Thesis : Improvements and Evaluation of the Monte-Carlo Tree Search Algorithm

Arpad Rimmel

15/12/2009

Contents

Chapter 1

Introduction

1.1 Motivations

In this thesis, we deal with the problem of taking decisions in a discrete observable uncertain environment with finite horizon. The reward is only given at the end. We focus in particular on problems where the number of states is huge (too large to be explored entirely) . In those cases, traditional methods like alpha-beta (for two player games) or dynamic programming (for control) have been outperformed by some recent techniques, namely the Monte Carlo Tree Search (MCTS). This holds even if the problem is very well formalized like for the game of Go, which is the current main testbed for Monte Carlo Tree Search methods.

For some problems, it is possible to create a function f that associates to every positions after a certain depth D a value $f(pos)$ correlated to the reward you would obtain by following the best path from pos. We call such function an *evaluation function*. A good evaluation function gives values highly correlated to the reward for positions at a low depth. By using such a function, it is possible to reduce the size of the problem by studying it only until the depth D. When this is the case (like in chess), classical methods like alpha-beta give good results. In the following, we will focus on problems where it is not possible (or would take too much time) to construct a good evaluation function.

1.1.1 The Game of Go

The game of Go is an ancient Asian game with the nice property that it is much harder for computers than is chess and is therefore still unsolved. The main reason for this is that is has turned out to be much simpler to devise a good evaluation function in chess than in Go. Whereas nowadays computers play handicap games in chess against top-level humans (the handicap is here for helping the human!), the top human Go players still win very easily handicap games against humans (with the handicap for helping the computer!). In 1998, a 6D human player could win with 29 handicap stones [Müller, 2002] against a top program, Many Faces of Go. However, we will see that the methods originating in [Coulom, 2006, Chaslot et al., 2007, Kocsis and Szepesvari, 2006], improved in this thesis and implemented in the program MoGo have greatly reduced this difference. In 2009, MoGo won a game against a top professional player (9Dan) with only 7 handicap stones. Additional results and analysis of Games from MoGo are given in the appendices.

The game of Go is a two-players (black and white) board game with complete information (see figure 1.1). It is mainly played on two different size of board: 19x19 and 9x9. The time setting of a game correspond to the amount of time given to each player for each move (*byo yomi*) or the amount of time given for the entire game (*sudden death*). A typical time setting for a 19x19 game is 30s per move or 60min for the game.

The rules are the following. Each player plays a move after another. Black plays first. A move consists in putting a stone on the board on an empty intersection. The stones of the same color that are next to each other horizontally or vertically belongs to a *group*. The empty intersections next to a *group* are the *liberties* of this group. If a *group* has no more *liberties*, it is removed from the board. To prevent infinite loop, it is forbidden to make a move that will put the board in a situation where it had already been before. At the end of the game, we count the points for each player. The player with the more points wins the game. The points are counted in the following way:

- each stone on the board gives one point to its owner;
- each empty intersection gives one point to the player with the closest stone;
- the white player has 7,5 more points to compensate the fact that he plays in second.

1.1.2 Other Problems

Games are not the only problems of decision taking in a large search space. Most of planning problems can be described in the same way. In fact, one of the papers introducing MCTS [Kocsis and Szepesvari, 2006] was applied to

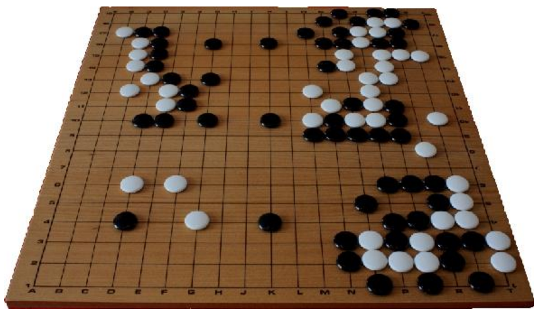


Figure 1.1: Example of a game of go. This game was played by MoGo.

the sailing problem. We will show another successful one-player application of this algorithm in section 5.

1.2 State of the art

In this section, classical methods for tree exploration in the case of one-player games or two-player games are described.

1.2.1 Minimax

Minimax is an algorithm for exploring the tree in the context of a two-player zero-sum game. It supposes the existence of a good evaluation function at a fixed depth D. The algorithm does an exhaustive search to the depth D, one player trying to maximize the evaluation function while the other is trying to minimize it (see algorithm 1 and figure 1.2).

Algorithm 1 : minimax(node,depth)

```
if node is terminal or $depth = 0$ then
  return value of node
else
  let $\alpha = -\infty$
  for each child of node do
    let $\alpha = max(\alpha, -minimax(child, depth - 1))$
  end for
  return $\alpha$
end if
```

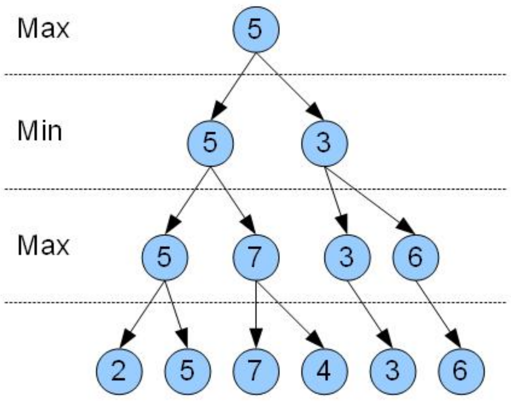


Figure 1.2: tree representation of the minimax algorithm.

This algorithm has been successfully applied in a lot of domains, usually with some modifications: transposition tables [Zobrist, 1990], iterative deepening [Korf, 1985], $MTD(f)$ [Plaat et al., 1996], best-first search [Korf and Chickering, 1994] and SSS* [Stockman, 1979].

This algorithm will return the best answer as it does an exhaustive search. However, the search time will increase exponentially with the depth.

1.2.2 Alpha Beta

AlphaBeta is an improvement of Minimax. It avoids exploring nodes when they can't modify the final result. In order to do that, the algorithm always remember:

- the minimum score the max player is assured of: α
- the maximum score the maximum player is assured of: β.

If β becomes inferior to α, there is no need to explore the situation any further (see algorithm 2 and figure 1.3).

This algorithm returns the same answer as Minimax, however, the number of nodes explored vary depending on the problem. Let b be the branching factor and let D be the depth. Minimax is exhaustive and therefore will explore $O(b^D)$ nodes.

In the worst case, Alphabeta will also explore $O(b^D)$ nodes.

In the best case, the algorithm will explore only $O(\sqrt{b^D})$ nodes, this corresponds to exhaustive search with a branching factor of $\sqrt{b}$.

Algorithm 2 : alphabeta(node,depth,α,β)

```
if node is terminal or $depth = 0$ then
    return value of node
else
    let $\alpha = -\infty$
    for each child of node do
        let $\alpha = max(\alpha, -alphabeta(child, depth - 1, -\beta, -\alpha))$
        if $\beta \leq \alpha$ then
            break
        end if
    end for
    return $\alpha$
end if
```

An other interesting value is the mean number of nodes explored by the algorithm. If we suppose that the repartition of the nodes is random, then Alphabeta will explore $O(b^{3D/4})$ in average and asymptotically, it will explore $O((b/log(b))^D)$.

A more detailed analyze of the Alphabeta algorithm is given in [Pearl, 1984].

When the size of the problem is too big, the search is limited to a certain depth and an evaluation function is used. One limitation of this algorithm is that the maximum depth is the same for all the tree while we would like to explore deeper the area of the tree are promising.

1.2.3 Nested Monte Carlo

This section is based on the article from Tristan Cazenave [Cazenave, 2009]. Nested Monte Carlo search uses nested calls with uniformly distributed simulations: Monte Carlo simulations. This algorithm is recursive. At each step, it evaluates every possible move by playing a game until the end using the lower level of Nested Monte Carlo. The level 0 is a Monte Carlo simulation.

The Nested Monte Carlo algorithm is presented in algorithm 3. It uses the function $monteCarlo(position)$ which plays a game until the end starting from *position* by choosing moves based on a uniform distribution and return the final score.

Tristan Cazenave shows that the results are greatly improved by modifying the algorithm in order to memorize the move associated to the best score of the lower level. As this algorithm explores only a fraction of the search space, it is able to deal with huge problems even if there is no good evaluation

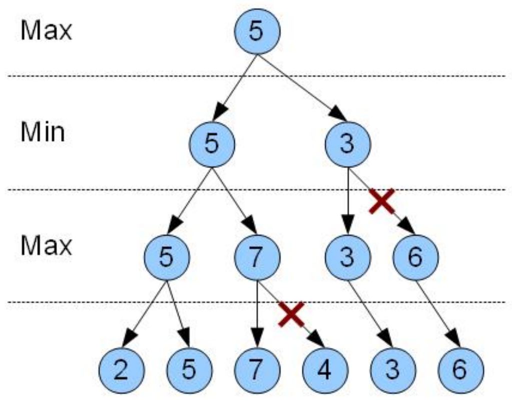


Figure 1.3: tree representation of the Alphabeta algorithm. The branches with a cross are not evaluated.

Algorithm 3 : nested(position,level)

```
while not end of the game do
  if $level = 1$ then
    $move = argmax_m(monteCarlo(play(position, m)))$
  else
    $move = argmax_m(nested(play(position, m), level - 1))$
  end if
  $position = play(position, move)$
end while
return score
```

function. It is very efficient in the case of one-player game like Samegame and Sudoku. It is also at the origin of a new world record in Morpion Solitaire. The main difference with classical algorithms like Alphabeta is the utilization of Monte Carlo simulations instead of an evaluation function in order to estimate a position. The problem of using an evaluation function is that it gives an uncertain value and that the confidence interval gets larger when the situation is far from a terminal position. A Monte Carlo simulation gives an rough estimation but the quality is independent of the depth. This will allow to achieve better results in all the games where there is a lot of states far from the end of a game. This is illustrated by the difference between the game of Chess and the game of Go. In chess, a situation can changed really fast to a won or a lost position. There are fewer situations far from the end of the game. Classical methods with evaluation functions works well. In the game of Go, on contrary, almost all positions are equally far from the end of the game. In this case, algorithms based on Monte Carlo simulations will give better results.

1.2.4 Dynamic Programming

Dynamic Programming is a method used to optimize the objective function in a planning problem. It is the one-player equivalent of Minimax. It can be applied when the problem possesses the optimal substructure property: when an optimal solution can be constructed from optimal solutions to its subproblems. In this case, the principle of Dynamic Programming is to recursively decompose the problem, solve the subproblems and construct the global solution. This method has the advantage of being exact when the property is verified.

An example of problem where this algorithm is efficient is the problem of finding the shortest path in a graph, where each arrow is associated to a distance. An example of this kind of problem is given on figure 1.4. If the problem is in such a way that the nodes of the graph are in n layers and that the path has to go from the first layer to the nth layer, then it possesses the substructure property: you can solve the problem recursively for each layer.

Let N^i be a node from the layer i. Let $f(N)$ be the function that associates the length of the shortest path starting from N to the node N. Let $l(N, M)$ be the distance between the nodes N and M.

$$f(N^i) = \min_{M \ in \ layer \ i+1} l(N^i, M^{i+1}) + f(M^{i+1})$$

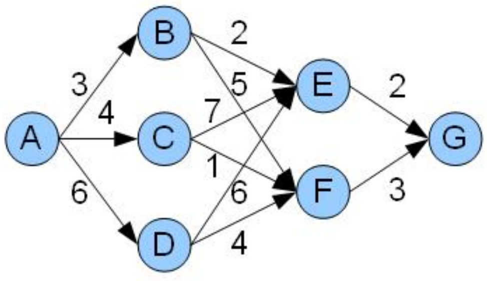


Figure 1.4: An example of a problem where the goal is to find the shortest path from A to G. The numbers on the arrows represent the distance. The nodes are arranged in 4 layers.

We can apply this formula to the problem from figure 1.4. There is 4 layers. Let's start with the layer 3.

- $f(E) = 2$
- $f(F) = 3$

Now we use the formula for the layer 2.

- $f(B) = minimum\ of$
  - $l(B, E) + f(E) = 2 + 2 = 4$
  - $l(B, F) + f(F) = 5 + 3 = 8$

  $\Rightarrow f(B) = 4$
- $f(C) = minimum\ of$
  - $l(C, E) + f(E) = 7 + 2 = 9$
  - $l(C, F) + f(F) = 1 + 3 = 4$

  $\Rightarrow f(C) = 4$
- $f(D) = minimum\ of$
  - $l(D, E) + f(E) = 6 + 2 = 8$
  - $l(D, F) + f(F) = 4 + 3 = 7$

  $\Rightarrow f(D) = 7$

And finally we apply the formula to the layer 1.

- $f(A) = minimum\ of$
  - $l(A, B) + f(B) = 3 + 4 = 7$
  - $l(A, C) + f(C) = 4 + 4 = 8$
  - $l(A, D) + f(D) = 6 + 7 = 13$

  $\Rightarrow f(A) = 7$

The shortest path from A to G is $ABEG$ and is of length 7.

This method is the state of the art in a lot of domains like in revenue management [Quante et al., 2009] and in energy management [Korpaas et al., 2003, Siu et al., 2001], when the dimension of the search space is not too large. It is often combined with approximation techniques for improved performance [Bertsekas, 2009].

However, the algorithm is not anytime; if stopped before then, there is no result.

1.3 Bandit Based Monte Carlo Tree Search

All the algorithms presented in the previous part have the drawback that they don't work very well in the cases where the tree is really large and there is no good evaluation function. In this section, we will introduce a recent exploration tree algorithm: Bandit based Monte Carlo Tree Search (BBMCTS). It achieves very good results in the case of the game of Go. This algorithm will be the main topic of this thesis.

The principle of this algorithm is based on the construction of a highly unbalanced tree. The nodes represent situations and the branches represent decisions. The nodes are evaluated by Monte Carlo simulations. The growth of the tree is directed by a bandit formula. A very efficient improvement of the algorithm is Rapid Action Value Estimation (RAVE).

1.3.1 Monte Carlo Simulations

Monte Carlo methods have been used for a very long time(see [Metropolis and Ulam, 1949]) and they are still used today in a lot of domains like physical sciences (see [DP Landau, 2005]) and finance (see [P Boyle, 1997]). The principle is to use several uniformly distributed samplings to explore the space, when it is too large to be explored entirely. For example, a Monte Carlo method can be used to compute an estimation of the value of π (see figure 1.5 and algorithm 4). The principle is to draw a quarter of circle in a square of size 1. Then we randomly select a points inside the square. An approximation of π is given by multiplying by 4 the proportion of points in the square in comparison to the total number of points.

The Monte Carlo method has been adapted for decisions-taking problems. The principle is to take decisions uniformly at each step until a situation that can be directly evaluated is reached. This is called a Monte Carlo simulation. This simulation will take bad decisions but *uniformly*. By using a large

Algorithm 4 : approximation of π by using a Monte Carlo method

```
$nbtotalpoints = 0$
$nbpointsinsquare = 0$
while there is some time left do
    $x = random(0, 1)$
    $y = random(0, 1)$
    $nbtotalpoints = nbtotalpoints + 1$
    if $\sqrt{x^2 + y^2} \leq 1$ then
        $nbpointsinsquare = nbpointsinsquare + 1$
    end if
end while
$\pi \approx 4 * \frac{nbpointsinsquare}{nbtotalpoints}$
```

number of those simulations, one can have an estimation of the evaluation of the initial situation.

