Balancing MCTS by Dynamically Adjusting Komi Value

Petr Baudiš[1]

Faculty of Math and Physics, Charles University, Prague, Czech Republic
SUSE Labs, Novell, Prague, Czech Republic

ABSTRACT

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. 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. We also compare performance of MCTS and traditional tree search in the context of extreme situations and measure the effect of dynamic komi on actual playing strength of a state-of-art MCTS Go program. Based on our results, we also conjencture on resilience of the game search tree to changes in the evaluation function throughout the search.

This work has not been submitted to peer review and official publication yet and is a work-in-progress, for your review only. Version 20110724a.

1. INTRODUCTION

The board game of Go has proven to be a challenge for traditional game-playing algorithms. Recently, the approach of using Monte Carlo simulations organized in a probabilistic minimax tree finally made significant headways in playing strength, far outperforming any other programmatic approaches and achieving the level of advanced human Go players on standard-sized boards and master level on small boards (Gelly and Silver, 2008). However, multiple obstacles hinder the effectivity of the Monte Carlo Tree Search (MCTS) approach.

To recapitulate, Go is a two-player full-information board game played with black and white stones on a square grid (we shall assume size 19×19 lines unless noted otherwise); the goal of the game is to surround the most territory and capture enemy stones. We assume basic familiarity with the game.

The Monte Carlo approach tries to solve the best move choice problem in Go by performing many simulations (randomized self-played games) per each candidate move, then choosing the move with the highest win rate. It turns out that a major leap in strength can be achieved by building a game tree from the Monte Carlo simulations and treating the decision in each node as a multi-armed bandit problem; the UCT algorithm was first to be used for MCTS in Go programs. An additional improvement has been to use information gathered in the simulations within the tree search, resulting in the RAVE search algorithm[2] that forms the basis of the current strongest programs.

In section 2, we detail the state of art Monte Carlo Tree Search technique used in computer Go (and our test program Pachi in particular) and the problematic points that we seek to address. In section 3, we outline our approach to tackle the problem. In sections 4 and 5, we present several specific algorithms we have developed, and we then measure and compare their performance in section 6. Eventually, we note our conclusions and outline future research directions in section 7.

[1] email:pasky@ucw.cz. Supported by the GAUK Project 66010 of Charles University Prague.

[2] Originally RAVE was concieved as UCT modification, but some programs (including our Pachi test program) do not use the UCT-specific term anymore, resulting in a wholly independent algorithm.

2. BACKGROUND

2.1 The RAVE Search Algorithm

The RAVE search algorithm (Chaslot *et al.*, 2010) repeatedly descends the minimax tree, runs a Monte Carlo simulation from the reached tree leaf, propagates the result back through the tree and expands the tree leaf when it is reached n times.[3] During the descent, for each node the algorithm chooses the child with the maximum value of:

$$\beta \frac{wins_{AMAF}}{sims_{AMAF}} + (1 - \beta) \frac{wins}{sims}$$

$$\beta = \frac{sims_{AMAF}}{sims_{AMAF} + sims + sims_{AMAF} sims / 3000}$$

The terms in the equation represent:

- $sims$, $wins$ is the number of simulations taken within the child and the number of won simulations, respectively
- $sims_{AMAF}$, $wins_{AMAF}$ has similar meaning, but all simulations from the current node where the player at any point played the move represented by the child are considered (so-called "all moves as first" heuristic)
- β gives large weight to AMAF values inferred from sibling simulations when few actual results are available and small weight when more simulations are run from the actual evaluated node

The RAVE algorithm ensures that the most thoroughly searched nodes are the ones that yield the highest likelihood of won simulation, giving initial search preference to moves highly correlated with achieving a win in the current situation in general.

The RAVE algorithm is usually complemented by other important domain-specific extensions — most notably, prior values of newly expanded nodes are assigned heuristically, and a lot of Go knowledge and heuristics are used within the Monte Carlo simulations.

2.2 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.[4] This is due to the fact that MCTS considers and maximizes expected win probability, 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.

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. Usually, the handicap amount ranges between one stone[5] 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.[6]

[3] We use the value $n = 2$ in Pachi.

[4] See Figures 2a and 2b for comparison with a classical program performance.

[5] 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.

[6] 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.

