go, spatial and temporal locality often coincide. Situations in parts of the board are played out until sufficiently settled, and the focus of play changes to another area.

value estimates in a similar fashion to how MCTS maintains value estimates for the moves following the root state. But while the MCTS tree only grows into the root state's future—not generalizing between any two positions it encounters—the move answer tree would grow into the future of *any* state. It is intended to learn optimal sequences of actions independent of state.

The idea of the move answer tree is treating any two positions of the game as identical if the previous x moves leading up to the positions have been identical. Moves to play in a simulation would be chosen on the basis of value estimates conditioned on these x moves. For situations and move sequences that keep reappearing, x would grow, and thus the policy would become more precise. More and more complex partial policies would be represented in the move answer tree, leaving an increasingly smaller part of move decisions to random sampling.

5.2.1 Outline of Implementation

The move answer tree has not been fully implemented yet. While this section proposes its basic workings, the next section presents the results of first experiments and explains the obstacles encountered.

Figure 5.1 shows a toy instance of the tree data structure[2]. Depending on technical details such as whether the move representation of a program includes the player ("White e6", "Black q17" etc.) or not ("e6", "q17"), the move answer tree can be implemented as a single tree for both players or as two seperate trees with Black and White to move at the respective root nodes. In this example, seperate trees are chosen for clarity. Only one of two move answer trees is shown: The one with Black to move at the root node.

Thus, the root r represents any game state where it is Black's turn. Every branch of the tree represents a move by one of the players—Black's moves at the first level of

[2] Just like the MCTS tree, the tree becomes a directed acyclic graph if transpositions are accounted for.

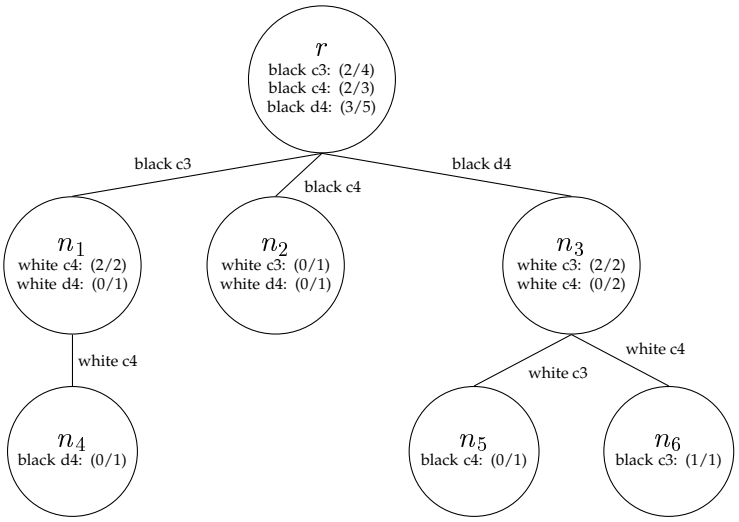

Figure 5.1: The move answer tree.

the tree, White's moves at the second level, and so on alternatingly. Every non-root node n stands for the set of game states whose immediately preceding moves are the moves on the path from the root to node n. For example, n_1 in the toy example stands for any game state that occurs directly after the move black c3.

Similar to the action value estimates or state-action value estimates in the nodes of the regular MCTS tree, move answer tree nodes contain value estimates for the represented states, or actions following the represented states. In this example, state-action value estimation is used, so nodes contain run and win counts for the emanating branches—the actions made from the represented states. For example, n_1 in the tree above contains the information that white d4 has been played one time from a position reached by black c3, and that this playout has been eventually lost by white (0 wins in 1 run). Optionally, additional variables for RAVE estimates could be included.

From the point of view of generalized policy iteration, learning in the move answer tree proceeds as follows:

- Policy evaluation: After each episode of experience, the value estimates of each chosen action are updated

with the return from that episode. In the regular MCTS tree this update is limited to a single state-action value estimate in a single node. In the move answer tree however, which does not feature a one-to-one correspondence of states and tree nodes, each action can potentially lead to updates for several nodes.

As an example: After the moves black d4 – white c4 – black c3 in a simulation won by Black, the run and win counts for Black's move c3 would be updated in the root node r (where the values of black moves are estimated independent of context), as well as in node n_6 (where the values of black moves after black d4 – white c4 are estimated)[3]. The tree resulting from all updates is pictured in figure 5.2.

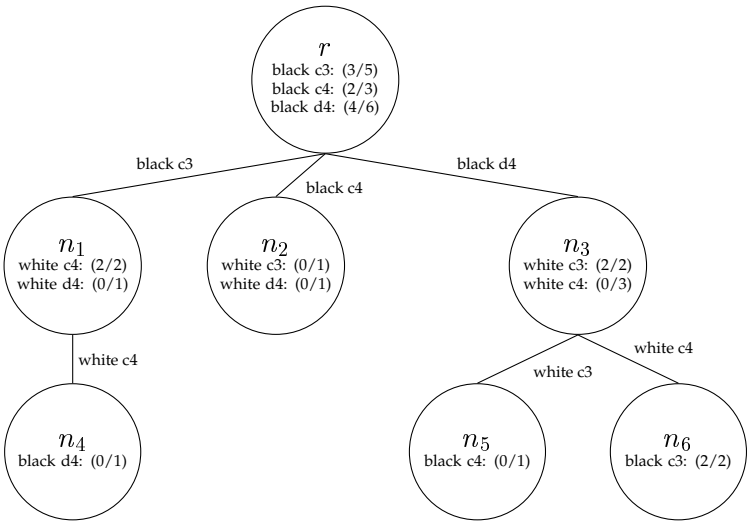

Figure 5.2: Updated move answer tree.

New nodes are created whenever the parent node's run count exceeds a given value. As an example, let the threshold value be one. The moves black c4 – white c3 have only been observed once so far, as recorded in node n_2 . If this sequence is played again, white c3's run count in n_2 is raised to two, which triggers the creation of a new child node of n_2. This child node can store more detailed information about move

[3]The value of black c3 after white c4 would be updated in a node at depth one of the other move answer tree, with White to move at the root.

sequences starting with black c4 – white c3 in the future. For the case that the playout containing black c4 – white c3 has been won by White, the result is shown in figure 5.3.

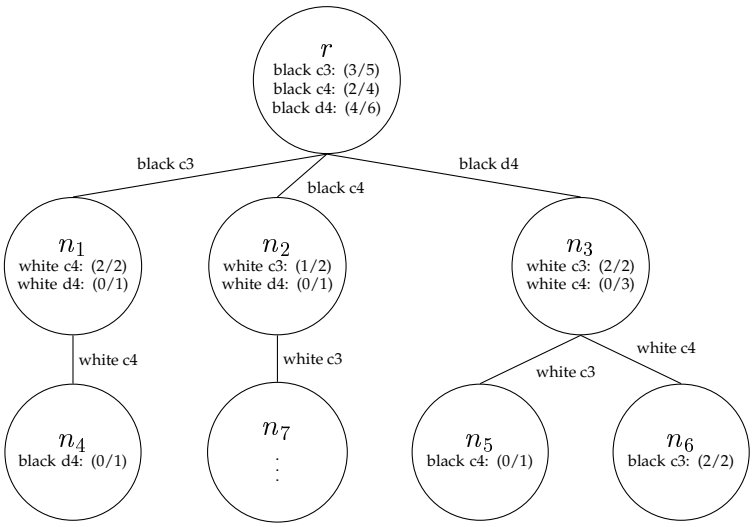

Figure 5.3: Extended move answer tree.

- Policy improvement: During each episode, the agent chooses its actions depending on their current value estimates. Just like there can be several value estimates involved in the update step of a single action, several estimates may need to be considered to determine the total value of a legal action during simulation. After the moves black d4 – white c3, for example, the total value estimate of the possible next move c4 by Black can be influenced by the value of c4 independent of context (represented in the root node r), but also by the value of c4 as an answer to white c3 (represented in the the tree not pictured here) and as an answer to black d4 – white c3 (represented in node n_5).

  These values, found at different depths in the tree, represent differently broad generalizations. Their combination into a single floating-point figure could be handled analogously to the combination of MCTS and RAVE estimates. Shallower contexts would provide noisier, but more readily available estimates; they would be discounted as more information for

deeper contexts arrives. To encourage exploration, a measure of uncertainty would be added, similar to the confidence bounds in UCT (section 2.7.1).

In this way, the objectives laid out in chapter 4 could be addressed: Solutions to subproblems could be stored and reused while searching—in the form of successful replies to frequent move sequences. Uncertainty of solutions could be handled through managing exploration and exploitation with a bandit algorithm, similar to MCTS. All solutions found would be independent of state and hence applicable at any time in a simulation; however, they would be dependent on increasingly long move sequences, which could provide for appropriate generalization.

Two issues are not completely resolved yet: First, the precise relative *weighting of estimates* from different tree depths needs a principled derivation. If RAVE turns out to be successful in this context as well, estimates will need to be weighted across two dimensions of generalization.

Second, due to the fast tree growth it might be advantageous or even necessary to delete and *reuse tree nodes* while the tree is being built. It remains to be tested whether this is done best on the basis of node age, or on the basis of a more sophisticated measure of the success of a node.

5.3 Preliminary Experiments

Before fleshing out all details of the move answer tree, several experiments were conducted to test the viability of the approach. In these experiments, the tree structure was reduced to a single level: Move value estimates were solely conditioned on one previous move of the opponent, as described in equation 5.2. The purpose was to examine the effect of conditional value estimates in their simplest form.

5.3.1 Improvement of Local Play

The game position shown in figure 5.4, adapted from Coulom (2009), was used as a basis for the first two tests.

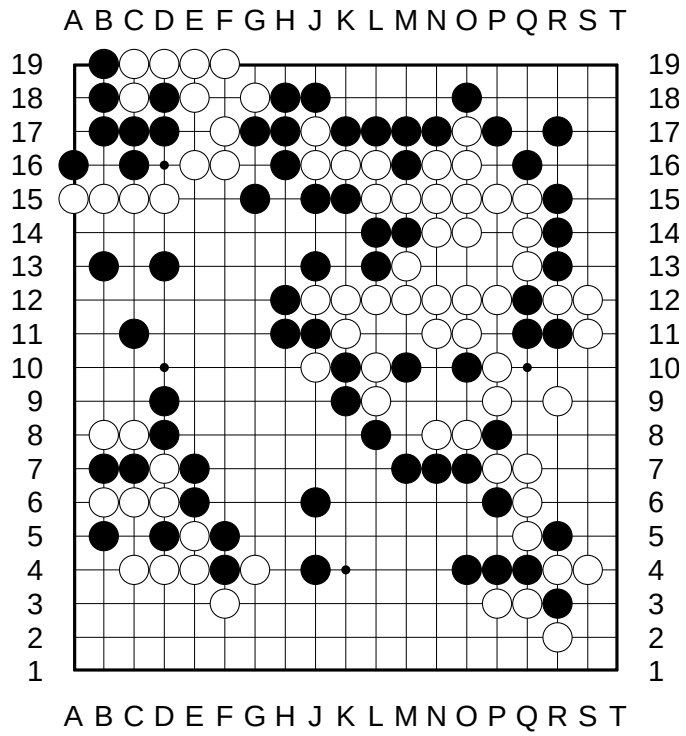

Figure 5.4: A position with a local fight.

Key to the outcome of this game is the situation in the top left corner, where a black group has been surrounded by a white group. Crucially, Black would have to play on both b16 and a18 to form two eyes and save his group from being captured. With competent play by White, he will not be able to do so, since White can take at least one of the two intersections even if Black plays first. For this reason, the black group is dead.

However, White needs to answer Black's moves correctly: b16 needs to be answered with a18, and a18 with b16. If this is not recognized and adhered to during the simulations of Monte Carlo Tree Search, the statistical result will be skewed in favor of Black's winning chances, and the po-

sition will be assessed incorrectly. It is a position in which local plays, more specifically local move replies, are decisive.

In the first experiment, both the standard MCTS player of OREGO (referred to as MCTS from here on) and a *best-move-reply* player with conditional value estimates according to equation 5.2 (henceforth called BMR) ran 25,000 simulated games on the above position. After every playout, the eventual owner of every intersection of the board was stored. If local moves are properly answered, the intersections of both the black group and the surrounding white group belong to White at the end of a game; if the correct move sequences are not played, Black has a chance of saving his group (and even killing the surrounding white group). Therefore, the hypothesis is: The percentage of simulations in which intersections a16, b17, b18, b19, c16, c17, d17, and d18 (the dead black group) end up belonging to White should be higher in the BMR condition than in the MCTS condition, indicating superior local play.

The policy of the BMR player was a modified version of equation 5.4, intended to increase exploration. It does not play the highest-valued *legal* move answer, but tries to play the highest-valued move answer *among all vacant intersections*. If this is illegal, the algorithm falls back to the standard MOGO policy used by MCTS.

$$\pi(s_t) = \begin{cases} \underset{a \in A(s_t)}{\operatorname{argmax}} \left(\hat{Q}^{\pi}(s_t, a) \cdot (1 - c_{s_t,a}) + \hat{Q}^{\pi}_{RAVE}(s_t, a) \cdot c_{s_t,a} \right) & \text{if } s_t \in T \\ \underset{a \in A_{vacant}(s)}{\operatorname{argmax}} \left(\hat{V}^{\pi}_{reply}(a_{t-1}, a, p_t) \right) & \text{if } \underset{a \in A_{vacant}(s)}{\operatorname{argmax}} (\ldots) \in A(s_t) \\ \pi_{MoGo}(s_t) & \text{otherwise} \end{cases} \tag{5.5}$$

where $A_{vacant}(s)$ is the set of vacant intersections in position s. This results in about 76% of playout moves being chosen for exploitation of value estimates, and 24% for exploration according to the standard policy.

The experiment was repeated 100 times for both the MCTS and the BMR condition.

The results of the first experiment confirmed the hypothesis: The intersections in question fell to White on average in 58.1% of simulations in the MCTS condition (mean absolute error 2.0%), and on average in 69.2% of simulations in the BMR condition (mean absolute error 3.2%). Using value estimates of move replies seems to improve local play.

In the second experiment, the MCTS and BMR players again ran 25,000 simulations on the above position. This time, the number of correct move replies to black b16 was stored; white a18 was counted as correct, any other white move as incorrect. In order to reduce noise, all simulations where a17, a18 or a19 had been played before black b16 were discarded. This excludes most exceptions in which white a18 is not the optimal answer to black b16.

This was, again, repeated 100 times for both the MCTS and the BMR condition.

The results of the second experiment again suggested a positive effect of conditional move value estimates: In the MCTS condition, Black played b16 on average 3498 times in 25,000 playouts (mean absolute error 667.5), and White answered correctly in on average 1308 (37.4%, mean absolute error 407.3) of cases. In the BMR condition, Black attempted to save his group on average 7423 times in 25,000 playouts by playing b16 (mean absolute error 3415.2), and White countered correctly in on average 5298 (71.4%, mean absolute error 2899.7) of these simulations.

5.3.2 Failure in Game Performance

As shown in the previous experiments, the BMR playout policy shows promise in the handling of locally correct move sequences. In the next step, the BMR player's performance in actual game-playing was tested.

In figure 5.5, the performance of the BMR policy is compared to that of the standard MCTS player of OREGO.

BMR played 94 games against GNUGO (47 as White, 47 as Black), and MCTS played 1302 games (651 as White, 651 as Black) against the same program. OREGO simulated 25,000 games per move.

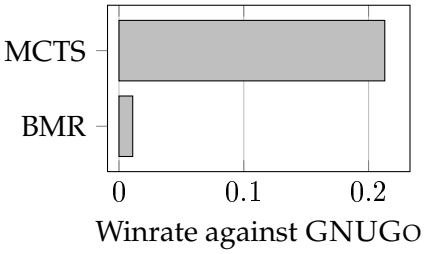

Figure 5.5: Performance of the BMR policy.

The results show that BMR has a significantly lower winrate than plain MCTS ($p < 0.001$). In fact, BMR has only been able to win a single game against GNUGO out of 94.

