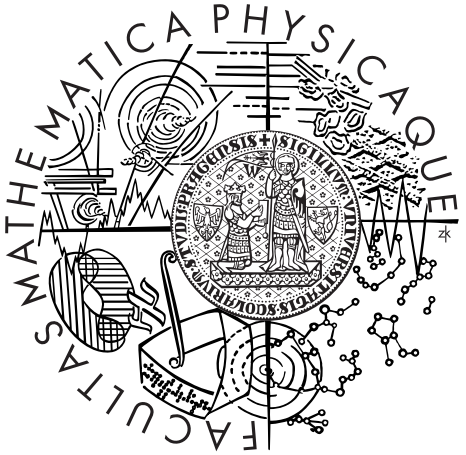
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Petr Baudiš

MCTS with Information Sharing

Department of Theoretical Computer Science
and Mathematical Logic

Supervisor of the master thesis: RNDr. Jan Hric
Study programme: Computer Science
Specialization: Theoretical Computer Science

Prague 2011

I would like to thank my advisor Jan Hric for guidance and many comments on both my research and this thesis in particular. Jean-loup Gailly not only pointed out mistakes in this text, but he has been a great help in Pachi development in all regards; his contributions keep up my motivation to work on Pachi. Robert Šámal also patiently listened to all my ideas and helped me with some statistical aspects of criticality. Jonathan Chetwyng has shown Pachi in a different angle by his work on Go visualization.

Nick Wedd has been patiently and reliably organizing regular Computer Go events for many years now, offering a steady benchmark for progress in the field. Many other people of the Computer Go community have been very helpful over the years, sharing details about their work, commenting and answering questions: David Fotland, Shih-Chieh (Aja) Huang, Rémi Coulom, Olivier Teytaud, Hiroshi Yamashita, Hideki Kato and everyone else on the `computer-go` mailing list. Code by Łukasz Lew, Martin Müller and Markus Enzenberger inspired me much when I started working on my own Go program.

Last but foremost, Pachi is made of two syllabes. Chido has been my greatest inspiration and without her, this thesis any many other things would not happen; also, she has drawn the cute Pachi pictures.

Work on this thesis has been in part supported by the GAUK Project 66010 of Charles University Prague. Computing resources of the Department of Applied Mathematics, Charles University Prague were used for play-testing and performance tuning of Pachi.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature

Název práce: MCTS se sdílením informací

Autor: Petr Baudiš

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: RNDr. Jan Hric

Abstrakt: Představíme naši výkonnou implementaci algoritmu Monte Carlo stromového vyhledávání (MCTS) pro hraní deskové hry Go: Pachi. Program je založeno na dříve publikovaných algoritmech i našich původních vylepšeních. Následně se zaměříme na zlepšování efektivity prohledávání pomocí sběru informací týkajících se taktických situací a obecného stavu hry z jednotlivých Monte Carlo simulací a jejich sdílení v rámci herního stromu. Navrhneme konkrétní metody takového sdílení — dynamické komi, měření kritičnosti tahů a mapy svobod — a předvedeme jejich pozitivní účinek na základě naměřené výkonnosti vůči jiným programům. Na závěr načrtneme několik zajímavých navazujících témat souvisejích s naším výzkumem.

Klíčová slova: Herní stromy, Minimax, Monte Carlo stromové vyhledávání, Go

Title: MCTS with Information Sharing

Author: Petr Baudiš

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jan Hric

Abstract: We introduce our competitive implementation of a Monte Carlo Tree Search (MCTS) algorithm for the board game of Go: Pachi. The software is based both on previously published methods and our original improvements. We then focus on improving the tree search performance by collecting information regarding tactical situations and game status from the Monte Carlo simulations and sharing it with and within the game tree. We propose specific methods of such sharing — dynamic komi, criticality-based biasing, and liberty maps — and demonstrate their positive effect. based on collected play-testing measurements. We also outline some promising future research directions related to our work.

Keywords: Game Trees, Minimax, Monte Carlo Tree Search, Go

Contents

Introduction

Go is an ancient Asian board game popular for its simple rules that nevertheless produce extremely complex gameplay. It has been studied for many centuries and vast amount of research on strategy and tactics involved has been done by both historical and contemporary professional players of the game, but it also inspired mathematicians and computer scientists. For example, one of motivations of the Combinatorial Game Theory has been the exact study of Go [BW94], and the game became well known among researchers interested in computer game-playing since it turned out to be much more difficult to write a strong Go program than for example a strong Chess program. Some consider beating the top human Go players one of the grand challenges of Artificial Intelligence.

Traditional game tree search techniques used in Chess do not work very well in Go. They achieve strength of an intermediate player (5-kyu) but are easy to beat by advanced players. The reasons are much speculated about; it is commonly accepted that the branching factor in Go is much higher than in Chess [All94], while pruning moves is tricky and writing a reliable evaluation function is not easy as stone groups can have varying tactical status and human strategy is guided by abstract, intuition-based principles. Also, reuse of previous evaluations is difficult since simple transposition tables cause only a minor reduction of the search space.

However, the field of Computer Go has experienced a quick dash forward after the seminal paper [KS06]. It turned out that the Monte Carlo Tree Search (MCTS) algorithm works extremely well for Go, and after much further work, top programs are now difficult to beat even for professional players on the small 9×9 board, and can compete with advanced players (4 dan) on normal board sizes. Most current research in the field focuses on improvements of the MCTS techniques.

However, much work still remains to be done, and there are significant hurdles preventing further strength growth on large boards. While MCTS usually makes good strategy decisions, its tactical evaluation can degrade rapidly if the situation does not align well with implemented heuristics, and its strategy may cease to be effective in case of situations with large advantage or disadvantage (e.g. in handicap games; Sec. 4.1.1).

In this work, we present the architecture and algorithms of our Go-playing program Pachi. It features a state-of-art implementation of MCTS for Go and consistently ranks among the top programs in competitions; we believe it is currently the strongest open source Go program and only a few proprietary programs can outperform it. We detail the variation of mainstream algorithms we use and reason our implementation choices based on collected performance measurements.

Furthermore, we shall outline an approach to solving some of the current issues in Computer Go. The common underlying theme is *information sharing:* we have observed that many problems stem from the lack of coordinated exploration in independent branches of the game tree in the standard MCTS algorithms. There are no established ways of adjusting the search for overly adverse or beneficial situations, and no method for representing and using tactical findings that do not directly align with the in-simulation heuristics.

The gist of this thesis is an amalgam of three papers that are currently going through the peer review process or being prepared for publication: **Balancing MCTS by Dynamically Adjusting Komi Value,**[1] **Pachi: State of Art Open-Source Go Program,**[2] and **Some Information Sharing Methods in Go MCTS.**[3] We have further rearranged, revised and extended the contents of the papers with the goal of presenting a full and coherent picture of the currently used algorithms and our improvements.

In the first chapter, we present the game of Go and the field of Computer Go. The second chapter follows up with the overview of MCTS in general and its application in Go; we also present the information sharing problem in more detail. The rest of the thesis describes our original work: we present our program Pachi and its specific variant of MCTS in the third chapter, and all our proposed methods of information sharing in the fourth to sixth chapter. In the seventh chapter, we consider some related research ideas.

The attached CD contains the thesis PDF, Pachi source code, Pachi executable compiled for Debian Wheezy x86_64 Git repository with the development history and shown game records in the SGF format.

[1] Submitted to the ICGA Journal in March 2011, currently a revision is waiting for peer review.

[2] Most likely will be submitted for the 2011 Advances in Computer Games conference.

[3] In preparation.

1. Game of Go

Go has its roots at least in the 6th century B.C. of ancient China [Fai08] and is commonly recognized as the oldest board game in existence. In the first millenium A.D. it has spread to Japan and Korea by cultural exchange, and became an important part of the East Asian culture. By the beginning of the 20th century it was most popular in Japan where the strongest players also lived and it has spread from there to the Western world; most Go terms used are still based on Japanese words. However, in the second half of the 20th century, Go regained popularity in China (where it is known as *Weiqi*) and especially Korea (its Korean name is *Baduk*) and by the beginning of the 21th century, it is fairly well known in the Western world as well. While Go is commonly played in clubs and tournaments, play on dedicated internet servers (e.g. KGS [Sch]) also became very popular and represents a useful testing venue for computer Go programs.

While Go is a game with simple rules, the array of strategical possibilities and tactical techniques are vast and mastering them requires many years of study. Therefore, it is important to measure the skill level ("strength") of players. Beginner and intermediate players are assigned so-called "kyu" ranks decreasing from 30-kyu for beginner[1] to 1-kyu for a fairly skilled player. Advanced players receive master "dan" ranks increasing from 1-dan to 9-dan[2] and professional players also use "dan" ranks, but on a different scale.[3] The rank difference of two players may be used to determine the amount of handicap stones so that they play a roughly even game (see Sec. 4.1.1). Nowadays, ranks are determined from Elo ratings [Cie+]. Various regional Go associations and internet servers use their own ranking systems that work as described above but are slightly differently calibrated.[4]

1.1 Rules

Go is a two-person, zero-sum game with full information. It is played on a square grid of a given size — *board*; 19×19 is the most popular, but e.g. 9×9 and 13×13 are also common. Each intersection of the grid is a *point*. Two players alternate in making moves consisting of placing a stone of their color on a point of their choice. Black and white colors are used, black makes the first move.

Stones of the same color that are adjacent and directly connected by grid lines form a *group*[5] and they share *liberties* — unoccupied points adjacent to a group. When a group runs out of all liberties, it is *captured* — the stones are physically removed from the board (and therefore, points are turned from occupied back to empty). For example, in Fig. 1.1 the stone A can be captured by black playing

[1] In some systems 20-kyu is the weakest rank while other start at 50-kyu. The ranking is not very acccurate for the least skilled players.

[2] Again, some systems have a cap at 7-dan while others have no top limit at all.

[3] Weakest professional players may be around the level of amateur 7-dan.

[4] Therefore, European 3-kyu might possibly be 1-kyu on KGS and 1-dan in United States. [Sen]

[5] We use the term "group" as this is the terminology used in Pachi sources and among most Go players. However, some of the scientific literature uses the term "chain" to emphasize that only solid connections are considered here.

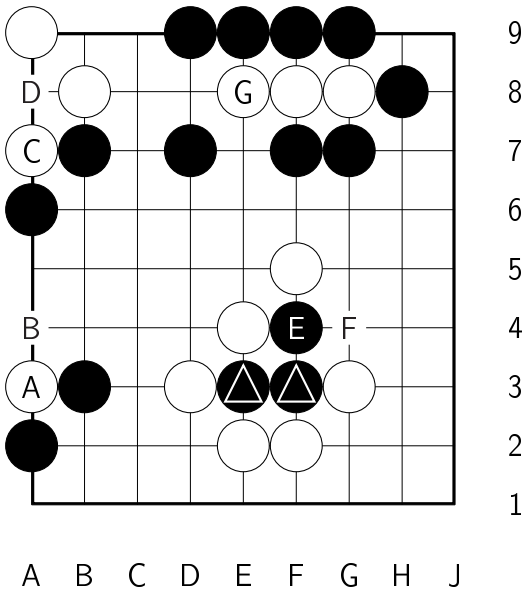


Figure 1.1: Example figure showing various Go situations.

B, stone C can be captured by D and E together with the triangled stones may be captured by white F. It is prohibited to play a stone that would have no liberties (unless it captures a neighboring group).

The goal of the game is to achieve higher *score* than the opponent, i.e. number of points occupied or completely surrounded by stones of the player (plus *komi*, several points given to white as a compensation for black playing first). The game ends when both players play a *pass* move (giving up their right to place a stone), usually when the score cannot change anymore.[6]

The capture rule would enable game loops. For example, consider the situation around C and D in Fig. 1.1: if one player captures the opponent's stone, the opponent could capture back the player's capturing stone right away, and so on. To prevent this, the *ko rule* prohibits immediately playing at the point of a single-stone capture; first, at least one other move must be made. The ko rule is triggered fairly often and it is tactically important. Frequently, capture or survival of a group might hinge on the result of a "ko fight": after one player captures, the other will play a move elsewhere such that the capturing player will face a dilemma — fill in the ko point or reply to the last move (ignoring it will usually mean some loss to the player). This way, ko fights may connect otherwise unrelated parts of the board.

1.2 Rulesets

There are multiple formulations of the rules. Rulesets take different approaches to the scoring phase, dealing with repeated situations *(the superko rule)* and other minor points, but the gameplay, tactics and strategy involved is almost the same.

[6]Sometimes, the players pass before all stones of the opponent that could be captured are removed from the board, if both players agree the owner cannot save them anymore. Formally, passing rules are usually more complicated to facilitate this agreement, but in casual games just two pass moves are used.

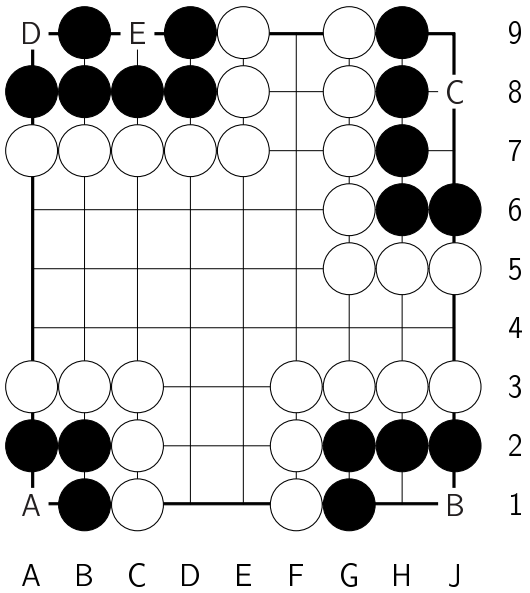


Figure 1.2: Life and death of groups.

Various rulesets strike different balance between precision, practicality and tradition. The rules we have described above closely match the mathematically precise Tromp-Taylor rules [TT]. By disallowing "suicide moves" (removing the last liberty of one's group), we essentially gain the Chinese rules that are most commonly used by MCTS-based programs. Japanese rules are usually used among western players and they take a very different approach to scoring ("territory scoring" vs the "area scoring" of Chinese rules), but it can be shown that in most situations, the score difference between the two methods would be at most one point.

1.3 Basic Gameplay

We shall mention few basic implications of the rules to better understand the general gameplay and roles of various heuristics and establish some vocabulary.

We use the term "tactics" for techniques directly involving survival or capture of groups. The most crucial is the tactical concept of "eyes": a single point completely enclosed by stones of same color.[7] [8] Eyes have direct relation to the *status* of the group. A group will eventually get surrounded by opponent stones, its liberties reduced. If it surrounds no empty points or has only a single eye, all its liberties will eventually be removed and the group is doomed to be captured — we call it *dead*. However, if the group manages to enclose two eyes, it becomes *alive* and its liberties cannot ever be reduced below two[9] as the opponent cannot fill both eyes at once.

[7] In colloquial usage, an eye can also describe an area of multiple points that cannot be reduced to multiple eyes.

[8] When we use phrases like "cannot be reduced," usually we assume "given competent play by both sides."

[9] Finding a way to reduce a small territory to a single eye or split it in two eyes may be very hard problem, but it is crucial to strong play. Exercises that require solving this problem are called "life and death" or "tsumego".

Fig. 1.2 shows examples of some basic life and death situations. The black group around A has the last liberty in its eye A; if white plays a stone at A, the move will capture all black stones. The black group around B has two liberties, but white can still capture it. If white plays a stone at B, it will be in atari and black can capture it, but by playing the capturing move the black group will be also down to a single liberty and white can play at B again. However, the group around C is unsettled. If white plays at the spot, the situation may be further reduced in turn to B and then to A and white will eventually capture. However, if black plays at C, the group will gain two eyes (like D and E) and white cannot capture the black group anymore.

As the game proceeds, stones begin to enclose *territories*; a group encloses territory if it is large enough (or properly structured) so that the surrounded area cannot be reduced to a single eye, and small enough that the opponent cannot start playing within the territory and be able to form a two-eyes group. It is important to realize that capturing opponent's stones has at least twice the score value of the territory underneath the stones: normally, each played stone intrinsically gains the player a single point of score, but if a stone has been captured, the capturing player gains the point underneath the stone while the captured player loses the point coming from presence of her stone on the board.

The whole game therefore revolves around two goals to be balanced: fighting the enemy groups to capture them and enclosing enough territory. In the game opening ("fuseki"), players stake out their general areas of influence and potential territories; usually, they start in corners (which are already enclosed from two sides) and then divide the sides.[10] The middle game then follows — frequently, one of the players stakes out a territory claim so large that the opponent must play a move among the strategically placed opponent stones and try to convert it to a living group. The game may evolve to a series of complex fights with many groups being split, connected and struggling for survival. Eventually, the status of all groups is settled and the endgame ("yose") wraps the game up as the players negotiate precise boundaries of territories.

A group that has but one liberty is called to be *in atari*. Putting one's group in "self-atari" is usually a bad move since the opponent may capture the group in the next move, but it is sometimes necessary: e.g. when preventing formation of two eyes within a group territory (the "nakade" technique). distinguishing between good and bad self-atari moves is a fairly complex problem (Sec. 3.4.1), but solving it may be important for program's performance.

Sometimes, two neighboring groups may become involved in a *capturing race* ("semeai") — both are surrounded by groups of opponent color and neither can enclose two eyes. Each player thus tries to capture the opponent's group first by reducing its liberties as fast as possible. Sometimes, this fight reaches a stalemate *seki* where the two groups share their last two liberties and therefore one cannot reduce the opponent's group liberties without putting itself in atari.

After describing the basic concepts, we shall mention at least two most basic actual tactical techniques. One may catch a group with only two liberties in a *net* ("geta") by playing nearby so that if it attempts to escape at any point, the player may catch it in atari that cannot be broken. For example, in Fig. 1.1 the

[10]Tactical exchanges may happen in the opening, usually following "joseki" sequences commonly agreed to lead to an equal result for both players.

group G cannot escape anymore.

On the other hand, catching a group in *ladder* means that even if the group in atari attempts to escape by "pulling out" at its last liberty, the opponent can continue putting it in atari from alternating sides until the group reaches the board edge or a supporting stone of the opponent and cannot escape anymore — if the group instead reaches a stone of its own color, the ladder would not work and the group would be saved.[11] Ladders are interesting in two regards — they can tactically connect two distant areas of the board, e.g. opposing corners if the middle of the board is yet unoccupied, and they are easily read even by beginner players while traditional game-search techniques are quickly daunted by length of the (principially straightforward) sequences. In Fig. 1.1 the group E cannot escape anymore — if black tried to play at F, white may put the group in atari again.

We have just scratched the surface of all the possible tactics and strategies stemming from hundreds of years of intense Go research. Interested readers may find a wealth of information e.g. on the Sensei's Library community wiki [HP+] where they will also find explanations for any other Go terms we might use.

1.4 Computer Go

We shall give only a very brief overview of the techniques used in Computer Go in the past before diving into the Monte Carlo Tree Search in detail. Surveys like [BC01] provide good overview of the pre-Monte Carlo approaches.

Early Go programs [Zob70b] were based around an influence function describing effect of stones on potential territory. Pattern-based play [Boo90] and attempts to mimic human understanding of the game by division to zones [RW79]. Standard techniques such as α, β-search, specialized submodules for concrete tasks and rule-based expert systems had limited success; the efforts in "classical go techniques" culminated with GNUGo [BFB+] and non-MCTS version of Many Faces of Go [Fotb], both ranked around 5-kyu on the KGS Go server. Popular alternative approaches like neuron networks [Enz96] were not successful.

However, some subproblems of Go were tackled successfully. The Combinatorial Game Theory has been applied to very late Go endgame and solvers based on CGT analysis are able to beat top professional players [BW94]. Also, solvers of isolated tsumego (life and death) problems based on α, β-search are now capable of solving even problems that are considered of professional-level difficulty [Wol07].

[11] A mistake of trying to escape a working ladder or trying to catch a group with a non-working ladder usually results in a dramatically disadvantageous position.

2. Monte Carlo Tree Search

Monte Carlo methods invoke the law of big numbers by performing large amounts of randomized "simulations", each giving a semi-random reply to the original question. We then assume that the most likely answer gained the largest percentage of replies, assuming that the results are unbiased and follow an appropriate random distribution. We use the randomized simulations to evaluate Go positions and efficiently build a game tree based on the randomized simulations.

2.1 Monte Carlo in Go

The first experiments regarding Monte Carlo approach in Computer Go date to 1993 [Bru93], but further research [BH03] did not have large impact. The approach consisted of examining a situation and using Monte Carlo simulations to do single-level sampling of possible next moves.