This has been used for the game of Go in [Bruegmann, 1993] and [Bouzy and Helmstetter, 2003]. In this case, a Monte Carlo simulation consists in playing random moves from the current position until the game is finished (see figure 1.6). Then the result of the game (win or loss) is the result of the simulation.

The underlying assumption is that the space explored by taking decisions uniformly is not biased in comparison to the space explored by taking good decisions. This assumption is not always true. In section 4.2.2, we give example of ways to modify those Monte Carlo simulations in order to make them more efficient. A detailed description of Monte Carlo algorithms is given in [Krauth, 2006].

1.3.2 Bandits

A classical k-armed bandit problem is defined as follows (see figure 1.7):

- A finite set $A = \{1, \dots, k\}$ of arms is given.
- Each arm $a \in A$ is associated to an unknown random variable X_a with an unknown expectation μ_a.
- At each time step $t \in \{1, 2, \dots\}$, the algorithm chooses $a_t \in A$ depending on $(a_1, \dots, a_{t-1})$ and $(r_1, \dots, r_{t-1})$.
- The bandit gives a reward r_t, which is a realization of X_{a_t}.

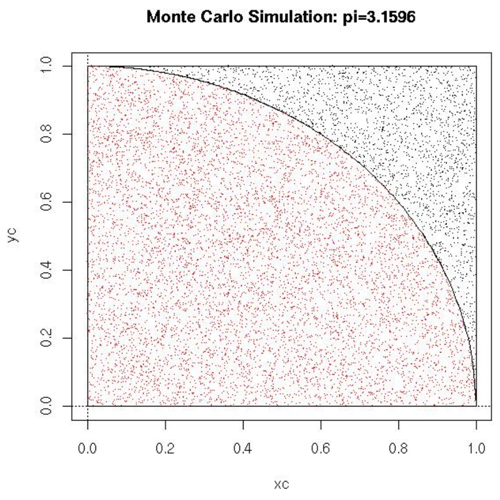


Figure 1.5: illustration of the Monte Carlo method used to compute an approximation of π. Image from the website of Vincent Zoonekynd (*http://zoonek2.free.fr/UNIX/48_R/16.html#2*).

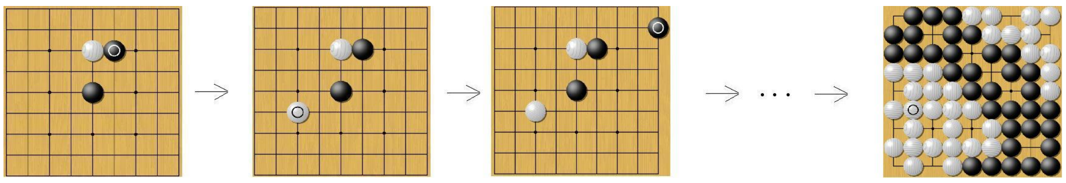


Figure 1.6: illustration of a Monte Carlo simulation for the game of Go.

The goal of the problem is to maximize the cumulative reward at each time step.

A concrete example of the k-armed bandit problem is in a casino. You have k slot machines. Each machine have an unknown reward function based on a random variable. You have a certain amount of coins. Your goal is for each coin to chose to next machine in order to maximize the sum of your gains.

Maximizing the cumulative reward is equivalent to minimizing the so-called *regret*: the loss due to the fact that the algorithm will not always chose the best arm.

Let $T_a(n)$ the number of times an arm has been selected during the first n steps. The *regret* after n steps is defined by

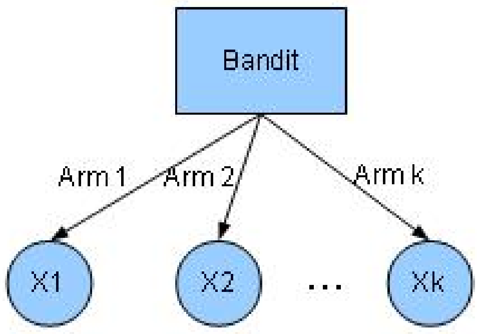


Figure 1.7: illustration of the k-armed bandit problem.

$$\mu^* n - \sum_{j=1}^{k} \mu_j \mathbb{E}[T_j(n)] \text{ where } \mu^* = \max_{1 \leq i \leq k} \mu_i$$

As the other terms of the formula are fixed, the only way to minimize the *regret* is to minimize $\mathbb{E}[T_j(n)]$.

In the paper [Lai and Robbins, 1985], Lai and Robbins give an inferior bound on the regret by proving that, for any strategy and for any suboptimal arm j:

$$\mathbb{E}[T_j(n)] \geq \frac{ln(n)}{D(p_j||p^*)} \text{ where } D(p_j||p^*) = \int p_j ln(\frac{p_j}{p^*})$$

p_j is the reward density if the machine j.

The $D(p_j||p^*)$ are constants that depend on the problem. This formula means that the best regret possible asymptotically is in $O(ln(n))$.

Auer & al achieve such a logarithmic regret uniformly over time in [Auer et al., 2002] with the following algorithm: first, tries one time each arm; then, at each step, plays the arm j that maximizes

$$\bar{x_j} + \sqrt{\frac{2ln(n)}{n_j}} \tag{1.1}$$

$\bar{x_j}$ is the average reward for the arm j.

n_j is the number of times the arm j has been selected so far.

n is the overall number of trials so far.

This formula is chosen at each step the arm that has the highest upper confidence bound. It is called the UCB formula.

To give an intuition of the meaning of this last formula, we can look at both parts of the sum separately.

The first part $\bar{x_j}$ has a high value when the arm j have given good rewards so far. This corresponds to the fact that we want to play often the good arms. This is the *exploitation* part.

The second part $\sqrt{\frac{2ln(n)}{n_j}}$ has a high value when the arm j has not been tested often in comparison to the other arms. This corresponds to the fact that we want to verify that we have not missed a good arm. This is the *exploration* part.

1.3.3 Monte Carlo Tree Search

The idea of the MCTS algorithm is to construct in memory a subtree $\hat{T}$ of the global tree T representing the problem in its whole (see algorithm 5 and figure 1.8). It was introduced by Kocsis and Szepesvari in [Kocsis and Szepesvari, 2006].

The construction of the tree is done by the repetition while there is some time left of 3 successive steps:

1. descent,
2. evaluation,
3. growth.

Descent

The descent in $\hat{T}$ is done by considering that taking decision (choosing a branch to follow) is a k-armed bandit problem. We use the formula 1.1 to solve this problem. In order to do that, we suppose that the necessary informations is stored for each node. Once a new node has been reached, we just repeat the same principle until we reached a situation S outside of $\hat{T}$.

Evaluation

Now that we have reached S and that we are outside of $\hat{T}$, there is no more information available to take a decision. As we are not at a leaf of T, we can not directly evaluate S. Instead, we use a Monte Carlo simulation (see section 1.3.1) to have a value for S.

Algorithm 5 MCTS(s) //s a situation

```
$\hat{T} = ()$
$info = ()$ //$info$ associates statistical information to a node
while there is some time left do
  $s' = s$
  $game = ()$
  //DESCENT//
  while $s'$ in $\hat{T}$ and $s'$ not terminal do
    $s'$ = reachable situation chosen according to the UCB formula (1.1)
    $game = game + s'$
  end while
  $S = s'$
  //EVALUATION//
  while $s'$ is not terminal do
    $s'$ = random reachable situation
  end while
  $result = result(s')$
  //GROWTH
  $\hat{T} = \hat{T} + S$
  for each $s$ in $game$ do
    $info(s) = update(info(s), result)$
  end for
end while
```

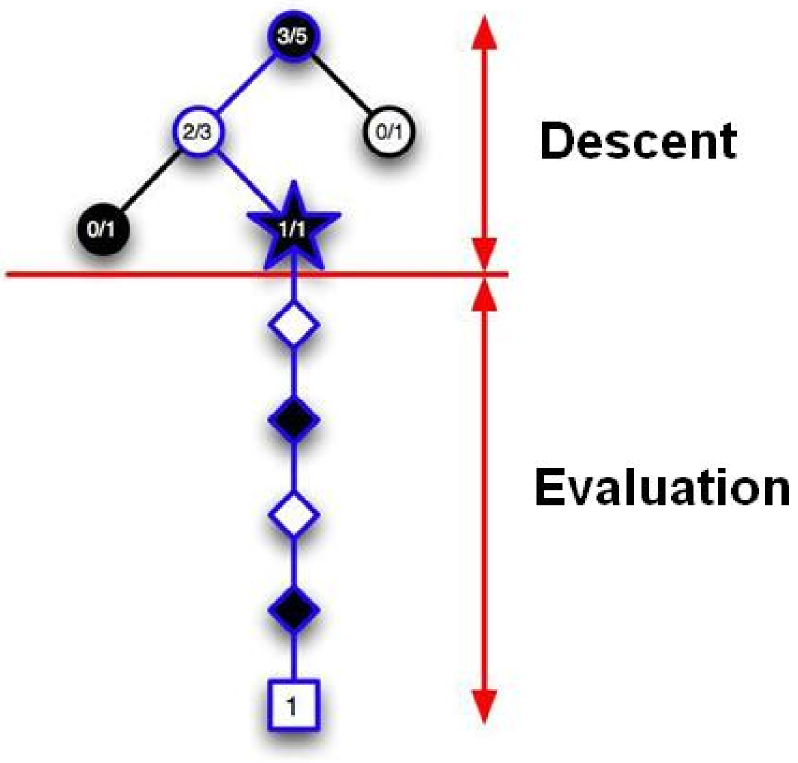


Figure 1.8: Illustration of the Monte Carlo Tree Search algorithm.

Growth

We add the node S to $\hat{T}$[1]. We update the information of S and of all the situations encountered during the descent with the value obtained with the Monte Carlo evaluation.

1.3.4 Classical Improvements

Since the creation of this algorithm, several improvements have been proposed.

Improvements of the bandit formula

The UCB formula (1.1) proposes a fixed coefficient for the trade-off exploration exploitation. However, it appears that the weight the more efficient

[1]it is possible to add several nodes per growth step or to add on node every few growth steps depending on the problem and the memory limitation.

for the exploration part depends on the application. The formula becomes the following:

$$\bar{x_j} + \alpha\sqrt{\frac{ln(n)}{n_j}} \tag{1.2}$$

$\bar{x_j}$ is the average reward for the arm j.

n_j is the number of times the arm j has been selected so far.

n is the overall number of trials so far.

α is a parameter that controls the trade-off between exploration and exploitation.

The classical way of solving the bandit problem is to first try each arm in order to have information and then use a formula to chose the next one. This is a problem when the number of trials is small in comparison to the number of arms. In the MCTS algorithm, this situation arises often for the nodes near the end of $\hat{T}$. Two solutions have been proposed to solve this problem in [Chaslot et al., 2007]: *progressive unpruning* and *progressive bias*. This methods are described in section 4.1.2.

Improvements of the Monte Carlo simulations

Monte Carlo simulations can be improved by the addition of rules. If a rule matches, then it is applied, if not, the decision is taken uniformly (see algorithm 6). This has been successfully applied for the game of Go with rules like *not filling eye*, *saving group* or *killing group*.

Algorithm 6 Monte Carlo Simulation improved by the addition of rules. $rule(s)$ returns a reachable situation from s.

```
input=$s$
while $s$ is not terminal do
  if $rule(s)$ exists then
    $s = rule(s)$
  else
    $s$ = random reachable situation
  end if
end while
```

Other examples of rules and discussions about the improvement of Monte Carlo simulations are given in section 4.2.2.

Rapid Action Value Estimation (RAVE)

RAVE was introduced in [Gelly and Silver, 2007]. It can be applied if the problem is such that the decision sequences can be transposed. Then, instead of averaging the results of simulations where the arm j has been immediately selected ($\bar{x_j}$ in 1.1), we average the results of simulations where the arm j has been selected at any subsequent time during the simulation: $\bar{R_j}$. This idea is also known as *all moves as first* in the domain of computer Go ([Bruegmann, 1993]). This term will have a low variance very quickly and therefore can be used to guide the exploration of the arms when the number of simulations is still low. However, this term is biased, so we want to use the real average at the limit. The new bandit formula will then be the following one.

$$\bar{x_j} + \alpha\sqrt{\frac{ln(n)}{n_j}} + \beta(n)\bar{R_j} \tag{1.3}$$

$\bar{x_j}$ is the average reward for the arm j.

$\bar{R_j}$ is the average reward of simulations where the arm j has been subsequently selected.

n_j is the number of times the arm j has been selected so far.

n is the overall number of trials so far.

α is a parameter that controls the trade-off between exploration and exploitation.

$\beta(n)$ decreases with n

Chapter 2

Parallelization

2.1 Introduction

This chapter is based on the article [Gelly et al., 2008] written with Sylvain Gelly, Jean-Baptiste Hoock, Olivier Teytaud and Yann Kalemkarian.

The efficiency of BBMCTS algorithms highly depends on the number of descents in the tree. However, the time allowed to take the decision as well as the computational power of a computer are limited. On the other hand, it becomes easier and easier to have access to a very large number of machines. Furthermore, the very principle of BBMCTS algorithms is based on the repetition of a single elementary part: the descent in the tree. Therefore, it is possible to decompose the execution of the code. For those reasons, in this chapter we will focus on the parallelization of the algorithm.

2.1.1 Difficulties

While the descents are similar, they are not entirely independent. Each new descent depends on the results of the previous ones. Even if the parallelization consists only in the repartition of the descents among the machines, it won't be equivalent to the sequential algorithm.

It is possible to directly extend the sequential algorithm to a parallel version by launching the sequential version on each machine and sharing all the information. However, the speed-up of such an algorithm is necessarily limited. This point will be further developed in section 2.2.2.

2.1.2 State of the art

In all this chapter, when we refer to "sharing the information of a node X of the tree between the computation units $C_1, ..., C_k$", we mean "averaging the number of wins and losses of the node X over the computation units $C_1, ..., C_k$". Every affected unit is supposed to have the node X.

Cazenave and Jouandeau in [Cazenave and Jouandeau, 2007] compare three ways of parallelizing the BBMCTS algorithm: single-run parallelization, multiple-runs parallelization and at-the-leaves parallelization.

- The *single-run parallelization* consists in running independently one MCTS on each computation unit. Each unit has its own tree in memory. At the end, the information of the root of each tree(one tree per machine) is shared in order to take the decision.

- The *multiple-runs parallelization* is similar to the single-run parallelization but the information of the root is shared periodically during the process. The period is typically in the order of the second.

- The *at-the-leaves parallelization* consists in replacing the random game at a leaf of the tree with multiple random games run in parallel. There is only one tree stored in the memory of the "master" and the "slaves" are doing the simulations.

They conclude that all the parallelizations give similar results. However, the at-the-leaves parallelization seems to be more efficient in the case of multiple cores on a single machine.

In his article [Chaslot et al., 2008], Guillaume Chaslot and al discuss three parallelization methods for BBMCTS: at-the-leaf parallelization, root parallelization (corresponds to the single-run parallelization) and a new method: the tree parallelization. A summary of those methods is presented on the figure 2.1. The new method uses one shared tree from which several simultaneous games are played. This method needs the memory of the system to be shared and therefore is designed for multi-core. The tests were run on his program Mango. The root parallelization performs very well overall. However, the tree parallelization is the most promising method on small boards. In the following, we will refer to the multi-core parallelization. This method is similar to the tree parallelization. We will show that it achieves very good results.