Another variant of the best-move-reply policy, BMR-2, is based on the idea to give the static MOGO policy priority over V^{π}_{reply} value estimates—using escaping, capturing or pattern moves whenever applicable and thus supporting search with human knowledge. Only in cases where no heuristic returns a move and the MOGO policy would resort to random move choice, BMR-2 tries to play the intersection with highest reply value estimate. As last fallback, a move is chosen randomly.

$$\pi(s_t) = \begin{cases} \underset{a \in A(s_t)}{\operatorname{argmax}} \left(\hat{Q}^{\pi}(s_t, a) \cdot (1 - c_{s_t,a}) + \hat{Q}^{\pi}_{RAVE}(s_t, a) \cdot c_{s_t,a} \right) & \text{if } s_t \in T \\ \pi_{MoGo}(s_t) & \text{if a capturing, escaping or pattern move is available} \\ \underset{a \in A_{vacant}(s)}{\operatorname{argmax}} \left(\hat{V}^{\pi}_{reply}(a_{t-1}, a, p_t) \right) & \text{if } \underset{a \in A_{vacant}(s)}{\operatorname{argmax}} (\ldots) \in A(s_t) \\ \operatorname{random}(s_t) & \text{otherwise} \end{cases} \tag{5.6}$$

where $\mathrm{random}(s)$ picks any move $a \in A(s)$ with uniform probability. This policy is intended to change the balance between heuristic and acquired knowledge. It results in only about 21% of playout moves being chosen according to reply value estimates, in comparison to 76% for BMR as described above.

In figure 5.6, the performance of the BMR-2 policy is compared to that of the standard MCTS player of OREGO. BMR-2 played 128 games against GNUGO (64 as White, 64 as Black), and MCTS played 1302 games (651 as White, 651 as Black) against the same program. OREGO simulated 25,000 games per move.

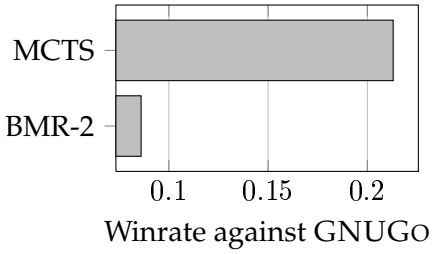

Figure 5.6: Performance of the BMR-2 policy.

Like BMR, BMR-2 performs significantly worse than plain MCTS ($p < 0.001$).

Reply value estimates seem to succeed in finding and exploiting narrow move sequences, but fail in the multitude of positions arising in real games—including many positions in which narrow sequences do not play a major role. It could be speculated that this is a result of the simultaneous adaptation of all move answers throughout a playout: Whenever a given reply shows good results in its context, the other player is likely to change earlier moves in each playout, creating a changed context. The estimated returns may not be able to adapt quickly enough to these changes.

But even if they could be modified to adapt more quickly by increasing exploration, the risk of getting trapped in local optima could only be traded against the risk of never converging to a meaningful solution. By attempting to optimize *every* move reply throughout a playout, the search

algorithm is given the task of finding the optimal playout, essentially the optimal game of Go. This is obviously infeasible and does not meet the requirements stated in chapter 4, which asked for *partial* strategies and solutions to *local* problems.

To approach this problem, it may be necessary to introduce a distinction between replies that actually belong to narrow move sequences—that are "forced moves" in a wider sense— and replies without forcing context, which should be sampled according to more general heuristics such as those provided by the MoGo policy. In other words: Conditional value estimates could be used to recognize and exploit narrow sequences without interfering with the rest of simulations.

In order to explore this idea, two variants of BMR have been implemented that employ move replies depending on their estimated return. If the maximal $\hat{V}^{\pi}_{reply}$ in a given position is considerably higher than the average $\hat{V}^{\pi}_{reply}$ of all legal moves (if the best reply stands out), it is considered a forced move and played with high probability. If however all moves are valued similarly as reply to the previous move (if $\hat{V}^{\pi}_{reply}$ does not identify a clear favourite answer), the standard policy is used with high probability instead. In these cases, the policy takes advantage of the current position's features instead of replying to the previous move. This way, narrow sequences are intended to be distinguished from regular play.

Formally, in the BMR-P (for probabilistic best-move-reply) policies the probability of playing the highest-valued move answer to move a_{t-1} in position s_t is defined by:

$$P_{answer}(a_{t-1}, s_t) = \frac{\hat{V}^{\pi}_{reply}(a_{t-1}, a^*, p_t) - \overline{V}^{\pi}_{reply}(a_{t-1}, p_t)}{1 - \overline{V}^{\pi}_{reply}(a_{t-1}, p_t)} \tag{5.7}$$

where

$$a^* = \underset{a \in A_{vacant}(s_t)}{\operatorname{argmax}} \hat{V}^{\pi}_{reply}(a_{t-1}, a, p_t) \tag{5.8a}$$

$$\overline{V}^{\pi}_{reply}(a_{t-1}, p_t) = \frac{1}{|A_{vacant}(s_t)|} \sum_{a \in A_{vacant}(s_t)} \hat{V}^{\pi}_{reply}(a_{t-1}, a, p_t) \tag{5.8b}$$

Similar to the BMR policy, the first variant BMR-P gives $\hat{V}^{\pi}_{reply}$ value estimates priority over the static MoGo policy:

$$\pi(s_t) = \begin{cases} \underset{a \in A(s_t)}{\operatorname{argmax}} \Big(\hat{Q}^{\pi}(s_t, a) \cdot (1 - c_{s_t,a}) + \hat{Q}^{\pi}_{RAVE}(s_t, a) \cdot c_{s_t,a} \Big) & \text{if } s_t \in T \\ & \text{with probability } P_{answer}(a_{t-1}, s_t) \\ \underset{a \in A_{vacant}(s)}{\operatorname{argmax}} \Big(\hat{V}^{\pi}_{reply}(a_{t-1}, a, p_t) \Big) & \text{if } \underset{a \in A_{vacant}(s)}{\operatorname{argmax}} (\ldots) \in A(s_t) \\ \pi_{MoGo}(s_t) & \text{otherwise} \end{cases} \tag{5.9}$$

In figure 5.7, the performance of the BMR-P policy is compared to that of the standard MCTS player of Orego. BMR-P played 186 games against GNUGo (93 as White, 93 as Black), and MCTS played 1302 games (651 as White, 651 as Black) against the same program. Orego simulated 25,000 games per move.

The winrate of BMR-P could not shown to be significantly different from that of MCTS.

Analogously to BMR-2, the second variant BMR-P-2 prioritizes escape, pattern and capture moves, if available, over highest-rated move replies:

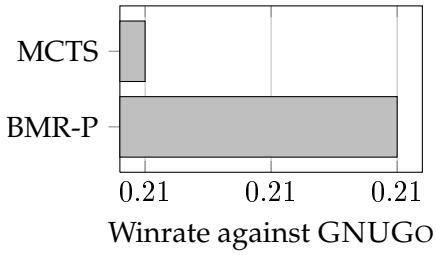

Figure 5.7: Performance of the BMR-P policy.

$$\pi(s_t) = \begin{cases} \underset{a \in A(s_t)}{\operatorname{argmax}} \left(\hat{Q}^{\pi}(s_t, a) \cdot (1 - c_{s_t,a}) + \hat{Q}^{\pi}_{RAVE}(s_t, a) \cdot c_{s_t,a} \right) & \text{if } s_t \in T \\ \pi_{MoGo}(s_t) & \text{if a capturing, escaping or pattern move is available} \\ \underset{a \in A_{vacant}(s)}{\operatorname{argmax}} \left(\hat{V}^{\pi}_{reply}(a_{t-1}, a, p_t) \right) & \text{with probability } P_{answer}(a_{t-1}, s_t) \text{ if } \underset{a \in A_{vacant}(s)}{\operatorname{argmax}} (\ldots) \in A(s_t) \\ \operatorname{random}(s_t) & \text{otherwise} \end{cases} \tag{5.10}$$

In figure 5.8, the performance of the BMR-P-2 policy is compared to that of the standard MCTS player of OREGO. BMR-P-2 played 728 games against GNUGO (364 as White, 364 as Black), and MCTS played 1302 games (651 as White, 651 as Black) against the same program. OREGO simulated 25,000 games per move.

The winrate of BMR-P-2 is significantly lower than that of MCTS ($p < 0.01$).

5.3.3 Discussion

No tested variant of move reply value policies has so far shown promise in preliminary experiments. For this rea-

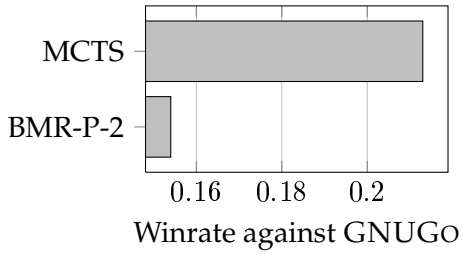

Figure 5.8: Performance of the BMR-P-2 policy.

son, work on the move answer tree algorithm has stalled.

Conditional value estimates seem to be a viable way of finding narrow move sequences and improving local play, but appear to severely hinder global search at the same time. In further work, better methods might be found to seperate these search objectives and divide move choices more effectively between adaptive, move reply based and traditional, position feature based sampling techniques.

A first attempt in this direction are the BMR-P policies, which do perform better than plain BMR—unfortunately, they are still not superior to the baseline of MCTS+RAVE with static playouts. Furthermore, it is possible that their relative success can simply be attributed to a smaller percentage of adaptive move choices: BMR chooses ca. 76% of moves according to learned reply value estimates, BMR-P only ca. 27%.

The basic idea of BMR-P is the concept of distinguishing between moves with "forced" answers and moves that can be "freely" answered in various ways without drastically lowering winning chances. BMR-P's simple way of making this distinction (equation 5.7) could be refined: For example on the basis of a statistical measure of relation between playing a given move reply and winning a simulation. This would represent an extension of the work of Coulom (2009) and Pellegrino et al. (2009), who examined measures of relation between playing a standalone move and winning a simulation.

The tentative goal of this approach is the restriction of best

move replies to the cases where playing the best reply actually correlates highly with success in the game[4]. The next chapter of this thesis takes a completely different point of view: Instead of focusing on the question, "Which move reply works best?" it asks "Which move reply worked last?"

[4]The approach potentially generalizes to other aspects of the game. As an example, high correlation between winning a simulation and killing a given group of the opponent could be used as motivation for playing corresponding attacks with higher probability.

Chapter 6

Adaptive Playouts via Last Good Replies

This chapter reports on the second approach to using move replies in playout policies—the use of *last good replies*, i.e. the move replies that have last appeared in a won playout. More generally, this chapter deals with simulation algorithms that modify the policy without explicitly estimating and maximizing values.

The basis for this approach is the last-good-reply policy (Drake, 2010) as outlined in section 3.4.2. It is therefore presented in more detail in the first section. The following sections examine various extensions and variants.

6.1 The Last-Good-Reply Policy

The last-good-reply or LGR policy by Drake chooses a very different route to adaptive playouts than that described in the previous chapter. Whenever a player wins a playout, every move a_t made after a move a_{t-1} of the opponent during the simulation is considered a successful answer. Instead of using a large number of sampled games to estimate reply values by averaging returns, the last-good-reply policy maintains only *one move answer* for each opponent move—the last successful one. In subsequent playouts, this

move is the first choice to play every time the corresponding move of the opponent appears. If no LGR-1 move is available, the standard MoGo policy is the fallback.

For updating the LGR policy, the following additional assignment step is performed for each action $a_t, t \geq 2$ after a simulation is completed:

$$\text{Reply}^{p_t}_{LGR}(a_{t-1}) \longleftarrow \begin{cases} a_t & \text{if } p_t \text{ won the playout} \\ \text{Reply}^{p_t}_{LGR}(a_{t-1}) & \text{otherwise} \end{cases} \tag{6.1}$$

where $\text{Reply}^{p}_{LGR}(a)$ is the last successful reply of player p to move a, initialized to a special constant `no_point` before the start of the search. The policy of *Orego* becomes

$$\pi(s_t) = \begin{cases} \underset{a \in A(s_t)}{\text{argmax}} \left(\hat{Q}^{\pi}(s_t, a) \cdot (1 - c_{s_t,a}) + \hat{Q}^{\pi}_{RAVE}(s_t, a) \cdot c_{s_t,a} \right) & \text{if } s_t \in T \\ \text{Reply}^{p_t}_{LGR}(a_{t-1}) & \text{if } \text{Reply}^{p_t}_{LGR}(a_{t-1}) \in A(s_t) \\ \pi_{MoGo}(s_t) & \text{otherwise} \end{cases} \tag{6.2}$$

The last-good-reply-2 policy (LGR-2) naturally extends LGR to the context of two previous moves: Every move a_t is here considered an answer to the player's own a_{t-2} and the opponent's a_{t-1}. Whenever $\text{Reply}^{p}_{LGR-2} : A \times A \mapsto A$ provides no stored answer in the simulation phase, the policy defaults to LGR-1.

The reply table of LGR-1 contains $2x$ entries in total, where x is the number of intersections—361 on the standard 19×19 board. The total amount of memory needed for LGR-2 is $2(x + x^2)$ integers (261,364 for 19×19).

6.1.1 Experimental Results

In figure 6.1, the performance of the LGR-1 and LGR-2 policies is compared to that of the standard MCTS player of OREGO. LGR-1 played 1196 games against GNUGO 3.8 (598 as White, 598 as Black), LGR-2 played 636 games (318 as White, 318 as Black), and MCTS played 1302 games (651 as White, 651 as Black) against the same program. OREGO simulated 25,000 games per move.

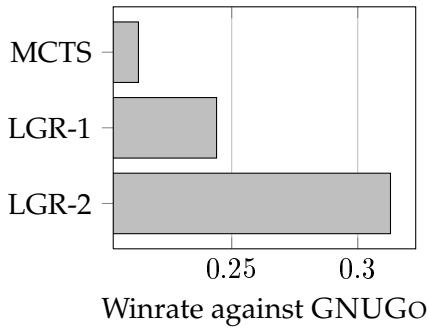

Figure 6.1: Performance of the LGR policies.

The results show that LGR-2 has a significantly higher winrate than LGR-1 ($p < 0.01$), while LGR-1 is almost significantly stronger than MCTS ($p < 0.07$). More samples will here be needed to show a significant difference.

6.2 Forgetting

One drawback of the last-good-reply policy is that move replies, once stored, cannot be overwritten easily. Only if a different reply is chosen in a simulated game—either because the stored reply is illegal in the situation at hand, or because the move choice is made in the selection phase—and if the game is also won eventually, will an existing entry in the reply table change.

The idea of the last-good-reply policy with forgetting (LGRF) is to not only attribute *success* to the move replies made during a simulation, but also *failure*. In addition to

learning that the winner's replies were good, LGRF learns that the loser's replies did not work, and deletes them from the reply table if present. While policy improvement remains essentially unchanged, the policy evaluation step of LGRF is for each $a_t, t \geq 2$:

$$\text{Reply}^{p_t}_{LGRF}(a_{t-1}) \longleftarrow \begin{cases} a_t & \text{if } p_t \text{ won the playout} \\ \texttt{no\_point} & \text{if } p_t \text{ lost the playout and } \text{Reply}^{p_t}_{LGRF}(a_{t-1}) = a_t \\ \text{Reply}^{p_t}_{LGRF}(a_{t-1}) & \text{otherwise} \end{cases} \tag{6.3}$$

As a result, the contents of the table fluctuate more quickly. The algorithm increasingly explores alternate replies, and while suboptimal replies can be forgotten more easily, answers of higher quality still reappear frequently.

Analogously to LGR without forgetting, LGRF can be conditioned on longer lists of previous moves: LGRF-2, LGRF-3 etc.

The space requirements of LGRF are identical to those of LGR: 722 integers for LGRF-1, 261,364 for LGRF-2 on the 19×19 board.

6.2.1 Experimental Results

In figure 6.2, the performance of the LGRF-1 and LGRF-2 policies is compared to that of LGR-1 and LGR-2 as described in the previous section. LGRF-1 played 790 games against GNUGO (395 as White, 395 as Black), LGRF-2 played 756 games (378 as White, 378 as Black), LGR-1 played 1196 games (598 as White, 598 as Black), and LGR-2 played 636 games (318 as White, 318 as Black). OREGO simulated 25,000 games per move.

The experiments show the following significant differences:

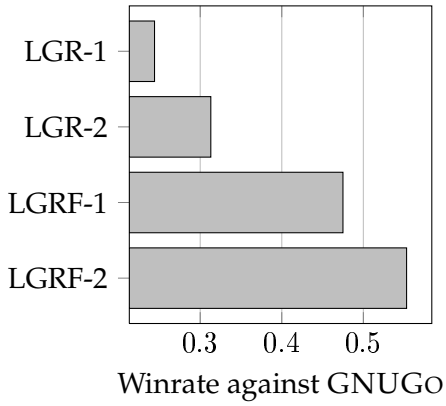

Figure 6.2: Performance of the LGRF policies.

LGR-1 < LGR-2 < LGRF-1 < LGRF-2 ($p < 0.01$). LGR with forgetting is clearly stronger than without.