Various techniques, e.g. progressive pruning strategies or the all-moves-as-first heuristic (see Sec. 2.3.3), were important inspiration for their MCTS equivalents. However, single-level Monte Carlo search never overcame its fundamental limitation of being unable to discover move replies in arbitrary depth. Therefore, while it was surprisingly good at finding strategically interesting moves that set up a large attack or enclose good chunk of territory, it suffered problems in tactical situations — a moves was rated high if the random simulation was not likely to play the appropriate reply, and vice versa.

2.2 Multi-armed Bandit Problem

The archetypal problem of iterated decisions in stochastic environment is the Multi-armed Bandit Problem. A "one-armed bandit" is term often used for a simple slot machine; the player inserts a coin and according to a fixed but unknown probability distribution, the bandit may give a reward. The "multi-armed bandit" then corresponds to a slot machine with multiple arms, each with an independent stationary probability distribution. The player's objective is then to repeatedly choose arms in such a way to maximize the total reward.[1]

The exploration-exploitation dilemma then concerns the fact that the player must follow two goals — go for the highest reward choosing the best arm according to her knowledge and improve the estimate of rewards for all arms. At the beginning, the player has to resort to random tries and only as the rewards start to come in, she gets the picture of which arms are most profitable; however, there is always a possibility of statistical error, thus exploration is always necessary for an optimal player.

More formally, [LR85], [KS06] and [GW06] define the K-armed bandit as a sequence of random rewards $X_{it} \in [0, 1], i = 1, \ldots, K, t \in \mathbb{N}$, where each arm has an index i and t denotes successive plays of the arm. A bandit policy π is then a function that selects the next arm to be played based on previous rewards.

[1] As far as we know, the problem was, albeit named differently, first proposed in [Rob52].

The accepted measure of a policy is its "regret" — how much reward we miss because of not always choosing the optimal arm (i.e. either we explore too much and do not choose the best arm often enough, or we explore insufficiently and have a wrong idea of the best arm). Let μ_i denote (originally unknown) expectation of arm i, μ^* the expectation of the optimal arm, $T_i(n)$ the number of times arm i has been played after the first n plays:

$$\mu_i = \mathbb{E}\left[\frac{1}{T_i(n)}\sum_{t=1}^{T_i(n)} X_{it}\right]$$

$$\mu^* = \max_i \mu_i$$

$$R_n = n\mu^* - \sum_{i=1}^{K} \mathbb{E}\left[T_i(n)\right]\mu_i$$

It was proven early [LR85] that the lower asymptotic bound[2] for the regret R_n is $\Omega(\ln n)$. In other words, the optimal arm is chosen exponentially more often (asymptotically) than any other arm. Multiple policies have been proposed; a popular one is the "ϵ-greedy policy," choosing the arm with the highest estimated expectation with probability $1-\epsilon$ and a different random arm with probability ϵ.

Another class of policies are such that in each step, we compute an "upper confidence index" for each arm and pick the arm with the highest index. The Upper Confidence Bound (UCB) policy[3] [ABF02] implicitly negotiates the exploration-exploitation dilemma by adding a relative measure of uncertainty to the estimated expectation; therefore, even low-expectation arms are occassionally explored when the uncertainty is too high compared to other arms:

$$\pi_{UCB1}(n) = \text{argmax}_i \left(\mu_i + c\sqrt{\frac{2\ln n}{T_i(n)}}\right)$$

Discovery of this policy has proven to be an important result since the policy follows the logarithmic bound not just asymptotically but also uniformly, and it is much more versatile in the face of unknown distributions than the ϵ-greedy strategy.When used in a particular domain, the parameter c of UCB can be tuned for the optimal exploration-explotation ratio.

Eventually, a quite different policy has gained prevalence in Computer Go: the RAVE policy, instead of giving an universal upper bound, takes full advantage of the tree and simulation structure of MCTS and estimates the upper confidence index based on previous performance of the arm in related bandits. We describe the details below.

[2]This is assuming a general probability distribution of the arms.

[3]We describe the UCB1 policy introduced in [ABF02]. The paper also describes UCB2 and UCB1-TUNED algorithms and the authors of Mogo found it works well in their case [GW06]. However, in all our experiments, UCB1 performed much better as the tree policy of UCT described below.

2.3 Bandit-based Game Tree Search and Upper Confidence Tree

Let us now bring together the concepts introduced in this chapter so far. We have discussed the Monte Carlo approach, but it has not been successful due to its lack of support for sequence evaluation. But with the multi-armed bandit policies, we have gained an excellent tool to integrate the Monte Carlo approach with a minimax tree that will dispatch the simulations efficiently.

Let us examine the general Monte Carlo Tree Search algorithm:

Algorithm 1 MCTS

Require: s is a certain whole-board situation.
Require: node(s) is a tree node corresponding to the given situation.
Require: $\pi_T(s)$ is the new situation after tree policy decision at a given situation.
Require: $\pi_S(s)$ is the new situation after simulation policy decision at a given situation.

```
while time is available do
  s ← RootSituation
  while node(s) is not leaf do
    s ← π_T(s)
  end while
  n ← node(s)
  MaybeExpandNode(n)
  while s is not final position do
    s ← π_S(s)
  end while
  r ← Evaluate(s)
  while n exists do
    UpdateNode(n, r)
    n ← parent(n)
  end while
end while
Play(argmax_{n∈child(RootSituation)} sims(n))
```

The Monte Carlo tree search consists of repeated episodes, where each episode descends the tree from the root to a leaf, takes a single sample based on Monte Carlo simulation at the leaf position, then updates the node values with the sampled data.[4] Initially, the tree consists only of the root node, and more nodes are added when their parents are visited.[5] Eventually, the root child with the highest number of simulations is chosen.[6]

[4] Of course the game tree is minimax. Therefore, either the expectations are for the to-play player in each tree layer, or the π_T policy minimizes the expectation in the opponent-to-play layers.

[5] The concrete expansion strategy may vary between programs. In Pachi, after visiting a node n times, children for all possible followup nodes are added.

[6] It might seem natural to choose the node with the highest expectation, but these values may fluctuate; MCTS compensates for the uncertainity and the number of simulations is thus a much more robust estimate.

Note that the values used to update the expectations are generally just either 1 for a win or 0 for a loss; score-based values were used initially, but it was quickly discovered that using a sharper division is much more robust [GW06] since Go games are usually fairly close and we want to avoid preferring moves that have just a small chance of large score swing — after all, a win is a win, no matter by how many points.[7]

Let's now join the UCB multi-armed bandit policy and the general MCTS algorithm — the specific algorithm is called UCT (Upper Confidence Tree) [KS06] and this combination has made the first significant headways in the field of Computer Go [GW06].[8] Let's give a simpler context-specific formula for *expectation*[9] μ_i of node i; it is simply the average outcome of simulations[10] going through node i:

$$\mu_i = \frac{wins(i)}{sims(i)}$$

The MCTS algorithm has several remarkable properties. It is an online algorithm that can be stopped anytime and the confidence in the current result will be proportional to time spent. The tree is highly memory efficient since it grows asymmetricaly based on the prevailing direction of the search. It also requires no extra domain knowledge per se — MCTS works even when π_S is uniformly random policy. In that case, it is only required to recognize and score the final position, i.e. just the most straightforward rules implementation is sufficient.

2.3.1 Simulation Policy

After we descend the tree to a leaf node, the simulation phase begins with the purpose of getting a Monte Carlo evaluation of the board position corresponding to the leaf node. The simulation policy π_S picks move after move until the final position is reached. Since MCTS engines usually use Tromp-Taylor or Chinese rules (see Sec. 1.2) for simulations, all the territory may be filled with stones and playing within the opponent's territory is not harmful.[11] Therefore, a common termination condition is to stop playing moves when no more moves are available, combined with a rule that prohibits filling own one-point eyes.[12] [13]

Of course, in practice π_S is packed with a rich mix of (mostly domain-specific) heuristics. In fact, perhaps most of the day-to-day program improvements and

[7] Pachi does allocate a small portion of the propagated value to rescaled score information (we call this *value scaling*), but the effect of this improvement is small.

[8] One must be however very careful when trying to apply results of multi-armed bandit research to MCTS. Some of the general assumptions may not hold, for example the arm probability distributions are not stationary as the tree gets expanded.

[9] In case of MCTS, we also use the term *winrate* synonymously.

[10] In case of Go, we also use the term *playouts* synonymously.

[11] At least when all dame has been played or both players are equally likely to do it.

[12] Filling one's own eye is reasonable only in exceptional circumstances; usually, it will only prevent life of one's group. Furthermore, without this prohibition, in the final position all groups would eventually have only a single liberty, the opponent would capture them and playing would resume in the freed up space.

[13] To avoid disturbing seki positions, some other non-eye points should be left empty. Simple MCTS implementations usually ignore this issue. Pachi addresses it by prohibiting self-atari moves in playouts with high probability (Sec. 3.4.1).

tuning is spent by tweaking the simulation policy. The policy often involves spatial patterns, tactical suggestions (the simplest being protecting or performing capture), handling of ko fights or protection against tactically unsound (e.g. self-atari) moves. We will explain Pachi's playout policy in detail in Section 3.4.

An important issue is how to combine all the heuristic suggestions. Two general models are usual in MCTS programs:

- **Rule-based policy** (based on the program Mogo [Gel+06]). The heuristics are applied in a fixed order, the suggestion of the first matching heuristic is chosen (or a uniformly random move if no heuristic triggers).
- **Probability distribution policy** (introduced in the program CrazyStone [Cou07]). On each move, all available moves (or their large subset) are assigned an expectation as a product of evaluations by individual heuristics matching the particular move and surrounding spatial patterns. The expectation then corresponds to relative probability of choosing the move.

The probability distribution policy is a prominent feature of the currently strongest programs. However, while we have spent a lot of effort on implementing this policy in Pachi, we have not been successful so far, therefore we still use the rule-based policy, which is also much simpler and faster.

Tuning the simulation policy is difficult and still considered largely a "dark art". It is not a matter of merely increasing its strength as a stand-alone player, or of its predictive power. Often the overall program strength is reduced when improving the domain-specific knowledge; the MCTS may converge slightly faster in general, but the heuristics always handle specific cases wrong and the possible resulting MCTS biases are a serious problem resulting in the horizon effect (Sec. 2.4.2).

We must also mention a recently proposed methodology to tackle this problem when automatically learning heuristic weights. "Simulation balancing" [ST09] [HCL10] is based on the idea that instead of minimizing the error of each individual prediction, the combined error of both players' predictions should be minimized: if a heuristic consistently favors one player in some situations, it will lead to undesirable bias, but if it makes errors fairly for both players, the biases will compensate.[14]

2.3.2 Prior Values

When we described leaf expansion, we have not specified one important thing — how are the newly created nodes initialized, or how are the nodes initially selected.

In the original algorithm, it was prescribed that each newly created node is tried once to seed the expectation. However, this involves a large overhead since typically, only few of the moves are likely to succeed. "First play urgency" represents a minor improvement of this method; unexplored moves are assigned a constant UCB value such that if moves explored so far are yielding good results, they will be explored further before other moves are tried.

[14] We are not performing any automatic simulation policy learning in Pachi yet, but we value the general idea.

However, an important part (see Table 3.2) of a modern MCTS-based algorithm is a more sophisticated way to choose moves to be explored when no actual statistics have been collected yet. A set of heuristic functions is applied to newly created nodes to perform some static evaluation; e.g. moves near to the last move are preferred to encourage local tactical search, solitary moves at the board edge are discouraged since they have little influence on the game and preference is given to nodes agreeing with the domain-specific playout policy rules. Two distinct approaches for applying the heuristics in practice are popular [Cha+07]:

- **Progressive widening (unpruning)** [Cou07] applies a ranking function to each newly created move. Then, given n visits to the node so far, only the first $f(n)$ children (ordered by the ranking function) are considered.[15]
- **Progressive bias** [GS07] adjusts the expectation of nodes instead; virtual simulations are considered together with the real simulations, with the amount and results of the virtual simulations based on the applied heuristics. The strategy is also progressive since the fixed number of virtual simulations has decreasing effect as the nodes are explored.

There are successful programs using either strategy, though we feel progressive bias is more widespread. In Pachi at least, we have found progressive bias more effective. We will talk in detail about the heuristics we use in Sec. 3.3.2.

2.3.3 Rapid Action Value Estimation (RAVE)

We shall finally describe the first mean of information sharing, first proposed by [GS07] and representing a major leap in the MCTS strength. In plain MCTS, the only data gathered from a simulation is its ultimate outcome, and it is used only for update of expectations along the path from tree leaf to the root node. RAVE both extends the amount of data gathered in simulations and distributes it among a larger set of nodes.

Aside of the regular expectations based on immediate simulation outcomes, we shall also introduce *AMAF (all-moves-as-first) expectations*. The AMAF heuristic [BH03] was introduced even before MCTS; normally, we consider only results of random games where we played the examined move in the current situation, while all-moves-as-first suggests to consider results of random games where we played the examined move *anytime* (after the current situation).

The RAVE algorithm is two-fold. First, it describes the propagation of AMAF expectations based on "common game prefix". AMAF expectation of move m in situation s is made based on outcomes of all simulations visiting situation s and playing m anytime later in the game.[16] Therefore, when propagating the simulation result, AMAF expectations of all siblings along the path are updated.

Second, the RAVE formula provides a way to combine regular and AMAF expectations. Let $sims_{RAVE}$ be the number of AMAF samples at the node and $wins_{RAVE}$ the number of sampled AMAF wins:

[15] $f(n)$ is a monotonically increasing function (e.g. $\log n$).

[16] This includes moves made in the Monte Carlo simulations. This much increases the importance of simulation quality as any biases within the simulations will reflect in the AMAF statistics.

$$\beta = \frac{sims_{RAVE}}{sims_{RAVE} + sims + sims_{RAVE} sims / sims_{EQUIV}}$$

$$\mu^{RAVE} = \beta \frac{wins_{RAVE}}{sims_{RAVE}} + (1 - \beta) \frac{wins}{sims}$$

$$\pi_{RAVE} = \mathrm{argmax}_i \, \mu_i^{RAVE}$$

Note the lack of exploration term. Some programs combine RAVE with the UCB-style exploration term (usually using a fairly small c multiplier), but we have found we obtain the best behavior by guiding the exploration purely by the (very noisy, after all) AMAF statistics. The β parameter[17] is designed to give higher precedence to AMAF expectation when the number of simulations has been much smaller than $sims_EQUIV$ while regular expectation takes over as the number of simulations goes up. For absolutely unexplored nodes, regular expectation also takes precedence as it is seeded with the prior value simulations while AMAF has gathered no simulations yet.

RAVE enables much quicker convergence to good moves and sequences as it picks up promising moves from simulations even if the prior heuristics do not suggest them or have too weak effect. However, it is more sensitive to simulation biases — if the simulation will never choose a good move at the right time, it will never gain the required amount of positive AMAF samples.

2.4 Information Sharing

After introducing all the basic concepts of Monte Carlo Tree Search, we now have a chance to discuss the main theme of this thesis: information sharing.

The game tree search considers each branch separately, and the only knowledge gained and stored is whether the specific branch leads to a positive or negative conclusion. There are three obvious approaches for improving the search: pruning branches, improving accuracy of the evaluation, and extending the set of data collected and used during the search.

A prominent example of the latter and the emblem of successful information sharing is RAVE. It speeds up identification of crucial moves by sharing statistical information with sibling branches. In this thesis, we tackle two general problems using information sharing methods.

2.4.1 Situational Information Sharing

Some of the game search parameters would benefit from assessing the general board situation, as recognized based on statistics collected over all performed situations: if most simulations reply the same result (i.e. the situation is extremely advantageous or disadvantageous), the resolution of the results would be improved if the score threshold is adjusted (dynamic komi, Ch. 4).

[17] Derived to minimize the mean square error of the sum when assuming binomially distributed expectations.

2.4.2 Horizon Effect

Even more prominent problem concerns local positions. On large boards, the situation can be loosely decomposed to many local positions, each concerning a set of groups that interact together and an optimal sequence to resolve this interaction. Of course, the positions are usually not entirely independent — any positions can influence each other in ko fights, and nearby positions often interact simply by changing status of involved groups. A rule of thumb is that if one position is resolved aggressively by one of the players, it is usually at the expense of safety of some of her groups, and the player therefore has to play more defensively in other positions involving this group to secure the required eyespace or escape routes.

Nevertheless, there are still many positions that may be considered almost isolated — usually corner and side situations with inner group of one player entirely surrounded by outer alive groups of the opponent (tsumego) or two groups involved in a capture race (semeai). The status of the inner group then depends on its eyespace and even though it may not be very large, the sequences to kill or secure life of the group may be fairly complex and even advanced human players may have trouble finding solutions easily. There exist dedicated solvers for these positions based on α, β-search (as noted in Sec. 1.4) but MCTS in itself is surprisingly weak at solving them:

- The correct solution is usually an exact sequence and any deviation can be catastrophic for the player. Therefore, the tree may not gradually converge to the solution.
- If a move is made elsewhere on the board, the exact sequence must be re-read repeatedly in each tree branch that does not touch the actual position.
- The simulation policy behaves in a certain manner in the position. Either the heuristics used in the policy resolve the local situation correctly[18] or the results are to some degree biased against the correct solution.

The third aspect is especially daunting when RAVE is used, since the tree exploration completely relies on the implication of good move by positive AMAF statistics. If the simulations produce biased results, they also feed the tree wrong AMAF information and the proper move may never be searched. But even if it is discovered in the tree, it is never fully read out; for the side that is disadvantaged in the proper solution, the in-tree result for the proper move is worse than in-tree result for an unrelated move that leads to a branch that resolves the position in the incorrectly behaving simulation. Therefore, the minimax game tree has a tendency to avoid the situation and push it out to the simulations. This is commonly called the "horizon effect".

When a MCTS-based program loses a game, in our experience it is mostly due to a misbehaving heuristic and ensuing horizon effect; a losing position is evaluated incorrectly in the tree, therefore the program avoids playing in the vicinity, but based on data from the simulations it behaves like the position would be unclear or favorable. Eventually, such positions tend to accumulate

[18] One could even argue that almost all the heuristics used in simulations have this purpose.

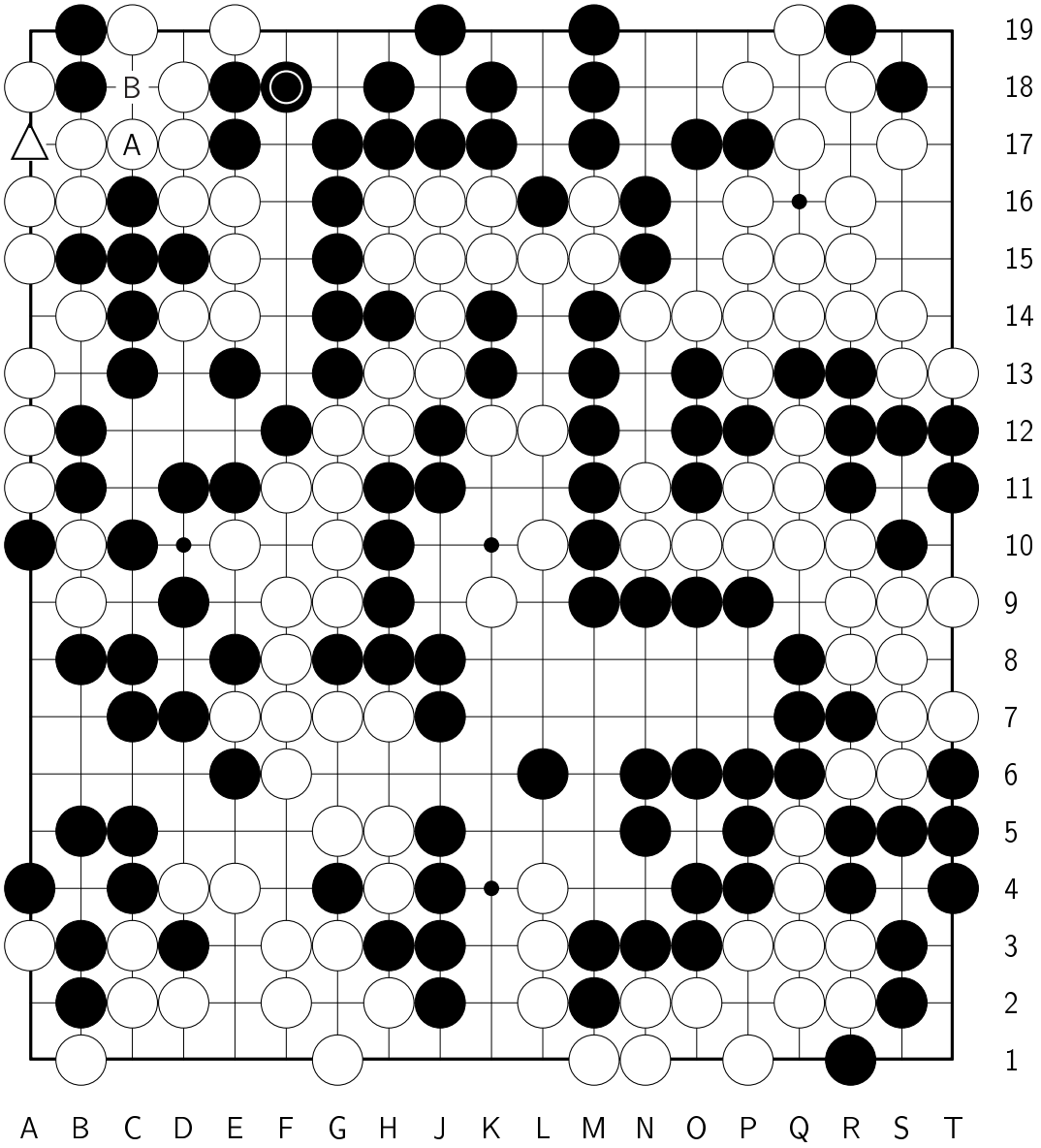


Figure 2.1: Example horizon effect position (black **Pachi30s** vs. white **botkiller2**, Jun 9 2011, KGS). In simulations, white is likely to play at the triangled position and therefore reducing the A group to a single eye. Therefore, the program incorrectly believes the group is effectively dead with high probability. The game tree would in time find that B for white is much superior to the triangled move, but that will only cause the minimax tree to switch from the previous black move to a different one until the game tree again discovers that B is better than the triangled move. Then it switches again and so on.

on the board, and either the opponent eventually makes clarifying moves, or the game proceeds to an almost-final position when there are no alternative moves in the tree anymore.