Hideki Kato and Ikuo Takeuchi propose another way to parallelize the BBMCTS algorithm by using a client-server model in [Kato and Takeuchi, 2008]. This method is a variation of the at-the-leaves parallelization in the sense that there is only one tree stored on the "master" computer and that the "slave" computers are doing Monte Carlo simulations. But instead of launching several Monte Carlo simulations at the same time from one leaf, the algorithm consists in launching only one simulation on

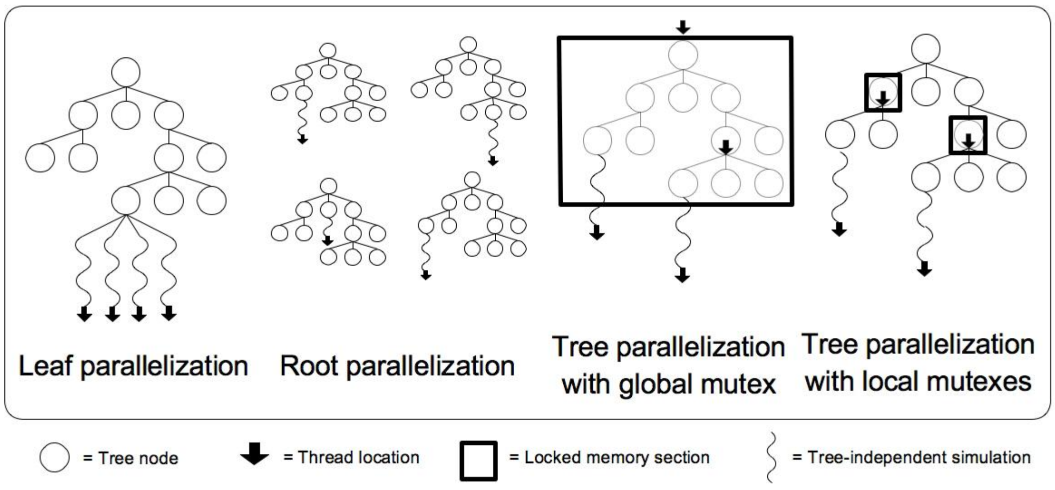

Leaf parallelization
Root parallelization
Tree parallelization with global mutex
Tree parallelization with local mutexes
= Tree node
= Thread location
= Locked memory section
= Tree-independent simulation

Figure 2.1: (a) Leaf parallelization (b) Root parallelization (c) Tree parallelization with global mutex (d) and with local mutexes. Figure from Guillaume Chaslot. Extracted from [Chaslot et al., 2008]

a free ”slave” and then beginning a new descent in the tree. This method allows us to get on-the-fly connection or disconnection of servers. It obtains good results with four computers.

In those previous works, an efficient parallelization has been proposed for the case where the memory is shared: the multi-core parallelization. However, the experiments are done only on a small number of cores. We propose a study of this algorithm when the number of cores becomes large.

For the case where the memory is not shared, no efficient algorithm has been proposed that is efficient on a large number of machines, we propose our own and experiment it in real conditions.

2.2 Contribution

We will first present new results about the tree parallelization (here named multi-core parallelization) after a detailed explanation on its algorithm. Then we will present a new parallelization algorithm: cluster parallelization. This algorithm was created in order to parallelize on several computers (without shared memory).

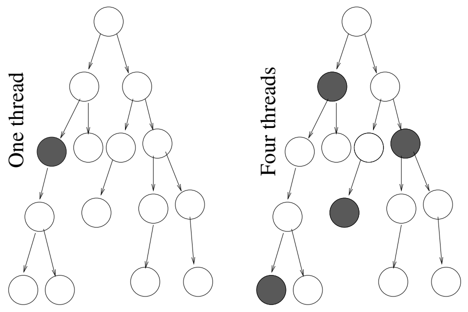


Figure 2.2: Multi-core parallelization. Left: sequential algorithm (one simulation at a time). Right: quad-core case: four simulations at a time, and each simulation updates the tree when it reaches the end of a game. Therefore, the difference with the 4 times accelerated sequential algorithm is that when a simulation is launched at the root, there are 3 updates which have not been performed (because 3 simulations are in progress).

2.2.1 Results on the Multi-Core Parallelization

The multi-core parallelization is intuitively the most natural one: the memory is shared. We just have to distribute the global loop (the succession of the 3 steps, see section 1.3.3) on various threads (each thread performs simulations independently of other threads as in Fig. 2.2, with just mutexes protecting the updates in memory), leading to algorithm 7.

Consider N the number of threads. This algorithm is not equivalent to the sequential one: possibly, $N-1$ simulations are running when one more simulation is launched, and the updates of the tree corresponding to these $N-1$ simulations are not taken into account. There is a $N-1$ delay, and the analysis of this delay is not straightforward - we will quantify this effect experimentally. Table 2.1 presents results that were established with 4-core machines on 9x9 board and table 2.2 presents the results on 16-core machines on 9x9 board.

Table 2.1 confirms the roughly 63% success rate known when doubling the computational power. Table 2.2 shows that the speed up is linear up to 4 cores but the results are not so good with more threads.

In the following, we will compare the speed up in term of number of simulations per second as we can't hope to have a better winning rate if we don't have more simulations. The results are shown on figure 2.3 and 2.4. The runs have been done on a Power-5 of Huygens, a 32-cores computer. The threads above 32 cores are based on the supposition that the hyperthreading is efficient. The speed up is much better on a board of size 19x19. It is

Algorithm 7 Multi-core Monte Carlo planning algorithm.

```
$\hat{T} = ()$
$info = ()$ //$info$ associates statistical information to a node
**for** each thread simultaneously **do**
  **while** there is some time left **do**
    $s' = s$
    $game = ()$
    //DESCENT//
    **while** $s'$ in $\hat{T}$ and $s'$ not terminal **do**
      $s'$ = reachable situation chosen according to the UCB formula (1.1)

      $game = game + s'$
    **end while**
    $S = s'$
    //EVALUATION//
    **while** $s'$ is not terminal **do**
      $s'$ = random reachable situation
    **end while**
    $result = result(s')$
    //GROWTH
    $\hat{T} = \hat{T} + S$
    **for** each $s$ in $game$ **do**
      $info(s) = update(info(s), result)$
    **end for**
  **end while**
**end for**
```

Nb threads $\times$ comp. time	10 sec.procs	20 secs.procs	40 secs.procs
mono- -thread	51.1 $\pm$ 1.8	62.0 $\pm$ 2.2	74.4 $\pm$ 2.4
two threads		62.9 $\pm$ 1.8	
four threads			66.4 $\pm$ 2.1

Table 2.1: Success rate against mogoRelease3, for various computation times on 9x9 board on a 4-cores computer with the multi-core parallelization. We see that a linear speed-up leads to a success rate 74.4 $\pm$ 2.4 %. The improvement with 4 cores is 66.4% $\pm$ 2.1 %. Under the two-assumptions (1) almost N times more simulations per second with N cores (2) no impact of the "delay" point pointed out in the text, the speed-up would be linear and all the raws would be equal. We see a decay of performance for 4 threads.

4s vs 1s	74.4 % $\pm$ 2.4 %
4 cores against 1 core	80.0 % $\pm$ 3.4%
16 cores against 4 cores	69.5% $\pm$ 4.5%

Table 2.2: Success rate against mogoRelease3 on 9x9 board on a Power-5 node of Huygens, a 32-cores computer with the multi-core parallelization. The results are very good with 4 threads but drop when the number of threads increases.

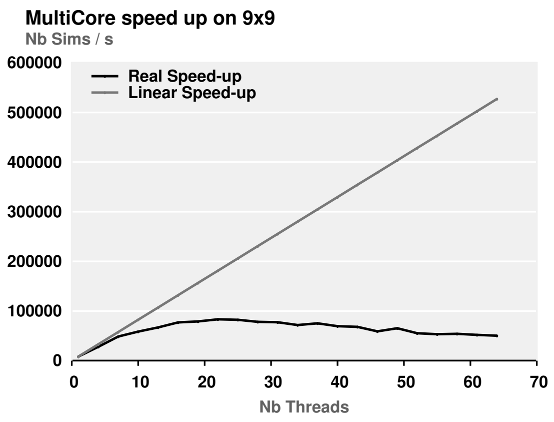


Figure 2.3: Comparison between the linear speed up and the speed up we obtain on 9x9 board with the multi-core parallelization. The speed-up is measured in term of number of simulations. The speed up is almost linear until 8 threads. Then the performances are still going up until 24 cores. After that, adding new threads doesn't improve the number of simulations and can even make it worse.

interesting to note that on 9x9, using too much threads has a negative impact on the performance. Also, the best performance is reached with only 24 cores. The results of the figure 2.4 show that in 19x19, the parallelization is much more efficient when the Monte Carlo simulations are slow. This has to be taken into consideration when analyzing the effect of a modification of the Monte Carlo simulations.

2.2.2 Cluster Parallelization

In this section, we will compare two different solutions:

- the generalization of the multi-core approach to a cluster;
- an alternative solution.

Generalization of the multi-core approach

First, let us consider the generalization of the multi-core approach to a cluster, i.e. a version with massive communications in order to keep exactly the same state of the memory on all nodes. As the memory is not shared here, we have to broadcast on the network many update-information; each simulation

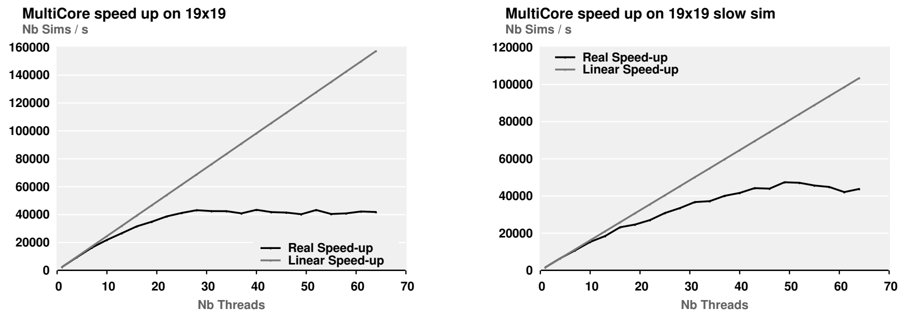


Figure 2.4: Left: comparison between the linear speed up and the speed up we obtain on 19x19 board with the multi-core parallelization. The speed-up is measured in term of number of simulations. The speed up is almost linear until 10 threads. Then the performance are still going up until 28 cores. After that, adding new threads doesn't improve the number of simulations. Right: comparison between the linear speed up and the speed up we obtain on 19x19 board with the multi-core parallelization with slow simulations. The speed-up is measured in term of number of simulations. The speed up is almost linear until 16 threads. Then the performance are still going up until 50 cores.

on one node leads to one broadcast. Possibly, we can group communications in order to reduce latencies. This leads to the algorithm 8.

A time delay of 0 between the sharing of the information is perhaps possible, for high performance clusters or processors inside the same machine. Let's consider the idealized case of a cluster with negligible communication cost (this is a reasonable hypothesis for many case, as the simulations and the updates are expensive, e.g. in computer-Go) and infinite number of nodes. Let's assume that a proportion α of time is spent in the updates. Also, let's assume that the delay of updates does not reduce the overall efficiency of the algorithm. What is the speed-up in that case ?

Consider M the number of simulations per second on one node in the (mono-node) case. With N nodes, at each time steps, we get NM simulations. The number of updates is therefore NM per second of simulation. If the time of one update is T, for each group of M simulations,

- each node performs M simulations (costs $1-\alpha$ second);
- each node sends M update-information (costs 0 second by assumption);
- each node receives $(N-1)M$ update-information (costs 0 second by assumption);

Algorithm 8 Cluster algorithm for Monte Carlo planning. As there are many paths leading to the same node, we must use a hash table, so that with the right key (the goban) we can find if a node is in the tree and what are its statistics in constant time.

```
for Each node do
  $\hat{T} = ()$
  $info = ()$ $//info$ associates statistical information to a node
end for
for each computer simultaneously do
  for each thread simultaneously do
    while there is some time left do
      $s' = s$
      $game = ()$
      $//DESCENT//$
      while $s'$ in $\hat{T}$ and $s'$ not terminal do
        $s'$ = reachable situation chosen according to the UCB formula
        (1.1)
        $game = game + s'$
      end while
      $S = s'$
      $//EVALUATION//$
      while $s'$ is not terminal do
        $s'$ = random reachable situation
        $game = game + s'$
      end while
      $result = result(s')$
      $//GROWTH$
      $\hat{T} = \hat{T} + S$
      for each $s$ in $game$ do
        $info(s) = update(info(s), result)$
      end for
      Add $game$ to a stack of to-be-sent simulations.
      if time delay has passed and this is the first thread then
        Send all the $game$ in the stack to all other nodes.
        Reset the stack.
        Receive many $game$ from all other nodes.
        for Each $game$ received do
          $\hat{T} = \hat{T} + S$(if not present).
          for each $s$ in $game$ do
            $info(s) = update(info(s), result)$
          end for
        end for
      end if
    end while
  end for
end for
```

- each node updates its tree with these $(N-1)M$ update information (costs $\alpha(N-1)$ second).

If we divide by the number N of nodes and let $N \to \infty$, we get

- a cost $(1-\alpha)/N \to 0$ for simulations;
- a cost 0 for sending update-information;
- a cost 0 for receiving update-information;
- a cost $\alpha(N-1)/N \to \alpha$ for updates.

This implies that the main cost is the update-cost, and that asymptotically, the speed-up is $1/\alpha$. This limitation of the overall system when only part of the system is improved is called the Amdahl's law [Amdahl, 1967]. In the case of MoGo, this leads to $\alpha \simeq 0.05$ and therefore roughly 20 as maximal speed-up for the case of a tree simultaneously updated on all nodes. As communications are far from negligible, as preliminary experiments were disappointing and as we expect better than a speed-up of 20, we will not keep this algorithm in the sequel.

An alternate solution with less communications

Whenever communications are perfect, the speed-up[1] of the approach above is limited to some constant $1/\alpha$, roughly 20 in MoGo. We propose the following algorithm (Algorithm 9), with the following advantages:

1. much less communications (can run on usual Ethernet);
2. tolerant to inhomogeneous nodes (as other algorithms above also);
3. our implementation is not yet fault-tolerant, but it could be done;
4. self-assessment possible (in the case of the game of Go, an amount of time is given for the whole game. It is therefore possible to spend more or less time on some decisions. If we had a distance between trees, we could allow more time for one particular move if, when doing the sharing, we realize that the trees from the different computers are far from each other.).

The algorithm is detailed in Algorithm 9. The function *Share*() is called 20 times per second in our implementation.

[1]This is not a “real” speed-up, as the parallel algorithm, even in the multi-core case, is not equivalent to the sequential one - the difference is the “delay” detailed in section 2.2.1. We here consider that the speed-up is k if the parallel algorithm is as efficient as the sequential one with k times more time.