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 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, creating the danger of losing the game.[7]

3. THE DYNAMIC KOMI TECHNIQUE

One possible way of tackling the described problem is using the "dynamic komi" technique. Komi is Go term for 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*[8] 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 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.[9]

We have decided to implement and test several approaches to dynamic komi. We test their 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.

3.1 Prior Work

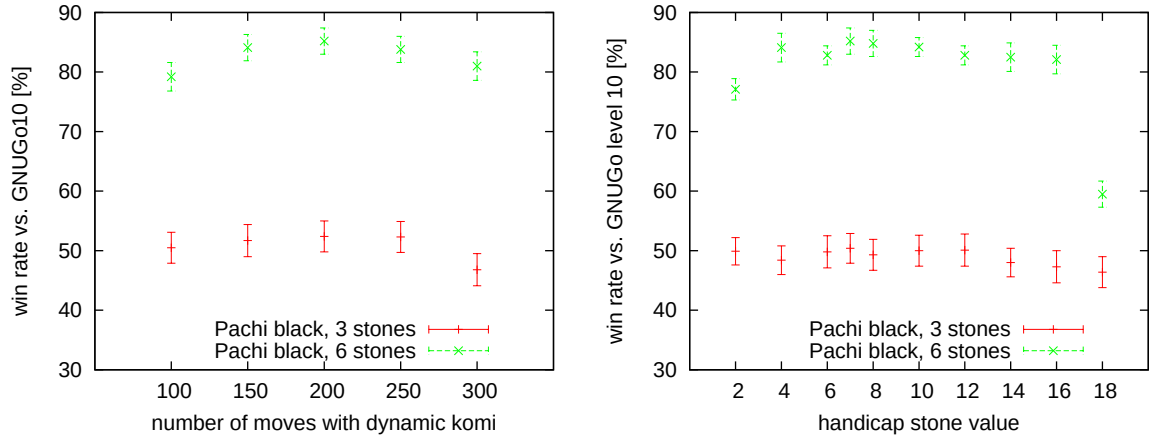
It has been long suggested especially by non-programmer players in the Computer Go community to use the dynamic komi approach to balance the pure winrate orientation of MCTS, however it has been met with scepticism from the program authors since it introduces artificial inaccuracy to the game tree and 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. Perhaps the first report of dynamic komi usage has been posted by Hideki Kato (Kato, 2008). Many Faces of Go uses an algorithm analogous to our Linearly Decreasing Handicap Compensation with slightly different constants (Fotland, 2010). 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 (Yamashita, 2010).

[7] In our test program Pachi, 0.04 of the win/loss value is actually allocated to reflect the score difference, however the effect on larger win margins preference appears to be very limited and the constant cannot be raised further without incurring strength loss.

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

[9] 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 1: Performance in handicap games.

4. 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 1 LINEARHANDICAP

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

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. 1a). h is the point value of single handicap stone; values around 7 work best for us (Fig. 1b).[10]

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.

5. 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.

[10]The value of one extra handicap stone is twice the value of the standard komi (Gailly, 2010) 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.

5.1 Situation Measure

5.1.1 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 2 SCORESITUATIONAL

Require: $\mathbb{E}[\text{score}]$ is the average score over reasonable amount of simulations (including then-used dynamic komi).
  if $\text{MoveNum} < 20$ **then**
    $\text{Komi} \leftarrow$ LINEARHANDICAP
    return
  end if
  $\text{BoardOccupiedRatio} \leftarrow \frac{\text{OccupiedIntersections}}{\text{Intersections}}$
  $\text{GamePhase} \leftarrow \text{BoardOccupiedRatio} + s$
  $\text{KomiRate} \leftarrow (1 + \exp(c \cdot \text{GamePhase}))^{-1}$
  $\text{Komi} \leftarrow \text{Komi} + \text{KomiRate} \cdot \mathbb{E}[\text{score}]$

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.

5.1.2 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.