The usual solution is to add and improve heuristics used in the simulations. While this is certainly worthwhile to a degree, it quickly evolves to fighting a time-consuming losing battle. The heuristics soon need to be more specific, covering an increasingly narrower set of situations, and the simulations are bogged down by multitudes of checks. Therefore, we focused our research on the investigation of generic ways that would allow us to (i) identify critical moves that need extra attention, (ii) share resolutions of local positions between tree branches and (iii) dynamically adjust local position resolutions in simulations.

One previously proposed way to find critical positions has been the "criticality" function measuring covariance of owning a given point and winning the game; we have found a good way to use it during the search directly (Ch. 5). Also, we have developed a way to share information on effectivity of various heuristical moves at ensuring survival of a group (liberty maps, Ch. 6)

2.4.3 Local Value

We need a general way to assess move effectivity in a local position — its **local value**. Let $(coord, color)$ be the move and fcolor_c be a color[19] of the point c in the final game position. A simple local value measure we have devised is (usually averaged over many simulations):

$$\mathrm{nei}(c) = \{\mathrm{left}(c), \mathrm{right}(c), \mathrm{up}(c), \mathrm{down}(c)\}$$

$$\mathrm{lvalue}(color, coord) = \frac{1}{3} \cdot [\mathrm{fcolor}_{coord} = color] + \sum_{c \in \mathrm{nei}(coord)} \frac{1}{6} \cdot [\mathrm{fcolor}_c = color]$$

In other words, we assume that the move is locally successful if the stone turns out to be still alive at the game end, and furthermore as many of its neighbors as possible turn out to be owned by the player. The latter is important especially for "outside moves" affecting a semeai — it gives precedence to moves that lead to the capture of the opponent group.

[19] We also declare the intersection to be of some color if the intersection is an eye of a group.

3. The Pachi Software

We have implemented a general framework for Go playing program and a Monte Carlo Tree Search engine on top of this framework. In this chapter, we will detail the architecture of our software and re-state the Computer Go problem and MCTS algorithm from the point of view of our specific implementation.

3.1 The Pachi Framework

The design goals of Pachi have been simplicity, minimum necessary level of abstraction, clarity of programming interfaces and focus on maximum playing strength. This has various consequences quickly apparent to new Pachi developers and users.

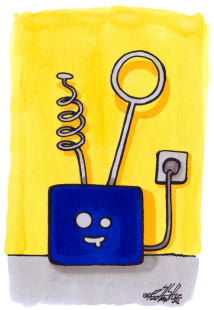


Figure 3.1: Official Pachi artwork by Radka Hanečková.

At the time we started our work on Pachi,[1] several programs with open source code already existed — most notably, GNU Go [BFB+], Fuego [Enz+10], libego [Lew] and Orego [Dra+]. We have borrowed useful ideas (mainly various implementation tricks) from most, but still we have chosen to develop a new program. This has been in part since we were simply unaware about some of these back in 2007, but also because we were uncomfortable with some design choices we have seen and we desired to learn the most about Computer Go by starting from scratch.

To put Pachi in contrast with these: Fuego is mainly a generic framework that provides a generic library for the game of Go and a generic library for game tree search; the go-playing engine emerges as a combination of these two libraries with several heuristics included. However, for our taste the Fuego architecture has too many layers and is therefore overly complex and difficult to extend for inexperienced users. GNU Go is a classical game search engine that has been largely abandoned and while it boasts a great wealth of heuristics, patterns and tactical modules, adapting it to MCTS appeared to be a formidable task. We weren't aware of libego and Orego at first, moreover while libego is tiny and very efficient, it was not ready for implementation of more detailed heuristics at that moment, and we weren't excited about Orego's choice of Java.

3.1.1 Software Development

Pachi is licenced under the GNU General Public Licence (version 2) [Fre91], making it a free and open source software. New versions are released regularly;[2] the last release at this time is 8.00 from May 22, 2011. Current snapshot can be retrieved at any time from the public Git repository where further development is done.[3] Some codebase statistics may be found in Appendix A.

[1] Pachi is named after the clicking sound stone makes when placed on a board.

[2] The latest version is always available for download at the Pachi homepage `http://pachi.or.cz/`.

[3] The Git repository is accessible at `http://repo.or.cz/w/pachi.git`.

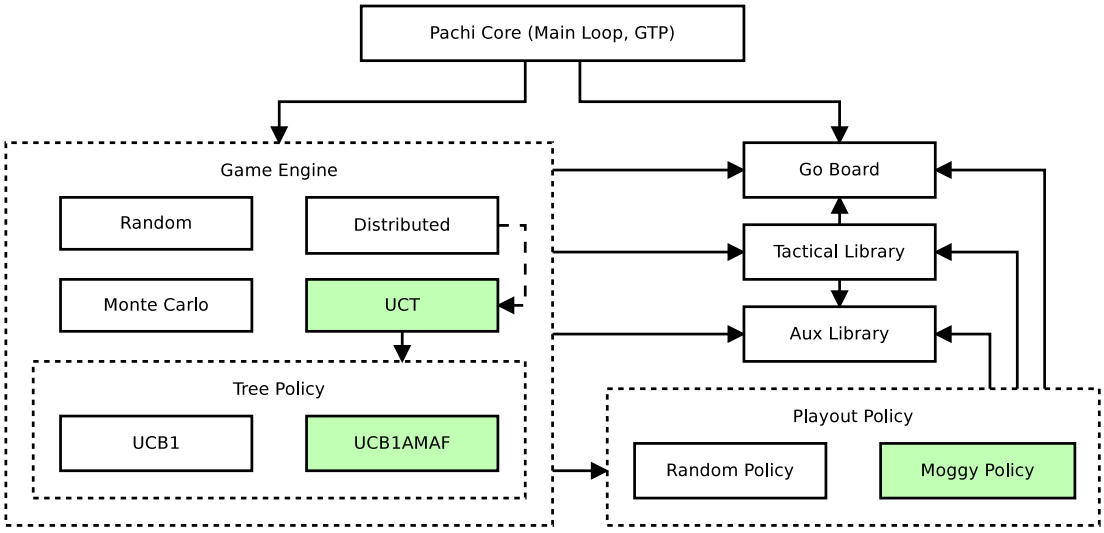


Figure 3.2: Block schema of the Pachi architecture. When multiple options of the same class are available, the default module used is highlighted.

Petr Baudiš has remained the principal developer and author of most of the architecture and the code. Jean-loup Gailly has been important contributor, improving tree memory management, co-authoring the flexible time management strategy and most importantly implementing highly scalable distributed computation support. Few minor patches came from other authors.

Pachi is written in pure C. Most of the code is richly commented and follows a clean coding style. Header files provide clearly defined interface for each build unit. Most Pachi modules are fully encapsulated and reentrant, using structures to maintain object instances and function pointers to provide a class-like interface in a manner similar e.g. to the Linux kernel.

3.1.2 General Architecture

Pachi is built with a complete focus on the game of Go, but otherwise features a modular architecture (Fig. 3.2). The common *Pachi core* provides common basic facilities, while the choice of next move is offloaded to a *game engine* that usually performs some kind of search. An *auxiliary library* provides optional supporting features for the game engine while the *Go board module* provides basic Go rules implementation and the *tactical library* offers more advanced Go heuristics. The *playout* modules provide the Monte Carlo simulation functionality (we assume multiple game engines might want to share the same playout modules).

The primary function of Pachi core is to provide an engine-agnostic foundation and the elementary shared code (with focus on the needs of Monte Carlo based engines). Its responsibilities are proper initialization, the main game loop and the GTP interface. It also provides common debugging facilities, fast random number generator (simplistic Park-Miller [PM88] implementation) and the time management strategy implementation (Sec. 3.3.3).

There are four main[4] game engines implementing various move selection strategies. The *random* engine simply plays randomly and it is meant as an example engine. The *Monte Carlo* engine performs single-level Monte Carlo search akin to [Bru93]; it is not maintained much anymore. The *UCT* engine finally contains a state-of-art implementation of MCTS for Go (the name is misleading by now, as the tree search uses pure RAVE instead of classic UCT). The *distributed* "meta-engine" is a work of Jean-loup Gailly and takes care of distributing and collecting work from multiple Pachi instances in a computation cluster; currently, only support for UCT engine instances is implemented.

The tree policy used within the UCT engine is also modular. While *UCB1* implements the classic formula (Sec. 2.2), *UCB1AMAF* incorporates management of the AMAF (and other) statistics and uses the RAVE formula (Sec. 2.3.3; the name is again a historical relic as the optimum for the UCB-specific exploration term constant is $c = 0$). Other parts of the engine (game tree, prior values, dynamic komi, ...) are also kept in separate modules with well-defined interface.

The Go board module maintains the core data structure of Pachi representing the current board situation (Sec. 3.1.4), provides related accessors and iterators and implements the Go rules (playing stones and counting score). Definitions of stone colors and the coordinate system are kept in small separate files.

The tactical library implements heuristics and checks related to tactics and strategy above the basic Go rules. Most notably, single-level tactical search for n-liberty groups, a sophisticated self-atari detector (Sec. 3.4.1) and a ladder testing tool.

The auxiliary library provides some extra data structures useful for game engines and playout modules — move queues, win/loss statistics, board point ownership statistics, 3×3 board pattern matcher, and probability distribution chooser.

The playout module task is to choose the next move in a Monte Carlo simulation; a secondary task is to provide hints for move prior values assigned by a game engine. Right now, two modules are used: the *random* module picks moves in a uniformly random manner, while the *moggy* module implements a rule-based policy making full use of all the available tactical heuristics (Sec. 3.4).

3.1.3 Program Interface

Pachi does not have a built-in user interface. It focuses just on the task of playing the game and for comfortable playing experience, a separate frontend program should be used.

All settings are specified on the command line. Command line options starting with a dash are processed by the Pachi core and specify universal settings like time constraints and debugging level. A possible other parameter is passed to the game engine and is expected to form a set of comma-separated `attribute=value` pairs. For example, the UCT engine accepts settings like number of threads, maximum tree size, exactly how AMAF statistics are collected, etc. Some of the attributes enable or choose further modules to use (e.g. tree and playout policy, dynamic

[4] The engine facility is also used internally and for debugging. The *replay* engine will simply offload the move selection to a playout module while the *joseki* engine is used for building the joseki dictionary.

komi, liberty maps, ...) and provide means to pass on further comma-separated configuration options (constants to use in the RAVE equation, probabilities of the stochastic playout rules, shape of the dynamic komi curve, and so on). Nearly all aspects of Pachi behavior are tunable by these options.

The main game-related communication of Pachi happens using the Go Text Protocol (GTP) [Far]. This is an "industry standard" used by nearly all Computer Go programs, enabling use of a variety of frontends, easy connection to internet servers and also automated computer–computer matches. The protocol is text-based and in principle can be also used directly; with the default debugging level, Pachi will print updated board diagram each time a move is made. Appendix B shows a sample Pachi session.

Furthermore, various supplementary simple scripts are included. The `sgf2gtp` script is crucial for program testing and debugging as it converts a SGF game record [Hol] as available for example from the KGS archive of played games to a stream of GTP commands. This can be also used for computer-aided analysis of previously played games. By cutting the GTP stream at an appropriate moment and appending a `genmove` command, the user may obtain the program's opinion on the situation. Similarly, the script `sgf-analyse` will plot a graph of Pachi's view of the game balance (in terms of best move winrate) after each move when given a game record. Instead of the `genmove` command, a custom command `uct_evaluate` may be used to in turn evaluate and rate each possible move in a situation.

In cooperation with Jonathan Chetwyng, we are developing[5] a new JSON standard for progressive reporting of current view of the game during ongoing move search. The periodical reports include a list of best moves and the "best move sequence", but also per-point information on territory owner and group survival (based on average color of each point in the final playout positions). The main intended application is smooth visualization using a web-based frontend (Sec. 7.2).

3.1.4 Board Data Structure

The foremost focus of most authors of new MCTS software is on raw number of simulations per second. Usually, "light" simulations are implemented and optimized for at first. Few or no heuristics are applied and the focus dwells on minimal implementation of the Go rules and uniformly random choice of moves. We have walked the same path at first.

However, the true strength improvements come from an elaborate mix of heuristics, and with RAVE (see below) they must be applied mainly within the simulations. At that point, the balance changes — while speed of move-playing routines is still important (about 20% of run time is spent there), it is not critical to increase it at any cost; instead, enough information must be maintained so that heuristics may be computed efficiently, especially with regard to liberties. For example, with the advent of MCTS, the method of pseudo-liberties[6] [Hou],

[5] The implementation is not merged in the master branch yet.

[6] Instead of the number of unoccupied points neighboring the group, pseudo-liberties represent the number of edges from the group to unoccupied neighboring points. This still allows to determine whether the group is captured and the count is much more efficient to maintain.

allowing Go rules to be implemented without maintaining explicit list of liberties, was popular — however, when implementing self-atari heuristics or semeai checks, having a list of real liberties is crucial and it does not make sense to optimize it away anymore.

Our data structure contains several $(n+1) \times (n+1)$ arrays; the arrays include a "border" of special color, allowing point neighbor iterators to skip explicit boundary checks. We maintain the following information for each intersection:

- Color of the intersection: none, black, white, board border.
- Group identifier for the current intersection. If no stone is at the intersection, this is zero. Group identifier is simply a position of the move that has founded the group.
- Position of the next stone in the group. Allows effortless iteration over all stones in a group.
- Counts of immediate neighbors of each color.
- Zobrist hash (Sec. 6.1) for the given intersection.
- 3×3 pattern code for the given intersection.
- "Traits" — cached simple tactical information (e.g. "it is safe to play here" or "number of neighbors we can capture").[7]

Furthermore, we maintain the following general data structures:

- Per group information containing details on liberties. We do not maintain the list of all liberties, but use a fixed-size list of at most $m = 10$ liberties; the list is refilled if it becomes shorter than $m/2 = 5$ and the group has more liberties. This allows rich heuristics based on low-liberty tactical situations and fast atari/capture handling without too much list maintenance overhead for large unencumbered groups.[8]
- Queue of capturable groups.
- Queue of free intersections. This allows fast and unbiased random move selection.
- List of the k last moves and superko hash table.

Pachi's implementation of the Go board supports a symmetry folding reduction in the game beginning as moves may be isomorphic by rotation or flipping. In the initial position, all moves are isomorphic to moves in a triangle marking just 1/8 of the board. If a Go engine can reduce the set of moves based on this symmetry information, move search in the opening is much more efficient.

However, it is not possible to easily distinguish atari situations or get a list of all liberties without duplicates.

[7] This functionality is currently turned off as it is not worth the extra performance hit with the current set of used heuristics. We plan to enable it again in the future.

[8] We were inspired by GNUGo in our liberty handling.

Pachi's board implementation follows the Chinese rules, except that it does not prohibit multi-stone suicide[9] [10] for performance reasons. Game engines usually allow suicide in playouts and prohibit it just in the game tree. The UCT engine also supports an approximate emulation of the Japanese rules.

3.2 Play Testing

Two complementary approaches are used when testing Go programs. The standard way of unit testing is supported in Pachi, although the set of tested features is very small and Pachi needs to be improved in this regard. But aside of that, it is important to watch and tune the overall playing strength of the program by "play testing" and observing its performance against other players, computer or real-world. Programs usually regularly play over the internet, either with humans or other programs. They may acquire a rating or rank that can serve as a crude measure of general strength.

But another common way of play testing is by choosing a reference opponent (or set of reference opponents)[11] and testing several variants of the program against the opponent by automatically playing hundreds or thousands of games. Observing and comparing their winning rates is then the usual way of quantifying new improvements and tuning the constants (stochastic rule probabilities, equation coefficients, etc.).

We use this kind of play testing extensively and developed a custom tool for this task as described below. It is the most reliable way of assessing improvements and the results shown in the rest of this work have been obtained this way. Unfortunately, the win rate differences gained by individual improvements tend to be fairly small in practice and therefore obtaining statistically meaningful results can be extremely demanding.[12]

Our testing methodology evolved over time. First, we focused on testing on 9×9 boards since the games are much shorter; however, there is much less demand for efficient information sharing methods on smaller boards and the general strategy and game-play is not that similar to large boards. On 19×19 boards, we originally tested with 220 seconds per game (no overtime, single thread); however, the tree search has very little time to grow the tree in this scenario, which means that changes quickly improving initial convergence will be overrated while changes that affect long-term search behavior will get underrated — and we have experienced that effect of many changes strongly depends on time per move. Lately, we have

[9] Originally, it was intended to follow the Tromp-Taylor rules, but these are not used widely for actual play and differ in their superko handling.

[10] Real-world rulesets may define complex protocols for ending the game and agreeing on stones that are dead or alive. Pachi, and probably no other computer program either, does not support these rules since in practice they are rarely used even in serious tournament settings.

[11] Another common practice is "self-playing", i.e. testing the improved version against an older version of the same program. We usually avoid this since we have observed the results to be frequently overamplified or otherwise distorted.

[12] Often, several thousands of games are necessary to distinguish performance of two variants with acceptable (95%) confidence. If we assume a single game takes 10 minutes on average, we can play 144 games per day; on 8-thread machine, we could play 1152 games per day. With a handful of candidate versions, this means a latency of several days even when using several computers in parallel and non-stop. And still, on average 1 in 20 measurements will be off.

moved to testing with 500 seconds per game (no overtime, single thread) as the current compromise between computing power available and getting results at least just in the order of days.[13] Overall, we gradually adopt testing settings that are slower but mirror real-world playing conditions more closely.

We are aware that these measurements still may not perfectly reflect behavior with longer thinking times and many-threaded search, unfortunately we just have to rely on this approximation for now, also noting that almost all other publications related to Computer Go show measurements obtained under similar conditions. This is also the reason why not all measurements presented here use the same time settings or base Pachi version — it is simply not practical to repeat all earlier measurements due to the extreme time and computational demands.

We usually perform play-testing against GNUGo v3.7.12 level 10 [BFB+]. GNUGo is a classical program that does not directly maximize winning probability like MCTS, therefore games against it can be more diverse and similar to games against humans. It is a popular benchmark for Computer Go programs performance due to its availability and speed, even though it is by far not as strong as the top programs. Level 1 is the weakest practical setting while level 10 is the strongest difficulty setting before testing becomes too slow.

3.2.1 Testing Infrastructure "Autotest"

Pachi features a unique and flexible infrastructure for play-testing, **autotest.** The infrastructure is based on several non-trivial POSIX shell scripts that take advantage of concurrent execution and shared filesystem, not requiring any other means of synchronization or a central dispatching service.