Algorithm 9 The "share" algorithm. The MPI_ALL_REDUCE has a cost logarithmic in the number of nodes. B has been empirically set to 10, α to 0.05.

```
  Begin of function Share($node$)
  Let $n$ be the total number of simulations at the root of the tree.
  Let $x_i$ be the number of simulation of the $i^{th}$ son of $node$.
  MPI_ALL_REDUCE($x$,sum), i.e. $x$ is replaced by its sum over all nodes.
  If $x_i > \alpha n$ and depth($node_i$)$< B$, then Share($node_i$) with $node_i$ the $i^{th}$ son of $node$.
  End of function.
```

Configuration of game	Winning rate in 9x9	Winning rate in 19x19
32 against 1	75.85 ± 2.49 %	95.10±01.37
32 against 2	66.30 ± 2.82 %	82.38±02.74
32 against 4	62.63 ± 2.88 %	73.49±03.42
32 against 8	59.64 ± 2.93 %	63.07±04.23
32 against 16	52.00 ± 3.01 %	63.15±05.53
32 against 32	48.91 ± 3.00 %	48.00±09.99

Figure 2.5: Experiments showing the speed-up of "slow-tree parallelization" in 9x9 and 19x19 Go. Experiments were reproduced with different parametrizations without strong difference; in this table, the delay between two calls to the "share" functions is 0.05s, and x is set to 5%.

An important advantage of this technique is that averaging vectors of statistics is possible quickly on a large number of nodes: the computational cost of this averaging over N nodes is $O(\log(N))$.

Using the function $Share()$ only at the end of the thinking time (no communication before the final step of the decision making) is just a form of root parallelization. For computer-go, we get 59 %± 3% of success rate with an averaging over 43 machines versus a single machine, whereas a speed-up 2 leads to 63%. This means that averaging provides a speed-up less than 2 with 43 machines; this is not a good parallelization.

We present in figure 2.5 the very good results we have in 19x19 and the moderately good results we have in 9x9.

We see that a plateau is reached somewhere between 8 and 16 machines in 9x9, whereas the improvement is regular in 19x19 and consistent with a linear speed-up - a 63% success rate is equivalent to a speed-up 2, therefore the results still show a speed-up 2 between 16 and 32 machines in 19x19.

2.3 Conclusion

A main advantage of the approach is that it is fully scalable. Whereas many expert-based tools have roughly the same efficiency when doubling the computational power, bandit-based Monte Carlo planning with time $2t$ has success rate roughly 63% against a bandit-based Monte Carlo planning algorithm with time t. This leads to the hope of designing a parallel platform, for an algorithm that would be efficient in various tasks of planifications.

The main results are the followings:

- Doubling the computational power (doubling the time per move) leads to a 63% success rate against the non-doubled version.
- The straightforward parallelization on a cluster (imitating the multi-core case by updating continuously the trees in each node so that the memory is roughly the same in all nodes) does not work in practice and has strong theoretical limitations, even if all communication costs are neglected.
- A simple algorithm, based on averages which are easily computable with classical message passing libraries, a few times per second, can lead to a great successes; in 19x19, we have reached, with 32 nodes, 95 % success rate against one equivalent node. This success rate is far above simple voting schemas, suggesting that communications between independent randomized agents are important and that communicating only at the very end is not enough.
- Our two parallelizations (multi-core and cluster) are orthogonal, in the sense that:
  - the multi-core parallelization is based on a faster breadth-first exploration (the different cores are analyzing the same tree and go through almost the same path in the tree; in spite of many trials, we have no improvement by introducing deterministic or random diversification in the different threads.
  - the cluster parallelization is based on sharing statistics guiding the first levels only of the tree, leading to a natural form of load balancing. The deep exploration of nodes is completely orthogonal.

  Moreover, the results are cumulative; we see the same speed-up for the cluster parallelization with multi-threaded versions of the code or mono-thread versions.

- In 9x9 Go, we have roughly linear speed-up until 8 cores and 9 nodes. Beyond 24 cores, each new thread has a negative effect.
- In 19x19 Go, the speed-up remains linear until 16 cores and at least 32 machines. Beyond 50 cores, each new thread has a negative effect.
- Using slow simulations improves a lot the speed up.

Chapter 3

Opening Database

3.1 Introduction

This chapter is based on the article [Audouard et al., 2009] written with Pierre Audouard, Guillaume Chaslot, Jean-Baptiste Hoock, Julien Perez and Olivier Teytaud.

In the case of planification tasks, the beginning is the most difficult part for a BBMCTS algorithm. As a matter of fact, this is the time where the number of possibilities is the largest, so the algorithm needs more simulations to find the best path. It is also the moment where the end is the furthest away, so the result of the Monte Carlo simulation will be less correlated with the leaf of the tree it originates in.

However, for the first few steps, it often happens that the situation at hand has already been encountered by the algorithm in a previous run. By using an Opening Book (OB), we could take advantage of that fact by spending a lot of time off-line to compute the best move in those situations.

Finally, in the case of go, it exists a lot of human knowledge about opening; this knowledge is however difficult to formalize.

3.1.1 Difficulties

Complexity of OB in Go.

Handcrafting OB in Go is quite hard because the usual knowledge is not encoded as a mapping *situation* $\rightarrow$ *answer* but much more as a set of local rules (termed Joseki), to be applied only after consideration of the global situation of the board. Also, a well known proverb in Go says "Learn Joseki, loose two stones" (LJLTS): when a player learns these rules, he becomes weaker in a first time. The simple explanation to this fact is that local context in Go must always be related to the global situation of the board;

so, blindly applying newly learned opening sequences leads to likely mistakes concerning the global context of the game known as "errors in the direction of play". Indeed, choosing between several opening sequences depending on the global context is one of the hardest skill to acquire in the game of go, because the amount of experience required to handle the extraordinary degree of freedom at the beginning of the game.

In the case of 9x9 Go, efficient computer players are very recent and there's no positive result on 9x9 Go OB. Also, as these recent (and strong) players are mainly best-first search, i.e. focus on a small number of moves and not on all moves, it is not clear whether they can be used for building OB. As a matter of fact, you need some diversity in order to build an OB. We will here present an efficient approach. This result, besides its positive efficiency, is also interesting for understanding how best-first search algorithms can be efficient for games in spite of the fact that they only study a small set of moves (without efficient expert pruning).

Best first research in games.

One main element in games is to take into account the branching factor (number of possible decisions in each situation). More than the incredible number of legal situations [Tromp and Farnebäck, 2006], the trouble in Go is that pruning rules are quite inefficient and therefore the "practical" branching factor is not much less than the "theoretical" branching factor. This is in particular true for building OB. One way of doing it is to ask a professional player for the correct answer in some positions. But requesting suggestions from professional players is quite difficult when the number of possible situations is huge as in the case of Go. How do we handle this problem? A naive approach would require 1 request to the professional player for the first move, 81 for the second move, and so on. How to reduce this combinatorial explosion? In the same way that MCTS is efficient in front of the combinatorial explosion of Go, we show here that a meta-level of MCTS (MCTS on top of MCTS, see below) is efficient against the combinatorial explosion of OB.

MCTS leads to a "best-first" approach[1] in the sense that when we build a tree of future situations, a move is further analyzed insofar as it is the move with best success rate in the simulations. Is this reasonable for OB? The experimental answer given in this chapter is positive, and we justify it

[1]MCTS has often been emphasized as a trade-off between exploration and exploitation, but many practitioners have seen that in the case of deterministic games, the exploration constant, when optimized, is zero or indistinguishable from zero, leading to a best-first approach.

as follows: (i) assume that for a fixed depth d (i.e. there are d moves at most before the end), the estimate of the probability of winning converges respectively to 1 or 0, depending on whether this situation is a winning position or not; (ii) then for a situation with depth $d+1$, the move with best success rate will be evaluated more often than the others. If this is a losing move, its success rate will converge to 0. As all the success rate are between 0 and 1, this move will not stay the one with best success rate, and this will ensure diversification. Of course, this is quite "sequential": no alternate move is explored if the first explored move is still at a high success rate in spite of the fact that it is not the best move. But MCTS has shown that with a strong enough computational power, this approach is in fact very reliable and better than classical approaches; we here show that it is also reliable in the difficult framework of OB in spite of the intuitive diversity requirement.

3.1.2 State of the art

Michael Buro presents in [Buro, 2001] a generic way to automatically generate OB for various board games. This algorithm guaranties that you won't lose two games in the exact same way: if you lost one game and the opponent plays the same way for a new game, you will change one move.

The algorithm supposes the existence of a heuristic function H able to associate a value to every situation. A good heuristic will associate to a position a value highly correlated to the winning chance of the position.

The algorithm is based on a tree constructed from all previous games (see figure 3.1). The root is the beginning of the game. Each branch represents a move played in a previous game and each node represents the situation reached after playing the move. The leaves are the end of the games and therefore have a result: win or loss (or draw for some games). We associate the value $-\infty$ to losing leaves and $+\infty$ to winning leaves.

Now, for each non-terminal node N, we will create a new child. This child corresponds to a situation never played before, reachable from N and with the best value according to H. This node corresponds to the best variation from N. We associate the value given by H to this leaf.

Finally, we use an Alphabeta algorithm to determine which move to play. As the values given by the heuristic are different from $-\infty$, the algorithm won't choose a path that lead to a previous loss.

The problem with this algorithm is that it assumes that the opponent has the same information as the player and the algorithm will propose desperate moves in situations that could have been good if the opponent had been less informed.

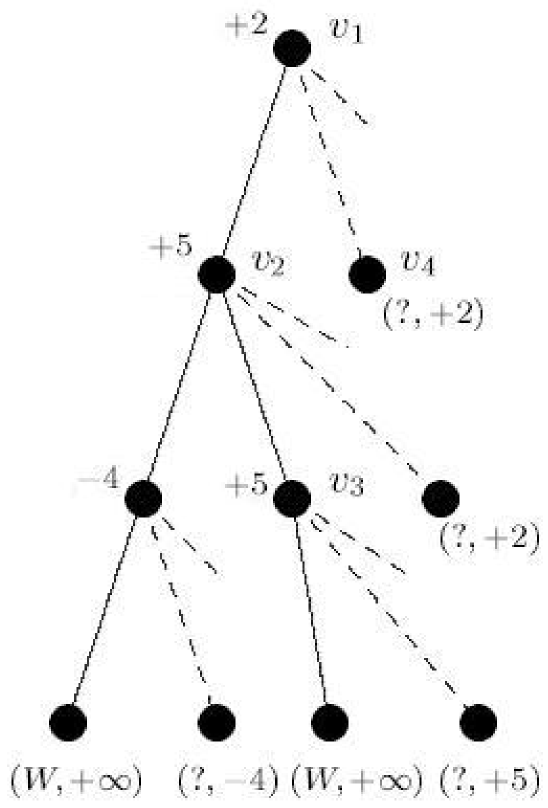


Figure 3.1: Example of opening book tree. Plain lines represent move from previous games, doted lines represent best variations. The move $v4$ will be selected next. Extracted from [Buro, 2001].

Donninger and Lorentz in [Donninger and Lorenz, 2006] propose a way of constructing an OB from a collection of games. It is supposed to have information describing each game.

The classical way of using such a collection of games is to compute the percentage of win for each situation and then play the move with the highest score. They propose instead to first remove non relevant games of the OB by applying filters (for example, remove every game with the word “blitz”). Then they use a formula based on a linear combination of the winning percentage and other term depending on the information about the games to give a value to each move.

For example, in the case of chess, they propose a linear combination of the following term:

- winning percentage
- draw percentage
- winning percentage of top games (games played by top professional)
- draw percentage of top games
- number of times the move has been played in top games
- number of times the move has been played in recent games (after 1.1.2000)
- number of times the move has been played by aggressive players

The weight of each term are tuned by experiments. They apply this method to the top chess program Hydra. They compare the effect of this OB against the chess program Shredder and they obtain a difference of 70 elo between the version with OB and the version without OB.

The main drawbacks of this method is that it supposes the existence of a large number of games played by very good players and with additional information. Also, once the optimal weights have been found, this algorithm does not propose a way to improve the OB further.

3.2 Contribution

We will present two new algorithms to automatically generate an Opening Book (OB): **mutate bad moves (MBM)** and **mutate very bad moves (MVBM)**. Then we will present the results obtained by using the two resulting OB in MoGo.

3.2.1 Algorithm

The goal is the incremental building of a set of games (an OB), supposed to be used by a (human or computer) player as follows: *If the current situation is in the set of games, choose the most frequent move in won games.*

We choose the most simulated move in won games, instead of the move with highest success rate - there are far less erroneous moves with the most simulated move in won games. This is consistent with the MCTS literature (see e.g. [Wang and Gelly, 2007]). We use self-play as a basic tool for building OB, as in [Nagashima et al., 2006] (for computer-Shogi). Precisely, while looking for a methodology for building an OB with coevolution, we understood that MCTS is precisely a tool for building OB, usually applied to a naive random player and with small time settings, and that what we need is essentially MCTS with however

1. large time settings

2. a good computer player

In this chapter, the good player is itself the original MCTS.

We first present two non-evolutionary methods for building OB: the expert OB and the 4H-OB. The best one will serve as a reference for comparison with the two new automatic methods we propose after.

Expert OB

A handcrafted set of games played by top level players and optimized manually (even pro games can be optimized off line by experts). This method gives good results and will be used for comparison.

4H-OB:

A version of MoGo spending 4 hours studying each move plays against a MoGo spending 1s per move. After 4 moves, the MoGo playing 1s per move resigns. These two MoGos use no OB and are not adaptive (no learning). The idea was that MoGo with 1s per move will have a lot of diversity and will explore a lot of different moves. The MoGo plays the role of the expert. However, the 4H-OB was weaker than the expert OB. In fact, 4 hours is too close to the time used in real games and we don't have enough computing power to generate an OB with more time per side. This OB won't be used in the rest of this chapter.

Now we will present our two new algorithms based on coevolution. Coevolution has already been used for building partial strategies in Chinese chess [Ong et al., 2007] and also in the game of Go [Drake and Chen, 2008], but with less impressive results. Our coevolutionary algorithm is "MCTS applied to MoGo-black and MoGo-white", as well as MoGo is "MCTS applied to a naive random black player and a naive random white player" (not "very" naive; see [Wang and Gelly, 2007] for details). Our co-evolutionary techniques for building OB are as follows, starting from a part of the handcrafted OB (the complete version was not available at that time).

Mutate bad moves (MBM, Algo. 10):

MoGo plays against itself (each player has 6h per side on a quad-core, roughly equivalent to 24h per side as the speed-up of MCTS for 4 cores is very good). The two MoGo use the OB and choose the move with highest success rate if at least one move with success rate > 50 % is available. Otherwise, the standard MoGo algorithm is used.

Mutate very bad moves (MVBM):

Mogo plays against itself (each player has 6h per side). The two MoGo use the OB and choose the move with highest success rate if at least one move with success rate > 10 % is available. Otherwise, the standard MoGo algorithm is used.

MVBM was introduced because, with MBM, there were too many cases in which black did not follow any OB at all because the success rate of black was lower than 50 %.

Both algorithms are a coevolution, in which an individual is a game (the games won by black (resp. white) are the black (resp. white) population; each won (resp. lost) game is a good (resp. bad) partial strategy), and as in the Parisian approach [Collet et al., 2000], the solution (the OB) is made of all the population. This algorithm is used on a grid (www.grid5000.fr, a large grid provided freely for scientific experimentation). This introduces some constraints: λ is not known in advance and not constant (λ depends on the number of available CPUs, and jobs might be preempted). In order to have preliminary results, we first run MBM on a simplified setting (see section 3.2.1). We see first that the depth-first approach works fine, which is a great success as building OB in Go is quite difficult; these results, quite good for white, are based on MBM. We will see however that this approach is not satisfactory for black which lead us to derive the MVBM approach.