Figure 6.3 shows how LGR and LGRF policies scale with computation time. LGR-1 played 826 games with 10,000 playouts per move, 374 games with 25,000 playouts per move, and 444 games with 50,000 playouts per move. LGR-2 played 1248 games with 10,000 playouts per move, 608 games with 25,000 playouts per move, and 708 games with 50,000 playouts per move. LGRF-1 played 554 games with 10,000 playouts per move, 448 games with 25,000 playouts per move, and 454 games with 50,000 playouts per move. LGRF-2 played 714 games with 10,000 playouts per move, 1256 games with 25,000 playouts per move, and 316 games with 50,000 playouts per move. In all conditions, equal numbers of games were played as Black and White.

The strength relations LGR-1 < LGR-2 < LGRF-1 < LGRF-2 are significant at all tested numbers of playouts ($p < 0.01$), except for LGR-1 and LGR-2 at 50,000 playouts which did not perform significantly different in the experiment[1].

[1]The scaling results shown in figure 6.3 and the following experiments in this thesis show weaker performance than the previous experiments shown in figures 6.1 and 6.2. This is due to a bug in OREGO that was brought to my attention shortly before the end of my thesis and has only been removed in figures 6.1 and 6.2 so far. The bug consists in a missing initialisation of tree nodes. All following experiments will be repeated; until then, the results up to 6.2 can only be directly compared to each other, not to those shown in later figures. Relative improvements

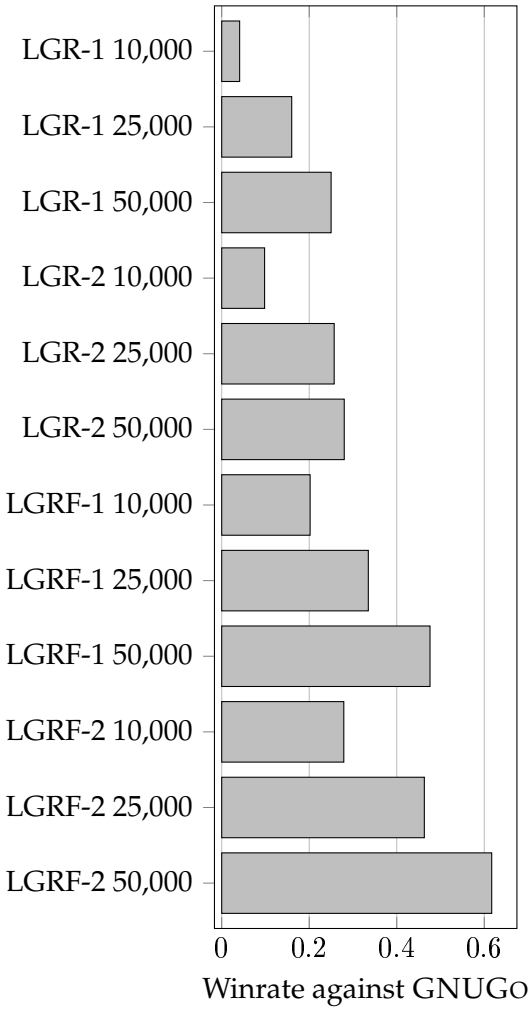

Figure 6.3: Scaling of the LGR and LGRF policies.

The LGRF policy has been described in a paper prepared in collaboration with the author of OREGO (Baier and Drake, 2010).

as shown by individual experiments should remain untouched by this problem.

6.3 Variants

This section presents further experiments with various forms of learning on the basis of the last-good-reply idea. None of the following policies make use of the law of large numbers by averaging simulation returns; instead, they use single appearances of moves in successful playouts as evidence for their efficacy.

6.3.1 Per-Node LGR

The search tree of OREGO, similar to that of other MCTS-based programs, grows to tens of thousands of nodes during a move search. It spans across very different positions, and move replies learned in one branch of the tree might not necessarily be useful in another branch.

The idea of per-node LGR (PNLGR) is the integration of the reply tables into the tree nodes, such that move answers can be learned separately for every position added to the tree. Whenever a new node is added in the expansion phase of MCTS, move answers are inherited from the mother node. During the simulation phase, PNLGR only uses move replies that are stored in the leaf node from which the playout was started. After a simulation has ended, information about successful or unsuccessful move replies is included in the information backpropagated through the visited nodes in the tree; it is not used to update global reply tables.

Forgetting is implemented in PNLGR. The policy evaluation step used by the policy is, for all positions s in the tree that the simulation traversed and for all $a_t, t \geq 2$:

$$\text{Reply}_s^{p_t}(a_{t-1}) \longleftarrow \begin{cases} a_t & \text{if } p_t \text{ won the playout} \\ \texttt{no\_point} & \text{if } p_t \text{ lost the playout and } \text{Reply}_s^{p_t}(a_{t-1}) = a_t \\ \text{Reply}_s^{p_t}(a_{t-1}) & \text{otherwise} \end{cases} \tag{6.4}$$

where $\text{Reply}_s^p(a)$ is the last successful reply of player p to move a in a simulation that traversed position s. The policy improvement step is

$$\pi(s_t) = \begin{cases} \underset{a \in A(s_t)}{\text{argmax}} \left(\hat{Q}^\pi(s_t, a) \cdot (1 - c_{s_t,a}) + \hat{Q}^\pi_{RAVE}(s_t, a) \cdot c_{s_t,a} \right) & \text{if } s_t \in T \\ \text{Reply}_{s_l}^{p_t}(a_{t-1}) & \text{if } \text{Reply}_{s_l}^{p_t}(a_{t-1}) \in A(s_t) \\ \pi_{MoGo}(s_t) & \text{otherwise} \end{cases} \tag{6.5}$$

where s_l is the last node of the simulation that is contained in the tree.

PNLGR requires space for $2xn_T$ integers, where n_T is the maximal number of tree nodes—the default in OREGO is 25,000. This amounts to 18,050,000 for the 19×19 board. Due to memory constraints, PNLGR has not been extended to two-move contexts.

Experimental Results

In figure 6.4, the PNLGR policy is compared to LGRF-1 as described in section 6.2. PNLGR played 476 games against GNUGO (238 as White, 238 as Black), and LGRF-1 played 448 games (224 as White, 224 as Black). OREGO simulated 25,000 games per move.

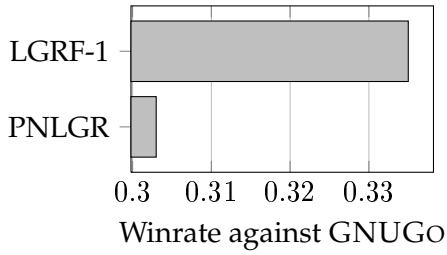

Figure 6.4: Performance of the PNLGR policy.

No significant difference between PNLGR and LGRF-1 could be shown.

6.3.2 Pattern LGR

As discussed in section 4.2.2, correct move answers in Go often depend on their local context on the board. The previous move is only a small aspect of that context, and a sequence of several previous moves is increasingly unlikely to be concentrated in a small local area.

In Pattern LGR (PLGR), move replies are conditioned not only on the previous move, but also on the 3×3 intersection neighborhood of that move on the board. For each such neighborhood, a distinct reply is stored. Due to data sparseness, regular LGRF is implemented as fallback in case the pattern around the previous move is not connected to a successful answer yet.

PLGR uses forgetting. The policy evaluation step added by the policy is for all $a_t, t \geq 2$:

$$\text{Reply}^{p_t}_{N(a_{t-1})}(a_{t-1}) \longleftarrow \begin{cases} a_t & \text{if } p_t \text{ won the playout} \\ \texttt{no\_point} & \text{if } p_t \text{ lost the playout and } \text{Reply}^{p_t}_{N(a_{t-1})}(a_{t-1}) = a_t \\ \text{Reply}^{p_t}_{N(a_{t-1})}(a_{t-1}) & \text{otherwise} \end{cases} \tag{6.6}$$

where $N(a)$ is the 3×3 intersection neighborhood of move a, and $\text{Reply}^p_N(a)$ is the last successful reply of player p to move a when $N(a) = N$. The policy improvement step is

$$\pi(s_t) = \begin{cases} \underset{a \in A(s_t)}{\text{argmax}} \left(\hat{Q}^\pi(s_t, a) \cdot (1 - c_{s_t,a}) + \hat{Q}^\pi_{RAVE}(s_t, a) \cdot c_{s_t,a} \right) & \text{if } s_t \in T \\ \text{Reply}^{p_t}_{N(a_{t-1})}(a_{t-1}) & \text{if } \text{Reply}^{p_t}_{N(a_{t-1})}(a_{t-1}) \in A(s_t) \\ \text{Reply}^{p_t}_{LGR}(a_{t-1}) & \text{if } \text{Reply}^{p_t}_{LGR}(a_{t-1}) \in A(s_t) \\ \pi_{MoGo}(s_t) & \text{otherwise} \end{cases} \tag{6.7}$$

PLGR requires space for $2xn_{3\times3}$ integers, where $n_{3\times3}$ is the number of possible 3×3 neighborhoods of a given intersection (7641). For the 19×19 board, that is 5,516,802 integers.

Experimental Results

In figure 6.5, the PLGR policy is compared to LGRF-1 as described in section 6.2. PLGR played 894 games against GNUGO (447 as White, 447 as Black), and LGRF-1 played 448 games (224 as White, 224 as Black). OREGO simulated 25,000 games per move.

No significant difference between PLGR and LGRF could be shown. Even if the slight improvement turns out to be significant with more samples, it must be noted that the

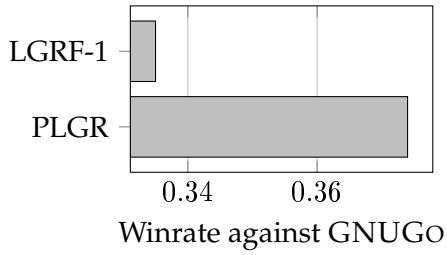

Figure 6.5: Performance of the PLGR policy.

PLGR policy is about three times slower than the LGRF policy.

6.3.3 Last Good Follow-ups

The underlying observation of the last-good-reply policy is that certain moves are answered almost reflexively by humans, which makes them relatively independent units in planning. However, other types of moves are closely connected as well—for example several moves of the same player in a planned invasion of the opponent's territory. No matter at which precise timing the invasion is started, these moves will have to be played in the same order in order to be effective.

The last-good-follow-up policy (LGF) conditions moves a_t not only on previous moves a_{t-1} by the opponent, but also on previous moves a_{t-2} by the same player. By storing moves that were successful follow-ups to other moves in past simulations, a player is enabled to stick to a partial strategy without being distracted by possibly ineffective replies by the opponent. Follow-ups are played as fallback when no replies are stored.

LGF uses forgetting. The policy evaluation step added by the policy is for all $a_t, t \geq 2$:

$$\text{Followup}^{p_t}_{LGF}(a_{t-2}) \longleftarrow \begin{cases} a_t & \text{if } p_t \text{ won the playout} \\ \texttt{no\_point} & \text{if } p_t \text{ lost the playout and } \text{Followup}^{p_t}_{LGF}(a_{t-2}) = a_t \\ \text{Followup}^{p_t}_{LGF}(a_{t-2}) & \text{otherwise} \end{cases} \tag{6.8}$$

where $\text{Followup}^{p}_{LGF}(a)$ is the last successful follow-up of player p to move a. The policy improvement step is

$$\pi(s_t) = \begin{cases} \underset{a \in A(s_t)}{\text{argmax}} \left(\hat{Q}^{\pi}(s_t, a) \cdot (1 - c_{s_t,a}) + \hat{Q}^{\pi}_{RAVE}(s_t, a) \cdot c_{s_t,a} \right) & \text{if } s_t \in T \\ \text{Reply}^{p_t}_{LGR}(a_{t-1}) & \text{if } \text{Reply}^{p_t}_{LGR}(a_{t-1}) \in A(s_t) \\ \text{Followup}^{p_t}_{LGF}(a_{t-2}) & \text{if } \text{Followup}^{p_t}_{LGF}(a_{t-2}) \in A(s_t) \\ \pi_{MoGo}(s_t) & \text{otherwise} \end{cases} \tag{6.9}$$

LGF requires space for $2 \cdot 2x$ entries in total, where x is the number of intersections—half of that for replies, and half for follow-ups.

Experimental Results

In figure 6.6, the LGF policy is compared to LGRF-1 as described in section 6.2. LGF played 362 games against GNUGO (181 as White, 181 as Black), and LGRF-1 played 448 games (224 as White, 224 as Black). OREGO simulated 25,000 games per move.

No significant difference between LGF and LGRF could be shown.

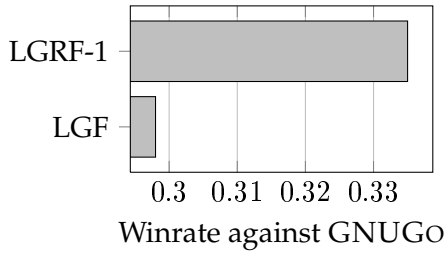

Figure 6.6: Performance of the LGF policy.

6.3.4 Indirect LGR

The idea of the successful Rapid Action Value Estimation technique (section 3.3.2) is that the value of playing a move from a given position can be approximated by the value of playing the move at any time after that position. The last-good-reply simulation policy, however, only stores actual direct replies.

The indirect last-good-reply policy (ILGR) extends LGR to moves played in a window of fixed length after a given opponent move. It stores any move a_t in a won playout not only as a (direct) reply to move a_{t-1}, but also as an (indirect) reply to moves a_{t-3} and a_{t-5} by the opponent[2]. When simulating a game, direct answers are tried first; if illegal, indirect answers are tried.

When a simulation is lost,

- ILGR-a deletes all moves a_t as direct or indirect move reply to a_{t-1}
- ILGR-b deletes a_t as a direct or indirect reply to a_{t-1}, a_{t-3} and a_{t-5}

The policy evaluation step added by ILGR for playouts won by p_t is for all $a_t, t \geq 2$:

$$\forall i \in \{1, 2, 3\} \,.\, \text{Reply}_i^{p_t}(a_{t-(2i+1)}) \longleftarrow a_t \tag{6.10}$$

[2]The number of three previous moves was chosen arbitrarily.

where $\text{Reply}_i^p(a)$ is the ith reply of player p to move a. For playouts lost by p_t, the following step is added by ILGR-a for all $i \in \{1,2,3\}$ and for all $a_t, t \geq 2$:

$$\text{Reply}_i^{p_t}(a_{t-1}) \longleftarrow \texttt{no\_point} \text{ if } \text{Reply}_i^{p_t}(a_{t-1}) = a_t \quad (6.11)$$

while ILGR-b instead adds, for all $i \in \{1,2,3\}$ and for all $j \in \{1,3,5\}$ and for all $a_t, t \geq 2$:

$$\text{Reply}_i^{p_t}(a_{t-j}) \longleftarrow \texttt{no\_point} \text{ if } \text{Reply}_i^{p_t}(a_{t-j}) = a_t \quad (6.12)$$

The policy improvement step of both policies is

$$\pi(s_t) = \begin{cases} \underset{a \in A(s_t)}{\text{argmax}} \left(\hat{Q}^\pi(s_t,a) \cdot (1 - c_{s_t,a}) + \hat{Q}^\pi_{RAVE}(s_t,a) \cdot c_{s_t,a} \right) & \text{if } s_t \in T \\ \text{Reply}_1^{p_t}(a_{t-1}) & \text{if } \text{Reply}_1^{p_t}(a_{t-1}) \in A(s_t) \\ \text{Reply}_2^{p_t}(a_{t-1}) & \text{if } \text{Reply}_2^{p_t}(a_{t-1}) \in A(s_t) \\ \text{Reply}_3^{p_t}(a_{t-1}) & \text{if } \text{Reply}_3^{p_t}(a_{t-1}) \in A(s_t) \\ \pi_{MoGo}(s_t) & \text{otherwise} \end{cases} \quad (6.13)$$

ILGR in this form requires space for $3 \cdot 2x$ entries in total—for each player, three sets of answers to every intersection on the board.

Experimental Results

In figure 6.7, the ILGR policies are compared to LGRF-1 as described in section 6.2. ILGR-a played 540 games against GNUGO (270 as White, 270 as Black), ILGR-b played 282 games (141 as White, 141 as Black), and LGRF-1 played

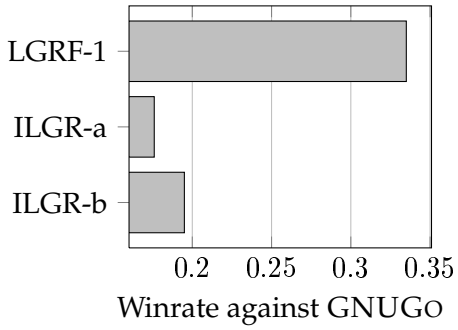

Figure 6.7: Performance of the ILGR policies.

448 games (224 as White, 224 as Black). OREGO simulated 25,000 games per move.