Algorithm 3 VALUESITUATIONAL

Require: Value is the winning rate over reasonable amount of simulations (including then-used dynamic komi).
  if $\text{MoveNum} < 20$ **then**
    $\text{Komi} \leftarrow$ LINEARHANDICAP
    $\text{Ratchet} \leftarrow \infty$
    return
  end if
  if $\text{Value} < \mathbf{red}$ **then**
    if $\text{Komi} > 0$ **then**
      $\text{Ratchet} \leftarrow \text{Komi}$
    end if
    $\text{Komi} \leftarrow \text{Komi} - 1$
  else
    if $\text{Value} > \mathbf{green} \wedge \text{Komi} < \text{Ratchet}$ **then**
      $\text{Komi} \leftarrow \text{Komi} + 1$
    end if
  end if

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.

The optimal values we have found are $\mathbf{red} = 0.45$, $\mathbf{green} = 0.5$ — 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.[11]

5.2 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[12] 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.

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[13] 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[14] in order to resign in cases when we cannot catch up anymore.

6. PERFORMANCE DISCUSSION

We have implemented the methods above within the state-of-art MCTS Go-playing program Pachi. (Baudišet al., 2010) As of October 2010, multi-core instance of the program was ranked at the 2k level on the KGS internet Go server (Schubert, 2010) and reached rating around 2450 on the CGOS computer-go server (Dailey, 2010). We use the RAVE algorithm as described in Section 2.1 with heuristic node value priors analogous to (Chaslot *et al.*, 2010), the simulations use Mogo-like heuristics and 3×3 patterns (Gelly *et al.*, 2006). In the play tests, we use a single core, reuse appropriate tree portions but do not ponder on the opponent's move.

[11] Note that the same tree policy, choosing the best value (expectation) available at the moment, is of course used.
[12] We update dynamic komi every 1000 simulations in our implementation.
[13] We use $n = 20$ for 19×19, $n = 4$ for 9×9.
[14] Estimation based on board fill ratio.

Table 1: Dynamic Komi Performance — Even Games

Method	Opponent	Time per Game	Win Rate
Pachi NONE	GNU Go	3.6min	$27.3\% \pm 3.2\%$
Pachi SCORESIT	GNU Go	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	GNU Go	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	$58.3\% \pm 3.4\%$

Our tests were performed on the 19×19 board.[15] We perform play-testing against GNU Go v3.7.12 level 10 (Bump *et al.*, 2010)[16] and self-play testing.

In table 1, we present our measurements of various dynamic komi methods in even games. Even though the improvement against GNU Go 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 2a, we present measurements of dynamic komi effect in handicap games when taking black and varying amount of stones. We compare Pachi with 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. GNU Go 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 is still as lacking as it might appear here.

In Figure 2b we show measurements of dynamic komi effect in handicap games when taking white and giving varying amount of stones; the opponent is GNU Go Level 1 and GNU Go 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.