The user can declare players (as program invocations) and pairings to benchmark; independently, she starts as many autotest client instances as desired. In each iteration, the client reloads the configuration file, picks a random pairing, uses the **twogtp** tool[14] to play out the match and records the result (also archiving the game played). Each client has a unique identifier it uses as directory name for storing results and games.

When the user wants to explore different pairings or new player version, updating the configuration file is enough and the clients will pick up the new pairings as soon as their current matches are finished. To get current results, a dedicated script inspects directories of all clients and produces summarized results.

In our setup, we share the autotest working directory over NFS to the workstations and computing servers of a university department; when the machines are idle (at most nights and on weekends), the autotest clients are spawned on the machines (the number of instances per machine corresponds to the number of cores). One of the pairings is always a "reference" (previous version of the program) and performance of other pairings is compared against this pairing to account for any winrate changes due to different combinations of hardware on the particular machines where clients run at the time, etc.

[13] Thanks to Jean-loup Gailly, the default Pachi settings were tuned to behave well with realistic number of simulations too.

[14] TwoGtp is a tool that runs two programms and connects their GTP streams so that they play a game together. Multiple implementations exist, we use the version distributed with GoGui [Enz].

Table 3.1: Performance with Various Exploration Coefficients against GNU Go level 10, 500s/game, 19×19 with no komi and GNUGo black

UCB coefficient	Win Rate
$c = 0.2$	$22.5\% \pm 2.5$
$c = 0.05$	$49.4\% \pm 1.9$
$c = 0$	$57.2\% \pm 1.9$

The highlights of autotest include trivial configuration, simplicity of setup and high degree of automation. Much work remains to be done, though: implementation of some automated tuning algorithm, more elaborate automated client spawning and performance improvements.

3.3 Tree Search Policy

We use the RAVE tree policy basically as described in Sec. 2.3.3, using $\text{sims}_{\text{EQUIV}} = 2500$ (found by parameter tuning). The $sims_{RAVE}$ and $wins_{RAVE}$ terms include the criticality value as detailed in Sec. 5.2. The $sims$ and $wins$ terms include the prior values (Sec. 3.3.2) for progressive bias:

$$sims = sims_{Prior} + sims_{Reg}$$

$$wins = wins_{Reg} \cdot \frac{sims_{Reg}}{sims} + wins_{Prior} \cdot \frac{sims_{Prior}}{sims}$$

Our choice of abandoning the UCB exploration term altogether is not unique, but not too common either. Table 3.1 shows the rationale for this choice. We speculate that this is allowed by our quite rich set of heuristics that can guide exploration alone by the RAVE term. However, this also currently limits our further strength growth as explained in Sec. 2.4.2.

We expand leaf nodes when they have been visited $n = 8$ times. For simplicity, propagated and stored values are always from black's perspective; in case of black win, value around 1 is propagated, while in case of white win, value around 0 is propagated. However, by **value scaling**,[15] small portion of the value $\text{Scale} = 0.04$ is reserved for linearly rescaled representation of score difference up to $\text{Range} = 40$, designed to nudge the tree to prefer branches achieving wins by largest amount possible:

Algorithm 2 WinValue

Require: Score is the final territory difference; negative in case of black loss.
Require: Result is $\{0, 1\}$; zero in case of black loss.
$\text{ScaledScore} \leftarrow \text{Scale} \cdot \max\{\min\{\frac{\text{Score}}{\text{Range}}, 1\}, -1\}$
$\text{WinValue} \leftarrow \text{Result} \cdot (1 - 2 \cdot \text{Scale}) + \text{Scale} + \text{ScaledScore}$

[15] Originally introduced in Fuego [Enz+10].

We represent the tree explicitly using node structures linked with pointers: to the parent, the first child and the next sibling in a linked list of children.[16] Each node structure contains separate statistics for regular winrate, prior winrate, AMAF winrate and ownership data (for criticality). The statistics are maintained as $(value, sims)$ pairs (where $value = wins/sims$). The safe lock-free value update scheme is inspired by Fuego [EM10].

When a move is played and the opponent replies, the appropriate sub-tree of the original tree is reused. We also support "pondering" — running a search on the background while waiting for the opponent's move. If the upper tree memory limit is hit, the search is stopped; we currently do not support pruning the tree during search and it is crucial that search does not continue when nodes are not expanded anymore — it soon starts exhibiting the same issues as the single-level Monte Carlo search (Sec. 2.1).

3.3.1 Opening Book

Building a large and effective opening book has not been Pachi's focus so far since we do not optimize much for the 9×9 board and we have not regarded opening book as particularly important on 19×19 before. We are planning to revisit this topic in the near future. Nevertheless, Pachi supports two opening book approaches:

- **Forced book** is stored in a Fuego-compatible format and contains, for each board size, a set of positions (defined by sequence of moves) and a single follow up move for each position. If a position included in the book is reached, the move is played unconditionally. No such book is included with Pachi by default, but the book distributed with Fuego may be used as-is.[17]

- **Tree book** is basically a serialized form of the game tree to use within the UCT engine in the game beginning. This approach is more flexible; there is some space for stochastic behavior and larger set of candidate moves may be considered in each position. The tree book file may be generated by previous Pachi search, so it is possible to use this simply as a cache of the search results for the initial position. However, the script included with Pachi to generate this book is very crude and it is very easy to get out of the book if opponent replies were not completely expected by Pachi.

Our usage of opening books has not been particularly effective yet. However, Pachi has the requisite facilities to use a book and now only a way to build a good opening book needs to be devised and implemented.

3.3.2 Prior Values

After $n = 8$ visits to a leaf tree node, the node is expanded and nodes for all possible followup moves are created. In order to kick-start exploration within the node meaningfully using progressive bias (Sec. 2.3.2), the winrate value for

[16] We are aware that our approach is somewhat inefficient here; all siblings could reside in a single array, and ideally transposition tables would be used instead of coordinate sequences.

[17] The `book/book.dat` in Fuego source tree.

Table 3.2: Performance of Various Prior Value Heuristics vs. GNU Go level 10, 500s/game, 19×19 with no komi and GNUGo black

Heuristic	Win Rate
Full combination	$58.5\% \pm 2.2\%$
w/o 19×19 lines	$56.3\% \pm 2.4\%$
w/o CFG distance	$49.1\% \pm 2.3\%$
w/o eye malus	$54.1\% \pm 2.3\%$
w/o ko prior	$56.4\% \pm 2.3\%$
w/o playout policy	$26.8\% \pm 2.8\%$

each new node is seeded based on various heuristics. In general, each heuristics contributes given number of virtual simulations[18] ("equivalent experience") all with the same result (win or loss). The sum of the heuristic contributions is represented by the $wins_{Prior}$ and $sims_{Prior}$ variables above. Performance of the heuristics is shown in Table 3.2. Most heuristics we use are inspired by the original paper on Mogo [Cha+10].

- First, we apply the "even game" heuristic; unlike other heuristics, it does not add a win or loss, but the value of 0.5. The heuristic is important to stabilize reading when RAVE is used; if turned off, RAVE tends to get overly biased by initial results and other prior values, unwilling to explore nodes outside of local maxima.[19]

- Next, we apply the "eye" heuristic, adding lost games to all moves that play within single-point true eyes of own groups. Such moves are generally useless and not worth considering at all; the only exception is the completion of the "bulky five" shape by filling corner eye, this situation is rare but possible, so we only discourage the move using prior values instead of pruning it completely.

- We attempt to encourage proper handling of ko fights (see Sec. 1.1) by encouraging exploration (adding wins) of a move that re-takes a ko no more than 10 moves old.

- We try to encourage sane 19×19 play in the opening by giving a malus to first-line moves and bonus to third-line moves if no stones are in the vicinity.

- We encourage exploring local sequences by providing won simulations bonus to moves that are topologically close to the last move; we use a metric based

[18] Small boards require less virtual simulations than large boards; for most heuristics, we use 14 simulations for 9×9 and 20 simulations for 19×19. Some heuristics are contributing half or twice this amount.

[19] We did not even include this heuristic in the performance table since the RAVE tree search simply does not work without it at all.

on the shortest path in **Common Fate Graph**.[20] This has two motivations — first, with multiple interesting sequences available on the board, it is required to ensure the tree does not swap between situations randomly but instead reads each sequence properly; this is of most importance on large boards. Second, it is well rooted in the traditional Go strategy where large portion of moves is indeed "sente", requiring a local reply. We provide bonus decaying with the distance, up to a distance of 3.

Additionally, we match board quadrants within a joseki dictionary and give additional priors according to suggestions by the playout policy (see Sec. 3.4). There is also some support for node prior evaluations based on external plugins, but this has not been used in practice yet.

3.3.3 Time Control

The tree search can be limited either by a fixed number of playouts, by fixed time per move, or by specific amount of time for the whole game (possibly further split to overtime periods etc.). In close cooperation with Jean-loup Gailly, we have developed a flexible thinking time allocation strategy for the latter case with the goal of spending the most time in the most critical parts of the game — in the middle game (see Sec. 1.3) and particularly in situations where the best move is unclear.

The main idea of the strategy is that it assigns two time limits for the next move — the *desired* time t_d and *maximum* time t_m. Normally, only t_d time is spent during the search (minus a fixed amount designated to compensate for network lag and tree management overhead), but this time may be extended up to t_m in case the results provided by the tree are too unclear (and this triggers very often in practice).

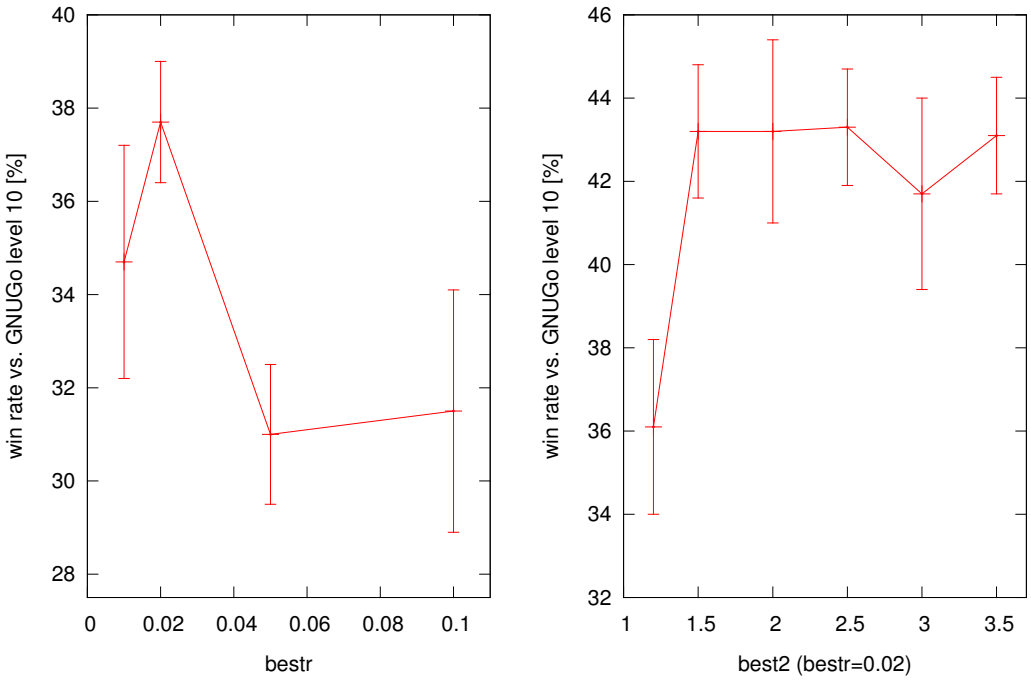
Given the main time T and estimated number of remaining moves in the game[21] R, the default allocation is $t_d = T/R$ and $t_m = 2t_d$; this allocation is recomputed on each move so that we account for time overspending. Furthermore, we tweak this allocation based on the number of moves since game beginning so that we spend most time per move in the middle game.

For *overtime (byoyomi)*, we use our *generalized overtime* specification: after the main time elapses, fixed-length overtime T_o for each next m moves is allocated, with n overtime periods available. I.e., a chunk of m moves must be played within the given time limit; otherwise, the period counter is decreased and when it hits zero, the game is forfeit on time exceeding. *Japanese byoyomi* is a specific case with $m = 1$ while *Canadian byoyomi* can be expressed with $n = 1$ fixed.

If overtime is used, the main time is still allocated as usual, except that $t_m = 3t_d$ and the main time t_d is at least the byoyomi t_d. In overtime, the first $n - 1$ overtime periods are spent as if they were part of the main time, then the

[20] Moves near liberties of the same group, though possibly far apart on the board, should be considered nearby. We consider a graph corresponding to the Go board grid with edges between stones of the same color (i.e. within a single group) having weight 0 and the rest of edges weighted at 1. Common Fate Graphs were introduced in [Gra+01].

[21] We base this on an expectation that on average, 25% of board points will remain unoccupied in the final position. We always assume at least 30 more moves will be required.



(a) Best — best child delta $best_r$ (b) Best — second best ratio $best_2$

Figure 3.3: Performance of various time setting parameters.

time per move is allocated as $t_d = T_o/m$ and $t_m = 1.1 \cdot t_d$. Of course, t_m is upper-bounded by the remaining period length.

The tree search continues even after t_d time already elapsed if the current state of the tree is suspicious — the expectations of the two best children are too similar ($best_2$ ratio), the expectations of the best children and its best followup are too different ($best_r$ difference), or the root child chosen based on most simulations is not the move with the highest expectation. As mentioned above, the concrete thresholds are tuned so that these conditions trigger except in the most clear-cut situations. On the other hand, the search is terminated earlier than t_d if the best move cannot change anymore without regard to results of all future simulations[22] or the chosen move expectation is over a given threshold and a sufficient number of simulations has been performed to confirm that.

Fig. 3.3 shows that using flexible time strategy may result in a significant performance increase compared to simply allocating fixed number of simulations or time slice per move. Overall, performance of Pachi with 300s per move would be $32.2\% \pm 1.1\%$ with no prolonged reading conditions vs. $44.1\% \pm 1.4\%$ with optimal settings.

3.3.4 Parallelization

Historically, various parallelization approaches for MCTS have been explored. **Root parallelization** and **leaf parallelization** are the simplest approaches [CWH08]. Root parallelization runs an independent search on a separate tree in

[22] This check is suppressed in overtime with $m = 1$ and $n = 1$; we gain nothing by stopping earlier and we may reuse parts of the tree in the next move.

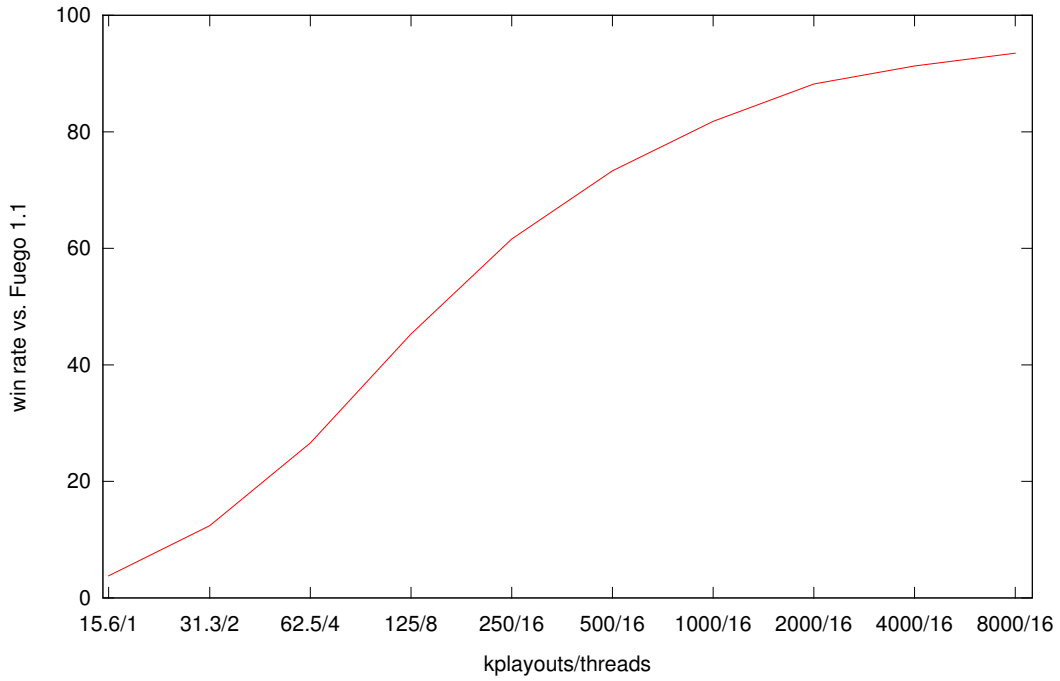


Figure 3.4: Performance of parallelization and various time settings vs. Fuego 1.1, 550kplayouts in 16 threads, 19×19 with komi -22.5 (Fuego as black).

each thread; when the search stops, the trees are merged and a winner is chosen from the final tree, i.e. the threads effectively give a weighted vote on the winner. On the contrary, leaf parallelization uses only a single thread for the search and then executes multiple simulations in parallel.

Root parallelization is not giving very good results since no information exchange usually happens and all parallel searches are likely to proceed in the same general direction; for leaf parallelization, tree search is the bottleneck and it turns out that playing multiple games at once of course provides a more accurate sample, but in fact does not improve MCTS effectivity too much.

The third obvious approach is **in-tree parallelization**, with multiple threads performing both searches and simulations in parallel on a shared tree. Furthermore, [CWH08] introduced the concept of *virtual loss* — to spread the parallel tree descents, a virtual lost simulation is added to each node visited during the descent and removed again in the update phase. Moreover, [Enz+10] has shown that no locking of the tree is needed and small inaccuracies resulting from non-transactional sequence of atomic operations are tolerable.

Lockless in-tree parallelization is the approach used in Pachi. Fig. 3.4[23] demonstrates both the effectivity of parallelization and also scaling when number of simulations is simply doubled.

Cluster parallelization of Go is still not clearly solved. Pachi has elaborate support for distributed computations with much information exchange between the nodes thanks to the work of Jean-loup Gailly, but even so it scales much slower when multiplying number of nodes than in the case of processors with single low-latency shared tree.

[23] Data courtesy of Jean-loup Gailly.

Table 3.3: Performance of Various Playout Heuristics vs. GNU Go level 10, 500s/game, 19×19 with no komi and GNUGo black

Heuristic	Win Rate
Full combination	$59.5\% \pm 2.3\%$
w/o Capture	$5.4\% \pm 2.1\%$
w/o 2-lib.	$47.2\% \pm 2.4\%$
w/o 3× pats.	$18.5\% \pm 1.7\%$
w/o self-atari	$54.7\% \pm 2.3\%$

3.4 Simulation (Playout) Policy

As already mentioned, Pachi uses a rule-based simulation policy originally inspired by the Mogo algorithm [Gel+06]. Heuristics are tried in a fixed order, each is applied with certain probability (in most cases, the probability is around 0.8 for large boards, 0.9 for small boards). A heuristic returns set of suggested moves; if the set is non-empty, a random move from the set is picked and played. If the set is empty (the common case), next heuristic is tried. If no heuristic matches, a uniformly random move is chosen. See Table 3.3 for performance impact of the major heuristics.

First, with $p = 0.2$, ko is re-captured if the opponent played a ko in the last 4 moves.

Then, local checks are performed — heuristics applied around the last move (sometimes also the second-to-last). They are restricted in this way partially for performance reasons, but it has been also shown [Gel+06] that a preference for playing local continuous sequences of moves is important for good performance:

- *Nakade* is a common technique of killing a group — a move is played inside the eyeshape to prevent the opponent from splitting it into two eyes. With $p = 0.2$, we check if the liberties of the last move group form the appropriate shape.

- If the last move has put its own group in atari, we capture it; if it has put a group of ours in atari, we attempt to escape (or counter-capture a neighboring group in atari). We are careful not to continue escaping a ladder that works against us.

- If the last move has reduced its own group to just two liberties, we put it in atari, trying to prefer atari with low probability of escape. If the opponent has reduced our group to two liberties, we attempt to either escape or put some neighboring opponent group in atari (thus attempting to handle the simplest forms of semeai).

- With $p = 0.2$, we attempt to do a simplified version of above for groups of three and four liberties.

- Points neighboring the last two moves are matched for 3×3 board patterns centered at these points. The patterns represent various common

shapes like "hane", cut and "magari". They are similar to patterns presented in [Gel+06], though few patterns have been replaced and information on "in atari" status of stones is also used.