Both ILGR-a and ILGR-b perform significantly worse than LGRF-1 ($p < 0.001$).

6.3.5 Multiple Reply LGR

The last-good-reply policy stores only one reply to any given opponent move. If this move gets deleted after a lost simulation, or if it is just illegal in a playout situation, no good reply is available anymore, and the policy falls back to the classic MOGO policy. In this section, all experiments are based on the idea of storing several successful move replies to one opponent move.

Multiple Replies sorted by Age

The MLGR-A policy stores successful moves and deletes unsuccessful moves a_t conditioned on the previous two moves a_{t-1} and a_{t-2}, similar to LGR-2. Other than LGR-2, it provides a variable number of reply slots per context instead of just one.

When a move reply a_t is made in a won playout, but a reply to a_{t-1} is already stored, LGR overwrites the old reply with

the new one. The intuition behind this is improvement—the new reply is probably going to be more relevant in future playouts than the old one. However, if the new entry is deleted at some point, it would be desirable to know which other candidate replies already had at least one successful trial—instead of starting exploration from scratch, the old entry could be reused.

In MLGR-A, a predetermined number of reply slots per opponent move is maintained. New successful replies fill up empty slots; when all slots are full, the oldest entry is replaced. Unsuccessful replies are deleted, allowing older replies to advance in rank and be chosen again. The same holds for answers to two-move contexts.

As for the simulation phase, a new question arises: It is known from experiments with LGR-1 and LGR-2 that an answer conditioned on two moves (a_{t-1} and a_{t-2}) should be preferred to an answer conditioned on just one move (a_{t-1}). But when old replies are available, is even an old two-move answer more useful than a new one-move answer? In other words, does age matter more than context length?

In order to test this, three different simulation strategies have been implemented for MLGR-A.

- In MLGR-A-a, only the youngest available replies are played. The order of priority is: Youngest two-move answer → youngest one-move answer → fallback to MoGo policy. Older replies only serve as backups for the case the younger ones fail in a later playout.
- In MLGR-A-b, older replies are tried as well, but context is given priority over age. The ranking is: Two-move answers, from youngest to oldest → one-move answers, from youngest to oldest → fallback to MoGo policy.
- In MLGR-A-c, age is given priority over context. The ranking is: Youngest answers, from two-move to one-move context → oldest answers, from two-move to one-move context → fallback to MoGo policy.

MLGR-A requires space for $2xn_{slots}$ entries in total, where n_{slots} is the number of stored replies per opponent move.

Experimental Results

In figure 6.8, the MLGR-A policies are compared to LGRF-2 as described in section 6.2. MLGR-A-a with two replies per context played 666 games against GNUGO (333 as White, 333 as Black), MLGR-A-b with two replies per context played 606 games (303 as White, 303 as Black), MLGR-A-c with two replies per context played 450 games (225 as White, 225 as Black), MLGR-A-c with three replies per context played 348 games (174 as White, 174 as Black), and LGRF-2 played 1256 games (628 as White, 628 as Black). OREGO simulated 25,000 games per move.

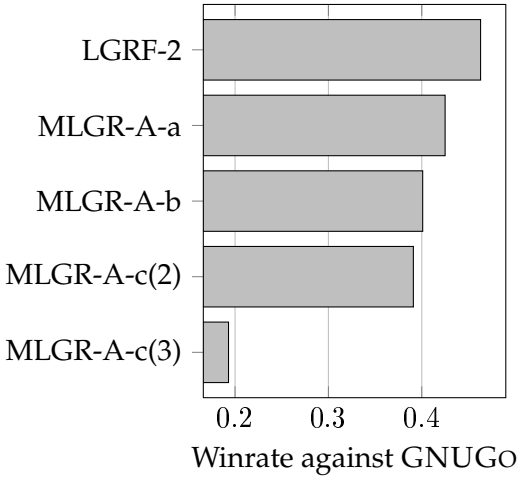

Figure 6.8: Performance of the MLGR-A policies.

MLGR-A-b ($p < 0.05$) and MLGR-A-c ($p < 0.01$ for two replies per context, $p < 0.001$ for three replies per context) are significantly weaker than LGRF-2. MLGR-A-a could not shown to be significantly different in strength from LGRF-2.

Multiple Replies Unsorted

MLGR-A, as defined in the previous section, stores several replies per context. These replies can be distinguished by their age, as newer ones are always ranked ahead of older ones. The MLGR-U policy stores several replies without any ranking or sorting: Winning replies are saved, losing replies are deleted, and whenever the policy is called for a move choice, one of the available replies is picked at random.

Only one-move contexts have been implemented in MLGR-U so far. Two variants have been compared that differ in their handling of replacement: Whenever a new successful move reply arrives, but the corresponding reply slots are already filled,

- MLGR-U-a discards the arrival and uses the old replies until forgetting frees up slots;
- MLGR-U-b randomly picks one of the old replies and overwrites it with the new one.

The space requirements of MLGR-U are identical to those of MLGR-A.

Experimental Results

In figure 6.9, the MLGR-U policies are compared to LGRF-1 as described in section 6.2. MLGR-U-a played 472 games against GNUGO (236 as White, 236 as Black), MLGR-U-b played 478 games against GNUGO (239 as White, 239 as Black), and LGRF-1 played 448 games (224 as White, 224 as Black). OREGO simulated 25,000 games per move.

Both MLGR-U-a and MLGR-U-b perform significantly worse than LGRF-1 ($p < 0.001$).

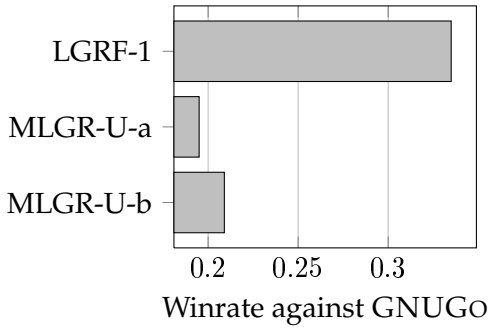

Figure 6.9: Performance of the MLGR-U policies.

6.3.6 Decaying LGR

Experiments with forgetting suggest that fluctuation has a positive effect on LGR. Newer replies seem to be more valuable than older ones, and holding on to outdated information seems to decrease playout quality.

LGRF, however, offers only two ways to replacing old replies: Deletion after a lost playout, or replacement after a won playout that offers a new response in the respective context. The last-good-reply policies with forgetting and decaying (LGRF-D) seek to answer the question: Should a move reply also be forgotten on the grounds that it is simply old, i.e. that it has not been tried for a longer period of time and is therefore less likely to be applicable anymore?

Two variants have been implemented. LGRF-2-D-a uses only move replies found in the last won playout; all stored replies stem from the same simulation. This reduces the average number of available replies, but could also improve their compatibility—it is not obvious that move replies from a number of different simulations can work when applied together.

The second variant, LGRF-2-D-b, stores move replies for only two simulations and forgets them afterwards. If the last two simulations were won by the same player, answers from both are available to him; if the last two simulations were lost, the reply table is cleared completely. Assuming that the search has now switched to another branch, replies

have to be relearned[3].

The policy evaluation step of both LGRF-2-D policies is identical to that of LGRF-2. Additionally, a playout counter is maintained throughout the search, and creation dates are stored with every move answer. During sampling, only replies created at the last won playout or at most two playouts ago, respectively, are considered.

LGRF-2-D needs the same amount of space as regular LGRF-2 (261,364 integers for the 19×19 board).

Experimental Results

In figure 6.10, the LGRF-2-D policies are compared to LGRF-2 as described in section 6.2. LGRF-2-D-a played 302 games against GNUGO (151 as White, 151 as Black), LGRF-2-D-b played 340 games (170 as White, 170 as Black), and LGRF-2 played 1256 games (628 as White, 628 as Black). OREGO simulated 25,000 games per move.

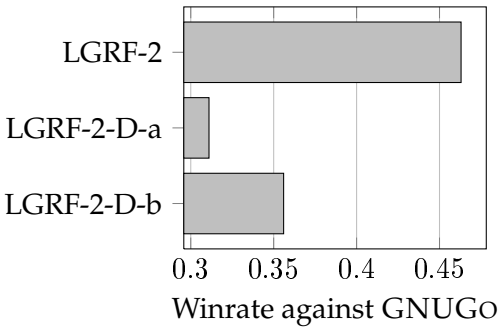

Figure 6.10: Performance of the LGRF-2-D policies.

Both LGRF-2-D-a and LGRF-2-D-b perform significantly worse than LGRF-2 ($p < 0.001$).

[3]The number of two simulations was chosen arbitrarily.

6.3.7 Last Bad Replies

The last-good-reply policy stores information about successful move answers. These answers are preferred in subsequent playouts. Forgetting allows for the deletion of such preference when a stored move answer fails. However, there is no opposite to preference—no mechanism for storing information about failed move replies in order to avoid them in the future.

The last-bad-reply policy (LBR) maintains, in addition to the last-good-reply table, a table for the last move replies that appeared in a lost playout. When a move a_t has to be chosen, the last good reply to a_{t-1} is tried first, if available; if it does not exist or cannot be played, the MoGo policy is used as fallback—but the last bad reply to a_{t-1} is excluded from the move choice. Last bad replies are never deleted, but quickly overwritten as soon as a playout is lost despite having used a different move in the corresponding context.

LBR-2 provides both last-good-reply tables of LGRF-2, and additionally two last-bad-reply tables, with unsuccessful move answers conditioned on one or two moves, respectively.

The policy evaluation step added by LBR-1 is for all $a_t, t \geq 2$:

$$\mathrm{Reply}^{p_t}_{LBR}(a_{t-1}) \longleftarrow \begin{cases} a_t & \text{if } p_t \text{ lost the playout} \\ \mathrm{Reply}^{p_t}_{LBR}(a_{t-1}) & \text{otherwise} \end{cases} \tag{6.14}$$

where $\mathrm{Reply}^{p}_{LBR}(a)$ is the last reply of player p to move a in a playout that was eventually lost by p. Policy improvement with LBR-1 combines LGR and LBR information as follows:

$$\pi(s_t) = \begin{cases} \underset{a \in A(s_t)}{\operatorname{argmax}} \left(\hat{Q}^{\pi}(s_t, a) \cdot (1 - c_{s_t,a}) + \hat{Q}^{\pi}_{RAVE}(s_t, a) \cdot c_{s_t,a} \right) & \text{if } s_t \in T \\ \text{Reply}^{p_t}_{LGR}(a_{t-1}) & \text{if Reply}^{p_t}_{LGR}(a_{t-1}) \in A(s_t) \setminus \text{Reply}^{p_t}_{LBR}(a_{t-1}) \\ \pi_{MoGo}\left(s_t, A(s_t) \setminus \text{Reply}^{p_t}_{LBR}(a_{t-1})\right) & \text{if available} \\ \text{random}(s_t) & \text{otherwise} \end{cases} \tag{6.15}$$

where $\pi_{MoGo}(s, X)$ are the three heuristics of the MoGo policy, restricted to the moves in set X.

LBR-1 requires space for $2 \cdot 2x$ entries in total, where x is the number of intersections—half of that for good, and half for bad replies. LBR-2 needs $4(x + x^2)$ integers, consequently (522,728 for the 19×19 board).

Experimental Results

In figure 6.11, the LBR-2 policy is compared to LGRF-2 as described in section 6.2. LBR-2 played 500 games against GNUGo (250 as White, 250 as Black), and LGRF-2 played 1256 games (628 as White, 628 as Black). Orego simulated 25,000 games per move.

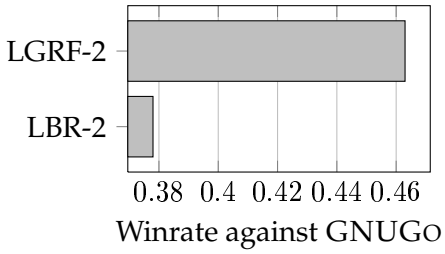

Figure 6.11: Performance of the LBR-2 policy.

LBR-2 performs significantly worse than LGRF-2 ($p <$

0.01).

6.3.8 LGR Priority

While LGR presents knowledge specific to the current situation—move replies that have been successfully used in the current search—it generalizes relatively strongly. Basic LGR-1 equates all positions s_t with equal a_{t-1}, and complex features of the position itself, such as whether a group is in atari, are only considered by the MoGo fallback policy if no last good move is available. However, a group being in danger seems to be far more informative than a single move being played by the opponent.

The experiment presented in this section represents a test whether reacting to the last move of the opponent should actually be ranked as first or as second priority. Contender for first priority is the most important heuristic of the classic MoGo policy—moves that avoid capture.

The escape-before-last-good-reply policy with forgetting (ELGRF) first tries to escape, if any group is in atari; if not, tries to play a stored move reply; if unavailable, tries to match a local pattern; if no pattern matches, tries to capture a group of the opponent and finally, makes a random move if nothing else was viable.

Space requirements of LGR remain unchanged.

Experimental Results

In figure 6.12, the ELGRF-2 policy is compared to LGRF-2 as described in section 6.2. ELGRF-2 played 236 games against GNUGo (118 as White, 118 as Black), and LGRF-2 played 1256 games (628 as White, 628 as Black). Orego simulated 25,000 games per move.

ELGRF-2 performs significantly worse than LGRF-2 ($p < 0.01$).

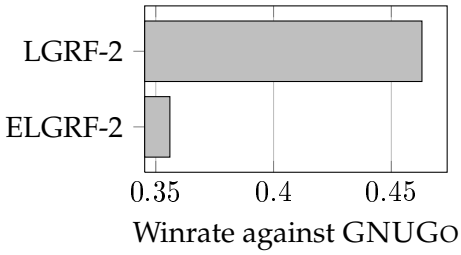

Figure 6.12: Performance of the ELGRF-2 policy.

6.3.9 Ignoring the Tails of Playouts

As mentioned in section 2.4, Monte Carlo estimates have very high variance. The success of a simulation depends on many move choices by both players—more than 400 on average—and the further down the tree these choices are made, the less informed they are. The majority of moves is chosen relatively random in the playout phase beyond the tree.

Regarding RAVE estimates that depend on all moves in a simulation, some researchers have therefore experimented with ignoring the *tails* of playouts (the last 30% to 50% of moves) during learning. The intuition is that these late moves are too far removed from reasonable play and introduce too much noise.

In the following experiment, it was explored whether the acquisition of move replies by LGRF should also be restricted to the first part of simulations, before too much noise accumulates through suboptimal moves. This first part was arbitrarily chosen to extend 200 moves starting from the root position. The LGRF-2 policy was modified accordingly, resulting in the LGRF-MAX200 policy.

The policy evaluation step of LGRF-MAX200 is identical to that of LGRF-2, but only executed for all $a_t, 2 \leq t \leq 200$.

Space requirements of LGR remain unchanged.

Experimental Results

In figure 6.13, the LGRF-MAX200 policy is compared to LGRF-2 as described in section 6.2. LGRF-MAX200 played 482 games against GNUGO (241 as White, 241 as Black), and LGRF-2 played 1256 games (628 as White, 628 as Black). OREGO simulated 25,000 games per move.

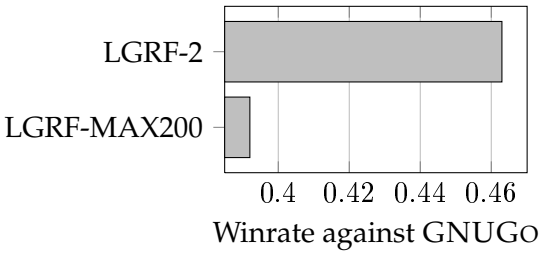

Figure 6.13: Performance of the LGRF-MAX200 policy.

LGRF-MAX200 performs significantly worse than LGRF-2 ($p < 0.01$).

6.3.10 Ignoring Captured Stones

During simulated games, many seemingly poor and meaningless moves are made by the simple playout policies. It would be desirable to restrict LGR learning to move replies that not only appear in good games—good enough to be won eventually—but are actually good moves themselves.

A naive definition of "good move" is: A good move is a move that remains on the board until the end of the game, a stone the opponent is not able to capture. In games between humans, this is obviously not true—stones and whole groups are regularly traded or sacrificed for territory in other parts of the board, and stones without a chance of survival can still have significant influence on the game. In the case of a playout policy, however, many stones are captured eventually because they were poorly played.

In the following experiment, the last-good-reply policy with forgetting for surviving stones (LGRF-2-S) is a mod-

ification of LGRF-2 which only stores surviving replies, stones that were still on the board when the simulation had ended[4].

Space requirements of LGR remain unchanged.

Experimental Results

In figure 6.14, the LGRF-2-S policy is compared to LGRF-2 as described in section 6.2. LGRF-2-S played 438 games against GNUGO (219 as White, 219 as Black), and LGRF-2 played 1256 games (628 as White, 628 as Black). OREGO simulated 25,000 games per move.