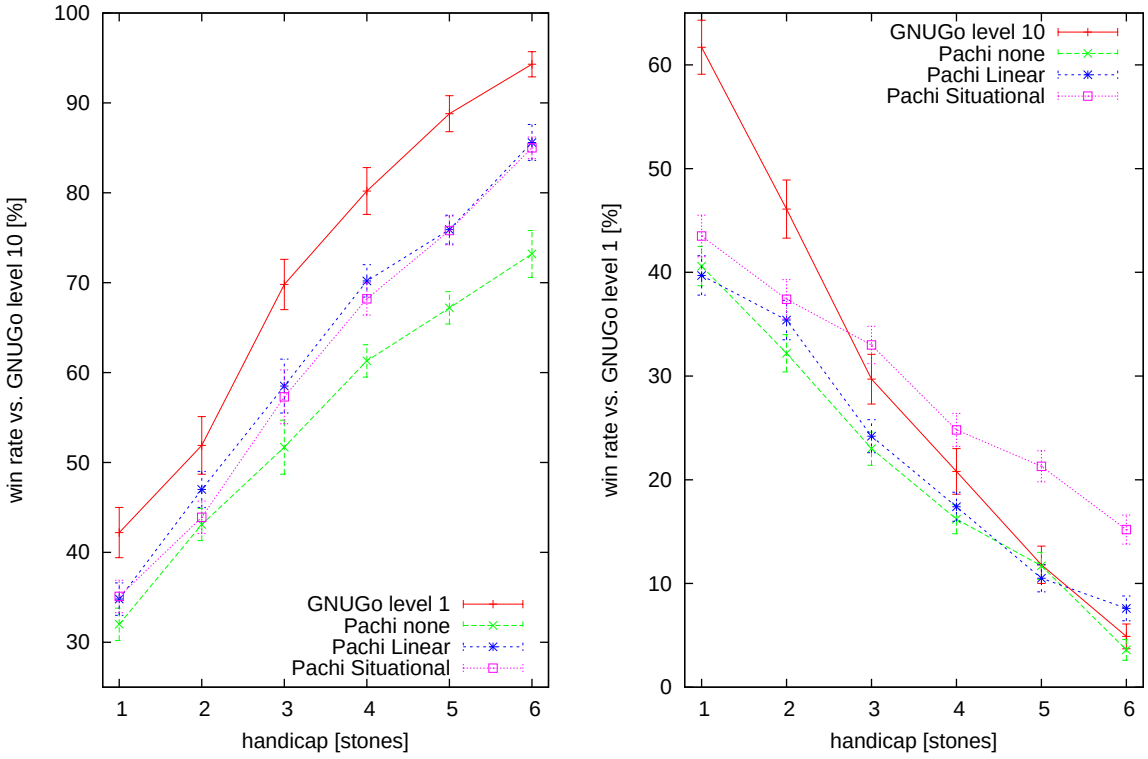
7. CONCLUSION

We have shown that balancing tree search using dynamic komi can be a viable approach to increase the search accuracy and thus the program strength. Non-trivial and statistically meaningful win rate increase has been measured in both handicap and even games, while accounting for the extra time spent on komi calculations. The proposed algorithms are easy to implement and offer many tunable parameters.

We have not researched other approaches to reduce noise and make the MCTS more score-focused; possible future research might investigate ways to co-guide the tree search by expected score difference or estimate and account for expected win probability variance when choosing the move to play. Better functions might be found both for decreasing handicap compensation and situational compensation parameter. The fact that it is best to never expire a ratchet might indicate that the dynamic komi allocation has some hidden structure we have not been able to discern yet.

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

[16] GNU Go is a classical program that does not directly maximize win probability like MCTS. 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.



(a) Playing black (b) Playing white

Figure 2: Dynamic komi in handicap games.

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.

7.1 Acknowledgements

We would like to thank Hiroshi Yamashita and others on the computer-go mailing list for helpful discussions, and Jan Hric and the anonymous reviewers for comments on the paper.

8. REFERENCES

Baudiš, P. et al. (2010). Pachi — Simple Go/Baduk/Weiqi Bot. http://pachi.or.cz/.

Bump, D., Farneback, G., Bayer, A., et al. (2010). GNU Go. http://www.gnu.org/software/gnugo/gnugo.html.

Chaslot, G., Fiter, C., Hoock, J.-B., Rimmel, A., and Teytaud, O. (2010). Adding Expert Knowledge and Exploration in Monte-Carlo Tree Search. *Advances in Computer Games* (eds. H. van den Herik and P. Spronck), Vol. 6048 of *Lecture Notes in Computer Science*, pp. 1–13. Springer Berlin / Heidelberg. http://dx.doi.org/10.1007/978-3-642-12993-3_1.

Dailey, D. (2010). Computer Go Server. http://cgos.boardspace.net/.

Fotland, D. (2010). Message to the computer-go mailing list, Feb 11. http://www.mail-archive.com/computer-go@computer-go.org/msg13470.html.

Gailly, J.-l. (2010). Komi and the value of the first move. http://www.mail-archive.com/computer-go@dvandva.org/msg00096.html.

Gelly, S. and Silver, D. (2008). Achieving master level play in 9x9 computer go. *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, pp. 1537–1540, AAAI Press. ISBN 978–1–57735–368–3.

Gelly, S., Wang, Y., Munos, R., and Teytaud, O. (2006). Modication of UCT with Patterns in Monte-Carlo Go. Research Report RR-6062, INRIA. http://hal.inria.fr/inria-00117266/en/.

Kato, H. (2008). Message to the computer-go mailing list, Feb 28. http://www.mail-archive.com/computer-go@computer-go.org/msg07357.html.

Schubert, W. (2010). KGS Go Server. http://gokgs.net/.

Yamashita, H. (2010). Message to the computer-go mailing list, Feb 11. http://www.mail-archive.com/computer-go@computer-go.org/msg13464.html.