Additionally, we match board quadrants within a joseki dictionary built from the Kogo Joseki Dictionary [Odo+] (we use the branches marked as "good variations").

Suggestions of the simulation policy are also used to assign prior values to new tree nodes. The same set of heuristics as listed above is used, but all heuristics are always applied (though some contribute reduced or increased number of virtual simulations) and local checks are performed globally. It may be of special note that we dynamically adjust the number of won virtual simulations for group captures based on the group size:

$$sims \cdot \frac{\max\{\min\{stones, 15\}, 2\}}{2}$$

3.4.1 The Self-atari Problem

Bad self-atari moves are pruned from heuristic choices and stochastically also from the final random move suggestions: in the latter case, if the other liberty of a group that is being put in self-atari is safe to play, it is chosen instead, helping to resolve some tactical situations involving shortage of liberties and false eyes.

The bad self-atari detection is quite complex process — not just because the move must not be actually played on the board yet but also since pruning legitimate self-atari moves will lead to many missed essential tactical moves. For example, when capturing a group with single large eye, it is required to fill the eye with own stones, eventually putting the inner group in atari; sometimes, stones need to be "thrown in" group connections along the edge to prevent formation of eyes, etc. On the other hand, the self-atari detector will also detect some dangerous moves extending three-liberty groups.

3.5 Performance

We have shown the performance effect of various aspects of Pachi in the text above, but we should also put the overall performance of Pachi in context of other Computer Go software.

The primary venue where Pachi plays open games with the members of public is the KGS internet Go server [Sch]. Petr Baudiš maintains a Pachi instance running with 8 threads on Intel i7 920 with hyperthreading enabled and 6 GiB of RAM available[24] that plays as users `Pachi`, `PachIV` and `PachIW`. These instances can hold a solid 1-dan rank[25] as shown in Fig. 3.5.[26]

[24] Until few months ago, only 4 GiB of RAM were available.

[25] The rank is computed automatically using a server-wide rating algorithm. Only handicap games up to six stones in either direction are allowed. The rank is based only on performance on the 19×19 board. On the 9×9 board, Pachi distributed over just a few machines has been tested to be of high dan strength [Bau10].

[26] This user has been used for playing only intermittently, therefore strength has been in fact increasing more linearly than apparent from here.

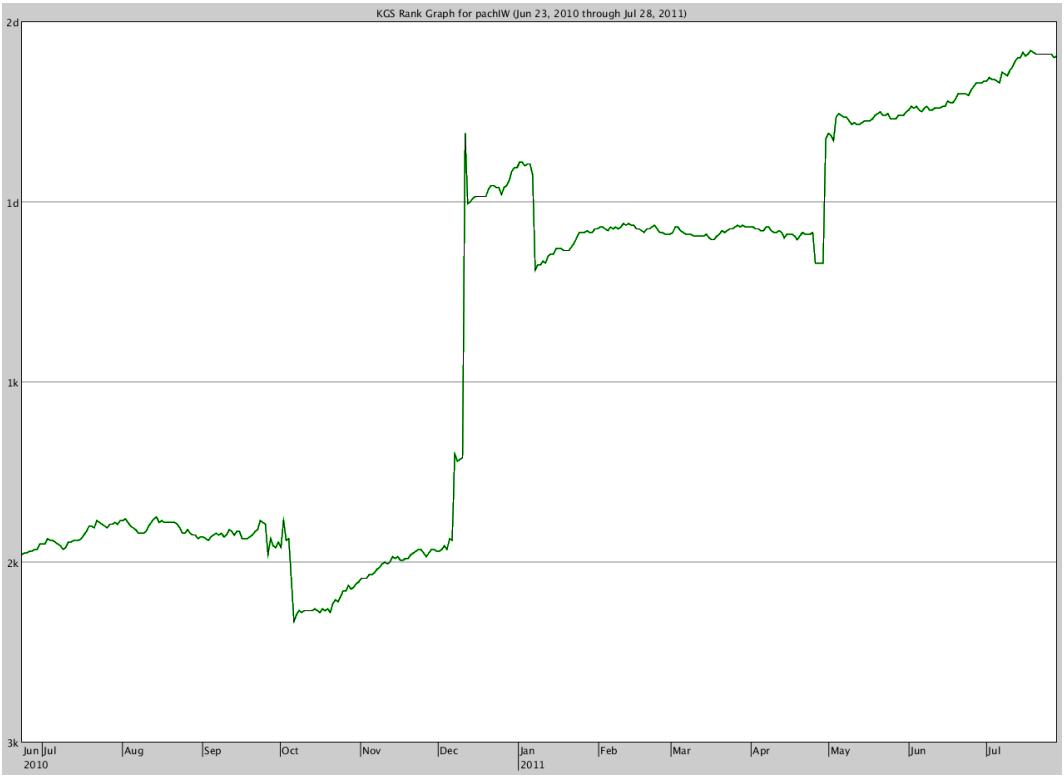


Figure 3.5: User `PachIW`

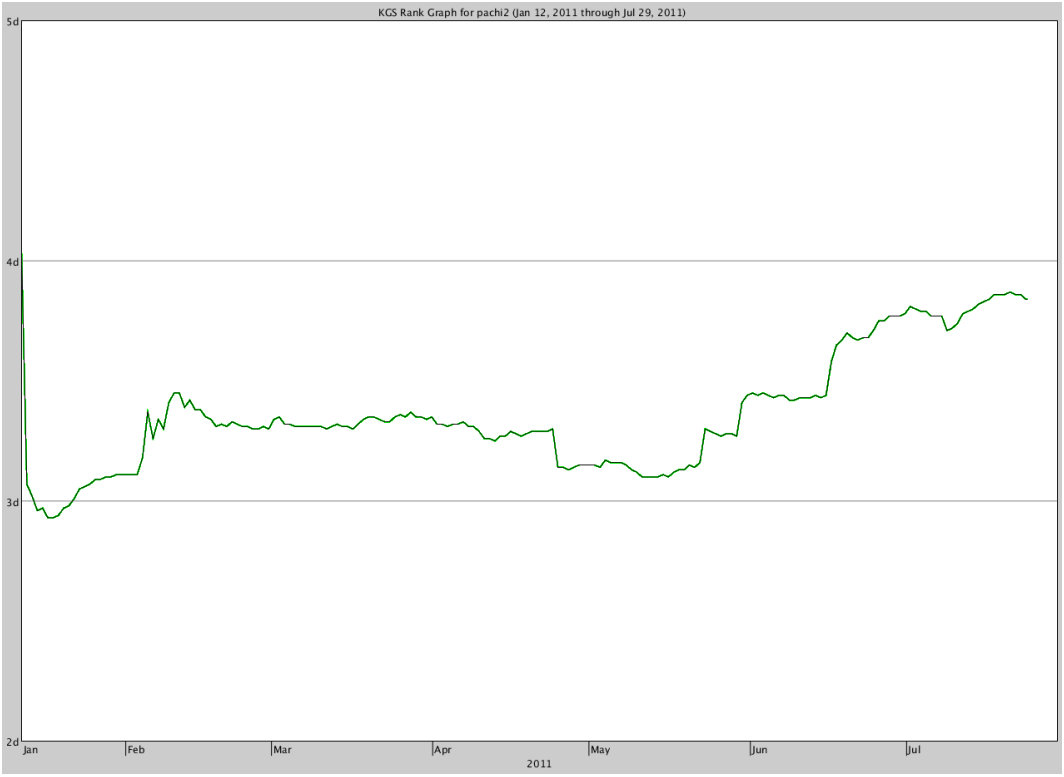


Figure 3.6: User `Pachi2`

Jean-loup Gailly runs KGS Pachi instances `Pachi2` and `Pachi30s` based on the distributed game engine. They use a cluster of 32 or 64 machines, each running 20 or 22 threads. Pachi2 (the higher rated of the two, using faster time settings) is ranked as 3-dan (Fig. 3.6).

Pachi (usually the distributed version) regularly participates in the monthly KGS tournaments [Wed05]. It is able to occassionally defeat even the top Computer Go program Zen and usually finishes on the second or third place. It tied with Many Faces of Go for the first place in the First 2011 Slow KGS Computer Go Tournament.

Measurements show that with both programs running 16-threaded on the same hardware configuration and with equal time settings, Pachi has 50% win chance against the next best open source MCTS program Fuego when giving it komi equivalent to 3.6 handicap stones. The main testing opponent we use, GNU Go on level 10, can be overcame with 50% time when Pachi plays with 500 seconds for the whole game (GNU Go with fixed level will not adjust its thinking time by the amount of time available) and the probability rises rapidly with more time available. On the other hand, some closed source programs are still stronger than Pachi: Many Faces is one rank higher than the single-machine Pachi on KGS, while single-machine Erica is about as strong as Pachi using a cluster and instances of the CrazyStone and Zen programs are even stronger. Overall, distributed Pachi is perhaps the fourth strongest program regularly playing on KGS while single-machine Pachi is the sixth strongest.

Pachi has not participated in real world tournaments and events much so far. The distributed Pachi participated in the Human vs. Computer Go Competition, SSCI 2011 Symposium Series on Computational Intelligence in Paris. It played three games against professional players on 19×19 and won a 7-handicap match against Zhou Junxun 9-dan professional [Tai11] (commonly accepted as one of the top current professional players). The game is shown in Fig. 3.7. It evolved to a complicated struggle for survival of invading white group; Zhou Junxun commented that Pachi played on professional level when killing it. In the European Go Congress 2011 Computer Go tournament [Eur11], distributed Pachi tied with Zen for the first place in the 19×19 section.

Pachi has some clear weaknesses. It frequently maintains even game or gains upper hand against higher ranked programs in the beginning, but is prone to misevaluating many positions (especially life and death problems) due to its aggressive use of many heuristics and relying purely on RAVE exploration (however, we have found these settings to be at least a local optimum, so simply encouraging more exploration is not a remedy). Pachi is perhaps the strongest program using rule-based playouts which might contribute to this issue.

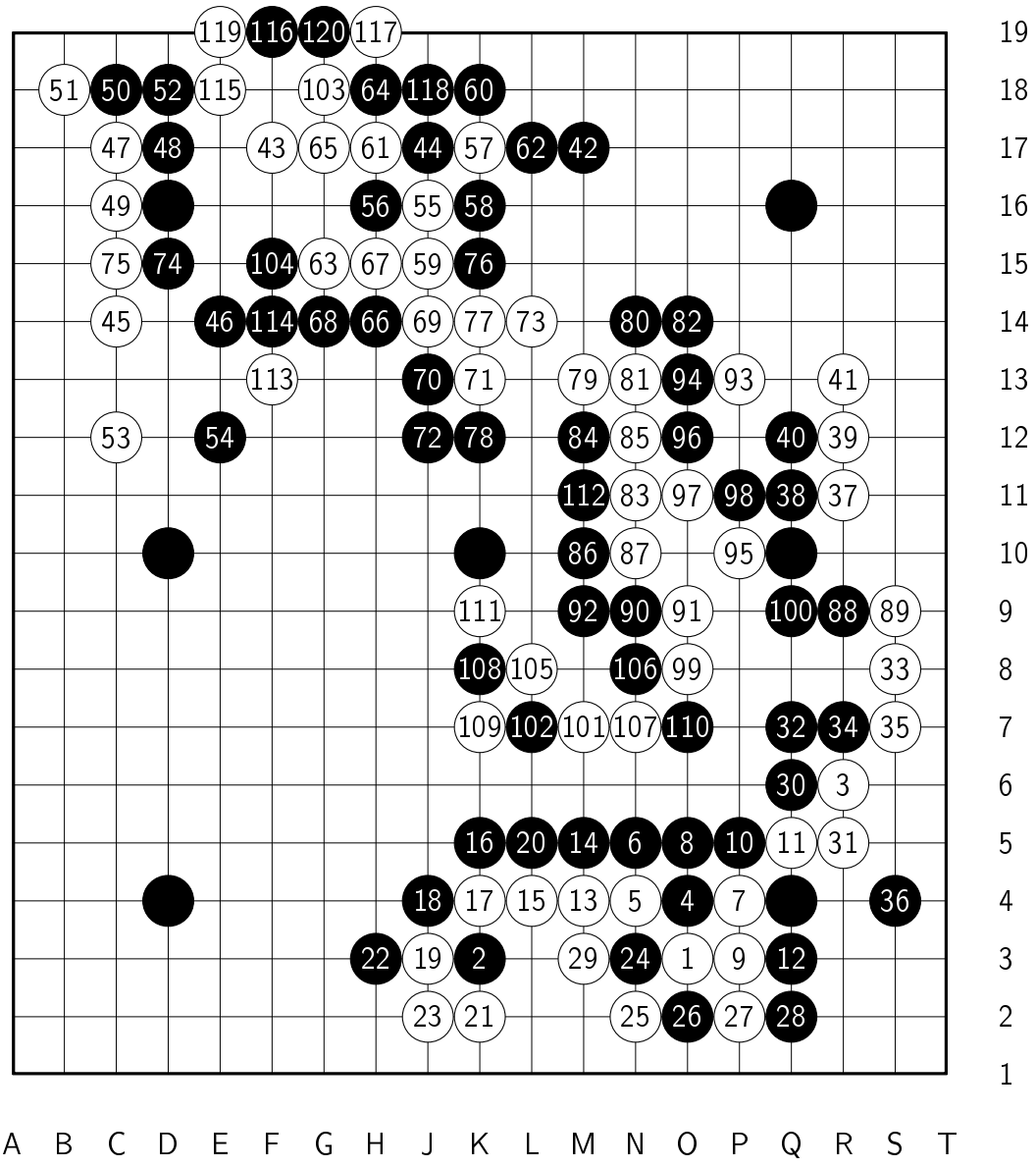


Figure 3.7: 7-handicap game between distributed Pachi (black) and Zhou Junxun 9-dan professional at SSCI 2011. White resigns after 120 moves.

4. Dynamic Komi

The Monte Carlo Tree Search in the game of Go tends to produce unstable and unreasonable results when used in situations of extreme advantage or disadvantage, due to poor move selection because of low signal-to-noise ratio; notably, this occurs when playing in high handicap games, burdening the computer with further disadvantage against the strong human opponent.

Here, we explore and compare multiple approaches to mitigate this problem by artificially evening out the game based on modification of the final game score by variable amount of points ("dynamic komi") before storing the result in the game tree.

4.1 The Extreme Situation Problem

One common problem of the Monte Carlo based methods is that by definition, they do not adapt well to extreme situations, i.e. when faced with extreme advantage or extreme disadvantage.[1] This is due to the fact that MCTS considers and maximizes winning expectation, not the score margin. If a game position is almost won, the "safe–active" move that pushes the score margin forward will have only slightly higher expected win expectation than a move with essentially no effect, since so many games are already implicitly won due to random noise in the simulations.

4.1.1 Handicap Games

A perfect example of such extreme situations occuring regularly are handicap games. When two players of different strength play together, a handicap is determined (as a function of the rank difference of the players). The handicap consists of the weaker player (always taking black color) placing a given number of stones on the board before the stronger player (white) gets to play her first move (Fig. 4.1). Usually, the handicap amount ranges between one stone[2] and nine stones. Thus, when playing against a beginner, the program can find itself choosing a move to play on board with nine black stones already placed on strategic points. Similarly, the program can begin with many stones and almost all simulations being won at the beginning of e.g. an exhibition game against a professional player.[3]

In practice, if a strong human player is faced with extreme disadvantage in a handicap game, they tend to play patiently and wait for opponent mistakes to catch up gradually; in even games, they usually try to set up difficult-to-read complications. An MCTS program however will seek the move that currently maximizes the expected win probability — ideally, it would represent a sophisticated trap, but in reality it tends to be rather the move the random simulations

[1] See Figures 4.3a and 4.3b for comparison with a classical program performance.

[2] The game begins as usual since black goes first in even game as well, but white will not receive any compensation for black playing the first move.

[3] This is especially troublesome since these games are high-profile and the general software level in Computer Go tends to be judged in part by performance in these games.

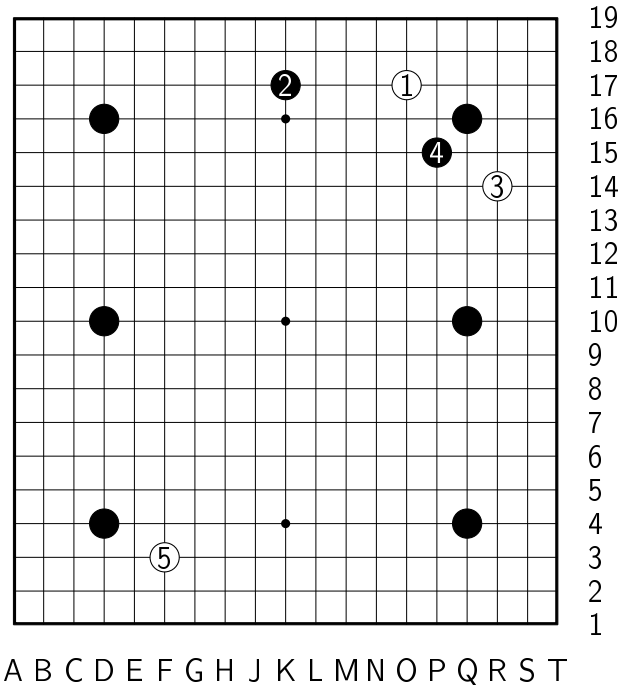


Figure 4.1: Example opening position from 6-handicap game (black `PachIW` vs. white `Lycoris`, Apr 29 2011, KGS).

mis-evaluate the most, ending up making trivially refutable moves.

Similarly, a strong human in position of large advantage will seek to solidify their position, defend the last remaining openings for attack and avoid complex sequences with unclear result; then, they will continue to enlarge their score margin if possible. MCTS program will make moves with minimal effect on the safety of its stones or territory and carelessly engage in sequences with high danger of mistakes; it will maximize the win expectation (of however biased and possibly misevaluating simulations) without regard to score margin,[4] creating the danger of losing the game.

4.2 The Dynamic Komi Technique

One possible way of tackling the described problem is using the "dynamic komi" technique. As mentioned in Sec. 1.1, komi is a point penalty imposed on one of the players; e.g. in even games, black has the right of the first move, but *gives white* 7.5 *komi*[5] as a compensation: at the game end, 7.5 points will be added to white score as a compensation for not moving first. (Conversely, komi value of -7.5 would be *black taking reverse* 7.5 *komi* and receiving a bonus of 7.5 points to her score at the game end; this would be a form of white handicapping herself. In handicap games, the usual komi — also used in our experiments — is 0.5, therefore ties are broken in favor of white but white gets no compensation for moving second, being the stronger player.)

The dynamic komi approach suggests that depending on the board situation, the program should adjust the internally used komi value to make the game more

[4] Up to value scaling.

[5] The exact komi value may vary; fractional value is used to avoid ties.

even — either giving the program virtual advantage in case it is losing in reality, or burdening it with virtual disadvantage when it is winning too much in the actual game.[6]

We have decided to implement and test several approaches to dynamic komi. Our test-case is performance in various handicap settings (since it is an obvious and well-defined example of the extreme situations described earlier) and also general performance in even games (where the extreme situations — with a chance of overtuning them — can occur naturally time by time).

In the following text and algorithmic descriptions, we will assume the black player's perspective — increasing the komi means giving extra advantage to the opponent, decreasing the komi corresponds to taking extra advantage on oneself. Real code needs to reverse the operations in case the computer is playing as white.

4.2.1 Prior Work

It has been long suggested especially by non-programmer players in the Computer Go community to use the dynamic komi approach to balance the pure winrate orientation of MCTS, however it has been met with scepticism from the program authors since it introduces artificial inaccuracy to the game tree and it was not clear how to deal with it in a rigorous way.

However, during our research we have become aware that some forms of dynamic komi are used in some programs. Many Faces of Go uses an algorithm analogous to our Linearly Decreasing Handicap Compensation with slightly different constants [Fota]. Hiroshi Yamashita has independently proposed an algorithm used in his program Aya that is similar to our Score-based Situational Compensation, along with some positive experimental results [Yam].