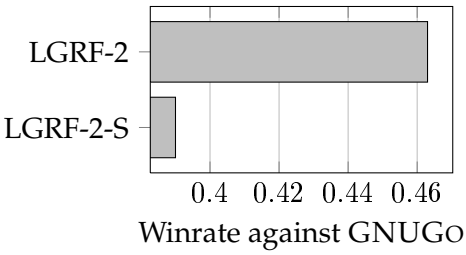

Figure 6.14: Performance of the LGRF-2-S policy.

LGRF-2-S performs significantly worse than LGRF-2 ($p < 0.01$).

6.3.11 Last Good Moves

LGR-2 returns moves conditioned on two previous moves, a_{t-2} and a_{t-1}; LGR-1 returns moves conditioned on one previous move, a_{t-1}. Instead of falling back to the static MOGO policy in the case no suitable reply is available, good replies conditioned on zero moves could be learned—or in other words, good moves.

The last-good-move policy (LGM) is related to what Brügmann called the "zeroth-order approach" (Brügmann,

[4]For the sake of simplicity, this does not detect stones that were captured mid-game, but played again later.

1993), the estimation of move values independent of context. LGM instead stores a single boolean value for every intersection and for every player: $\text{Goodmove}^{p}_{LGM}(a)$ is initialized to `false`, set to `true` whenever move a by player p is part of a won simulation, and set to `false` again when player p made move a in a simulation he eventually lost. The corresponding policy evaluation step is for all a_t:

$$\text{Goodmove}^{p_t}_{LGM}(a_t) \longleftarrow \begin{cases} \texttt{true} & \text{if } p_t \text{ won the playout} \\ \texttt{false} & \text{otherwise} \end{cases} \tag{6.16}$$

Two versions of LGM have been implemented, both combining zero-, one-, and two-move contexts.

- LGM-2-a tries two-move replies first when sampling. If unavailable or illegal, the policy falls back to replies with one-move context; and if these are not usable as well, a random move a is played such that $\text{Goodmove}^{p}_{LGM}(a) = \texttt{true}$. Last fallback is the MoGo policy.
- LGM-2-b uses two-move and one-move replies as first and second priority; afterwards, the policy looks for escaping moves, matching patterns or captures as in the MoGo policy (see section 3.4.1); and if still no move has been selected, a random "good move" is chosen. Last fallback is a uniformly random move choice.

LGM-2 requires space for 261,364 integers (identical to LGR-2) and for 722 booleans.

Experimental Results

In figure 6.15, the LGM-2 policies are compared to LGRF-2 as described in section 6.2. LGM-2-a played 296 games against GNUGo (148 as White, 148 as Black), LGM-2-b

played 394 games (197 as White, 197 as Black), and LGRF-2 played 746 games (373 as White, 373 as Black). OREGO simulated 45,700 games per move.

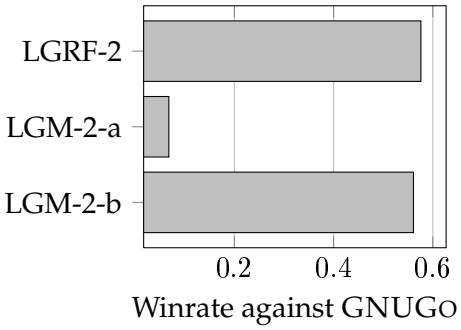

Figure 6.15: Performance of the MLGR-U policies.

While LGM-2-a performs significantly worse than LGRF-2 ($p < 0.001$), no significant difference between LGM-2-b and LGRF-2 could be shown.

6.3.12 Local LGR

A large part of the success of the well-known MOGO policy has been attributed to its local pattern matching, which creates sequences of moves resembling (short-sighted) human play (Gelly et al., 2006; Wang and Gelly, 2007). Although similar patterns had been used before, the surprising contribution of MOGO in this regard was the superiority of local matching over global matching. Small patterns like MOGO's create more meaningful play when only applied in the 3×3 neighborhood of the last move, than when used on the entire board.

In this section, this locality effect is examined in the context of the last-good-reply and the last-good-move policies. All experiments are based on the so-called *knight's neighborhood*, named after the area covered by the legal moves of the knight piece in chess. Figure 6.16 shows the knight's neighborhood of a black stone on e5.

The following variants of LGRF and LGM have been implemented:

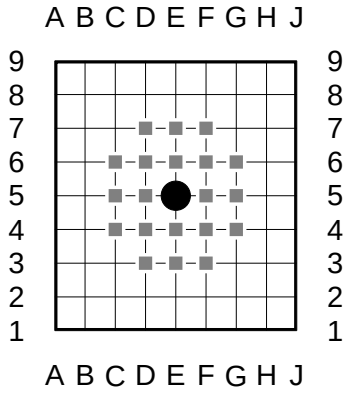

Figure 6.16: The knight's neighborhood of e5.

- LGRF-2-L-a is a local variant of LGRF-2. It only stores successful move replies a_t if they are played in the knight's neighborhood of the previous move a_{t-1}. Two-move replies are only stored if a_{t-2} and a_{t-1} are in each other's neighborhood, and a_{t-1} and a_t are in each other's neighborhood. If no local reply is available, the policy falls back to the MoGo policy.

- LGRF-2-L-b combines LGRF-2-L-a with LGRF-2 by maintaining both local and global reply tables. If no reply in the knight's neighborhood is available, it falls back to replies on the entire board; only as third priority, it falls back to the MoGo policy.

- LGM-2-L-a is a local variant of LGM-2-a. Like LGM-2-a, it gives "good moves" priority over the MoGo policy, but only searches the knight's neighborhood of a_{t-1} for "good moves".

- LGM-2-L-b is a local version of LGM-2-b. Like LGM-2-b, it gives escape, pattern and capture moves priority over "good moves", and uses uniformly random moves as last fallback; like LGM-2-L-a, it only uses "good moves" near the last move of the opponent.

LGRF-2-L-a has the same space requirements as LGRF-2 (261,364 integers for the 19×19 board). LRGF-2-L-b needs twice as much space. The LGM-2-L variants require the

same amount of space as LGM-2 (261,364 integers and 722 booleans).

Experimental Results

In figure 6.17, the LGRF-2-L policies are compared to LGRF-2 as described in section 6.2. LGRF-2-L-a played 228 games against GNUGO (114 as White, 114 as Black), LGRF-2-L-b played 888 games (444 as White, 444 as Black), and LGRF-2 played 746 games (373 as White, 373 as Black). OREGO simulated 45,700 games per move.

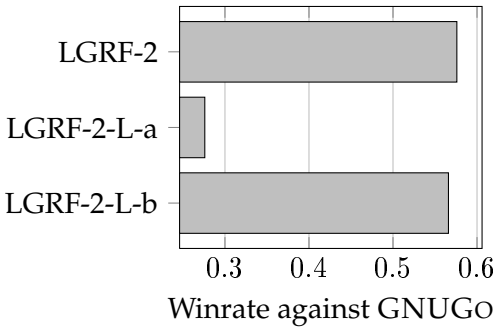

Figure 6.17: Performance of the LGRF-2-L policies.

While no significant difference between LGRF-2-L-b and LRGF-2 could be shown, LGRF-2-L-a performs significantly worse than LGRF-2 ($p < 0.001$).

In figure 6.18, the LGM-2-L policies are compared to LGRF-2 as described in section 6.2. LGM-2-L-a played 618 games against GNUGO (309 as White, 309 as Black), LGM-2-L-b played 1184 games (592 as White, 592 as Black), and LGRF-2 played 746 games (373 as White, 373 as Black). OREGO simulated 45,700 games per move.

LGM-2-L-a performed significantly worse than LGRF-2 ($p < 0.001$). No significant difference between LGM-2-L-b and LGRF-2 could be shown.

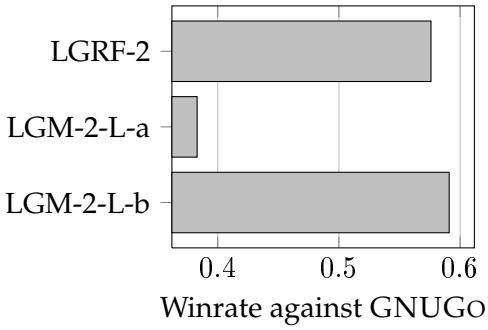

Figure 6.18: Performance of the LGM-2-L policies.

6.3.13 Scoring LGR

While chapter 5 considered the use of move answers with associated value estimates, the policies in this chapter have so far not differentiated between more or less successful replies. For LGR, a move reply that has proved itself in several won playouts in a row is indistinguishable from another reply that was stored due to a single win—both are in the reply table. Also, an excellent answer that has been deleted as a result of one unlucky playout is no different from a very weak answer that loses at every attempt—both are not in the table.

As the success of the last-good-reply policy suggests, performance in the most recent simulations is a better criterion for move replies than average performance over a longer timespan, possibly because it allows for quicker fluctuation in the reply tables and thus for more exploratory behavior. This section examines the possibility of a compromise: A ranking between different replies based on their very recent successes, instead of a large number of simulation returns.

The last-good-reply policy with scoring (LGRS) associates each move reply with an integer value. This integer is set to 1 when a new reply is stored in the table; it is incremented whenever the stored reply appears in another won simulation, and decremented whenever the reply is used and the simulation is lost. If the integer reaches zero, the corresponding move entry is deleted. This has the effect of more stability in the reply table, since replies are less likely to be

forgotten due to random noise; however, it still allows for relatively quick fluctuation, since the value counters do not stabilize over time.

LGRS has two parameters: The number of replies stored per opponent move, and the maximal value of the integer counters. This cap value is another measure intended to facilitate quicker fluctuation, as a move can only be a certain number of losses away from deletion at any given time. In the simulation phase, the stored reply with the highest integer value is chosen. If it is illegal, the standard MoGo policy serves as fallback.

The following variants have been tested:

- LGRS-1-5 maintains only one move reply for every opponent move, and caps the value counters at five.
- LGRS-3-5-a stores up to three move replies for every opponent move, and also caps the value counters at five. If three answers to a given move exist in the table and a new successful reply arrives, it is discarded in this version.
- LGRS-3-5-b is identical to LGRS-3-5-a, except that a newly arriving reply automatically overwrites the existing reply with the lowest value counter.
- LGRS-3-5-c is identical to LGRS-3-5-b, except that in case of a lost simulation, a reply is immediately removed from the table instead of just decrementing its counter. Incrementing remains unchanged.

LGRS-1 requires space for $2 \cdot 2x$ integers in total, where x is the number of intersections—half of that for reply moves, and half for value counters. LGRS-3 needs three times as much space (4332 integers).

Experimental Results

In figure 6.19, the LGRS policies are compared to LGRF-1 as described in section 6.2. LGRS-1-5 played 306 games

against GNUGO (153 as White, 153 as Black), LGRS-3-5-a played 790 games (395 as White, 395 as Black), LGRS-3-5-b played 648 games (324 as White, 324 as Black), LGRS-3-5-c played 522 games (261 as White, 261 as Black), and LGRF-1 played 448 games (224 as White, 224 as Black). OREGO simulated 25,000 games per move.

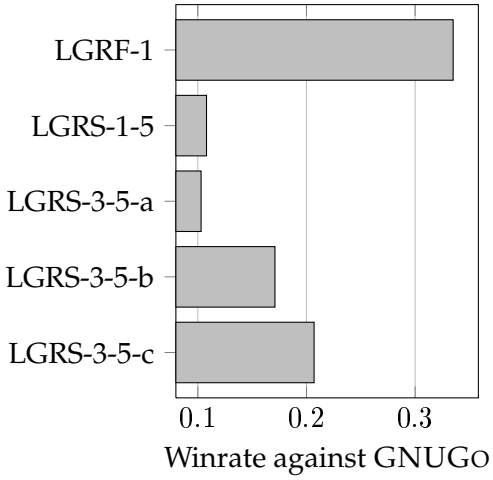

Figure 6.19: Performance of the LGRS policies.

All four tested variants of the LGRS policy perform significantly worse than LGRF-1 ($p < 0.001$).

6.3.14 Move Answer Tree with Last Good Replies

The last experiments described in this chapter combine ideas from the LGR policy and the move answer tree presented in chapter 5. In particular, the storage of move replies without value estimates is combined with the ability to selectively grow into the direction of frequently appearing contexts.

The LGR-MAT policy maintains a tree which differs from the move answer tree in two respects. First, the move answer tree stores win and run statistics for moves in order to compute conditional value estimates. In the LGR-MAT tree however, every node contains only the last successful answer to the move sequence leading from itself to the root.

Second, the move answer tree as introduced in chapter 5 grows into the future: In a path from the root to a leaf node, branches appear in the order of their corresponding moves in a simulation. The previously played move is always represented by a branch leading to a leaf. The LGR-MAT tree, on the other hand, grows into the past: The order of moves in a simulation corresponds to a path from a leaf to the root, and the previously played move is always a branch adjacent to the root[5].

Like the move answer tree, the LGR-MAT tree can be implemented as one unified tree or as two seperate trees. In the case of seperate trees, shown below for simplicity, one tree stores all of White's replies and one tree those for Black.

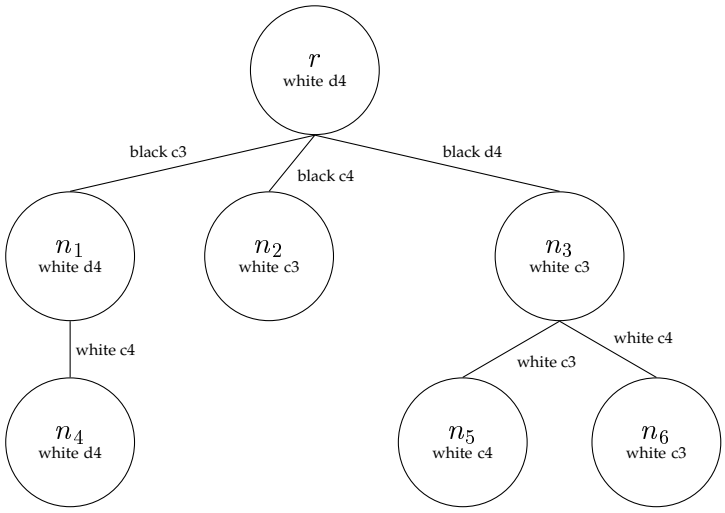

Figure 6.20: The LGR-MAT tree.

In the toy instance of an LGR-MAT tree shown in figure 6.20, node n_2 stores the information that White's last successful reply to black c4 was c3, and node n_5 stores White's last successful answer to white c3 – black d4: c4.

The LGR-MAT tree has two parameters: A limit to the maximal tree depth, and the number of times a deeper context (such as white c3 – black d4) has to appear before a node for it is created as a child to a shallower context (such as black

[5]Further experiments are needed to determine the effect of growth direction.

d4). The reason for the depth limit is the assumption that long contexts have very low chances of reappearing, making the storage of good answers futile. The reason for the node creation delay is the attempt to distinguish between randomly appearing move sequences, and move sequences that frequently reappear due to their efficiency.

Two variants of LGR-MAT have been tested so far: LGR-MAT-a with a maximum tree depth of five and creation of child nodes at second appearance, and LGR-MAT-b with a maximum tree depth of three and creation of child nodes at first appearance.

Experimental Results

In figure 6.21, the LGR-MAT policies are compared to LGRF-2 as described in section 6.2. LGR-MAT-a played 452 games against GNUGO (226 as White, 226 as Black), LGR-MAT-b played 370 games (185 as White, 185 as Black), and LGRF-2 played 316 games (158 as White, 158 as Black). OREGO simulated 50,000 games per move in the LGRF-2 condition, and 20,000 simulations per move in the LGR-MAT conditions in order to achieve comparable computation time.

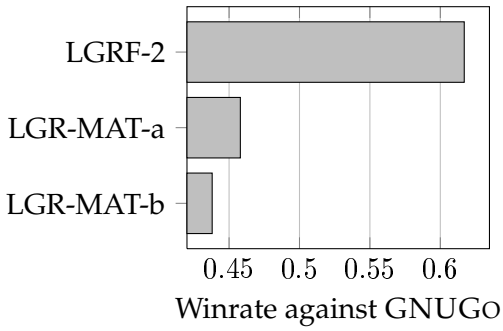

Figure 6.21: Performance of the LGR-MAT policies.

Both tested variants of the LGR-MAT policy perform significantly worse than LGRF-2 ($p < 0.001$). The difference between LGR-MAT-a and LGR-MAT-b is not significant.

Chapter 7

Other Experiments

This chapter reports on experiments on Monte Carlo Tree Search that are not directly related to the overarching topic of adaptive playout policies, but have been conducted in the process of exploring various facets of the algorithm.