Algorithm 10 The "mutate bad move" (MBM) algorithm. λ is the number of machines available. The random choice is performed by MoGo with long time settings, therefore it is quite expensive. Readers familiar with UCT/MCTS might point out that there is no exploration term in the version of MCTS presented here; however, this is the case in many successful MCTS implementations, and other implementations often have a very small constant which is equivalent, in fact, to 0.

```
$Population$ = small handcrafted OB.
while True do
    for $l = 1..\lambda$, generate one individual (a game) in parallel with two steps as follows do
        $s$ =initial state; $g = (s)$.
        while $s$ is not a final state do
            $bestScore = 0.5$
            $bestMove = Not\ A\ Move$
            for $m$ in the set of possible moves in $s$ do
                $score$ =frequency of won games in $Population$ with move $m$ in $s$
                if $score > bestScore$ then
                    $bestScore = score$
                    $bestMove = m$
                end if
            end for
            if $bestMove = Not\ A\ Move$ then
                Mutation: $bestMove = RandomChoice(s)$.
            end if
            $s = nextState(s, bestMove)$ (transition operator)
            $g = concat(g, s)$
        end while
        $Population = Population \cup \{g\}$
    end for
end while
```

Why we should mutate very bad moves only

We here experiment the MBM approach, on a simplified case of 10 seconds per move, 8-cores machine; in this easier framework we could generate 3496 games. The results follow:

Conditions	Before learning	After learning	Against handcrafted OB
Success rate (white)	51.5 % ± 1.8 %	64.3 % ± 1.9 %	64.1 % ± 1.8 %
Success rate (black)	48.5 % ± 1.8 %	48.0 % ± 1.9 %	46.1 % ± 1.8 %

The first column sums to 100 % as it is the success rate of white (resp. black) in the case of no learning. The second column shows the results for white (resp. black) after learning against a non-learning version: the sum is higher than one as MBM did a good job on average. The third column shows that the learning has provided better results than the handcrafted OB. However, we see no progress for black. This can be explained as follows: as long as black has not found a move with success rate $> 50\%$, he always mutates. Therefore, white improves his results by choosing moves with higher success rates, but not black. This is why in next sections, we will use the "mutate very bad moves" approach - asymptotically it is probably equivalent, but non-asymptotically it's better to use an approach that differentiates situations where the success rate is below 50%(e.g. 40% of success rate is better than 30% of success rate) instead of repeatedly mutating in order to find at least 50 %.

Robustness of the Opening Books

These results above look quite promising, but we then tested what happens if we use this learned OB in a stronger version of MoGo (i.e. with larger time settings), using 60s per move instead of 10s (still on 8-cores machines).

Success rate of the handcrafted OB against no OB for 60 s per move: 50 % as black, 63 % as white. Clearly the handcrafted OB still works for this stronger MoGo.

Success rate of the learned OB (learned as above, with 10s per move games) against handcrafted OB: 30.6 % as black, 40.7 % as white. The learned OB, learned for 10s per move, is definitely not sufficiently strong for 60s per move. These numbers show that a weak player can't estimate efficiently an opening sequence, whenever this weak player performs hundreds of games - the opening sequence might be good for this weak-player, but this opening sequence will become a handicap when the player will become stronger.

3.2.2 Validation

In this section we present the results of the real-size experiments on MVBM. Each game is played with 6 hours per side on a quad-core. 3000 games are played, i.e. the final population has 3000 individuals. MVBM is "Parisian": all the population is the solution. Also, there's no removal: the population is incrementally increasing, and no individual is never eliminated (it might just become negligible). The handcrafted OB used in this section is an improved version of the one used in the previous section.

We first show that the learning version becomes stronger and stronger in front of the non-learning version. Results are presented in figure 3.2 and 3.3. The curves are not monotonous because the OB is generated by self-play. Once that a color found a good move, the other color spend some time to found a refutation and during this time, the percentage of victory decreases. But in average, the winning percentage grows for both color against the non-learning version.

The figure 3.4 describes the evolution of the results during the run of the algorithm. It represents the winning percentage of the program with the generated OB against itself from the point of view of white (the curve for

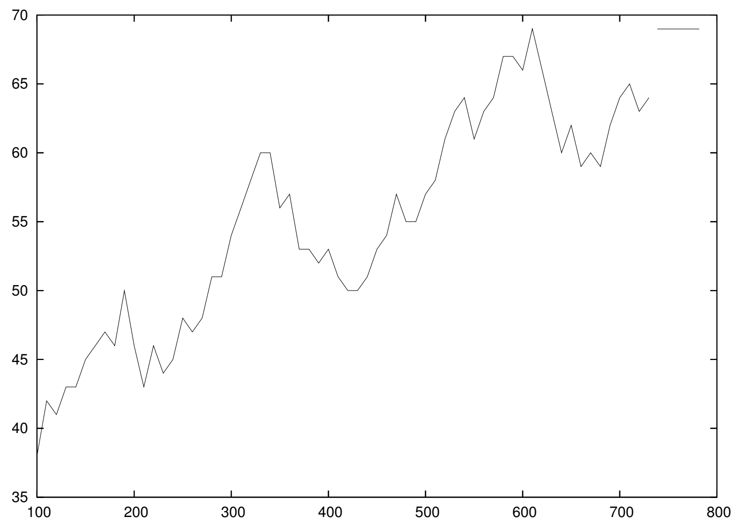


Figure 3.2: Success rate as black against the non-learning version (range from 35% to 70%). The x-axis is the time index (number of test games played); the plot is made on a moving average on 100 games.

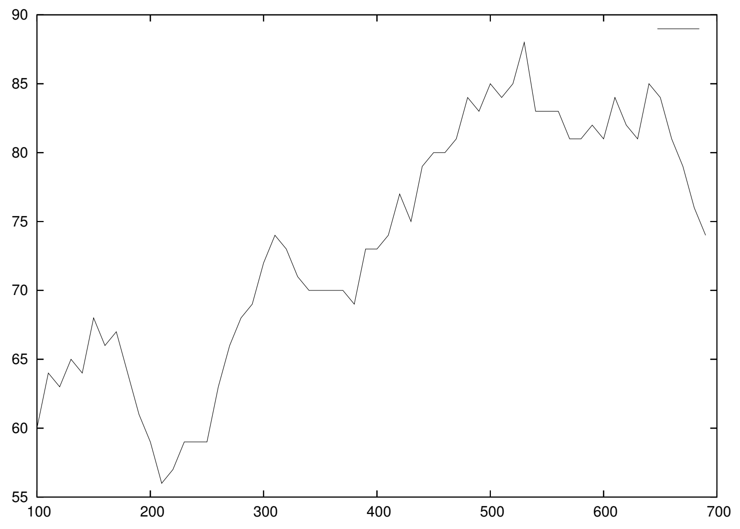


Figure 3.3: Success rate as white against the non-learning version (range from 55% to 90%). The x-axis is the time index (number of test games played); the plot is made on a moving average on 100 games.

Color	Coevolutionary opening book	Handcrafted opening book
White	74.5% ± 2.3 %	62.9 % ± 3.8 %
Black	64.6 % ± 2.4 %	49.7 % ± 3.8 %

Table 3.1: Comparison between the OB generated by the coevolution algorithm and the handcrafted OB. They are both tested against the version without OB.

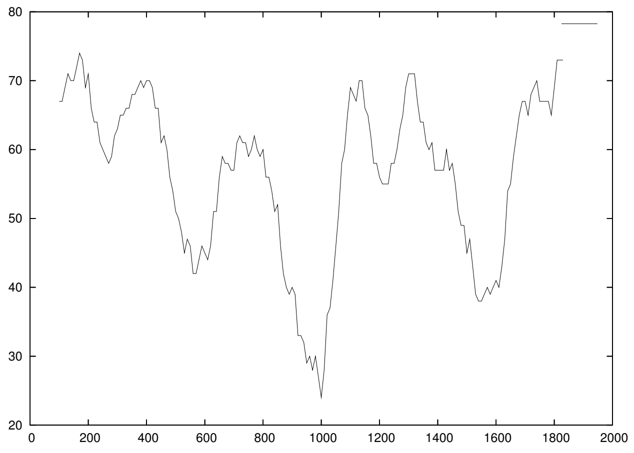


Figure 3.4: Success rate of the self-learning version against itself, as white. The x-axis is the time index (number of test games played); the plot is made on a moving average on 100 games. This is consistent with usual knowledge about 9x9 Go with komi 7.5 for strong players: the white player has the advantage.

black would be the same but horizontally inversed). There are some very important variations depending which color has currently the best opening line. It seems to converge to 100% of win for white. This is consistent with the opinion of experts: on 9x9 with a komi of 7.5, white should win.

Finally, we compare the automatically generated OB with the handcrafted OB. Both are tested against the version without OB. The results are shown on table 3.1. This approach is successful: for both colors, the generated OB wins 15% more than the handcrafted OB.

3.3 Conclusion

We used a coevolutionary algorithm for generating OB for a BBMCTS algorithm and tested it for the game of Go. There's no large efficient OB available. First, the positive results follow.

First, efficiency. The resulting OB is satisfactory. It leads to quite good results in the sense that the program is much stronger with the OB than without. It also outperforms a handcrafted OB. It is also satisfactory, in most cases, from the point of view of an expert human player: some known rules have been found independently by the algorithm (Kosumi, Mane Go,...).

Second, grid-compliance. Thanks to preemptable jobs (the algorithm is completely fault tolerant), the use of the grid was not too much a trouble for other users. It was also possible to have simultaneously a small number of non-preemptable jobs and plenty of preemptable jobs, so that the grid is used adaptively.

Third, parallelization. The speed-up is seemingly good, as (i) there's very little redundancies between individuals in the population (ii) the individuals are different just a very few moves after the initial sequence which is not randomly mutated. This is consistent with [Teytaud and Fournier, 2008] which has shown that evolution strategies have a very good speed-up when the problem has a huge dimensionality: the domain here is huge [Tromp and Farnebäck, 2006].

Fourth, user-friendly aspect. It is easy to monitor the optimization, as at any fixed time step we can see which sequence is currently analyzed by the algorithm.

Fifth, generality. We applied our algorithm to the empty Goban (initial situation), but it could be applied easily to a given situation (Tsumego analysis). Also, it can be used for analyzing a particular situation for any Markov Decision Process. After all, MVBM is an adaptive Monte Carlo algorithm: it is based on an initial strategy, and incrementally improves this strategy, based on simulations - it is like a Monte Carlo algorithm used for plotting a Value-At-Risk, but used adaptively. Applications in finance or power plant management are straightforward.

Some less positive points are as follows:

First, instability. One of the solution proposed by the algorithm is weak according to an expert. We have an algorithm which is therefore consistent, but we can't guess a priori that a solution is still unstable.

Second, no free lunch. The OB generated by a poor random player (MoGo with small time settings) is useless for a better MoGo (with longer time settings). This is consistent with result in

[Donninger and Lorenz, 2006]: the presence of erroneous moves is a major trouble when building OB. Therefore, only long time settings can be used.

Chapter 4

Expert Knowledge and Diversity Preservation

4.1 Introduction

This chapter is based on the article [Chaslot et al., 2009] written with Guillaume Chaslot, Christphe Fiter, Jean-Baptiste Hoock and Olivier Teytaud.

In a lot of domains, some expert knowledge (EK) already exists . The efficiency of an algorithm can often be improved by the addition of this knowledge but depending on the algorithm, this is not always easy. In this chapter, we will study two different ways of adding expert knowledge to the BBMCTS algorithm: adding it in the tree and adding it in the MC simulations. We will take the example of the game of go where the amount of existing expert knowledge is huge.

Adding expert knowledge as another term in the bandit formula is quite natural. As a matter of fact, with the original formula, the first choices at a node (before having any information) are done with a uniform distribution. But it can take a lot of simulations to realize that a choice is bad. So we can improve the algorithm by doing those first choices according to the expert knowledge instead. If the expert knowledge is relevant, we should find the good moves faster on average.

In spite of many improvements in the bandit formula, there are still situations which are poorly handled by MCTS. MCTS uses a bandit formula for moves early in the tree, but can't figure out long term effects which involve the behavior of the simulations far from the root. The situations which are to be clarified at the very end should therefore be included in the Monte Carlo part and not in the bandit.

We therefore propose three improvements in the MC part:

- Diversity preservation as explained in section 4.2.2;
- Nakade refinements as explained in section 4.2.2;
- Elements around the Semeai, as explained in section 4.2.2.

4.1.1 Difficulties

The difficulties encountered concern the validation part. As the expert knowledge consists of a lot of small rules, each one has only a small effect on the efficiency of the algorithm and requires a lot of testing to be validated.

For the Monte Carlo part, the problem is even more complicated as the effect of a modification is unpredictable and often counter intuitive: improving the level of the random player doesn't necessarily improve the global level of the algorithm. For the moment, the solution is to try and validate by extensive experimentations.

4.1.2 State of the art

[Chaslot et al., 2007] have already combined off-line learning (statistic from professional games) and on-line learning (bandit choice of the move). They first use the expert knowledge to create a heuristic function H_i that associates a value to the move i. Then, they propose two different ways of using it.

The first is called "progressive bias". It consists in adding a term in the bandit formula that decreases with the number of simulations. In the paper, they propose a linear decrease: $\frac{H_i}{n_i}$, where n_i is the number of simulation for the move i.

The second is called "progressive unpruning". It consists in choosing the next move only among the first $U(n_i)$ moves. $U(n_i)$ is a function that associates a integer value to n_i and that increases with n_i. The moves are ordered according to H_i.

They test the effect of those modifications by using them in the Go program Mango. The version with the modifications wins 80% of the time against the version without the modifications. Furthermore, the version with modifications wins 58% of the time against an other Go program: Gnugo, while the version without modifications wins only 25% of the time.

In the following, we will propose an improved version of the progressive bias. However, we won't use the progressive unpruning because not studying at all some moves has a very bad effect for some situations where the heuristic is wrong.

[Gelly and Silver, 2007] combines on-line learning and transient learning (RAVE values) and experiments the utilization of off-line learning. The off-line values are generated by using a reinforcement learning algorithm. They propose two different ways of using those values.

First, they use them to modify the Monte Carlo policy. However, they obtain only negative results. We will propose in this chapter some successful modifications of the Monte Carlo policy.

Then, they propose to use the off-line information to initialize the number of wins and the number of simulations in the RAVE part of the formula (see section 1.3.4). This method achieves an improvement of 9% of winning rate against the program Gnugo.

4.2 Contribution

We will present two different ways of adding expert knowledge to a BBMCTS algorithm. The first way is to modify the bandit formula used to descend in the tree by adding a bonus. The second is to change the Monte Carlo simulations. Instead of using only uniform distribution to chose the next move, we first check if some appropriate rules match and play accordingly if this is the case.

4.2.1 Adding Expert Knowledge in the Tree

In this section we present a modification of the bandit formula used in MCTS (see sections 1.3.2 and 1.3.4). We combine online learning (bandit module), transient learning (RAVE values), expert knowledge (detailed below) and offline pattern-information. RAVE values are presented in [Gelly and Silver, 2007]. We point out that this combination is far from being straightforward: due to the subtle equilibrium between online learning (i.e. naive success rates of moves) transient learning (RAVE values) and offline values, the first experiments were highly negative, and became clearly conclusive only after careful tuning of parameters[1].