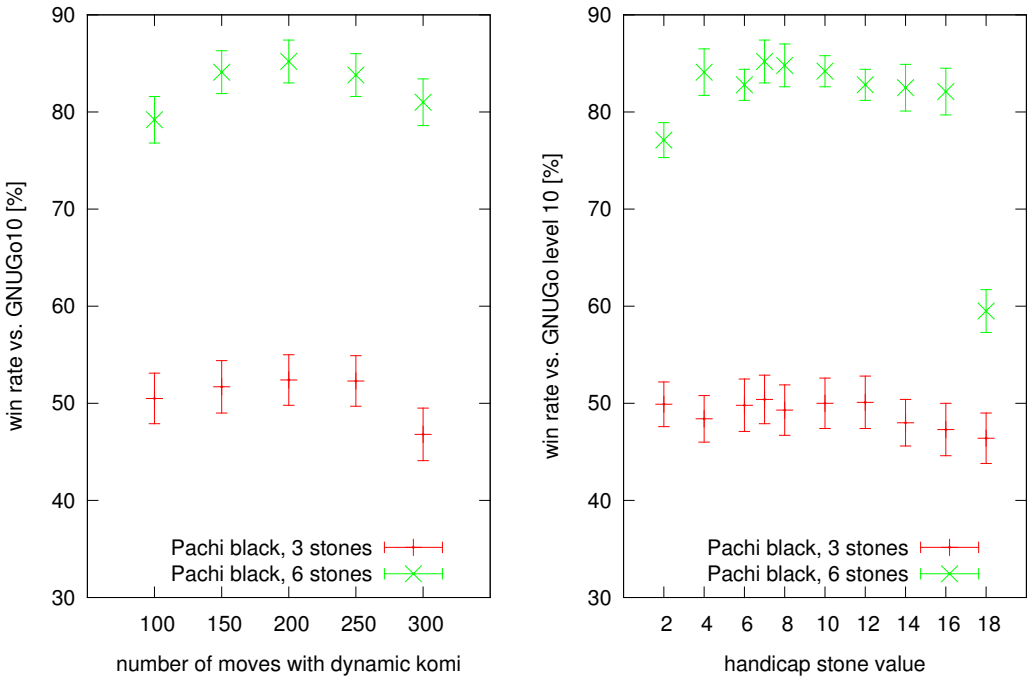
4.3 Linearly Decreasing Handicap Compensation

The simplest approach we used successfully was to simply special-case handicap games and try to stabilize the reading by imposing a komi on the program based on the number of handicap stones and linearly diminishing the komi throughout the game:

Algorithm 3 LINEARHANDICAP

Require: MoveNum is the number of the current move to play.
  if MoveNum $> M$ **then**
    Komi $\leftarrow 0$
    return
  end if
  KomiByHandicap $\leftarrow h \cdot$ NumHandi
  Komi $\leftarrow$ KomiByHandicap $\cdot (1 - \frac{\text{MoveNum}}{M})$

[6] The actual final score count is of course not affected by the dynamic komi and the opponent needs not even be aware of it; it is used only to change internal evaluation of the simulations within the program.



(a) $h = 7$ with various values of M (b) $M = 200$ with various values of h

Figure 4.2: Performance in handicap games.

M denotes the number of moves for which the dynamic komi effect should last; we have found the optimal value to be around 200 (see Fig. 4.2a). h is the point value of single handicap stone; values around 7 work best for us (Fig. 4.2b).[7]

We should note that we compute the exact komi value in the tree leaf where we start the simulation. That implies the nice property that even when reusing parts of the tree after generating a final move, each node has all games played with the same komi. Since our next models will have no such property, we have checked its effect; we were not able to measure a statistically significant performance decrease when applying the same komi value computed on tree root in all simulations. Therefore, we assume that it is safe not to preserve this property.

4.4 Situational Compensation

A more sophisticated way to adjust komi throughout the game is adapting to the currently perceived situation on the board. We are not limited to the case of handicap games, but determining, implementing and tuning the mechanism is more difficult.

The situational compensation can be decomposed to several aspects: how to judge the current situation, how often to re-adjust the komi, and how much to adjust the komi in various phases of the game.

[7] The value of one extra handicap stone is twice the value of the standard komi [Gai] so approximately 14 points. However, dynamic komi apparently works best when only a certain percentage of handicap stone value is added to the komi. At any rate, at least in the $(4, 14)$ range the effect of this constant is very small. The optimal value is not statistically clear, altough we have of course gathered more results during the development than just Fig. 4.2b.

4.4.1 Situation Measure

Score-based Situational Compensation

One way to assess the situation is by observing the *expected score* of the Monte Carlo simulations over a time period (e.g. the previous move) and adjust the komi so that $\mathbb{E}[\text{score}] - \Delta_{\text{komi}} \to 0$.

Algorithm 4 ScoreSituational

Require: $\mathbb{E}[\text{score}]$ is the average score over reasonable amount of simulations (including then-used dynamic komi).

```
if MoveNum < 20 then
  Komi ← LinearHandicap
  return
end if
BoardOccupiedRatio ← OccupiedIntersections / Intersections
GamePhase ← BoardOccupiedRatio + s
KomiRate ← (1 + exp(c · GamePhase))^−1
Komi ← Komi + KomiRate · E[score]
```

This way (assuming $c > 0$), at the game beginning we adjust the dynamic komi by measured average game result (except the first few moves where meaningless fluctuations are expected to be large), up until a certain point in the game when we dramatically reduce the amount of further komi changes. The phase parameter s determines the point in the game when the phase shift happens.

The best values we have found are $c = 20$ and $s = 0.75$, but they still perform worse than the approach presented below. We have also tried other KomiRate transformations, without much success.

Value-based Situational Compensation

The other way to assess the game situation and amplify winrate differences between candidate tree nodes is to look directly at the winrate values at the root of the tree and adjust the komi to put the values within a certain interval.

We will divide the winrate value to three ranges: *red* (losing), *yellow* (highest winrate resolution), and *green* (winning). We will call the upper bound of red zone **red**, the lower bound of green zone **green**. Our goal shall then be to dynamically adjust the komi to keep the winrate in the yellow zone, that is between **red** and **green**.

Furthermore, we need to avoid "flapping" around critical komi value especially in the endgame — when the true score of the game is well established (i.e., very near the game end), winrate will be high with komi n and much lower with komi $n + 1$. To alleviate this problem, we introduce a *ratchet* variable recording the lowest komi for which we reach red zone; we then never give this komi or more. (This applies only to giving extra komi to the opponent when our situation is favorable; in case of negative komi compensating for unfavorable situation, we are eager to flap into the red zone in order not to get fixed in a permanently lost position and keep trying to make the game at least even again.) Optionally, the ratchet can expire after several moves.

Algorithm 5 ValueSituational

Require: Value is the winning rate over reasonable amount of simulations (including then-used dynamic komi).

```
if MoveNum < 20 then
  Komi ← LinearHandicap
  Ratchet ← ∞
  return
end if
if Value < red then
  if Komi > 0 then
    Ratchet ← Komi
  end if
  Komi ← Komi − 1
else
  if Value > green ∧ Komi < Ratchet then
    Komi ← Komi + 1
  end if
end if
```

The optimal values we have found are **red** = 0.45, **green** = 0.5 — that is, locking oneself into a slightly disadvantageous position, allowing to give the opponent an advantage but never expiring the ratchet. This is curious: in our strategy, it seems best to allow giving extra komi (evening out a game we are winning) at the beginning, but if we at any point lose the advantage, we only allow taking extra komi (balancing a game we are losing) from then on.

Using this kind of situational compensation has an interesting consequence. Normally, the program's evaluation of the position can be determined by the winrate percentage, representing the program's estimate of how likely the color to play is to win the game. However, here the value is always kept roughly fixed and instead, the program gives a bound on the likely score margin in the given situation — the applied extra komi.[8]

4.4.2 Komi Adjustment

There is a question of how often to adjust the komi: we can either determine the dynamic komi for the whole next move based on information gathered throughout the whole previous move *(offline komi)*, or we divide the tree search for a single move into fixed-size time slices and re-adjust dynamic komi in each slice based on feedback from the previous slice *(online komi)*; all slices work on the same tree, so no simulations are needlessly lost.

We have found the latter to be a significantly better approach, allowing to quickly fine-tune the komi to an "optimal" value during the search. The downside is of course that values obtained with varying komi values are mixed within a single tree, however we have not found this too harmful.

[8] Note that the same tree policy, choosing the best value (expectation) available at the moment, is of course used.

Table 4.1: Dynamic Komi Performance — Even Games

Method	Opponent	Time per Game	Win Rate
Pachi NONE	GNUGo	3.6min	$27.3\% \pm 3.2\%$
Pachi SCORESIT	GNUGo	3.6min	$26.3\% \pm 2\%$
Pachi SCORESIT	Pachi NONE	3.6min	$46\% \pm 2\%$
Pachi SCORESIT	Pachi VALUESIT	3.6min	$43.4\% \pm 1.8\%$
Pachi VALUESIT	GNUGo	3.6min	$29\% \pm 2.6\%$
Pachi VALUESIT	Pachi NONE	3.6min	$54.2\% \pm 3.4\%$
Pachi VALUESIT	Pachi NONE	10min	$55.6\% \pm 2.2\%$
Pachi VALUESIT	Pachi NONE	20min	$59.4\% \pm 3.2\%$
Pachi VALUESIT	Pachi NONE	30min	$58.3\% \pm 3.4\%$

Another question is the size of single komi adjustment step in case of value-based dynamic komi: when using online komi, the finest adjustment amount of 1 point worked best for us. We expect that more complicated arrangement would have to be in place in case another method is used.

We have discovered that the situational dynamic komi methods are not stable at the game beginning, especially in handicap games. We have obtained small improvement by using the LINEARHANDICAP for the first n moves[9] and only then switching to the situational compensation.

The final question is how to limit the amount of favourable komi imposed on the player; surely, with extra 100 komi in favour, the board examination completely loses touch with reality — also, deciding when to resign may become complicated. We have found that 30 is the top useful value for favorable komi; moreover, we stop allowing negative komi altogether when we reach 95% of the game[10] in order to resign in cases when we cannot catch up anymore.

4.5 Performance Discussion

Our tests were performed on the 19×19 board.[11] We perform play-testing against GNUGo v3.7.12 level 10 and self-play testing.

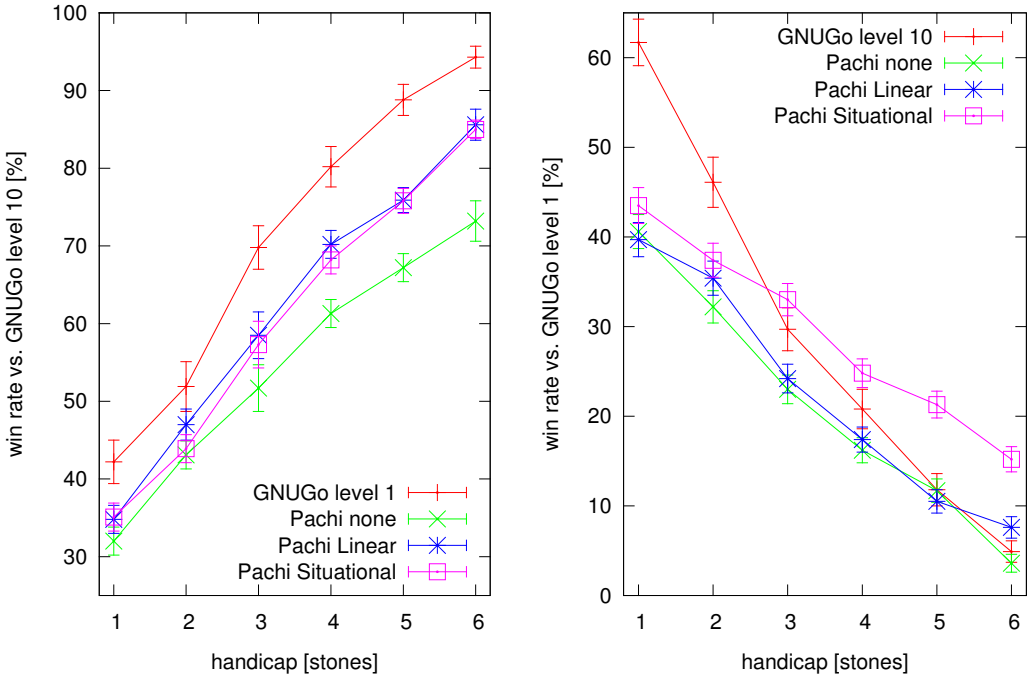
In table 4.1, we present our measurements of various dynamic komi methods in even games. Even though the improvement against GNUGo is not statistically significant in the presented table, based on many tests with various settings throughout the development we believe that there is a tangible small improvement. The improvement in self-play is much more pronounced and increases with alotted time.

In Figure 4.3a, we present measurements of dynamic komi effect in handicap games when taking black and varying amount of stones. We compare Pachi with

[9] We use $n = 20$ for 19×19, $n = 4$ for 9×9.

[10] Estimation based on board fill ratio.

[11] We have done some informal testing that indicates dynamic komi performing well on 19×19 does not deteriorate 9×9 performance.



(a) Playing black (b) Playing white

Figure 4.3: Dynamic komi in handicap games.

no dynamic komi, the linear komi and value-based situational compensation. The thinking time of Pachi is fixed on 3.6 minutes per game — this means very fast time controls, but allows to gather sufficient sample sizes. GNUGo Level 1 performance is included for further comparison; Level 1 achieves even better results than Pachi, but it is difficult to judge how much the usual self-play effects interfere with the performance, i.e. if MCTS performance in handicap games still has such a long way to go as it might seem here.

In Figure 4.3b we show measurements of dynamic komi effect in handicap games when taking white and giving varying amount of stones; the opponent is GNUGo Level 1 and GNUGo Level 10 is a reference. It is apparent that while linear komi is not effective in this case, value-based situational komi gives the expected improvement.

The results clearly show that dynamic komi is a significant playing performance improvement in handicap games. They also indicate that it can enhance program performance in even games. Value-based situational compensation fits the bill as a universal dynamic komi method, yielding an improvement for both handicap and even games. Alternatively, linear handicap compensation may be implemented very easily in any program to get just the shown major improvement in handicap game performance against stronger players.

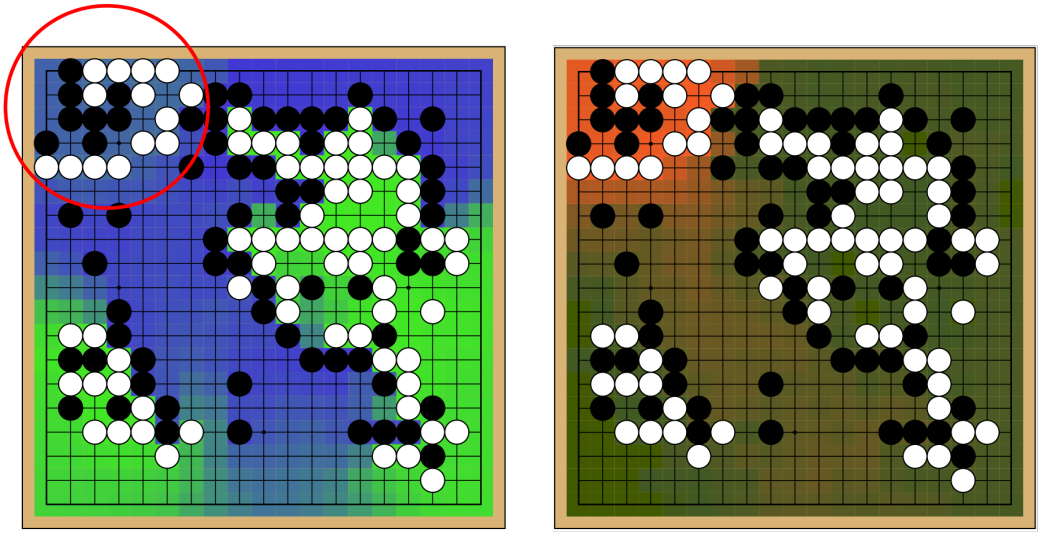
Our findings have a remarkable consequence — it appears that MCTS trees are *tinker resilient* to shifts in the evaluation function; if the evaluation function is adjusted to give a more even result in the middle of the search, the search tree can cope well and the end result is an improvement, even though the values in the tree have not been obtained consistently.

5. Criticality-based Biasing

We now proceed to a strategy that aims to focus tree search on "hot" parts of the board which are critical for winning the game, a statistics that we can gather quickly on the fly similarly to the AMAF information.

We call this the **point criticality.** While AMAF collects information on moves played after the current move and correlates them with winning the game, the criticality collects information on the ownership of points in the final position and correlates the ownership with winning the game — i.e., covariance of player owning the intersection at the game end and winning the game. (Fig. 5.1.)

We can then use criticality in a manner similar to the AMAF statistics to guide the exploration, or as a measure to use in a more high-level strategy for processing information on local positions.



(a) Ownership (blue: black territory, green: white territory)

(b) Criticality (brighter is higher)

Figure 5.1: A sample misevaluated position involving a semeai in the top left corner. White group is considered dead while it actually wins the semeai, but criticality of any moves in the vicinity is very high. (Reproduced from [Cou09] with permission.)

5.1 Criticality Measure

Let $C(x)$ be criticality for coordinate x; μ_{win} is expectation of simulation winner owning the coordinate, $\mu_{b(x)}$ and $\mu_{w(x)}$ are expectations of black and white owning the coordinate, respectively; μ_{b} and μ_{w} are expectations of black and white winning the simulation. Two concrete measures were proposed — [Cou09]:

$$C_{\text{Coulom}}(x) = \mu_{win(x)} - (\mu_{b(x)}\mu_{\text{b}} + \mu_{w(x)}\mu_{\text{w}})$$

and [Pel+09]:

$$C_{\text{Pell.}}(x) = \mu_{\text{b},b(x)}\mu_{\text{w},w(x)} - \mu_{\text{w},b(x)}\mu_{\text{b},w(x)}$$

Both formulas are equivalent to the classic covariance formula $Cov(X,Y) = \mathbb{E}XY - \mathbb{E}X\mathbb{E}Y$; we have found a rearranged version of the $C_{\text{Coulom}}(x)$ formula most efficient since it requires only a single multiplication and tracking of minimal amount of data in the tree:

$$C_{\text{Pachi}}(x) = \mu_{win(x)} - (2\mu_{b(x)}\mu_{\text{b}} - \mu_{b(x)} - \mu_{\text{b}} + 1)$$

5.2 Using Criticality

Our main contribution here is a method to efficiently use criticality in the RAVE equation. Originally, [Cou09] introduced criticality C_{Coulom} as a criterium for progressive widening (see Sec. 2.3.2), i.e. initially choosing only from tree nodes featuring highest criticality. He reported modest improvement in self-play experiments, but only minor strength increase against GNUGo. Independently, [Pel+09] proposed covariance measure $C_{\text{Pell.}}$ as an additional UCB1 urgency term:

$$\mu_i + c\sqrt{\frac{2\ln n}{T_i(n)}} + 2C_{\text{Pell.}}(i)$$

This was a good improvement to plain UCT, but did not improve MCTS that already used the RAVE policy.

Since Pachi uses progressive bias instead of progressive widening, we chose to also incorporate criticality in the tree policy formula. However, we see a similarity between AMAF statistics and the point ownership statistics for criticality, and therefore we decided to decrease the criticality weight of well explored nodes the same way we already handle AMAF expectation in the RAVE formula. Therefore, we incorporate criticality in the RAVE formula (Sec. 2.3.3):

$$sims_{Crit} = |C_{\text{Pachi}}(x)| \cdot sims_{AMAF}$$

$$wins_{Crit} = \begin{cases} C_{\text{Pachi}}(x) > 0 & 1 \\ C_{\text{Pachi}}(x) < 0 & 0 \end{cases}$$

$$sims_{RAVE} = sims_{AMAF} + sims_{Crit}$$

$$wins_{RAVE} = \frac{sims_{AMAF}}{sims_{RAVE}} wins_{AMAF} + \frac{sims_{Crit}}{sims_{RAVE}} wins_{Crit}$$

The β-formula is used as before to manage a transition from RAVE to regular expectation. We simply incorporate the criticality in the RAVE part by adding virtual won simulations to the AMAF statistics proprotional to both current amount of AMAF statistics and the strength of the move criticality. Therefore, the criticality contribution stays constant as AMAF samples are gathered, but its overall weight in tree policy is reduced properly as the move is actually explored. The win or loss is minimax-adjusted for the color to play, of course. Criticality can also indicate that moves are anticorellated, in this case we add losses.

We track criticality statistics for each node of the tree separately, therefore measuring move criticality only in a single context. We use criticality only when the node is visited at least $n = 250$ times.[1]

[1] Based on some informal experiments, it might be appropriate to adjust this value on the total number of playouts.