7.1 Multi-Start MCTS

In the context of Single-Player MCTS, Schadd et al. (2008) applied the concept of *Meta-Search*: a search method that uses other searches to compute its result. The simple form of Meta-Search used consisted in restarting the Monte Carlo algorithm several times and dividing available time among these searches, instead of spending it on a single long search process. Eventually, the result with the highest score found in all searches was returned. This method was intended to avoid getting caught in local optima.

Due to the indeterministic nature of Monte Carlo Tree Search, it is worthwile to test whether the idea applies to Go as well. The following two variants of LGRF-2 have been created to answer the question: Are two searches of length $\frac{1}{2}\alpha$, started with different random seeds, more informative than a single search of length α?

- LGRF-2-M-a plays the move with the highest number of wins summed up over the two searches.
- LGRF-2-M-b plays the move with the highest number of wins in any of the two searches.

7.1.1 Experimental Results

In figure 7.1, the performance of the LGRF-2-M-a policy is compared to LGRF-2 as described in section 6.2. LGRF-2-M-a played 364 games against GNUGO (182 as White, 182 as Black), and LGRF-2 played 1256 games (628 as White, 628 as Black) against the same program. OREGO simulated 25,000 games per move, or 2 times 12,500, respectively.

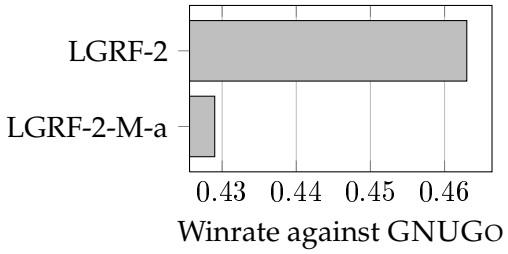

Figure 7.1: Performance of the LGRF-2-M-a policy.

LGRF-A-M-a could not be shown to perform significantly different from LGRF-2. LGRF-A-M-b's results are not yet available.

7.2 Feasibility

In the playout phase of simulated games, OREGO plays only moves that are found to be *feasible*. The definition of feasibility for MCTS Go traditionally excludes moves in the eyes of groups of the own color. This rule rarely excludes interesting moves, and helps to ensure eventual termination of playouts, since it avoids the killing of living groups by involuntary cooperation of players. The definition of eye (or *eye-like point*) used by OREGO is "an empty intersection surrounded by friendly stones and having no more

than one (zero at the board edge) diagonally adjacent enemy stones" (OREGO documentation, Drake et al. (2009)).

In order to restrict the branching factor of the search tree, even more moves can be declared infeasible. Moves on the first line, for example, are almost never useful if there is are no stones already present within a relatively small distance. Not considering these moves for sampling can potentially speed up convergence to optimal solutions.

Orego 6.10 defines only those moves as feasible that are, in addition to not being an eye-like point, either on the third or fourth line from the edge or within the knight's neighborhood of another stone. In particular at the beginning of the game, this drastically cuts down the number of move choices, and still seems to allow for reasonable play.

However, MCTS often prefers a very open, unorthodox style of play with opening moves far from the corners and edges. Moreover, staking out potential territory on the board often requires moves that are more "far out" than allowed by OREGO's feasibility definition. For this reason, two alternative definitions have been tested in combination with the LGRF-2 policy:

- LGRF-2-F-a defines any move as feasible that is not an eye-like point, and that is either not on the first line or within a *large knight's neighborhood* of another stone. Figure 7.2 shows the large knight's neighborhood of a black stone on e5.
- LGRF-2-F-b defines all moves as feasible that are not eye-like points.

7.2.1 Experimental Results

In figure 7.3, the performance of the LGRF-2-F-a and LGRF-2-F-b policies is compared to that of the standard LGRF-2 player of OREGO. LGRF-2-F-a played 1066 games against GNUGO (533 as White, 533 as Black), LGRF-2-F-b played 360 games (180 as White, 180 as Black), and LGRF-2 played

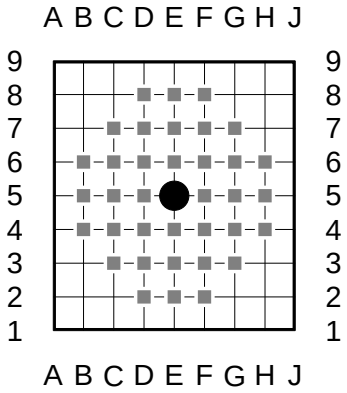

Figure 7.2: The large knight's neighborhood of e5.

1256 games (628 as White, 628 as Black) against the same program. OREGO simulated 25,000 games per move.

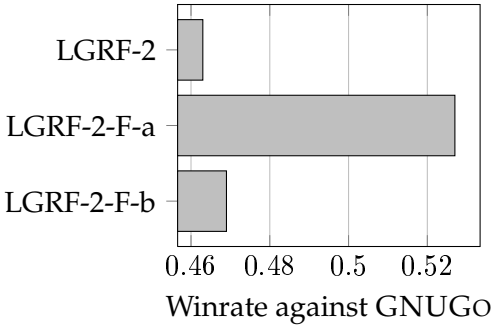

Figure 7.3: Performance of the LGRF-2-F policies.

The results show that LGRF-2-F-a has a significantly higher winrate than LGRF-2 ($p < 0.02$). No significant difference between LGRF-2-F-b and LGRF-2 could be shown.

7.3 Pattern Memory

The classic MOGO policy, as detailed in section 3.4.1, consists of three subheuristics ordered in priority: The escape heuristic, the pattern heuristic, and the capture heuristic. If none of the three heuristics return a move, the policy falls back to random move selection.

One of these policies, the pattern matcher, potentially loses information after every call: It tries to match a set of patterns around the previous move, and if matches are found, picks one of them at random to play the suggested move. Information about other matching patterns is discarded afterwards. While the creators of MoGo claim that they found local matching to be superior to global matching Gelly et al. (2006), it might be reasonable to store information about pattern-suggested moves that have not been played.

The experiment presented in this section tests a simple modification of the regular MoGo policy as employed by Orego. The pattern heuristic is extended to a *pattern collector* by saving every match found besides the returned one. Another heuristic, the *pattern retriever*, checks whether saves patterns are still applicable, and plays one of their associated moves. The last-good-reply policy with pattern memory (LGRF-2-PM) consists of five subheuristics in the following order: last-good-reply heuristic (LGRF-2), escape heuristic, pattern collector, capture heuristic, pattern retriever.

LGRF-2-PM has the same space requirements as LGRF-2, plus an additional $2x$ integers, where x is the number of intersections on the board.

7.3.1 Experimental Results

In figure 7.4, the LGRF-2-PM policy is compared to LGRF-2 as described in section 6.2. LGRF-2-PM played 916 games against GNUGo (458 as White, 458 as Black), and LGRF-2 played 398 games (199 as White, 199 as Black). Orego simulated 50,000 games per move.

No significant difference between LGRF-2-PM and LGRF-2 could be shown.

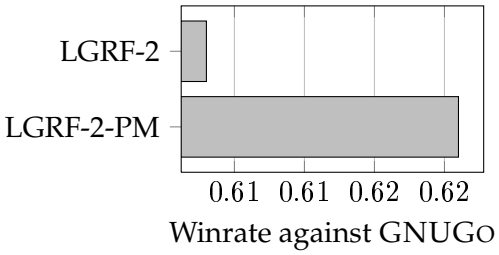

Figure 7.4: Performance of the LGRF-2-PM policy.

7.4 Dynamic Komi

As explained in section 4.1.1, Monte Carlo Tree Search often behaves suboptimally in extreme situations. When the algorithm is very far behind or very far ahead of his opponent, in particular in handicap games, it has difficulties distinguishing the winning chances of available moves.

Although no publications exist to the best of my knowledge, discussions on the computer go mailing list[1] have suggested the possibility of using *dynamic komi*. Variable komi can shift winrates into a less extreme and more easily managable range—e.g. from $95+\%$ down to $50-60\%$, which enables the algorithm to more clearly distinguish good from bad moves, and motivates it to play less passively as Black in a handicap game. However, it obscures the true goal of the game to the agent and can therefore clearly lead to suboptimal decisions.

As an example: If the regular komi is 7.5, Black has to achieve 8 more points than White to win the game. Increasing the komi dynamically to 11.5 means that in its internal, simulated games, Black has to achieve 12 points more than White now[2]. As a result, Black takes greater risks and plays more aggressively, which can result in stronger play. On the other hand, Black will not be able to take the secure path to a 9-point win, if one exists; dynamic komi can force the player to aim higher than advisable.

[1] http://dvandva.org/cgi-bin/mailman/listinfo/computer-go; the archives are available at http://blog.gmane.org/gmane.games.devel.go

[2] The real game is not affected by this, of course.

One preliminary experiment with a simple implementation of dynamic komi has been conducted with the OREGO player using the LGRF-2 policy. The modified algorithm, called LGRF-2-DK for last-good-reply with forgetting (two-move contexts) and dynamic komi, changes the komi value after every completed move search according to the following rules (without loss of generality, the algorithm plays Black):

- If the player won more than 55% of playouts in the past search, winning the game is artificially made more difficult. The komi value used in simulations is raised by one stone.
- If the player won more than 60% of playouts in the past search, internal komi is raised by two stones at once. Winrates are thus kept roughly in the $50 - 60\%$ range.
- If the player won less than 45% of playouts in the past search, and internal komi had been artificially increased in the past, it is decreased again by one stone. The estimation of being ahead has to be corrected. However, komi is not decreased below the regular komi value of the real game—this would create the illusion of safety while the algorithm is losing.
- If the player won less than 40% of playouts in the past search, and internal komi is above its regular value, it is decreased again by two stones at once, down to at least the regular value.

7.4.1 Experimental Results

In figure 7.5, the LGRF-2-DK policy is compared to LGRF-2 as described in section 6.2. LGRF-2-DK played 920 games against GNUGO (460 as White, 460 as Black), and LGRF-2 played 1256 games (628 as White, 628 as Black). OREGO simulated 25,000 games per move.

LGRF-2-DK is almost significantly stronger than LGRF-2 ($p < 0.06$).

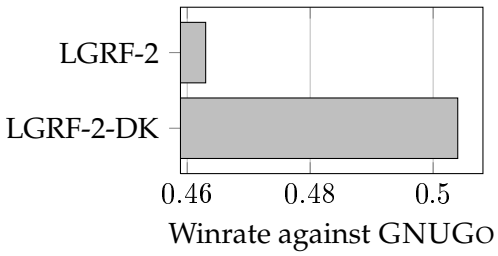

Figure 7.5: Performance of the LGRF-2-DK policy.

Chapter 8

Conclusion and Future Work

This chapter summarizes the results of the thesis, and addresses several directions for future research. It includes approaches that have only been skimmed due to time constraints, but could be taken up and explored further.

8.1 Conclusion

The goal of this work was to solve or mitigate the problem of *narrow sequences* in Monte Carlo Go, together with the related *horizon problem*. The chosen approach was the creation of *learning playout policies* that adapt to the structure of the search space, replacing or complementing the predominantly static playout policies traditionally used in MCTS.

In order to extend learning and adaptiveness from the tree part of MCTS to the simulations, a well-known phenomenon was utilized: The existence of locally optimal move replies in Go. The main idea was to explore a new form of generalization between states when judging and choosing a move. Instead of only taking the move's past performance from the current position into account—as in plain MCTS—and instead of considering the move's past performance from all (subsequent) states—as in AMAF

(RAVE)—only success or failure in states that have the same previous move[1] as the current position were considered.

Two strategies were devised to find promising move answers to previously played moves. The first strategy used estimates of a conditional value function depending on previous moves to compare possible answers and choose the *best* one (described in chapter 5). The second strategy, building on work by Peter Drake, used only information about the *last* playout-winning answers to a given move (covered in chapter 6).

The best-reply strategy, while seemingly successful in increasing the percentage of locally correct move decisions, did not achieve an overall improvement in playing strength (see section 5.3.2). Further refinements are necessary.

The last-good-reply strategy was successfully improved by the addition of forgetting (see section 6.2). Two extensions of the technique—replies to 3×3 neighborhoods instead of isolated moves (see section 6.3.2), and local moves from won playouts as zeroth-order replies (see section 6.3.12)—have shown promise and will be examined further.

These methods meet three of the five research objectives outlined in section 4.2.1: Solutions to subproblems of Go are stored, namely successful answers to single moves or sequences of moves. These solutions are tentative and changeable in character, as forgetting allows for quick fluctuation. The solutions are generalized by applying them not only to the state in which they were found, but to all states that have the same previous move or previous sequence of moves, or otherwise similar recent game history. The goals of automatically refining this generalization and of asymptotical convergence could not be met yet.

In summary, it can be stated that the results of using move replies in dynamic playout policies are encouraging and justify further research.

Apart from the main goal of this work, experiments with other elements of the OREGO program were realised as de-

[1] or sequence of previous moves, or otherwise similar recent history

tailed in chapter 7.

8.2 Future Work

8.2.1 Best Replies

Future research into the best-reply approach discussed in chapter 5 will need to find deeper insights into the reasons why the BMR and BMR-P policies failed. The exploration-exploitation tradeoff could be addressed by improved multi-armed bandit algorithms, while an appropriate measure of relation between playing a given reply and winning the game could help distinguishing between "forced" and "free" replies.

Due to the complexity of the problem, it might be advisable to take a step back from Go to a suitable toy game—with a game tree small enough to study the entire search process in detail.

8.2.2 Last Good Replies

As a next step, the LGRF policy (section 6.2) will be extended to LGRF-3 in order to find the maximal context length that still increases strength. The Pattern LGR policy (section 6.3.2) will be tested with a variety of different neighborhood definitions—e.g. only the four adjacent points instead of a 3×3 square, or neighborhoods of irregular form. These neighborhoods may also be used to improve the LGM-2-L-b policy (section 6.3.12).

Possible directions for future research into last-good-reply policies are:

- Varying the success criterion for stored replies: e.g. by storing only replies that are part of several won playouts.

- Varying the concept of move contexts: e.g. by allowing in a context not only the previous moves, but also moves further in the past of a simulation.
- Varying the definition of a trigger for a reply: e.g. by replacing moves on fixed intersections with more general events like a certain group having a low number of liberties.
- Varying the definition of a reply: e.g. by replacing moves on fixed intersections with more general actions like an attack on a certain group, or a connection between two friendly groups.

Concerning the LGR-MAT policy (section 6.3.14), future experiments will try to determine whether the growth direction of the tree affects performance, and whether high- and low-quality move replies can be distinguished in order to prune the tree for greater memory efficiency.

8.2.3 The Road Ahead

Several avenues for further investigation presented themselves over the course of this work.

First, the concept of correlation or covariance between executing a given action (not necessarily a simple move answer) and winning a game merits further exploration. It could point to ways how generalizable partial strategies can be filtered from the background noise of random simulations.

Second, incremental decision trees or other methods for mining high-speed data streams could be adapted to find distinguishing features of winning and losing moves, move answers or other partial strategies in Monte Carlo playouts. The problem here is the extreme speed necessary for MCTS simulation, as well as the one-pass scenario where large parts of the search history cannot easily be retained in memory.

Third, adaptiveness and learning could also be applied to

RAVE estimates and their underlying generalization over states, which has not been researched thoroughly. RAVE algorithms could learn from playouts which moves can be permuted in value backup, and which moves have to be played in a fixed order, e.g. because they are forced replies.

Fourth, the creation of efficient opening books and their seamless integration with MCTS search is a promising field of study. Go openings are usually categorized into standard move sequences in a single corner (*joseki*) and, far more rarely, whole-board move sequences (*fuseki*). Copying professional joseki, however, often neglects any interaction between the four corners, while copying high-level fuseki is problematic due to data sparseness.

Fifth, research into more efficient evaluation of MCTS algorithms would be worthwile. A technique that allows for quicker testing and parameter optimization, while still correlating highly with traditional measures of playing strength, could provide a considerable boost to work on computer Go and related algorithms.

Last but not least, the methods described in this work exploit domain-specific knowledge only to a small degree. It is an interesting question which other tasks besides Go can make use of recent histories of agents to improve behaviour in unknown states. The potential of Monte Carlo Tree Search in general, and adaptive playout policies in particular, has yet to be uncovered in many real-world domains with large search spaces.

Bibliography

Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y. Ng. An application of reinforcement learning to aerobatic helicopter flight. In *In Advances in Neural Information Processing Systems 19*, page 2007. MIT Press, 2007.

Naoki Abe, Naval Verma, Chid Apte, and Robert Schroko. Cross channel optimized marketing by reinforcement learning. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 767–772, New York, NY, 2004. ACM. ISBN 1-58113-888-1.