We modify the bandit formula of MCTS by adding the "progressive bias". However, instead of the original weight that decreases linearly with the number of trials: $1/n_i$, we propose a new formula for the weight: $(\gamma + \frac{C}{\log(2+n_j)})$. This weight still decreases with the number of trials but logarithmically. This new weight leads to better results. The progressive bias plays the role of the exploration; the original exploration part of the

[1]We used both manual tuning and cross-entropy methods. Parallelization was highly helpful for this.

UCB formula (see 1.1 is not necessary anymore. Therefore, the α parameter in 1.3 is set to 0 (this value has been verified by several experiments). The new formula for the score of a decision j (i.e. a legal move) is:

$$\underbrace{\bar{x}_j}_{Online} + \beta(n) \underbrace{\bar{R}_j}_{Transcient} + (\gamma + \frac{C}{\log(2+n_j)}) \underbrace{H_j}_{Offline} \tag{4.1}$$

where the coefficients β, γ and C are empirically tuned coefficients depending on n_j (number of simulations of the decision j) and n (number of simulations of the current board) as follows:

$$\beta = \frac{\#\{ravesims\}}{(\#\{ravesims\} + \#\{sims\} + c_1\#\{sims\}\#\{ravesims\})} \tag{4.2}$$

$$\gamma = \frac{c_2}{\#\{ravesims\}} \tag{4.3}$$

$$\tag{4.4}$$

$\#\{ravesims\}$ is the number of Rave-simulations.
$\#\{sims\}$ is the number of simulations.
C, c_1 and c_2 are empirically tuned constants.

For the sake of completeness, we make clear that C, c_1 and c_2 depend on the board size, and are not the same in the root of the tree during the beginning of the thinking time, in the root of the tree during the end of the thinking time, and in other nodes. Also, this formula is computed most often with an approximated (faster) formula, and sometimes with the complete formula - it was empirically found that the constants should not be the same in both cases. All these local engineering improvements make the formula quite unclear and the take-home message is mainly that MoGo has good results with $\gamma \simeq c_2/\#\{ravesims\}$ and with the logarithmic formula $C/\log(2+n(d))$ for progressive unpruning. These rules imply that:

- initially, the most important part is the offline learning;
- later, the most important part is the transient learning (RAVE values);
- eventually, only the “real” statistics matter.

H_j is the sum of two terms: patterns, as in [Bouzy and Chaslot, 2005, Chaslot et al., 2007, Coulom, 2007], and rules detailed below:

- capture moves (in particular, string contiguous to a new string in atari), extension (in particular out of a ladder), avoid self-atari, atari (in particular when there is a *ko*), distance to border (optimum distance = 3 in 19x19 GO), short distance to previous moves, short distance to the move before the previous move; also, locations which have probability nearly 1/3 of being of one's color at the end of the game are preferred.

The following rules are used in our implementation in 19x19, and improve the results:

- Territory line (i.e. line number 3), Line of death (i.e. first line), Peep-connect (ie. connect two strings when the opponent threatens to cut), Hane (a move which "reaches around" one or more of the opponent's stones), Threat, Connect, Wall, Bad Kogeima(same pattern as a knight's move in chess), Empty triangle (three stones making a triangle without any surrounding opponent's stone).

They are used both (i) as an initial number of RAVE simulations (ii) as an additive term in H. The additive term (ii) is proportional to the number of RAVE simulations.

These shapes are illustrated on Figure 4.1. With a naive hand tuning of parameters, only for the simulations added in the RAVE statistics, they provide 63.9±0.5 % of winning rate against the version without these improvements. We are optimistic on the fact that tuning the parameters will strongly improve the results. Moreover, since the early developments of MoGo, some "cut" bonuses are included (i.e., advantages for playing at locations which match "cut" patterns, i.e. patterns for which a location prevents the opponent from connecting two groups).

Following [Bouzy and Chaslot, 2005], we built a model for evaluating the probability that a move is played, conditionally to the fact that it matches some pattern. When a node is created, the pattern matching is called, and the value it returns is used as explained later (Eq. 4.1). The pattern matching is computationally expensive and we had to tune the parameters in order to have positive results.

The following parameters had to be modified, when this model was included in H:

- time scales for the convergence of the weight of online statistics to 1 (see Eq. 4.1) are increased;

- the number of simulations of a move at a given node before the subsequent nodes is created is increased (because the computational cost of a creation is higher).

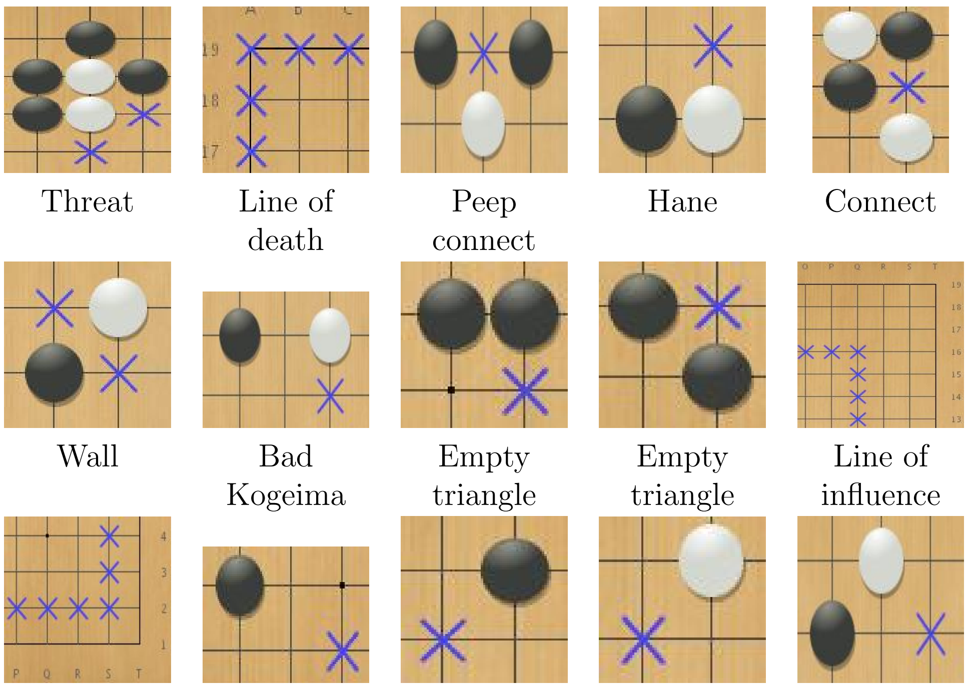


Figure 4.1: We here present shapes for which exact matches are required for applying the bonus/malus. In all cases, the shapes are presented for the black player: the feature applies for a black move at one of the crosses. The reverse pattern of course applies for white. Threat is not an exact shape to be matched but just an example: in general, black has a bonus for simulating one of the liberties of an enemy string with exactly two liberties, i.e. to generate an atari.

Tested version	Against	Conditions	Success rate
MoGo + P	MoGo	3000 sims/move	56 % ± 1%
MoGo + P	MoGo	2s/move	50.9 % ± 1.5 %
MoGo + P + TC	MoGo + P	1s/move	55.2 % ± 0.8 %
MoGo + P + TC + C	MoGo + P + TC	1s/move	61.7 % ± 3.1 %

Table 4.1: Effect of adding patterns extracted from professional games in MoGo. P: patterns, TC: tuning of coefficients, C: adding and tuning the constant C. The first tuning of parameters is the tuning of β and γ as functions of $n(d)$ (see Eq. 4.1) and of coefficients of expert rules. A second tuning consists in tuning constant C in Eq. 4.1.

- the optimal coefficients of expert rules are modified;
- importantly, the results were greatly improved by adding the constant C (see Eq. 4.1). This is the last line of Table 4.1.

Results are presented in Table 4.1.

4.2.2 Adding Expert Knowledge and preserving diversity in the Monte Carlo part

There exists no easy criterion to evaluate the effect of a modification of the Monte Carlo simulator on the global MCTS algorithm in the case of Go or more generally two players games. Many people have tried to improve the MC engine by increasing its level (the strength of the Monte Carlo simulator as a stand-alone player), but it is shown clearly in [Wang and Gelly, 2007, Gelly and Silver, 2007] that this is not the good criterion: a MC engine MC_1 which plays significantly better than another MC_2 can lead to very poor results as a module in MCTS, whenever the computational cost is the same. Some MC engines have been learned on datasets [Coulom, 2007], but the results are strongly improved by changing the constants manually. In that sense, designing and calibrating a MC engine remains an open challenge: one has to intensively experiment a modification in order to validate it.

Various shapes are defined in [Bouzy, 2005, Wang and Gelly, 2007, Ralaivola et al., 2005]. [Wang and Gelly, 2007] uses patterns and expertise. We present below two new improvements, both of them centered on an increased diversity when the computational power increases; in both cases, the improvement is negative or negligible for small computational power and becomes highly significant when the computational power increases.

Fill the Board: Random Perturbations of the Monte Carlo Simulations

The principle of this modification is to play first on locations of the board where there is large empty space. The idea is to increase the number of locations at which Monte Carlo simulations can find pattern-matching in order to diversify the Monte Carlo simulations.

As trying every position on the board would take too much time, the following procedure is used instead. A location on the board is chosen randomly; if the 8 surrounding positions are empty, the move is played, else the following $N-1$ positions on the board are tested; N is a parameter of the algorithm. This modification introduces more diversity in the simulations: this is due to the fact that the Monte Carlo player uses a lot of patterns. When patterns match, one of them is played. So the simulations have only a few ways of playing when only a small number of patterns match; in particular at the beginning of the game, when there are only a few stones on the goban. As this modification is played before the patterns, it leads to more diversified simulations (Figure 4.2). This modification is even more efficient on large boards(19x19) because there are less possible moves on small board(9x9) and therefor the diversity loss in less important in comparison. The detailed algorithm is presented in Algorithm 11, experiments in Figure 4.3.

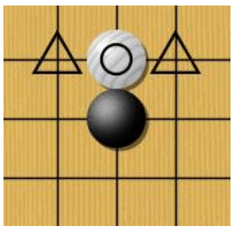


Figure 4.2: diversity loss when the “fillboard” option was not applied: the white stone is the last move, and the black player, starting a Monte Carlo simulation, can only play at one of the locations marked by triangles.

Algorithm 11 Algorithm for choosing a move in MC simulations, including the "fill board" improvement. We experimented also a constraint of 4, 12 and 22 empty locations instead of 8, but results were disappointing.

```
if the last move is an atari, then
  Save the stones which are in atari if possible.
else
  "Fill board" part.
  for $i \in \{1, 2, 3, 4, \ldots, N\}$ do
    Randomly draw a location $x$ on the goban.
    IF $x$ is an empty location and the eight locations around $x$ are empty,
    play $x$ (exit).
  end for
  End of "fill board" part.
  Sequential move, if any (see above).
  Capture move, if any (see above).
  Random legal move, if any (see above).
end if
```

The "Nakade" Problem

A known weakness of MoGo, as well as many MCTS programs, is that *nakade* is not correctly handled. We will use the term *nakade* to denote a situation in which a surrounded group has a single large internal, enclosed space in which the player won't be able to establish two eyes if the opponent plays correctly.

The group is therefore dead, but the baseline Monte Carlo simulator sometimes estimates that it lives with a high probability, i.e. the MC simulation does not necessarily lead to the death of this group. Therefore, the tree will not grow in the direction of moves preventing difficult situations with *nakade* — MoGo just considers that this is not a dangerous situation.

This will lead to a false estimate of the probability of winning. As a consequence, the MC part (i.e. the module choosing moves for situations which are not in the tree) must be modified so that the winning probability reflects the effect of a *nakade* .

Interestingly, as most MC tools have the same weakness, and also as MoGo is mainly developed by self-play, the weakness concerning the *nakade* almost never appeared before humans found the weakness (see post from D. Fotland called "UCT and solving life and death" on the computer-Go mailing list). It would be theoretically possible to encode in MC simulations a large set of known *nakade* behaviors, but this approach has two weaknesses: (i) it

9x9 board		19x19 board	
Nb of playouts per move or time/move	Success rate	Nb of playouts per move or time /move	Success rate
10 000	52.9 % ± 0.5%	10000	49.3 ± 1.2 %
5s/move, 8 cores	54.3 % ± 1.2 %	5s/move, 8 cores	77.0 % ± 3.3 %
100 000	55.2 % ± 1.4 %	100 000	73.7 % ± 2.9%
200 000	55.0 % ± 1.1 %	200 000	78.4 % ± 2.9 %

Figure 4.3: results associated to the “fillboard” modification. As the modification leads to a computational overhead, results are better for a fixed number of simulations per move; however, the improvement is clearly significant. The computational overhead is reduced when a multi-core machine is used: the concurrency for memory access is reduced when more expensive simulations are used, and therefore the difference between expensive and cheap simulations decays as the number of cores increases. This element also shows the easier parallelization of heavier playouts.

is expensive and MC simulations must be very fast (ii) abruptly changing the MC engine very often leads to unexpected disappointing effects. Therefore we designed the following modification: if a contiguous set of exactly 3 free locations is surrounded by stones from the opponent, then we play at the center (the vital point) of this “hole”. The new algorithm is presented in Algorithm 12.

We validate the approach with two different experiments: (i) known positions in which old MoGo does not choose the right move (Figure 4.4) (ii) games confronting the new MoGo vs the old MoGo (Table 4.2).

We also show that our modification is not sufficient for all cases: in the game presented in Fig. 4.4 (e), MoGo lost with a poor evaluation of a *nakade* situation, which is not covered by our modification.

Approach Moves

Correctly handling life and death situation is a key point in improving the MC engine. Reducing the probability of simulations in which a group which should clearly live dies (or vice versa) improves the overall performance of the algorithm. For example, in Fig. 4.5, black should play in B before playing in

Algorithm 12 New MC simulator, reducing the *nakade* problem.

```
if the last move is an atari, then
  Save the stones which are in atari if possible.
else
  Beginning of the nakade modification
  for $x$ in one of the 4 empty locations around the last move do
    if $x$ is in a hole of 3 contiguous locations surrounded by enemy stones
    or the sides of the goban then
      Play the center of this hole (exit).
    end if
  end for
  End of the nakade modification
  "Fill board" part (see above).
  Sequential move, if any (see above).
  Capture move, if any (see above).
  Random legal move, if any (see above).
end if
```

A for killing *A*. This is an approach move. We implemented this as presented in algorithm 13. This modification provides a success rate of

- 52.68 % (± 0.33 %) in 9x9 with 20 000 simulations per move;
- 54.69 % (± 2.27%) in 19x19 with 50 000 simulations per move.

We can see on Fig. 4.5 that some *semeai* situations are handled by this modification: MoGo now clearly sees that black, playing first, can kill on Fig. 4.5. Unfortunately, this does not solve more complicated *semeai* as e.g. Fig. 4.5 (e).

4.3 Conclusion

We showed some successful ways of adding expert knowledge and preserving the diversity of a BBMCTS algorithm and made the following observations.

First, as well as for humans, all time scales of learning are important: offline knowledge (strategic rules and patterns) as in [Coulom, 2006, Chaslot et al., 2007]; online information (i.e. analysis of a sequence by mental simulations) [Gelly and Silver, 2007]; transient information (extrapolation as a guide for exploration).