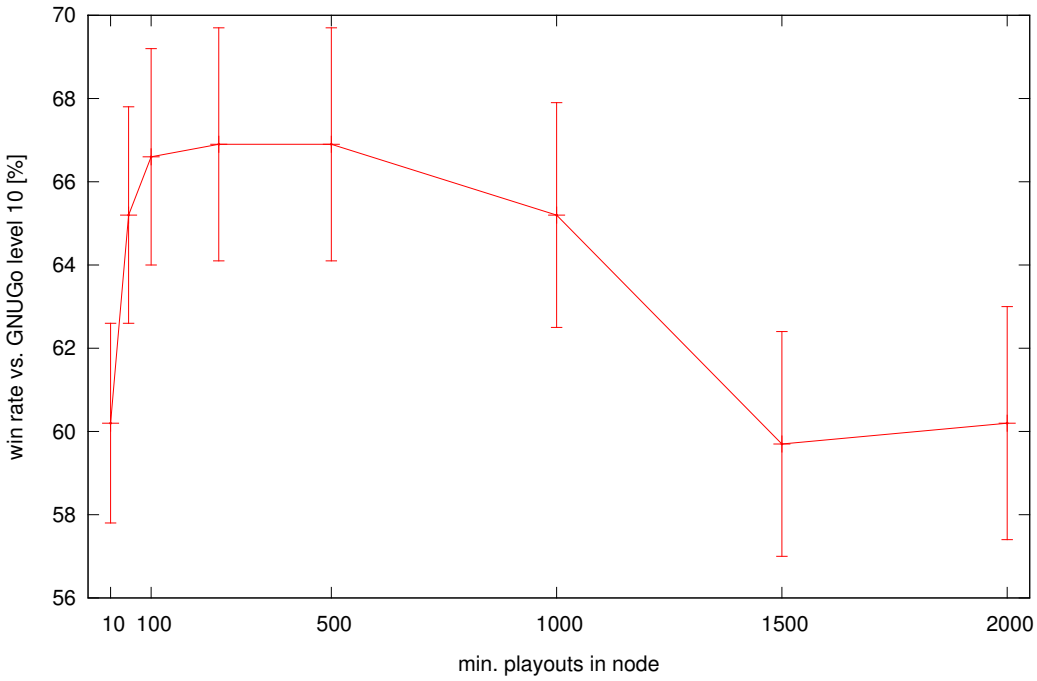


Figure 5.2: Performance of criticality with various minimum playouts bounds

5.3 Performance Discussion

When testing with 500s/game against GNUGo level 10 (on 19×19 with no komi and GNUGo black), the baseline version wins $57.6\% \pm 2.1\%$ while the version with criticality enabled wins $66.9\% \pm 2.2\%$, proving that criticality gives a measurable improvement. Fig. 5.2 shows that limiting the criticality to nodes with sufficient number of playouts is important (but being too strict diminishes the effect, at least with these time settings).

Normally, the criticality statistics simply test for presence of stone of the appropriate color at the given point in the final position. We have also tested using the point *local value* (Sec. 2.4.3) instead. While this appears beneficial with fixed number of simulations, the performance hit caused by local value computation makes this infeasible. We believe that we can optimize this computation enough in the future to make its usage in this case beneficial with real time settings.

Currently, the `master` branch contains an inferior default criticality settings. Use the parameter `policy=ucb1amaf:crit_negflip:crit_min_playouts=250`.

6. Liberty Maps

In Pachi, we explore purely based on RAVE, RAVE statistics come from moves played in the playouts and playout heuristics cause some moves to be played far more often than others. When the heuristics suggest useful moves, this greatly improves the game tree convergence and this founds the basis of Pachi's strength. However, when the heuristics are wrong, it causes misevaluations that often cause loss of the whole game.

We propose an approach to solve this problem by allowing online learning of the heuristics. Our idea is to track success rate of various heuristic candidates and then use this information to bias future move choice. We perform this learning on two group-specific heuristics that deal with 2-liberty and n-liberty ($2 < n \leq 4$) groups. They will examine the group's surrounding and suggest a set of moves that attempt to increase own group liberties or reduce enemy group liberties. Usually, multiple moves are possible and some of them are more efficient than others — it is our aim to play the more efficient moves more frequently.

6.1 Collecting Ratings

First, we need a way to key local group heuristic choices so that we can share collected data in similar situations. We assign each board point a hash number. Let a hash of a point be this hash number, deterministically modified based on the occupation of its direct neighbors. Let a hash of point set then be xor of all point hashes.[1] Liberty hash shall then be a hash of all liberty points of a group. Therefore, liberty hash of a group captures the particular configuration of the group liberties in the given situation, including prospects of escape by each of the liberties. **Liberty map** is then a mapping from liberty hash to a set of move ratings.

Ratings for the liberty map are collected as follows:

Algorithm 6 LibMapCollect

$H \leftarrow \{\}$
During the playout:
for all heuristic choices $(libhash, move)$ **do**
  $H \leftarrow H \cup (libhash, move)$
end for
In the final position:
for all $(libhash, move) \in H$ **do**
  UpdateLibMap($(libhash, move)$, lvalue($move$))
end for

As heuristics make choices for various groups within a simulation, $(hash, move)$ pairs are noted. When the simulation is finished, the rating of each noted move is updated based on its *local value* in the final position. The rating represents the average local value of the move,[2] i.e. $\mathbb{E}[\text{lvalue}(move) \mid libhash]$.

[1] This is a variant of the Zobrist hash [Zob70a].

[2] If the goal was to escape a group, high local value for the escaping move means the group

Table 6.1: Liberty Map Performance (19×19, no komi against GNU Go)

Player	Opponent	Time Settings	Win Rate
Baseline	GNU Go L10	500s/game	$56.4\% \pm 2.2\%$
LibMap	GNU Go L10	500s/game	$59.8\% \pm 2.1\%$
Baseline	GNU Go L10	6000p/move	$45.3\% \pm 2.2\%$
LibMap	GNU Go L10	6000p/move	$50.3\% \pm 2.3\%$
LibMap	Baseline	500s/game	$50.7\% \pm 2.6\%$
LibMap	Baseline	6000p/move	$55.1\% \pm 2.6\%$

6.2 Using Ratings

When a heuristical choice is made from a set of candidates, usually it is uniformly random selection. When using liberty maps, the move rating is accounted for:

Algorithm 7 LibMapChoose

Require: *libhash* describes the tackled group, M is array of proposed moves
  $i \leftarrow \text{random}(|M|)$
  With probability $p = 0.1$ unconditionally return $M[i]$
  $j \leftarrow i$
  repeat
    $m \leftarrow M[j]$
    if $\neg\exists\,\text{LibMap}(libhash, m)$ **then**
      return m
    end if
    if $\text{LibMap}(libhash, m) \geq 0.7$ **then**
      return m
    end if
    $j \leftarrow (j + 1) \bmod |M|$
  until $i = j$
  return $M[i]$

Therefore, given a group described by *libhash* and set of candidate moves M, preference is given to moves whose average eventual local value is higher than some constant (we use 0.7) or for which no local value has been stored yet.

The liberty map is shared for the whole tree and it is never reset, with the (certainly simplifying) assumption that the liberty hash fully describes the particular predicament of a group.

6.3 Performance Discussion

Table 6.1 shows that while liberty maps show demonstrable gain when given the same number of simulations, they are not quite convincing when taking the extra

survived successfully. Similarly if the goal was to kill a group, high local value for killing move means the area eventually became the attacker's territory.

time consumed into account; the gain is on the edge of statistical significance and more tuning and optimizations are needed for liberty maps to represent an important improvement.

We have tried using simple 0 or 1 instead of local value and the global result of a game as the move rating. Both seem to be inferior to local value.

As of the time of submission, the code is not merged in the `master` branch of our Git repository yet; it can be found in the `libmap` branch.

7. Future Directions

We hope the kind reader will bear in mind that much of our work presented here is just a snapshot of ongoing research. There is certainly a lot of room for improvement in our implementation. We expect to still gain large improvements in some of the algorithms presented here and we also have many ideas for completely new techniques.

7.1 Information Sharing

All the proposed techniques shall be investigated further. With regard to dynamic komi, the ratchet role and behavior should be looked into. Also, alternative dynamic komi scheme with more focus on the endgame seems possible; Jean-loup Gailly recently began investigating this.

There is a conceptual problem with liberty maps that still needs to be addressed — they select moves that are successful, but this may not mean moves that are good but moves that get bad replies in the simulation policy; in effect, optimizing for later failure.

7.1.1 Dynamic Score-value Scaling

Normally, we use the value of 0 as loss and value of 1 as win when updating the tree node expectations. In Pachi and some other programs, 0.04 of this has been allocated to score difference to slightly prefer larger wins and discourage large losses. Both likelihood of win and aesthetics of moves could be improved by dynamically giving more weight to average score margins of tree branches in suitable situations. Many guiding metrics come to mind — from simply the move number (like we use to determine the game phase for time controls) to the number of "unstable stones" on board.

7.1.2 Local Trees

We call our original and ultimate idea for efficient information sharing **local trees.** Aside of the main game tree, a "local tree" storing only non-tenuki sequences would be maintained by MCTS. It is built and descended in parallel with the main tree, but every time tenuki occurs (move is too far away from the last one in CFG topology), the node pointer in local tree is returned to the root.[1]

The aim is to record "local solutions" of various local situations. If the game tree discovers the proper sequence to resolve a tactical problem, the horizon effect (Sec. 2.4.2) may prevent it from being used in the tree, however the local tree should enable sharing the solution among various main game tree branches nevertheless.

Unfortunately, we have not been able to make the local trees truly effective so far. Two obvious questions arise — what values to store for local sequences and how to use the information stored in local trees. We have invented the *local value*

[1] Two trees are maintained, one for sequences starting with black move, another for white-first sequences.

metric (Sec. 2.4.3) for the evaluation purpose and update each tree branch with eventual local value of the first move in the branch — the idea is that successful move (and its followups) will convert the occupied point to the player's territory. Criticality might be also involved in selection of sequences from local trees.

As of using the information, integrating the value of played-through local sequences in the RAVE equation is not sufficient — to counter the horizon effect, simulations must be affected by the data. Our currently investigated approach is "sequence forcing", picking a given number of successful sequences from the tree and playing them out after the main tree leaf is reached but before the standard randomized playout takes over. There are many crucial choices to make when executing this approach and they still need thorough investigation.

7.1.3 Tactical Solvers

Liberty maps and local trees could be used to efficiently integrate external sources of information, e.g. tsumego solvers or other dedicated tactical search modules, in the simulations. There are successful algorithms to solve many tactical problems, but they are usually too slow to be used regularly in simulations and efficient means to reuse their results need to be found.

7.2 Other Research

Computer Go efforts should not be limited to achieving maximum strength and beating the humans. The tools may be also used to improve our understanding of the game and as a teaching aid to help people enjoy the game more. Jonathan Chetwyng is using Pachi while investigating ways to visualize the search and various metrics like point ownership and criticality at Peepo.com.

Conclusion

We have given a basic overview of the Computer Go problem and the MCTS algorithm. We presented our MCTS implementation Pachi, exploring the impact of various variables on playing strength and describing our minor enhancements. We then focused on the information sharing methods, explaining the motivation, describing the individual algorithms and again presenting measurements proving the effectivity of our proposals. While some of our ideas merely show promise and need further work to become truly effective, other clearly demonstrate their merit and lead to demonstrable significant improvements of tree search performance.

Computer Go is subject of intense research, while MCTS is making headways in other fields as well — not just different games, but also for solving other control and optimization problems. We sincerely hope our contribution to this research will be useful for pushing the Computer Go research forward as well as improving MCTS performance in other applications.

Bibliography

[ABF02] Peter Auer, Nicolò C. Bianchi, and Paul Fischer. "Finite-time Analysis of the Multiarmed Bandit Problem". In: *Machine Learning* 47.2/3 (2002), pp. 235–256. URL: http://citeseer.ist.psu.edu/auer00finitetime.html.

[All94] Victor Allis. "Searching for Solutions in Games and Artificial Intelligence". PhD thesis. University of Limburg, Maastricht, The Netherlands, 1994. ISBN: 9090074880.

[Bau+] Petr Baudiš et al. *Pachi — Simple Go/Baduk/Weiqi Bot*. URL: http://pachi.or.cz/.

[Bau10] Petr Baudiš. *40-core Pachi on KGS*. 2010. URL: http://groups.google.com/group/computer-go-archive/browse_thread/thread/e026ffa84c9c51a0.

[BC01] Bruno Bouzy and Tristan Cazenave. "Computer Go: an AI Oriented Survey". In: *Artificial Intelligence* 132 (2001), pp. 39–103.

[BFB+] Daniel Bump, Gunnar Farneback, Arend Bayer, et al. *GNU Go*. URL: http://www.gnu.org/software/gnugo/gnugo.html.

[BH03] Bruno Bouzy and Bernard Helmstetter. "Monte Carlo Go Developments". In: *Advances in Computer Games conference (ACG-10), Graz 2003*. Ed. by Ernst. Kluwer, 2003, pp. 159–174. URL: http://www.math-info.univ-paris5.fr/~bouzy/publications/bouzy-helmstetter.pdf.

[Boo90] Mark Boon. "A Pattern Matcher for Goliath". In: *Computer Go* 13 (1990), pp. 12–23.

[Bru93] Bernd Bruegmann. *Gobble — Monte Carlo Go*. 1993. URL: http://www.cgl.ucsf.edu/go/Programs/Gobble.html.

[BW94] Elwyn Berlekamp and David Wolfe. *Mathematical Go: Chilling Gets the Last Point*. A. K. Peters, 1994. ISBN: 1568810326.

[Cha+07] Guillaume Chaslot et al. "Progressive Strategies for Monte-Carlo Tree Search". In: *Joint Conference on Information Sciences, Salt Lake City 2007, Heuristic Search and Computer Game Playing Session*. 2007. URL: http://www.math-info.univ-paris5.fr/~bouzy/publications/CWHUB-pMCTS-2007.pdf.

[Cha+10] Guillaume Chaslot et al. "Adding Expert Knowledge and Exploration in Monte-Carlo Tree Search". In: *Advances in Computer Games*. Ed. by H. van den Herik and Pieter Spronck. Vol. 6048. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 1–13.

[Cie+] Aleš Cieply et al. *EGF ratings system — System description*. URL: http://www.europeangodatabase.eu/EGD/EGF_rating_system.php.

[Cou07] Rémi Coulom. "Computing Elo Ratings of Move Patterns in the Game of Go." In: *ICGA Journal* (2007), pp. 198–208.

[Cou09] Rémi Coulom. *Criticality: a Monte-Carlo Heuristic for Go Programs*. Invited talk, University of Electro-Communications, Tokyo, Japan. 2009. URL: http://remi.coulom.free.fr/Criticality/.

[CWH08] Guillaume Chaslot, Mark Winands, and H. van den Herik. "Parallel Monte-Carlo Tree Search". In: *Computers and Games*. Ed. by H. van den Herik et al. Vol. 5131. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 60–71.

[Dai] Don Dailey. *Computer Go Server*. URL: http://cgos.boardspace.net/.

[Dra+] Peter Drake et al. *Orego*. URL: http://legacy.lclark.edu/~drake/Orego.html.

[EM10] Markus Enzenberger and Martin Müller. "A Lock-Free Multithreaded Monte-Carlo Tree Search Algorithm". In: *Advances in Computer Games*. Ed. by H. van den Herik and Pieter Spronck. Vol. 6048. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 14–20.

[Enz] Markus Enzenberger. *GoGui — graphical user interface to programs that play the board game Go*. URL: http://gogui.sourceforge.net/.

[Enz+10] Markus Enzenberger et al. "Fuego — An Open-source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search". In: *IEEE Transactions on Computational Intelligence and AI in Games* 2:4 (2010), pp. 259–270.

[Enz96] Markus Enzenberger. *The Integration of A Priori Knowledge into a Go Playing Neural Network*. 1996. URL: http://webdocs.cs.ualberta.ca/~emarkus/neurogo/neurogo1996.html.

[Eur11] European Go Federation. *European Go Congress 2011 in Bordeaux, Computer Go*. 2011. URL: http://egc2011.eu/index.php/en/computer-go.

[Fai08] John Fairbarn. "History of Go". In: *Games of Go on Disk (GoGoD)*. 2008. URL: http://www.gogod.co.uk/.

[Far] Gunnar Farnebäck. *GTP — Go Text Protocol*. URL: http://www.lysator.liu.se/~gunnar/gtp/.

[Fota] David Fotland. *Re: Dynamic Komi's basics*. Feb 11, 2010. URL: http://www.mail-archive.com/computer-go@computer-go.org/msg13470.html.

[Fotb] David Fotland. *The Many Faces of Go*. URL: http://www.smart-games.com/manyfaces.html.

[Fre91] Free Software Foundation. *GNU General Public Licence*. 1991. URL: http://www.gnu.org/licenses/gpl-2.0.html.

[Gai] Jean-loup Gailly. *Komi and the value of the first move*. Apr 5, 2010. URL: http://www.mail-archive.com/computer-go@dvandva.org/msg00096.html.

[Gel+06] Sylvain Gelly et al. *Modification of UCT with Patterns in Monte-Carlo Go*. English. Research Report RR-6062. INRIA, 2006. URL: http://hal.inria.fr/inria-00117266/en/.

[Gra+01] Thore Graepel et al. "Learning on Graphs in the Game of Go". In: *Artificial Neural Networks — ICANN 2001*. Ed. by Georg Dorffner, Horst Bischof, and Kurt Hornik. Vol. 2130. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2001, pp. 347–352.

[GS07] Sylvain Gelly and David Silver. "Combining online and offline knowledge in UCT". In: *ICML '07: Proceedings of the 24th international conference on Machine learning*. New York, NY, USA: ACM, 2007, pp. 273–280. ISBN: 978-1-59593-793-3.

[GS08] Sylvain Gelly and David Silver. "Achieving master level play in 9x9 Computer Go". In: *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*. Chicago, Illinois: AAAI Press, 2008, pp. 1537–1540. ISBN: 978-1-57735-368-3.

[GW06] Sylvain Gelly and Yizao Wang. "Exploration exploitation in Go: UCT for Monte-Carlo Go". In: *Twentieth Annual Conference on Neural Information Processing Systems NIPS 2006* (2006). URL: http://eprints.pascal-network.org/archive/00002713/.

[HCL10] Shih-Chieh Huang, Rémi Coulom, and Shun-Shii Lin. "Monte-Carlo Simulation Balancing in Practice". In: *International Conference on Computers and Games*. Kanzawa, Japan, 2010. URL: http://remi.coulom.free.fr/CG2010-Simulation-Balancing/.

[Hol] Arno Hollosi. *SGF File Format*. URL: http://www.red-bean.com/sgf/.

[Hou] Jason House. *Groups, liberties, and such*. Oct 14, 2005. URL: http://go.computer.free.fr/go-computer/msg08075.html.

[HP+] Arno Hollosi, Morten Pahle, et al. *Sensei's Library*. URL: http://senseis.xmp.net/.

[KS06] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning". In: *Machine Learning: ECML 2006*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Vol. 4212. Lecture Notes in Computer Science. 10.1007/11871842.29. Springer Berlin / Heidelberg, 2006, pp. 282–293.

[Lew] Łukasz Lew. *libEGO — Library of effective Go routines*. URL: https://github.com/lukaszlew/libego.

[LR85] Tze Leung Lai and Herbert Robbins. "Asymptotically efficient adaptive allocation rules". In: *Advances in Applied Mathematics* 6 (1985), pp. 4–22.

[Odo+] Gary Odom et al. *Kogo's Joseki Dictionary*. URL: http://waterfire.us/joseki.htm.

[Pel+09] Seth Pellegrino et al. "Localizing Search in Monte-Carlo Go Using Statistical Covariance". In: *ICGA Journal* 32:3 (2009), pp. 154–160.

[PM88] Stephen K. Park and Keith W. Miller. "Random number generators: good ones are hard to find". In: *Communications of the ACM* 31 (10 1988), pp. 1192–1201. ISSN: 0001-0782.

[Rob52] Herbert Robbins. "Some aspects of the sequential design of experiments". In: *Bulletin of the American Mathematics Society*. Vol. 58. 1952, pp. 527–535.

[RW79] Walter Reitman and Bruce Wilcox. "The structure and performance of the interim.2 go program". In: *Proceedings of the 6th international joint conference on Artificial intelligence - Volume 2*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1979, pp. 711–719. ISBN: 0-934613-47-8.

[Sch] William Schubert. *KGS Go Server*. URL: http://gokgs.com/.