Bruce Abramson. Expected-Outcome: A General Model of Static Evaluation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(2):182–193, 1990.

L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, 1994.

Vadim V. Anshelevich. A Hierarchical Approach to Computer Hex. *Artificial Intelligence*, 134(1-2):101–120, 2002.

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-Time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2-3):235–256, 2002. URL http://homes.dsi.unimi.it/~cesabian/Pubblicazioni/ml-02.pdf.

Hendrik Baier. *Der Alpha-Beta-Algorithmus und Erweiterungen in Vier Gewinnt*. Bachelor's thesis, Technische Universität Darmstadt, 2006.

Hendrik Baier and Peter Drake. The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte-Carlo Go. Submitted, 2010.

Radha-Krishna Balla and Alan Fern. UCT for Tactical Assault Planning in Real-Time Strategy Games. In Craig Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009, Pasadena, California, USA, July 11-17, 2009*, pages 40–45, 2009. URL http://web.engr.oregonstate.edu/~afern/papers/rts-uct-ijcai09.pdf.

David B. Benson. Life in the game of Go. *Informational Sciences*, 10(1):17–29, 1976.

Elwyn Berlekamp and David Wolfe. *Mathematical Go Endgames, Nightmares for the Professional Go Player*. Ishi Press International, 1994.

Vincent Berthier, Hassen Doghmen, and Olivier Teytaud. Consistency Modifications for Automatically Tuned Monte-Carlo Tree Search. In Roberto Battiti, editor, *Learning and Intelligent OptimizatioN, LION 4*, Venice, January 18-22 2010. URL http://www.cs.bham.ac.uk/~wbl/biblio/cache/http___hal.archives-ouvertes.fr_docs_00_43_71_46_PDF_consistency.pdf.

Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2):201–240, 2002.

Ronald Bjarnason, Alan Fern, and Prasad Tadepalli. Lower Bounding Klondike Solitaire with Monte-Carlo Planning. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*. AAAI, 2009. ISBN 978-1-57735-406-2. URL http://www.informatik.uni-freiburg.de/~ki/teaching/ws0910/gamesem/bjarnason-et-al-2009.pdf.

M. Boon. A Pattern Matcher for Goliath. *Computer Go*, 13: 13–23, 1990.

Bruno Bouzy. Mathematical Morphology Applied to Computer Go. *IJPRAI*, 17(2):257–268, 2003. URL http://www.math-info.univ-paris5.fr/~bouzy/publications/Bouzy-IJPRAI.pdf.

Bruno Bouzy. Associating Shallow and Selective Global Tree Search with Monte Carlo for 9*9 Go. In H. Jaap van den Herik, Yngvi Björnsson, and Nathan S. Netanyahu, editors, *4th International Conference on Computers and Games, CG 2004, Ramat-Gan, Israel, July 5-7, 2004, Revised Papers*, volume 3846 of *Lecture Notes in Computer Science*, pages 67–80. Springer, 2004. ISBN 3-540-32488-7. URL http://www.math-info.univ-paris5.fr/~bouzy/publications/cg04-bouzy.pdf.

Bruno Bouzy. Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. *Information Sciences*, 175(4):247–257, 2005a. URL http://www.math-info.univ-paris5.fr/~bouzy/publications/Bouzy-InformationSciences.pdf.

Bruno Bouzy. History and Territory Heuristics for Monte-Carlo Go. In *JCIS-2005: Heuristic Search and Computer Game Playing Session*, Salt Lake City, UT, 2005b. URL http://www.math-info.univ-paris5.fr/~bouzy/publications/bouzy-jcis05.pdf.

Bruno Bouzy. Move Pruning Techniques for Monte-Carlo Go. In H. Jaap van den Herik, Shun chin Hsu, Tsan sheng Hsu, and H. H. L. M. Donkers, editors, *Advances in Computer Games, 11th International Conference, ACG 2005, Taipei, Taiwan, September 6-9, 2005. Revised Papers*, volume 4250 of *Lecture Notes in Computer Science*, pages 104–119. Springer, 2006. ISBN 3-540-48887-1. URL http://www.math-info.univ-paris5.fr/~bouzy/publications/bouzy-acg11.pdf.

Bruno Bouzy and Tristan Cazenave. Computer Go: An AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001. URL http://www.lamsade.dauphine.fr/~cazenave/papers/CG-AISurvey.pdf.

Bruno Bouzy and Guillaume Chaslot. Monte-Carlo Go Reinforcement Learning Experiments. In Sushil J. Louis and Graham Kendall, editors, *Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games (CIG06), University of Nevada, Reno, campus in Reno/Lake Tahoe, 22-24 May, 2006*, pages 187–194. IEEE, 2006. URL http:

//www.math-info.univ-paris5.fr/~bouzy/publications/bouzy-chaslot-cig06.pdf.

Bruno Bouzy and Bernard Helmstetter. Monte-Carlo Go Developments. In H. Jaap van den Herik, Hiroyuki Iida, and Ernst A. Heinz, editors, *10th International Conference on Advances in Computer Games, Many Games, Many Challenges, ACG 2003, Graz, Austria, November 24-27, 2003, Revised Papers*, volume 263 of *IFIP*, pages 159–174. Kluwer, 2003. ISBN 1-4020-7709-2. URL http://www.math-info.univ-paris5.fr/~bouzy/publications/bouzy-helmstetter.pdf.

Bernd Brügmann. Monte Carlo Go, March 1993. URL ftp://ftp.cse.cuhk.edu.hk/pub/neuro/GO/mcgo.tex. Unpublished manuscript.

Jay Burmeister and Janet Wiles. The challenge of Go as a domain for AI research: a comparison between Go and chess. In *Intelligent Information Systems, 1995. ANZIIS-95. Proceedings of the Third Australian and New Zealand Conference on*, pages 181–186, Perth, WA, 1995.

Jay Burmeister and Janet Wiles. AI techniques used in computer Go. In *Fourth Conference of the Australasian Cognitive Science Society*, Newcastle, 1997.

Michael Buro. The Othello Match of the Year: Takeshi Murakami vs. Logistello. *ICCA Journal*, 20(3):189–193, September 1997.

Michael Buro. From Simple Features to Sophisticated Evaluation Functions. In H. Jaap van den Herik and Hiroyuki Iida, editors, *First International Conference on Computers and Games, CG'98, Tsukuba, Japan, November 11-12, 1998*, volume 1558 of *Lecture Notes in Computer Science*, pages 126–145. Springer, 1998. ISBN 3-540-65766-5.

Michael Buro. Call for Research in RTS Games. In *Proceedings of the AAAI Workshop on AI in Games*, pages 139–141. AAAI press, 2004. URL http://webdocs.cs.ualberta.ca/~mburo/ps/RTS-AAAI04.pdf.

Xindi Cai and Donald C. Wunsch, II. Computer Go: A Grand Challenge to AI. volume 63 of *Studies in Computational Intelligence*, pages 443–465. Springer, 2007. ISBN 978-3-540-71983-0.

Murray Campbell, A. Hoane, and Feng-hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

Tristan Cazenave. Metaprogramming Forced Moves. In *ECAI*, pages 645–649, 1998. URL http://www.ai.univ-paris8.fr/~cazenave/ecai98.pdf.

Tristan Cazenave. The separation game. *New Mathematics and Natural Computation*, 2(2):161–172, 2006.

Tristan Cazenave. Playing the Right Atari. *ICGA Journal*, 30 (1):35–42, 2007.

Tristan Cazenave and Bernard Helmstetter. Combining Tactical Search and Monte-Carlo in the Game of Go. In Graham Kendall and Simon M. Lucas, editors, *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05), Essex University, Colchester, Essex, UK, 4-6 April, 2005*. IEEE, 2005a. URL http://www.ai.univ-paris8.fr/~bh/articles/searchmcgo.pdf.

Tristan Cazenave and Bernard Helmstetter. Search for transitive connections. *Informational Sciences*, 175(4):284–295, 2005b. URL http://www.lamsade.dauphine.fr/~cazenave/papers/transitive.pdf.

Tristan Cazenave and Nicolas Jouandeau. On the parallelization of UCT. In *Computer Games Workshop*, pages 93–101, 2007.

Hyeong Soo Chang, Michael C. Fu, Jiaqiao Hu, and Steven I. Marcus. An Adaptive Sampling Algorithm for Solving Markov Decision Processes. *Operations Research*, 53(1):126–139, 2005. URL http://www.rhsmith.umd.edu/faculty/mfu/fu_files/CFHM05.pdf.

Guillaume Chaslot, Mark Winands, H. Jaap van den Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive Strategies for Monte-Carlo Tree Search. In *JCIS-2005: Heuristic Search and Computer Game Playing Session*, Salt Lake City, UT, 2007. URL http://www.math-info.univ-paris5.fr/~bouzy/publications/CWHUB-pMCTS-2007.pdf.

Guillaume Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel Monte-Carlo Tree Search.

In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *6th International Conference on Computers and Games, CG 2008, Beijing, China, September 29 - October 1, 2008*, volume 5131 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 2008. ISBN 978-3-540-87607-6. URL http://www.personeel.unimaas.nl/m-winands/documents/multithreadedMCTS2.pdf.

Guillaume Chaslot, Christophe Fiter, Jean-Baptiste Hoock, Arpad Rimmel, and Olivier Teytaud. Adding expert knowledge and exploration in Monte-Carlo Tree Search. In *Advances in Computer Games*, Pamplona, Spain, 2009. Springer. URL http://www.lri.fr/~rimmel/publi/EK_explo.pdf.

Guillaume Chaslot, Christophe Fiter, Jean-Baptiste Hoock, Arpad Rimmel, and Olivier Teytaud. Adding expert knowledge and exploration in Monte-Carlo Tree Search, 2009. URL http://www.personeel.unimaas.nl/g-chaslot/papers/acg09.pdf.

Ken Chen and Zhixing Chen. Static Analysis of Life and Death in the Game of Go. *Informational Sciences*, 121(1-2): 113–134, 1999.

Pierre-Arnaud Coquelin and Rémi Munos. Bandit Algorithms for Tree Search. In *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence*, Vancouver, Canada, 2007. URL http://hal.inria.fr/docs/00/15/02/07/PDF/BAST.pdf.

Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, editors, *5th International Conference on Computers and, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006. ISBN 978-3-540-75537-1. URL http://hal.archives-ouvertes.fr/docs/00/11/69/92/PDF/CG2006.pdf.

Rémi Coulom. Computing Elo Ratings of Move Patterns in the Game of Go. *ICGA Journal*, 30(4):198–208, 2007. URL http://hal.archives-ouvertes.fr/docs/00/14/98/59/PDF/MMGoPatterns.pdf.

Rémi Coulom. Criticality: a Monte-Carlo Heuristic for Go Programs. Invited Talk, 2009. URL http://remi.coulom.free.fr/Criticality/.

D. Dailey. Computer Go Server, May 2010. URL http://cgos.boardspace.net/.

Frédéric de Mesmay, Arpad Rimmel, Yevgen Voronenko, and Markus Püschel. Bandit-based optimization on graphs with application to library performance tuning. In Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman, editors, *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382 of *ACM International Conference Proceeding Series*, page 92. ACM, 2009. ISBN 978-1-60558-516-1. URL http://spiral.ece.cmu.edu:8080/pub-spiral/pubfile/494_134.pdf.

Guy Van den Broeck, Kurt Driessens, and Jan Ramon. Monte-Carlo Tree Search in Poker Using Expected Reward Distributions. In Zhi-Hua Zhou and Takashi Washio, editors, *Advances in Machine Learning, First Asian Conference on Machine Learning, ACML 2009, Nanjing, China, November 2-4, 2009*, volume 5828 of *Lecture Notes in Computer Science*, pages 367–381. Springer, 2009. ISBN 978-3-642-05223-1. URL http://wwwis.win.tue.nl/bnaic2009/papers/bnaic2009_paper_85.pdf.

Peter Drake. The Last-Good-Reply Policy for Monte-Carlo Go. *ICGA Journal*, 32(4), 2010. URL https://webdisk.lclark.edu/drake/publications/drake-icga-2009.pdf.

Peter Drake and Steve Uurtamo. Heuristics in Monte Carlo Go. In Hamid R. Arabnia, Mary Qu Yang, and Jack Y. Yang, editors, *Proceedings of the 2007 International Conference on Artificial Intelligence, ICAI 2007, Volume I, June 25-28, 2007, Las Vegas, Nevada, USA*, pages 171–175. CSREA Press, 2007a. ISBN 1-60132-023-X. URL https://webdisk.lclark.edu/drake/publications/GAMEON-07-drake.pdf.

Peter Drake and Steve Uurtamo. Move ordering vs heavy playouts: Where should heuristics be applied in Monte Carlo Go? In *Proceedings of the*

3rd North American Game-On Conference, 2007b. URL https://webdisk.lclark.edu/drake/publications/GAMEON-07-drake.pdf.

Peter Drake et al. Orego 6.10, November 2009. URL http://legacy.lclark.edu/~drake/Orego.html.

Markus Enzenberger and Martin Müller. A Lock-free Multithreaded Monte-Carlo Tree Search Algorithm. In *Advances in Computer Games, 12th International Conference*, 2009. URL http://www.cs.ualberta.ca/~emarkus/publications/enzenberger-mueller-acg12.pdf.

David Fotland. Building a World-Champion Arimaa Program. In H. Jaap van den Herik, Yngvi Björnsson, and Nathan S. Netanyahu, editors, *4th International Conference on Computers and Games, CG 2004, Ramat-Gan, Israel, July 5-7, 2004, Revised Papers*, volume 3846 of *Lecture Notes in Computer Science*, pages 175–186. Springer, 2004. ISBN 3-540-32488-7.

Free Software Foundation. GNUGo 3.8, February 2009. URL http://www.gnu.org/software/gnugo/.

Sylvain Gelly. *A Contribution to Reinforcement Learning; Application to Computer-Go*. PhD thesis, Université Paris-Sud, September 2007.

Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In Zoubin Ghahramani, editor, *Proceedings of the Twenty-Fourth International Conference on Machine Learning (ICML 2007), Corvalis, Oregon, USA, June 20-24, 2007*, volume 227 of *ACM International Conference Proceeding Series*, pages 273–280. ACM, 2007. ISBN 978-1-59593-793-3. URL http://www.machinelearning.org/proceedings/icml2007/papers/387.pdf.

Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Technical report, HAL - CCSd - CNRS, 2006. URL http://hal.inria.fr/docs/00/12/15/16/PDF/RR-6062.pdf.

Michael R. Genesereth, Nathaniel Love, and Barney Pell. General Game Playing: Overview of the AAAI Competition. *AI Magazine*, 26(2):62–72, 2005.

Matthew L. Ginsberg. GIB: Imperfect Information in a Computationally Challenging Game. *Journal of Artificial Intelligence Research (JAIR)*, 14:303–358, 2001.

David P. Helmbold and Aleatha Parker-Wood. All-Moves-As-First Heuristics in Monte-Carlo Go. In Hamid R. Arabnia, David de la Fuente, and José Angel Olivas, editors, *Proceedings of the 2009 International Conference on Artificial Intelligence, ICAI 2009, July 13-16, 2009, Las Vegas Nevada, USA*, pages 605–610. CSREA Press, 2009. ISBN 1-60132-109-0. URL http://users.soe.ucsc.edu/~dph/mypubs/AMAFpaperWithRef.pdf.

James Hendler. Computers Play Chess; Humans Play Go. *IEEE Intelligent Systems*, 21(4):2–3, 2006.

Jean-Baptiste Hoock and Olivier Teytaud. Bandit-Based Genetic Programming. In Anna Isabel Esparcia-Alcázar, Anikó Ekárt, Sara Silva, Stephen Dignum, and A. Sima Etaner-Uyar, editors, *13th European Conference on Genetic Programming, EuroGP 2010, Istanbul, Turkey, April 7-9, 2010*, volume 6021 of *Lecture Notes in Computer Science*, pages 268–277. Springer, 2010. ISBN 978-3-642-12147-0. URL http://hal.inria.fr/docs/00/45/28/87/PDF/pattern.pdf.

Feng-Hsiung Hsu. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, Princeton, NJ, USA, 2002. ISBN 0691090653.

David R. Hunter. MM Algorithms for Generalized Bradley-Terry Models. *Annals of Statistics*, 32(1):384–406, 2004. URL http://www.stat.psu.edu/~dhunter/papers/bt.pdf.

Robert Jasiek. rec.games.go Rules FAQ, December 2007. URL http://home.snafu.de/jasiek/rulesfaq.txt.

Robert Jasiek. Go Rules, January 2010. URL http://home.snafu.de/jasiek/rules.html.