Second, reducing diversity has been a good idea in Monte Carlo; [Wang and Gelly, 2007] has shown that introducing several patterns and rule

Algorithm 13 New MC simulator, implementing approach moves. Random is a random variable uniform on $[0, 1]$.

```
if the last move is an atari, then
  Save the stones which are in atari if possible.
else
  Nakade modification (see above).
  "Fill board" part (see above).
  if there is an empty location among the 8 locations around the last move
  which matches a pattern then
    Randomly and uniformly select one of these locations.
    if this move is a self-atari and can be replaced by a connection with
    another group and random < 0.5 then
      Play this connection (exit).
    else
      Play the select location (exit).
    end if
  else
    Capture move, if any (see above).
    Random legal move, if any (see above).
  end if
end if
```

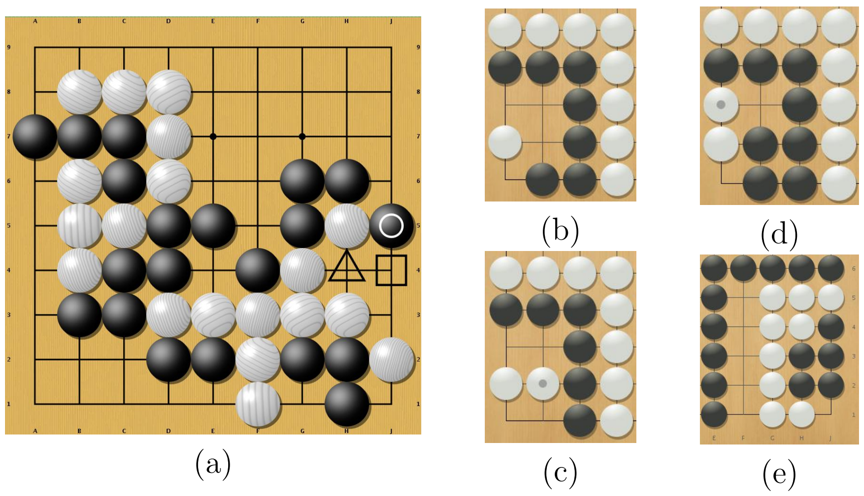


Figure 4.4: In Figure (a) (a real game played and lost by MoGo), MoGo (white) without specific modification for the *nakade* chooses H4 (triangle); black plays J4 (square) and the group F1 is dead (MoGo looses). The right move is J4; this move is chosen by MoGo after the modification presented in this section. Examples (b), (c) and (d) are other similar examples in which MoGo (as black) evaluates the situation poorly and doesn't realize that his group is dead. The modification solves the problem. (e) An example of more complicated *nakade* , which is not solved by MoGo - we have no generic tool for solving the *nakade* .

greatly improves the efficiency of Monte Carlo Tree-Search. However, plenty of experiments around increasing the level of the Monte Carlo simulator as a stand-alone player have given negative results - diversity and playing strength are too conflicting objectives. There is a trade-off between these two criteria.

Approach moves are an important feature. It makes MoGo more reasonable in some difficult situations in corners. We believe that strong improvements can arise as generalizations of this idea, for solving the important *semeai* case.

Importantly, whereas exploration by a UCT term

$$+\sqrt{\frac{\log(n)}{n_j}}$$

as in UCB (see formula 1.1) is important when scores are naive empirical success rates, the optimal constant in the exploration term becomes 0 when learning is improved (at least in MoGo, and the constant is very small in several UCT-like programs also). In MoGo, the constant in front of the

Nb sims per move	Success rate	Nb sims per move	Success rate
9x9 board		19x19 board	
10000	52.8 % ± 0.5%		
100000	55.6 % ± 0.6 %	100 000	53.2 % ± 1.1%
300000	56.2 % ± 0.9 %		
5s/move, 8 cores	55.8 % ± 1.4 %		
15s/move, 8 cores	60.5 % ± 1.9 %		
45s/move, 8 cores	66.2 % ± 1.4 %		

Table 4.2: Experimental validation of the *nakade* modification: modified MoGo versus baseline MoGo. Seemingly, the higher the number of simulations (which is directly related to the level), the higher the impact.

exploration term was not null before the introduction of RAVE values in [Gelly and Silver, 2007]; it is now 0. Another term has provided an important improvement as an exploration term: the constant C in Eq. 4.1.

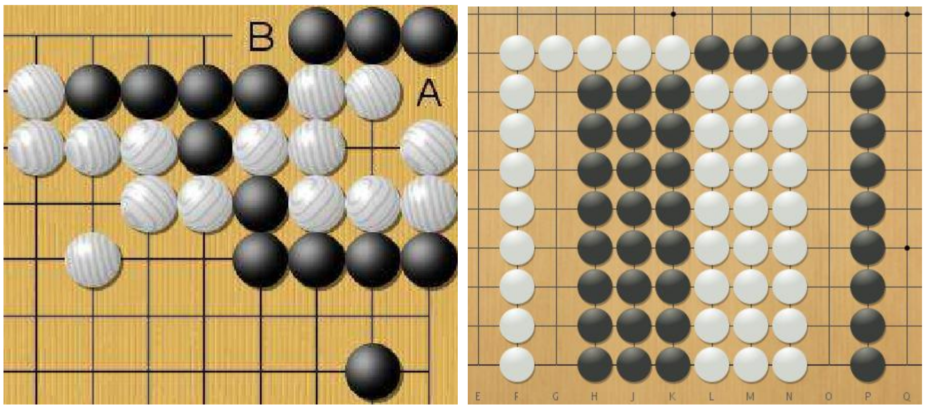


Figure 4.5: Left: Example of situation which is poorly estimated without approach moves. Black should play *B* before playing *A* for killing the white group and live. Right: situation which is not handled by the "approach moves" modification.

Chapter 5

Threshold Ascent applied to Graph

5.1 Introduction

This chapter is based on the article [De Mesmay et al., 2009] written with Frederic De Mesmay, Yan Voronenko and Markus Puschel.

In the previous chapter, several improvements of the BBMCTS algorithm have been presented. But the algorithm was always applied to the game of Go. In this chapter, we address three problems. We show that a BBMCTS algorithm can work with other bandits than the classical one. We present a way to adopt this algorithm in the case of graphs and call it Threshold Ascent applied to Graph (TAG). We apply the algorithm to an industrial problem and achieve better results than the previous method. The problem is to find the fastest among all the possible ways of computing a linear transform on a particular computer. Once it has been found, an already-existing program (Spiral) automatically generates a Library from it.

5.1.1 Library Performance Tuning

Our target application is the automatic performance tuning in adaptive libraries based on divide-and-conquer algorithms with inherent degrees of freedom. Specifically, we implemented TAG to operate as a search strategy in the adaptive general-size linear transform libraries generated by Spiral [Voronenko et al., 2009].

We first give brief background on transforms, transform algorithms, their implementations, and the notion of an adaptive library. Then we discuss the need for search and finally match the performance tuning problem to Problem 1, which shows that TAG is applicable.

Background: Linear Transforms

Transforms. A linear transform is a matrix-vector product $y = Mx$, where x is the input vector, y the output vector, and M the fixed transform matrix. We focus on the discrete Fourier transform (DFT) defined as

$$\mathbf{DFT}_n = [e^{-2\pi ik\ell/n}]_{0\leq k,\ell<n}, \quad i = \sqrt{-1}.$$

Naïve computation of the matrix-vector product incurs $O(n^2)$ operations, however, fast, $O(n\log(n))$, algorithms, which exploit the particular structure of matrix M, exist for many transforms including the DFT.

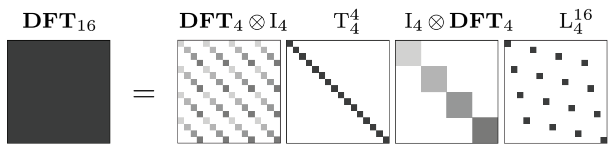
Fast algorithms. One way of writing transform algorithms is as sparse factorizations of the transform matrix. For example, the famous Cooley-Tukey fast Fourier transform (FFT) algorithm can be written as

$$\mathbf{DFT}_n = (\mathbf{DFT}_k \otimes \mathrm{I}_m)\, \mathrm{T}^n_m (\mathrm{I}_k \otimes \mathbf{DFT}_m)\, \mathrm{L}^n_k, \;\; n = km. \tag{5.1}$$

Here, I_n is the identity matrix of size n; T^n_m is a diagonal matrix and L^n_k a permutation matrix, whose precise definition is not relevant here. Finally, the tensor (or Kronecker) product $\otimes$ of two matrices is defined as

$$\mathrm{A} \otimes \mathrm{B} = [a_{k,l}\, \mathrm{B}], \text{ where } \mathrm{A} = [a_{k,l}].$$

We show below a visualization of the non-zero values in the matrices for $k = m = 4$.



In both tensor products, all parts of equal gray shade constitute a single $\mathbf{DFT}_4$. We observe that all four matrices are sparse, that the computation uses a divide-and-conquer approach, and that there is a degree of freedom (choice of $k|n$). Assuming that n is a power of two[1], recursive applications of the algorithm yield $O(n\log(n))$ computations.

Implementation and search space. The above FFT suggests a library implementation using a recursive function `dft`. Given the input x, the function would first permute ($t = \mathrm{L}^n_k\, x$), then call `dft` on multiple segments of x, then scale the result with the entries of T^n_m, and then call again `dft`

[1]Recursive application of equation (5.1) require to provide base cases for all prime factors of n. For simplicity, we will therefore only consider power-of-two sizes $n = 2^t$. Note that these sizes also happen to be the most important usage cases of the DFT.

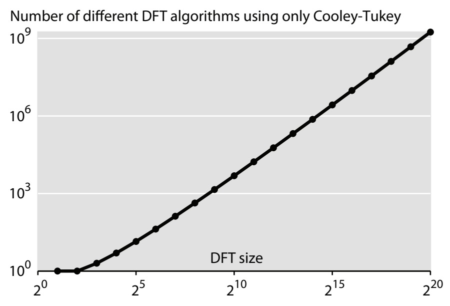


Figure 5.1: Number of DFT algorithms based on standard Cooley-Tukey FFT, implemented naïvely. All algorithms for a given DFT input size have roughly the same operations count.

on segments, extracted in a stride, of the result. The resulting library would have a simple call graph, as shown in Figure 5.2(a). Even such a simple implementation has a degree of freedom in the recursion due to the choice of k. Recursively compounded this yields an algorithm space of $\Theta(5^t/t^{3/2})$ that this library covers (see Figure 5.1) [Johnson and Püschel, 2000]. All of these have roughly the same operations count, yet, the performance can differ widely due to cache misses and other effects.

The above implementation makes four passes through a vector of length n and has hence poor memory hierarchy performance. The performance can considerably improve as done in FFTW 2.x by replacing the explicit (and expensive) permutation L^n_k with a readdressing in the subsequent smaller DFTs. Similarly, scaling by T^n_m can be fused with the subsequent DFTs. However, this creates the need for additional functions—variants of the DFT with modified interfaces. The call graph of such a library is shown in Figure 5.2(b).

The situation gets considerably more complicated with state-of-the-art libraries on current off-the-shelf computers. The reason is that to get maximal performance, the libraries need to apply several restructuring transformations to (5.1). In particular, the algorithm must be 1) vectorized, to take advantage of vector instructions (e.g., SSE on x86 architectures); 2) parallelized, to exploit multiple processor cores using threading; 3) transformed by loop optimizations for buffering; 4) allowed to load from a precomputed table the constant elements of T^n_m from (5.1), also called "twiddle factors" [Frigo and Johnson, 2005, Püschel et al., 2005, Voronenko, 2008, Voronenko et al., 2009].

Applying the restructuring transformations described above increases the number of different mutually recursive functions that comprise the library, and also enlarge the algorithm search space. For example the Spiral-generated DFT library with all optimizations 1–3 contains 31 different functions which form the call graph in Figure 5.2(c).

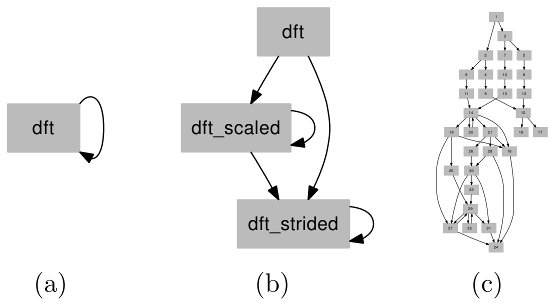


Figure 5.2: Call graphs of three different recursive libraries: (a) naïve, (b) optimized scalar and (c) optimized vectorized parallelized.

The above discussion holds for many other linear transforms including the discrete cosine transforms, the real discrete Fourier transform, finite impulse response filters, and the discrete wavelet transform. Furthermore, not all algorithms decompose a transform into transforms of the same type. In this case the search space is further increased.

Adaptive Libraries and Search

Consider a recursive library as discussed above. In each recursion step, the library has a degree of freedom. As a consequence, it can compute the transform in many different ways. What makes the library adaptive is the use of online search to find a fast recursion strategy. This search is part of an initialization routine (called planner in FFTW) that takes the input size n and returns a function pointer implementing the fastest known recursion strategy. After this initial overhead, the user can now compute as many transforms of size n as desired, compensating for the overhead.

The main search strategy in FFTW, UHFFT, and Spiral generated libraries is dynamic programming (DP). It is based on the assumption that the best solution to a problem is built out of optimal solutions to subproblems. Here, this means that an algorithm's performance is independent of its context which, unfortunately, does not always hold[2]. However, in practice, DP has shown to work quite well except for very large transforms as we will see later in our benchmarks. Over these large search spaces, DP has another weakness which is that it is not an anytime algorithm: one has to wait for

[2]It is fairly easy to build counter-examples: for instance, an algorithm running on one core will be slower if another core is active due to conflicts in the shared cache.

DP to solve all subproblems before it gives any solution. This waiting time is significant: for FFTW it can amount to days in the case of large transforms on multicore systems.

A simple anytime strategy is Monte-Carlo (MC) which, each time there is a decision to take, chooses according to a uniform distribution. At the end of the descent, it evaluates the candidate and restarts. At any point in time the user can interrupt the search to retrieve the best known candidate. Since at each step, there is an equal chance for all branches to be picked but branches are not laid out uniformly, the overall space is *not* sampled uniformly.

5.1.2 State of the art: Dynamic Programming

As described in section 1.2.4, Dynamic Programming (DP) is a method that recursively constructs solutions of large problems from solutions of smaller problems. It is implemented for the problem of library performance tuning in the following way. Given a transform T, we expand T using all applicable rules. A set of children is extracted from them. Then DP is called recursively to expand each children until a terminal node is reached. This node can be evaluated by a timer. Finally the set of rules with the minimal cost is returned.

There are two problems with using DP in this context. The first one is that it is very time consuming when used on linear transforms of large size. The second is that the property of optimal substructure doesn't hold in this case because the solution to a subproblem is context-dependent. Even if this is a good approximation and therefore can achieve very good results, there is no guaranty to obtain the optimal solution.

5.2 Contribution

After describing the problem in a formal way, we are going to present the new algorithm TAG. Then we will present the results on the problem of library performance tuning and conclude.

5.2.1 Formal Problem Statement

Below, we formally state the problem considered in this chapter. Later, we will show that automatic tuning in the considered transform libraries is an instantiation of this problem.