[Sen] Sensei's Library. *Rank — worldwide comparison*. URL: http://senseis.xmp.net/?RankWorldwideComparison.

[ST09] David Silver and Gerald Tesauro. "Monte Carlo Simulation Balancing". In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, 2009.

[Tai11] NUTN Taiwan. *Human vs. Computer Go Competition, SSCI 2011 Symposium Series on Computational Intelligence*. 2011. URL: http://ssci2011.nutn.edu.tw/result.htm.

[TT] John Tromp and Bill Taylor. *The Logical Rules of Go*. URL: http://homepages.cwi.nl/~tromp/go.html.

[Wed05] Nick Wedd. *Computer Go Tournaments on KGS*. 2005–2011. URL: http://www.weddslist.com/kgs/index.html.

[Wol07] Thomas Wolf. "Two Applications of a Life & Death Problem Solver in Go". In: *Journal of ÖGAI* 26 (2 2007), pp. 11–18.

[Yam] Hiroshi Yamashita. *Re: Dynamic Komi's basics*. Feb 11, 2010. URL: http://www.mail-archive.com/computer-go@computer-go.org/msg13464.html.

[Zob70a] Albert L. Zobrist. *A hashing method with applications for game playing*. Tech. rep. University of Wisconsin, 1970.

[Zob70b] Albert L. Zobrist. "Feature extraction and representation for pattern recognition and the game of Go." PhD thesis. University of Wisconsin, 1970.

List of Figures

List of Tables

List of Abbreviations

AMAF	All Moves as First
CFG	Common Fate Graph
CGT	Combinatorial Game Theory
GTP	Go Text Protocol
JSON	JavaScript Object Notation
KGS	KGS Go Server
MCTS	Monte Carlo Tree Search
NFS	Network File System
RAVE	Rapid Action Value Estimation
SGF	Smart Game Format
UCB	Upper Confidence Bound
UCT	Upper Confidence Tree

A. Pachi Source Code

The Pachi source code can be found at `http://repo.or.cz/w/pachi.git` and the current snapshot of the git repository (with the latest version checked out) is also saved on the attached CD. Here, we only list some general information.

Statistics generated using David A. Wheeler's 'SLOCCount':[1]

```
SLOC    Directory       SLOC-by-Language (Sorted)
4138    top_dir         ansic=4138
3987    uct             ansic=3987
1075    distributed     ansic=1075
1059    tactics         ansic=1059
777     playout         ansic=777
292     joseki          ansic=250,perl=42
213     montecarlo      ansic=213
213     t-play          sh=213
106     t-unit          ansic=106
99      tools           perl=69,sh=30
95      replay          ansic=95
36      random          ansic=36
0       media           (none)


Totals grouped by language (dominant language first):
ansic:        11736 (97.07%)
sh:             243 (2.01%)
perl:           111 (0.92%)


Total Physical Source Lines of Code (SLOC)                = 12,090
Development Effort Estimate, Person-Years (Person-Months) = 2.74 (32.87)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 0.79 (9.43)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 3.49
Total Estimated Cost to Develop                           = $ 369,989
 (average salary = $56,286/year, overhead = 2.40).
```

The Pachi source tree manifest follows:

```
COPYING       debug.h       joseki19.pdict  ownermap.h  random/   tactics/
CREDITS       distributed/  media/          pachi.c     random.c  timeinfo.c
HACKING       engine.h      montecarlo/     pattern3.c  random.h  timeinfo.h
Makefile      fbook.c       move.c          pattern3.h  replay/   tools/
Makefile.lib  fbook.h       move.h          playout/    stats.h   uct/
README        fixp.h        mq.h            playout.c   stone.c   util.h
TODO          gtp.c         network.c       playout.h   stone.h   version.h
board.c       gtp.h         network.h       probdist.c  t-play/
board.h       joseki/       ownermap.c      probdist.h  t-unit/

./distributed:
Makefile       distributed.h  merge.h     protocol.h
distributed.c  merge.c        protocol.c

./joseki:
Makefile  README  base.c  base.h  joseki.c  joseki.h  sgfvar2gtp.pl

./media:
pachi-small.png  pachi.jpg

./montecarlo:
Makefile  internal.h  montecarlo.c  montecarlo.h

./playout:
Makefile  light.c  light.h  moggy.c  moggy.h
```

[1] We have removed the third-party `twogtp.py` script beforehand.

```
./random:
Makefile  random.c  random.h

./replay:
Makefile  replay.c  replay.h

./t-play:
TESTS  autotest/  resum*  test_in_context.sh*

./t-play/autotest:
README  autotest-client*   autotest-gather*  autotest-prune*  autotest-worker*
TODO    autotest-clients*  autotest-lib      autotest-show*   rc

./t-unit:
Makefile  README  sar.t  test.c  test.h

./tactics:
1lib.c  2lib.c  Makefile  ladder.h  nakade.h  nlib.h       selfatari.h  util.h
1lib.h  2lib.h  ladder.c  nakade.c  nlib.c    selfatari.c  util.c

./tools:
complete-tromptaylor.gtp  genmove19.gtp      sgf-analyse.pl*  twogtp.py*
complete.gtp              gentbook.sh*       sgf2gtp.pl*
genmove.gtp               pattern3_show.pl*  spirit.gtp

./uct:
Makefile   internal.h  plugins.c  prior.c   search.h  tree.c  uct.h
dynkomi.c  plugin/     plugins.h  prior.h   slave.c   tree.h  walk.c
dynkomi.h  plugin.h    policy/    search.c  slave.h   uct.c   walk.h

./uct/plugin:
example.c  wolf.c

./uct/policy:
Makefile  generic.c  generic.h  ucb1.c  ucb1amaf.c
```

The Git commit history starts on Nov 11, 2007. Since then, 2606 commits were made in the `master` branch, with the by-author breakdown given in Table A.1.

Table A.1: By-author Git History Breakdown

Author	Commits
Petr Baudiš	2393
Jean-loup Gailly	208
Matthew Woodcraft	3
Francois van Niekerk	1

B. Sample Pachi Session

An example Pachi session given in full. The option `-t 5` sets maximum time to 5 seconds per move; `dumpthres=10000` reduces the subset of the game tree shown after move is chosen. User input is shown **in bold** while debugging output written to stderr is typeset *slanted*. Standard GTP output is shown in regular font; for each input command, it is just a single line beginning with the equal sign. Some superfluous lines have been culled.

```
$ ./pachi -t 5 dumpthres=10000
Random seed: 1312203069
boardsize 5
=

komi 24
Move:   0  Komi: 24.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E
    +-----------+
  5 | . . . . . |
  4 | . . . . . |
  3 | . . . . . |
  2 | . . . . . |
  1 | . . . . . |
    +-----------+

=

genmove b
Fresh board with random seed 1312203069
desired 4.50, worst 5.00, clock [1] 0.00 + 5.00/1*1, lag 2.00
[10000] best 0.552927 | seq  C3  C4  D4  B3 | can  C3(0.553)  C4(0.483)  E5(0.364)  D4(0.493)
[20000] best 0.559103 | seq  C3  C4  D4  B3 | can  C3(0.559)  C4(0.511)  E5(0.334)  D4(0.419)
[30000] best 0.592087 | seq  C3  C4  D4  B3 | can  C3(0.592)  C4(0.499)  E5(0.330)  C5(0.264)
(UCT tree; root white; extra komi 0.000000; max depth 30)
[pass] 0.585/32528 [prior 0.000/0 amaf 0.000/0 crit -0.415] h=0 c#=7 <f4240>
 [C3] 0.590/31603 [prior 0.750/28 amaf 0.590/31900 crit 0.129] h=0 c#=6 <f4244>
  [C4] 0.585/30185 [prior 0.167/42 amaf 0.585/30307 crit 0.021] h=0 c#=14 <f424b>
   [D4] 0.600/25244 [prior 0.875/56 amaf 0.598/27593 crit 0.128] h=0 c#=23 <f4259>
    [B3] 0.585/17170 [prior 0.167/42 amaf 0.579/19554 crit 0.284] h=0 c#=22 <f426a>
     [B4] 0.612/14939 [prior 0.900/70 amaf 0.606/15830 crit 0.119] h=0 c#=21 <f434e>
      [C2] 0.599/13077 [prior 0.125/56 amaf 0.597/13922 crit 0.246] h=0 c#=20 <f4381>
       [D3] 0.630/12165 [prior 0.900/70 amaf 0.630/12224 crit 0.055] h=0 c#=19 <f4aa2>
        [A3] 0.619/10907 [prior 0.125/56 amaf 0.619/11468 crit 0.287] h=0 c#=18 <f5223>
[32594] best 0.590394 | seq  C3  C4  D4  B3 | can  C3(0.590)  C4(0.497)  E5(0.324)  C5(0.273)
*** WINNER is C3 (3,3) with score 0.5904 (31603/32528:32528/32594 games), extra komi 0.000000
genmove in 2.50s (13021 games/s, 13021 games/s/thread)
playing move C3
Move:   1  Komi: 24.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . . . . . |  5 | x x x x x |
  4 | . . . . . |  4 | x x x X x |
  3 | . . X). . |  3 | x x X x x |
  2 | . . . . . |  2 | x x x x x |
  1 | . . . . . |  1 | x x x x x |
    +-----------+    +-----------+

= C3
```

play w c4

```
Move:   2  Komi: 24.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . . . . . |  5 | x x x x x |
  4 | . . O). . |  4 | x x x X x |
  3 | . . X . . |  3 | x x X x x |
  2 | . . . . . |  2 | x x x x x |
  1 | . . . . . |  1 | x x x x x |
    +-----------+    +-----------+
```

=

genmove b

```
desired 4.50, worst 5.00, clock [1] 0.00 + 5.00/1*1, lag 3.26
<pre-simulated 30185 games>
(UCT tree; root white; extra komi 0.000000; max depth 28)
[C4] 0.584/30703 [prior 0.167/42 amaf 0.585/30307 crit 0.021] h=0 c#=14 <f424b>
 [D4] 0.599/25760 [prior 0.875/56 amaf 0.597/28109 crit 0.128] h=0 c#=23 <f4259>
  [B3] 0.584/17679 [prior 0.167/42 amaf 0.577/20063 crit 0.285] h=0 c#=22 <f426a>
   [B4] 0.610/15424 [prior 0.900/70 amaf 0.604/16326 crit 0.119] h=0 c#=21 <f434e>
    [C2] 0.599/13077 [prior 0.125/56 amaf 0.595/14343 crit 0.246] h=0 c#=20 <f4381>
     [D3] 0.630/12165 [prior 0.900/70 amaf 0.630/12224 crit 0.055] h=0 c#=19 <f4aa2>
      [A3] 0.619/10907 [prior 0.125/56 amaf 0.619/11468 crit 0.287] h=0 c#=18 <f5223>
[518] best 0.599045 | seq  D4  B3  B4  C2 | can  D4(0.599)  C2(0.533)  D3(0.517)  C5(0.479)
*** WINNER is D4 (4,4) with score 0.5990 (25760/30703:518/518 games), extra komi 0.000000
genmove in 0.10s (5132 games/s, 5132 games/s/thread)
playing move D4
Move:   3  Komi: 24.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . . . . . |  5 | x x x X X |
  4 | . . O X). |  4 | , x x X X |
  3 | . . X . . |  3 | , , X X X |
  2 | . . . . . |  2 | , , , x x |
  1 | . . . . . |  1 | , , , x x |
    +-----------+    +-----------+
```

= D4

play w b3

```
Move:   4  Komi: 24.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . . . . . |  5 | x x x X X |
  4 | . . O X . |  4 | , x x X X |
  3 | . O)X . . |  3 | , , X X X |
  2 | . . . . . |  2 | , , , x x |
  1 | . . . . . |  1 | , , , x x |
    +-----------+    +-----------+
```

=

genmove b

```
desired 4.50, worst 5.00, clock [1] 0.00 + 5.00/1*1, lag 4.62
<pre-simulated 17679 games>
(UCT tree; root white; extra komi 0.000000; max depth 26)
[B3] 0.584/18273 [prior 0.167/42 amaf 0.577/20063 crit 0.285] h=0 c#=22 <f426a>
 [B4] 0.609/15933 [prior 0.900/70 amaf 0.604/16872 crit 0.119] h=0 c#=21 <f434e>
  [C2] 0.599/13077 [prior 0.125/56 amaf 0.595/14783 crit 0.246] h=0 c#=20 <f4381>
   [D3] 0.630/12165 [prior 0.900/70 amaf 0.630/12224 crit 0.055] h=0 c#=19 <f4aa2>
    [A3] 0.619/10907 [prior 0.125/56 amaf 0.619/11468 crit 0.287] h=0 c#=18 <f5223>
[594] best 0.609346 | seq  B4  C2  D3  A3 | can  B4(0.609)  C2(0.526)  D3(0.534)  A1(0.169)
*** WINNER is B4 (2,4) with score 0.6093 (15933/18273:594/594 games), extra komi 0.000000
genmove in 0.10s (5885 games/s, 5885 games/s/thread)
```

```
playing move B4
Move:   5  Komi: 24.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . . . . . |  5 | x x x X X |
  4 | . X)O X . |  4 | x x x X X |
  3 | . O X . . |  3 | , , X X X |
  2 | . . . . . |  2 | , , , x x |
  1 | . . . . . |  1 | , , , x x |
    +-----------+    +-----------+

= B4

play w c2
Move:   6  Komi: 24.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . . . . . |  5 | x x x X X |
  4 | . X O X . |  4 | x x x X X |
  3 | . O X . . |  3 | , , X X X |
  2 | . . O). . |  2 | , , , x x |
  1 | . . . . . |  1 | , , , x x |
    +-----------+    +-----------+

=

genmove b
desired 4.50, worst 5.00, clock [1] 0.00 + 5.00/1*1, lag 4.06
<pre-simulated 13077 games>
(UCT tree; root white; extra komi 0.000000; max depth 24)
[C2] 0.603/13718 [prior 0.125/56 amaf 0.595/14783 crit 0.244] h=0 c#=20 <f4381>
 [D3] 0.635/12774 [prior 0.900/70 amaf 0.635/12833 crit 0.055] h=0 c#=19 <f4aa2>
  [A3] 0.624/11393 [prior 0.125/56 amaf 0.624/12027 crit 0.288] h=0 c#=18 <f5223>
[642] best 0.634946 | seq  D3  A3  D2  C1 | can  D3(0.635)  B2(0.277)  A2(0.160)  A1(0.040)
*** WINNER is D3 (4,3) with score 0.6349 (12774/13718:641/642 games), extra komi 0.000000
genmove in 0.10s (6377 games/s, 6377 games/s/thread)
playing move D3
Move:   7  Komi: 24.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . . . . . |  5 | X X X X X |
  4 | . X O X . |  4 | X X X X X |
  3 | . O X X). |  3 | x x X X X |
  2 | . . O . . |  2 | x x x X X |
  1 | . . . . . |  1 | x x x X X |
    +-----------+    +-----------+

= D3

play w b2
Move:   8  Komi: 24.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . . . . . |  5 | X X X X X |
  4 | . X O X . |  4 | X X X X X |
  3 | . O X X . |  3 | x x X X X |
  2 | . O)O . . |  2 | x x x X X |
  1 | . . . . . |  1 | x x x X X |
    +-----------+    +-----------+

=

genmove b
desired 4.50, worst 5.00, clock [1] 0.00 + 5.00/1*1, lag 4.78
<pre-simulated 89 games>
(UCT tree; root white; extra komi 0.000000; max depth 22)
[B2] 0.730/733 [prior 0.167/42 amaf 0.616/8695 crit 0.260] h=0 c#=18 <f5220>
[644] best 0.796620 | seq  D2  B5  C5  A4 | can  D2(0.797)  A3(0.642)  A1(0.421)  C5(0.787)
*** WINNER is D2 (4,2) with score 0.7966 (516/733:644/644 games), extra komi 0.000000
genmove in 0.10s (6395 games/s, 6395 games/s/thread)
```

```
playing move D2
Move:   9  Komi: 24.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . . . . . |  5 | X X X X X |
  4 | . X O X . |  4 | X X X X X |
  3 | . O X X . |  3 | x x X X X |
  2 | . O O X). |  2 | x x x X X |
  1 | . . . . . |  1 | x x x X X |
    +-----------+    +-----------+

= D2

play w a4
Move:  10  Komi: 24.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . . . . . |  5 | X X X X X |
  4 | O)X O X . |  4 | X X X X X |
  3 | . O X X . |  3 | x x X X X |
  2 | . O O X . |  2 | x x x X X |
  1 | . . . . . |  1 | x x x X X |
    +-----------+    +-----------+

=

genmove b
desired 4.50, worst 5.00, clock [1] 0.00 + 5.00/1*1, lag 3.58
<pre-simulated 27 games>
(UCT tree; root white; extra komi 0.000000; max depth 20)
[A4] 0.840/772 [prior 0.125/56 amaf 0.786/316 crit 0.104] h=0 c#=16 <f8ce3>
[745] best 0.876386 | seq  B5  C5  D5  A3 | can  B5(0.876)  C5(0.739)  A1(0.222)  A3(0.192)
*** WINNER is B5 (2,5) with score 0.8764 (684/772:745/745 games), extra komi 0.000000
genmove in 0.10s (7401 games/s, 7401 games/s/thread)
playing move B5
Move:  11  Komi: 24.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . X). . . |  5 | X X X X X |
  4 | O X O X . |  4 | X X X X X |
  3 | . O X X . |  3 | X X X X X |
  2 | . O O X . |  2 | X X X X X |
  1 | . . . . . |  1 | X X X X X |
    +-----------+    +-----------+

= B5

play w d1
Move:  12  Komi: 24.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . X . . . |  5 | X X X X X |
  4 | O X O X . |  4 | X X X X X |
  3 | . O X X . |  3 | X X X X X |
  2 | . O O X . |  2 | X X X X X |
  1 | . . . O). |  1 | X X X X X |
    +-----------+    +-----------+

=

genmove b
desired 4.50, worst 5.00, clock [1] 0.00 + 5.00/1*1, lag 3.30
<pre-simulated 1 games>
(UCT tree; root white; extra komi 0.000000; max depth 18)
[D1] 0.864/694 [prior 0.167/42 amaf 0.897/382 crit 0.102] h=0 c#=14 <11a069>
[694] best 0.883252 | seq  C1  B1  C5  C1 | can  C1(0.883)  A1(0.825)  C5(0.790)  D5(0.842)
*** WINNER is C1 (3,1) with score 0.8833 (548/694:693/694 games), extra komi 0.000000
genmove in 0.10s (6876 games/s, 6876 games/s/thread)
```

```
playing move C1
Move:  13  Komi: 24.0  Handicap: 0  Captures B: 0 W: 0
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . X . . . |  5 | X X X X X |
  4 | O X O X . |  4 | X X X X X |
  3 | . O X X . |  3 | X X X X X |
  2 | . O O X . |  2 | X X X X X |
  1 | . . X)O . |  1 | X X X X X |
    +-----------+    +-----------+

= C1

play w b1
Move:  14  Komi: 24.0  Handicap: 0  Captures B: 0 W: 1
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . X . . . |  5 | X X X X X |
  4 | O X O X . |  4 | X X X X X |
  3 | . O X X . |  3 | X X X X X |
  2 | . O O X . |  2 | X X X X X |
  1 | . O). O . |  1 | X X X X X |
    +-----------+    +-----------+

=

genmove b
desired 4.50, worst 5.00, clock [1] 0.00 + 5.00/1*1, lag 3.44
<pre-simulated 200 games>
(UCT tree; root white; extra komi 0.000000; max depth 16)
[B1] 0.901/996 [prior 0.062/112 amaf 0.876/478 crit 0.079] h=0 c#=12 <11ac9b>
[798] best 0.909070 | seq  C5 pass  A3  A2 | can  C5(0.909)  A2(0.876)  A1(0.787) pass(0.000)
*** WINNER is C5 (3,5) with score 0.9091 (815/996:796/798 games), extra komi 0.000000
genmove in 0.10s (7923 games/s, 7923 games/s/thread)
playing move C5
Move:  15  Komi: 24.0  Handicap: 0  Captures B: 1 W: 1
      A B C D E        A B C D E
    +-----------+    +-----------+
  5 | . X X). . |  5 | X X X X X |
  4 | O X . X . |  4 | X X X X X |
  3 | . O X X . |  3 | X X X X X |
  2 | . O O X . |  2 | X X X X X |
  1 | . O . O . |  1 | X X X X X |
    +-----------+    +-----------+

= C5
```