Michael J. Kearns, Yishay Mansour, and Andrew Y. Ng. A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes. In Thomas Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999*, pages 1324–1231. Morgan Kaufmann, 1999. ISBN 1-55860-613-0. URL http://www.cis.upenn.edu/~mkearns/papers/sparseplan.pdf.

Akihiro Kishimoto and Martin Müller. Search versus Knowledge for Solving Life and Death Problems in Go. In Manuela M. Veloso and Subbarao Kambhampati, editors, *The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 1374–1379. AAAI Press / The MIT Press, 2005. ISBN 1-57735-236-X.

Hiroaki Kitano, Walther von Hahn, Lawrence Hunter, Ryuichi Oka, Benjamin W. Wah, and Toshio Yokoi. Grand Challenge AI Applications. In *IJCAI*, pages 1677–1683, 1993.

Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, Eiichi Osawai, and Hitoshi Matsubara. RoboCup: A challenge problem for AI and robotics. In *RoboCup-97: Robot Soccer World Cup I*, pages 1–19. Springer, 1998. URL http://dx.doi.org/10.1007/3-540-64473-3_46.

Donald E. Knuth and Ronald W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *17th European Conference on Machine Learning, ECML 2006, Berlin, Germany, September 18-22, 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006. ISBN 3-540-45375-X. URL http://zaphod.aml.sztaki.hu/papers/ecml06.pdf.

Tze L. Lai and Herbert Robbins. Asymptotically efficient

adaptive allocation rules. *Advances in Applied Mathematics*, 6:4–22, 1985.

John E. Laird and Michael van Lent. Human-Level AI's Killer Application: Interactive Computer Games. *AI Magazine*, 22(2):15–25, 2001.

Chang-Shing Lee, Mei-Hui Wang, Guillaume Chaslot, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, Shang-Rong Tsai, Shun-Chin Hsu, and Tzung-Pei Hong. The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in Games*, 1:73–89, 2009. ISSN 1943-068X.

D. Lefkovitz. Technical Note 60-243: A Strategic Pattern Recognition Program for the Game of Go. Technical report, University of Pennsylvania, the Moore School of Electrical Engineering, Wright Air Development Division, 1960.

Jens Lieberum. An Evaluation Function for the Game of Amazons. *Theoretical Computer Science*, 349(2):230–244, 2005.

Richard J. Lorentz. Amazons Discover Monte-Carlo. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *6th International Conference on Computers and Games, CG 2008, Beijing, China, September 29 - October 1, 2008*, volume 5131 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2008. ISBN 978-3-540-87607-6.

Simon M. Lucas. Computational Intelligence and Games: Challenges and Opportunities. *International Journal of Automation and Computing*, 5(1):45–57, 2008.

Simon M. Lucas and Graham Kendall. Evolutionary Computation and Games. *IEEE Computational Intelligence Magazine*, 1(1):10–18, 2006. URL http://www.cs.nott.ac.uk/~gxk/papers/gxkieeemag.pdf.

Mitja Luštrek, Matjaž Gams, and Ivan Bratko. A Program for Playing Tarok. *ICGA Journal*, 23(3):190–197, 2003. URL http://tarok.bocosoft.com/dl/A_Program_for_Playing_Tarok-03.pdf.

John McCarthy. Chess as the Drosophila of AI. In T. A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 227–238. Springer, New York, 1990.

John McCarthy. AI as Sport. *Science*, 276(5318):1518–1519, 1997.

Dylan L. McClain. Once Again, Machine Beats Human Champion at Chess. In The New York Times, December 5th, 2006.

David A. Mechner. All Systems Go. *The Sciences*, 38(1), 1998.

Martin Müller. Measuring the performance of Go programs. In *International Go Congress*, Beijing, 1991. URL http://www.cs.ualberta.ca/~mmueller/ps/mueller91a.ps.

Martin Müller. Playing it safe: Recognizing secure territories in computer Go by using static rules and search. In *Game Programming Workshop in Japan '97*, pages 80–86, 1997.

Martin Müller. Computer Go. *Artificial Intelligence*, 134(1-2): 145–179, 2002.

Martin Müller. Conditional combinatorial games and their application to analyzing capturing races in Go. *Informational Sciences*, 154(3-4):189–202, 2003.

Yuriy Nevmyvaka, Yi Feng, and Michael Kearns. Reinforcement learning for optimized trade execution. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 673–680, New York, NY, 2006. ACM. ISBN 1-59593-383-2.

Xiaozhen Niu and Martin Müller. An Open Boundary Safety-of-Territory Solver for the Game of Go. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, editors, *5th International Conference on Computers and, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, volume 4630 of *Lecture Notes in Computer Science*, pages 37–49. Springer, 2006a. ISBN 978-3-540-75537-1.

Xiaozhen Niu and Martin Müller. An improved safety solver for computer Go. In *Computers and Games: 4th In-*

ternational Conference, CG 2004. LNCS 3846, pages 97–112. Springer, 2006b.

Xiaozhen Niu, Akihiro Kishimoto, and Martin Müller. Recognizing Seki in Computer Go. In H. Jaap van den Herik, Shun chin Hsu, Tsan sheng Hsu, and H. H. L. M. Donkers, editors, *Advances in Computer Games, 11th International Conference, ACG 2005, Taipei, Taiwan, September 6-9, 2005. Revised Papers*, volume 4250 of *Lecture Notes in Computer Science*, pages 88–103. Springer, 2006. ISBN 3-540-48887-1. URL http://webdocs.cs.ualberta.ca/~mmueller/ps/seki.pdf.

Seth Pellegrino and Peter Drake. Investigating the Effects of Playout Strength in Monte-Carlo Go. In *2010 International Conference on Artificial Intelligence*, 2010. URL https://webdisk.lclark.edu/drake/publications/pellegrino-elo-2010.pdf.

Seth Pellegrino, Andrew Hubbard, Jason Galbraith, Peter Drake, and Yung-Pin Chen. Localizing Search in Monte-Carlo Go Using Statistical Covariance. *ICGA Journal*, 32(3):154–160, 2009. URL https://webdisk.lclark.edu/drake/publications/pellegrino-ICGA-2009.pdf.

Laurent Péret and Frédérick Garcia. On-Line Search for Solving Markov Decision Processes via Heuristic Sampling. In Ramon López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 530–534. IOS Press, 2004. ISBN 1-58603-452-9.

S. Proper and P. Tadepalli. Scaling Model-Based Average-reward Reinforcement Learning for Product Delivery. In *ECML 2006: Proceedings of the 17th European Conference on Machine Learning*, pages 735–742, 2006.

M. L. Puterman. *Markov Decision Processes*. Wiley, 1994.

Jan Ramon and Tom Croonenborghs. Searching for Compound Goals Using Relevancy Zones in the Game of Go. In H. Jaap van den Herik, Yngvi Björnsson, and Nathan S. Netanyahu, editors, *4th International Conference*

on Computers and Games, CG 2004, Ramat-Gan, Israel, July 5-7, 2004, Revised Papers, volume 3846 of *Lecture Notes in Computer Science*, pages 113–128. Springer, 2004. ISBN 3-540-32488-7.

Raj Reddy. Foundations and Grand Challenges of Artificial Intelligence: AAAI Presidential Address. *AI Magazine*, 9 (4):9–21, 1988.

J. Reitman and B. Wilcox. The Structure and Performance of the INTERIM.2 Go Program. In *Proceedings IJCAI79*, pages 711–719, 1979.

Horst Remus. Simulation of a Learning Machine for Playing GO. In *IFIP Congress*, pages 428–432, 1962.

Arpad Rimmel and Fabien Teytaud. Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search. In Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna Esparcia-Alcázar, Chi Keong Goh, Juan J. Merelo Guervós, Ferrante Neri, Mike Preuss, Julian Togelius, and Georgios N. Yannakakis, editors, *Applications of Evolutionary Computation, EvoApplicatons 2010: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM, and EvoSTOC, Istanbul, Turkey, April 7-9, 2010*, volume 6024 of *Lecture Notes in Computer Science*, pages 201–210. Springer, 2010. ISBN 978-3-642-12238-5. URL http://hal.inria.fr/docs/00/45/64/22/PDF/CMC.pdf.

Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematics Society*, 58:527–535, 1952.

Philippe Rolet, Michèle Sebag, and Olivier Teytaud. Optimal robust expensive optimization is tractable. In Franz Rothlauf, editor, *Genetic and Evolutionary Computation Conference, GECCO 2009, Montreal, Québec, Canada, July 8-12, 2009*, pages 1951–1956. ACM, 2009. ISBN 978-1-60558-325-9. URL http://hal.archives-ouvertes.fr/docs/00/37/49/10/PDF/balopt.pdf.

Arthur L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3(3):211–229, 1959.

Arthur L. Samuel. Programming Computers to Play Games. *Advances in Computers*, 1:165–192, 1960.

Maarten P. D. Schadd, Mark H. M. Winands, H. Jaap van den Herik, Guillaume Chaslot, and Jos W. H. M. Uiterwijk. Single-Player Monte-Carlo Tree Search. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *6th International Conference on Computers and Games, CG 2008, Beijing, China, September 29 - October 1, 2008*, volume 5131 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2008. ISBN 978-3-540-87607-6. URL http://www.personeel.unimaas.nl/m-winands/documents/CGSameGame.pdf.

Jonathan Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.

Jonathan Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer, Berlin, 1997.

Jonathan Schaeffer and H. Jaap van den Herik. Games, computers, and artificial intelligence. *Artificial Intelligence*, 134(1-2):1–7, 2002.

Jonathan Schaeffer, Joseph C. Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A World Championship Caliber Checkers Program. *Artificial Intelligence*, 53(2-3):273–289, 1992.

W. Schulz, P. Dayan, and P. R. Montague. A Neural Substrate of Prediction and Reward. *Science*, 275(5306):1593–1599, 1997.

Sensei's. Sensei's Library - Rules of Go, February 2010. URL http://senseis.xmp.net/?RulesOfGo.

Claude Shannon. Programming a Computer for Playing Chess. *Philosophical Magazine*, 41(314):256–275, 1950.

Shiven Sharma, Ziad Kobti, and Scott D. Goodwin. Knowledge Generation for Improving Simulations in UCT for General Game Playing. In Wayne Wobcke and Mengjie

Zhang, editors, *AI 2008: Advances in Artificial Intelligence, 21st Australasian Joint Conference on Artificial Intelligence, Auckland, New Zealand, December 1-5, 2008*, volume 5360 of *Lecture Notes in Computer Science*, pages 49–55. Springer, 2008. ISBN 978-3-540-89377-6.

Brian Sheppard. World-championship-caliber Scrabble. *Artificial Intelligence*, 134(1-2):241–275, 2002.

David Silver. *Reinforcement Learning and Simulation-Based Search in Computer Go*. PhD thesis, University of Alberta, November 2009. URL http://hdl.handle.net/10048/797.

David Silver and Gerald Tesauro. Monte-Carlo Simulation Balancing. In Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman, editors, *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382 of *ACM International Conference Proceeding Series*, page 119. ACM, 2009. ISBN 978-1-60558-516-1. URL http://www.cs.mcgill.ca/~icml2009/papers/500.pdf.

David Silver, Richard S. Sutton, and Martin Müller. Reinforcement Learning of Local Shape in the Game of Go. In Manuela M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007, Hyderabad, India, January 6-12, 2007*, pages 1053–1058, 2007. URL http://webdocs.cs.ualberta.ca/~mmueller/ps/silver-ijcai2007.pdf.

Herbert A. Simon and Allen Newell. Heuristic Problem Solving: The Next Advance in Operations Research. *Operations Research*, 6(1):1–10, 1958.

Satinder Singh, Diane Litman, Michael Kearns, and Marilyn Walker. Optimizing Dialogue Management with Reinforcement Learning: Experiments with the NJFun System. *Journal of Artificial Intelligence Research*, 16:105–133, 2002.

David H. Stern, Ralf Herbrich, and Thore Graepel. Bayesian pattern ranking for move prediction in the game of Go. In William W. Cohen and Andrew Moore, editors, *Proceedings of the Twenty-Third International*

Conference on Machine Learning (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006, volume 148 of *ACM International Conference Proceeding Series*, pages 873–880. ACM, 2006. ISBN 1-59593-383-2. URL http://www.autonlab.org/icml_documents/camera-ready/110_Bayesian_Pattern_Ran.pdf.

John Stogin, Yung-Pin Chen, Peter Drake, and Seth Pellegrino. The Beta Distribution in the UCB Algorithm Applied to Monte-Carlo Go. In Hamid R. Arabnia, David de la Fuente, and José Angel Olivas, editors, *Proceedings of the 2009 International Conference on Artificial Intelligence, ICAI 2009, July 13-16, 2009, Las Vegas Nevada, USA*, pages 521–524. CSREA Press, 2009. ISBN 1-60132-109-0.

Peter Stone and Richard S. Sutton. Scaling Reinforcement Learning toward RoboCup Soccer. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 537–544, San Francisco, CA, 2001. Morgan Kaufmann.

Richard S. Sutton and Andrew G. Barto. Time-derivative models of Pavlovian reinforcement. In *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pages 497–537. MIT Press, 1990.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998. ISBN 0262193981. URL http://www.cs.ualberta.ca/%7Esutton/book/ebook/the-book.html.

Istvan Szita, Guillaume Chaslot, and Pieter Spronck. Monte-Carlo Tree Search in Settlers of Catan. In H. Jaap van den Herik and Pieter Spronck, editors, *Advances in Computer Games, 12th International Conference, ACG 2009, Pamplona, Spain, May 11-13, 2009. Revised Papers*, volume 6048 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 2009. ISBN 978-3-642-12992-6. URL http://www.personeel.unimaas.nl/g-chaslot/papers/ACGSzitaChaslotSpronck.pdf.

Shogo Takeuchi, Tomoyuki Kaneko, and Kazunori Yamaguchi. Evaluation of Monte Carlo Tree Search and the Application to Go. In Philip Hingston and Luigi Barone, editors, *IEEE Symposium on Computational Intelligence and Games*, pages 191–198. IEEE, dec 2008. ISBN

978-1-4244-2974-5. URL http://www.csse.uwa.edu.au/cig08/Proceedings/papers/8046.pdf.

Gerald Tesauro. Programming Backgammon Using Self-teaching Neural Nets. *Artificial Intelligence*, 134(1-2):181–199, 2002.

Gerald Tesauro and Gregory R. Galperin. On-line Policy Improvement using Monte-Carlo Search. In Michael Mozer, Michael I. Jordan, and Thomas Petsche, editors, *Advances in Neural Information Processing Systems 9, NIPS, Denver, CO, USA, December 2-5, 1996*, pages 1068–1074. MIT Press, 1996.

Fabien Teytaud and Olivier Teytaud. Creating an Upper-Confidence-Tree program for Havannah. In *ACG 12*, Pamplona, Spain, 2009. URL http://hal.inria.fr/inria-00380539/en/.

John Tromp and Gunnar Farnebäck. Combinatorics of Go. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, editors, *5th International Conference on Computers and, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, volume 4630 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2006. ISBN 978-3-540-75537-1.

Erik van der Werf, Jaap van den Herik, and Jos Uiterwijk. Solving Go on small Boards. *ICGA Journal*, 26(2):92–107, 2003. URL http://erikvanderwerf.tengen.nl/pubdown/solving_go_on_small_boards.pdf.

Erik C. D. van der Werf and Mark H. M. Winands. Solving Go for Rectangular Boards. *ICGA Journal*, 32(2):77–88, 2009. URL http://erikvanderwerf.tengen.nl/pubdown/SolvingGoICGA2009.pdf.

Yizao Wang and Sylvain Gelly. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 175–182, Honolulu, HI, 2007.

Mark H. M. Winands and Yngvi Björnsson. Evaluation Function Based Monte-Carlo LOA. In H. Jaap van den Herik and Pieter Spronck, editors, *Advances in Computer Games, 12th International Conference, ACG 2009, Pamplona, Spain, May 11-13, 2009. Revised Papers*, volume 6048 of

Lecture Notes in Computer Science, pages 33–44. Springer, 2009. ISBN 978-3-642-12992-6. URL http://www.hr.is/faculty/yngvi/pdf/WinandsB09.pdf.

Thomas Wolf. The Program GoTools and its Computer-generated Tsume Go Database. In *Game Programming Workshop in Japan '94*, pages 84–96, 1994. URL http://eprints.kfupm.edu.sa/70954/.

Kazuki Yoshizoe, Akihiro Kishimoto, and Martin Müller. Lambda Depth-First Proof Number Search and Its Application to Go. In Manuela M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007, Hyderabad, India, January 6-12, 2007*, pages 2404–2409, 2007.

Albert Lindsey Zobrist. *Feature extraction and representation for pattern recognition and the game of go*. PhD thesis, The University of Wisconsin-Madison, 1970.