Problem 1 Given is an acyclic formal grammar $F = (T, N, P, S)$ with T the set of terminals, N the set of nonterminals, P the set of production rules or

$$
\begin{aligned}
T &= \{\text{a, b, ac}\} \\
N &= \{\text{S, A, B}\} \\
P &= \{\text{S} \rightarrow \text{AB}, \\
&\quad \text{A} \rightarrow \text{a}, \\
&\quad \text{B} \rightarrow \text{b}, \\
&\quad \text{aB} \rightarrow \text{ac}\}
\end{aligned}
$$

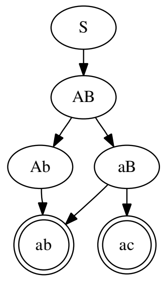


Figure 5.3: Formal grammar $F = (T, N, P, S)$ (*left*) and associated derivation graph $G(F)$ (*right*). S, A, B are nonterminals and a, b, c are terminals. The graph has two sinks (double circled), i.e., the language $\mathcal{L}(F)$ has two elements.

simply rules, and S the starting symbol. $\mathcal{L}(F)$ is the associated language and f is an objective function from $\mathcal{L}(F)$ into the positive reals $\mathbb{R}^+$. We want to compute

$$w_{\text{best}} = \operatorname*{argmax}_{w \in \mathcal{L}(F)} f(w).$$

F has an associated *derivation graph* $G = G(F)$ which is directed, acyclic and weakly connected as shown in Figure 5.3: S is the root, the directed edges (arrows) correspond to applications of rules in P, the nodes are partially derived words in the language, and the sinks (outdegree = 0) are precisely the elements of $\mathcal{L}(F)$. Hence we can reduce Problem 1 to:

Problem 2 Given a weakly connected, acyclic, directed graph $G = (V, E)$ and an objective function f (as above) on the sinks $S(G)$ of G. We want to compute

$$w_{\text{best}} = \operatorname*{argmax}_{w \in S(G)} f(w).$$

We assume the graph $G(F)$ to be large such that it is impossible to generate and evaluate all sinks in a reasonable time. Our goal is an algorithm that finds a "very good" sink with a small number of evaluations.

5.2.2 Algorithm

TAG is an *anytime* algorithm that determines an approximate solution to Problem 2. Due to the size of the graph, it is not meant to run until completion, in which case it would be equivalent to an exhaustive search.

TAG finds solutions by incrementally growing and exploiting the subgraph $\hat{G} = (\hat{V}, \hat{E})$ of $G = (V, E)$: $\hat{V} \subset V$, $\hat{E} \subset E$, starting with $\hat{G} = (\{S\}, \{\})$. Evaluations are used to direct the growth of $\hat{G}$ towards the expected bests sinks.

Assume the current subgraph is $\hat{G}$. Then TAG proceeds in three high level steps visualized in Figure 5.4:

1. *Descend:* G is traversed starting at its root. Each choice along the way is solved by a *bandit algorithm*. The descent stops when it uses an arrow e that is not in $\hat{E}$.

2. *Evaluate:* If e is incident with a vertex not in $\hat{V}$, this vertex is evaluated using a Monte-Carlo expansion.

3. *Backpropagate:* The evaluation is stored in all ancestors of the vertex.

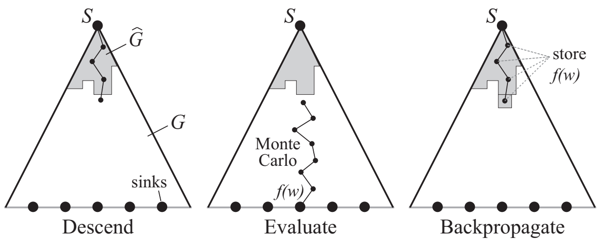


Figure 5.4: Visualization of the three main steps in TAG. Note that $\hat{G}$ (shaded area) and G are not trees (e.g., see Figure 5.3).

We proceed with describing the three steps in detail, describe the pseudocode and conclude the section with a presentation of related algorithms.

Descend

The goal of the *descent* step is to select the next edge to add to the subgraph $\hat{G}$. It is chosen so that $\hat{G}$ grows towards the sinks that present the best expected rewards. Starting from the root S, the most promising path is layed out by successively choosing the most promising outgoing edges. Each choice is solved using a bandit algorithm that we describe first.

Background: Max k-Armed Bandit Problem. The *maximum k-armed bandit problem* considers a slot machine with k arms, each one having a different pay-out distribution (Figure 5.5). The goal is to maximize the single best reward obtainable over $\bar{n}$ trials [Cicirello and Smith, 2005]. Formally, if each arm has distribution D_i and $R_j(D_i)$ denotes the j-th reward obtained on arm i, the goal is to solve

$$\max_{\sum_{i=1}^{k} \bar{n}_i = \bar{n}} \max_{1 \leq i \leq k} \max_{1 \leq j \leq \bar{n}_i} R_j(D_i).$$

We use a variation: an anytime version of the problem where the total number of pull $\bar{n}$ is not known in advance. Only the n previous pulls and their associated rewards are known.

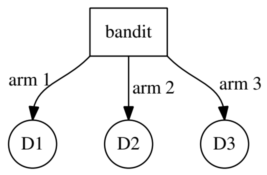


Figure 5.5: A 3-armed bandit. The choice of the arm i leads to a realization of the distribution D_i.

Streeter & Smith in [Streeter and Smith, 2006] solve the problem using *Threshold Ascend*, an algorithm that makes no assumptions on the form of the distributions. Using their notations, we present here a straightforward adaptation to the anytime variation.

The main idea of the algorithm is to track only the s best rewards and the arms they are coming from. Let s_i be the number of such rewards among the n_i rewards received by the arm i. Also, let δ be a positive real parameter. The algorithm advises to pull the arm i_{best} given by

$$\begin{aligned} i_{\text{best}} &= \underset{1 \leq i \leq k}{\text{argmax}}\ h(s_i, n_i), \\ \text{with } h(s_i, n_i) &= \begin{cases} \frac{s_i + \alpha + \sqrt{2 s_i \alpha + \alpha^2}}{n_i}, & \text{if } n_i > 0 \\ \infty, & \text{else} \end{cases} \\ \text{and } \alpha &= \ln(2nk/\delta). \end{aligned}$$

This formula keeps the same principle as the original formula from the classical k-armed bandit problem: the trade-off between exploration and exploitation. The first part, $\frac{s_i}{n_i}$, corresponds to the percentage of chance that a reward from the arm i be in the s best rewards. This is the exploitation term. In the second part, we find a term in $\frac{\sqrt{ln(n)}}{n_i}$ which is very similar to the exploitation in the classical k-armed bandit formula.

Descend. The graph descent is responsible for incrementally building the subgraph $\hat{G} \subset G$, initially restricted to the root. The purpose of the descent is to select an arrow in $E \setminus \hat{E}$ that leads towards an expected good sink. It does so by tracing a path starting from the root and considering each successor choice as a max k-armed bandit problem (Figure 5.6). For now, assume that a table of positive real rewards $R(v)$ has been maintained for each vertex $v \in \hat{V}$.

Let v denote the current vertex in the descent. Starting from v, there are multiple ways to continue the path since it can follow any of the arrows originating from v (we denote these with $E(v)$). The arrows in $E(v)$ that are also in $\hat{E}(v)$ lead to vertices of $\hat{V}$ corresponding to "arms" that have already been played (they have previous rewards attached to them). The other arrows lead to arms that have never been played. The bandit algorithm discussed above decides which arrow to follow, which has to be one that was not followed before if such an arrow exists (due to the infinite weight in $h(s_i, n_i)$). If the arrow belongs to $\hat{E}(v)$ and the successor is not a sink, the successor becomes the new descent vertex and the descent continues. If not, the descent ends.

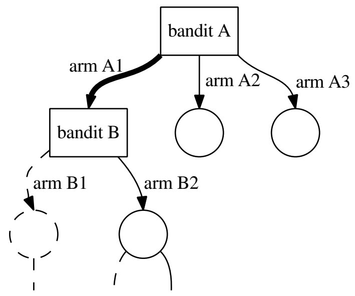


Figure 5.6: The descent in the graph is done as a cascade of multi-armed bandits. Solid arrows, circles and boxes are in $\hat{G}$, dashed arrows and circles are in $G \setminus \hat{G}$. For bandit A all arms had been played before, and A1 is chosen based on the stored rewards. Bandit B will now choose B1, since it is the only arrow not played before.

Evaluate

Assume the descent ended on an arrow pointing to a vertex v that is not part of $\hat{V}$. The arrow and vertex are then immediately added to $\hat{G}$ and v is *evaluated*.

If v is a sink of G, then $f(v)$ can be directly computed. Otherwise a path to a sink of G is chosen by "Monte-Carlo," which means in each step a (uniformly drawn) random choice is made until a sink w is obtained. The evaluation $f(w)$ gives a value for v.

Also, if the evaluation is better than $f(w_{\text{best}})$, the current best sink is replaced.

Backpropagate

After v has been evaluated, the reward is added to its reward list $R(v)$ and to the reward lists of all its ancestors.

Note that if the descent ended on an arrow pointing to a vertex v that is already a part of $\hat{V}$, we just discovered a new way to connect to an already evaluated vertex. In this case, we add the new arc to $\hat{E}$ and propagate the rewards of v only to the vertices that would not be ancestors of v without the new arrow (since the other ancestors already have these rewards).

Pseudocode and Remark

Pseudocode. Algorithm 1, the pseudocode of TAG, summarizes the previous discussion. After initialization, the graph $\hat{G} = (\hat{V}, \hat{E})$ is grown one arc at a time until the user signifies an interruption. The vertex pointed by an arrow e is denoted *head(e)*. `BANDIT` refers to the *Treshold Ascend* algorithm summarized in subsection 5.2.2. `RANDOM` refers to an uniform draw.

Remark. In practice, if the objective function is deterministic, it is useless to evaluate a sink twice. It is therefore possible to modify the algorithm to guarantee that it never returns in a branch where choices have been exhausted.

Applicability of TAG

Applying TAG in the context of adaptive libraries requires to identify the grammar $G = (T, N, P, S)$ and the objective function f such that the performance optimization can be mapped to Problem 1.

The start symbol S is the transform specification as entered by the user. The terminals T are the *base cases*, the set of problems that can be directly solved by the library. The non-terminals N are the set of all non base case subproblems that could be needed to solve the problem. The production rules P breakdown a problem from N into one or more subproblems by fixing a degree of freedom. The function to maximize, f, is the performance of the implementation. The acyclicity of the grammar is guaranteed by the fact that the underlying algorithms provably finish. Note that the grammar itself changes with the problem size.

For instance, if a naïve DFT library based on Cooley-Tukey is used to compute $\mathbf{DFT}_8$, we would define

$$\begin{aligned} S &= \mathbf{DFT}_8 & P = \{&\mathbf{DFT}_8 \to (\mathbf{DFT}_2, \mathbf{DFT}_4), \\ T &= \mathbf{DFT}_2 & &\mathbf{DFT}_8 \to (\mathbf{DFT}_4, \mathbf{DFT}_2), \\ N &= \{\mathbf{DFT}_8, \mathbf{DFT}_4\} & &\mathbf{DFT}_4 \to (\mathbf{DFT}_2, \mathbf{DFT}_2)\} \end{aligned}$$

5.2.3 Validation

Experimental Setup. We evaluated our search algorithm on a complex DFT C++ library generated by Spiral from (5.1). The library is vectorized using intrinsics, threaded using OpenMP, and optimized as explained in Section 5.1.1.

We add TAG and Monte-Carlo (MC) methods to the already existing DP search infrastructure. We compile using the Intel Compiler 10.1 and benchmark on a 64-bit Linux platform using two dual core 3 GHz Intel Xeon 5160 processors.

We display performance using pseudo mega floating-point operations per second (MFlops) with the complex DFT operation count assumed to be $5n \log_2 n$ (standard practice).

Parameter tuning. We tune the parameters for TAG on a specific problem, $\mathbf{DFT}_{2^{19}}$. The sensitivity of the algorithm with variations in the s parameter of the bandit is shown on Figure 5.7(a). Since s is the size of the best rewards vector, a low s tweaks the bandit towards exploitation of previous good branches, while a bigger s leads to the exploration of new promising branches. We find that $\delta = 0.1$ and $s = 30$ work best and we use them for *all* following experiments.

Comparison with Monte-Carlo. We compare the performance of TAG and MC on $\mathbf{DFT}_{2^{18}}$. Figures 5.7(b) and 5.7(c) show that TAG performs better (higher mean) and more reliably (lower standard deviation) than Monte-Carlo. Note that the plots are done with respect to a fixed "wall clock" time and not to a fixed number of simulations. This is realistic in that the simpler MC algorithm performs more simulations than our more complex algorithm in the same time frame. Also it is worth remembering that, asymptotically, TAG and MC match since they both explore the full finite search space.

Comparison with dynamic programming. We compare TAG with DP on a single problem on 5.8(a). We observe that TAG quickly reaches the same performance as DP and then caps 10% above it. On Figure 5.8(b) we plot the time it takes for TAG to get results of the same quality as DP. We observe that TAG finds solutions of equal performance significantly faster on various DFT sizes.

Comparison with other FFTs. Figure 5.8(c) shows the best performance attained by the generated library (with TAG and DP) and its competitors. We compare against FFTW 3.2 alpha 2 and Intel IPP 5.3. FFTW does platform adaptation using dynamic programming. As far as we know, IPP does not use search and branches out to a specialized implementation for each platform.

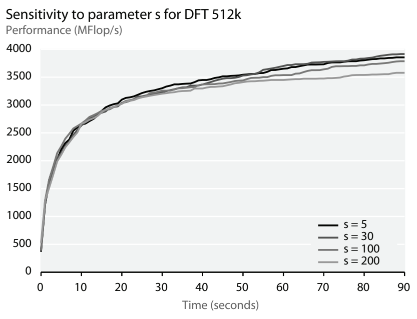
5.3 Conclusion

In this chapter, we tackled the problem of using a BBMCTS algorithm on an industrial problem. In order to do that, we needed to optimize an objective function over the sinks of a directed acyclic graph. We solved it using a new anytime algorithm, TAG, that grows a subgraph towards the expected best sinks. Similarly to UCT, TAG traces the most promising path by considering local bandits and valuates nodes using Monte-Carlo simulations. In our context however, the optimization problem requires to consider the *maximum* variant of the k-armed bandit problem.

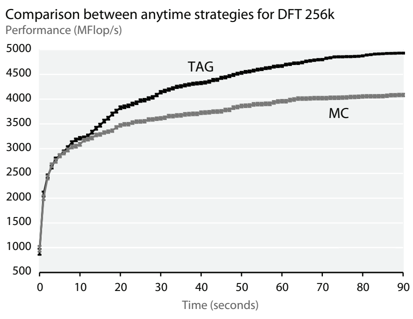
Implementation inside a high-performance adaptive library for linear transforms considerably decreased the search time while providing a 10% increase in the quality of the solutions. One interesting feature of our problem setup is that evaluating “bad” nodes is much more costly than evaluating “good” ones since the objective function is the timer from the processor. In future work, we will try to modify the algorithm to take advantage of that fact.

Algorithm 1 TAG

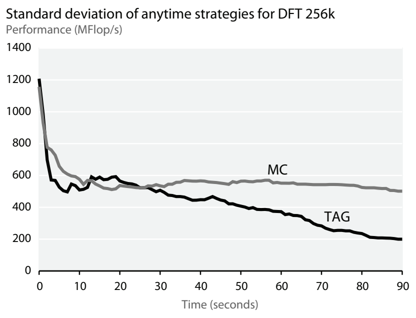
```
$\hat{G} \leftarrow S$
$w_{\text{best}} \leftarrow \emptyset$
$R(\hat{V}) \leftarrow \emptyset$
**while** not interrupted **do**
  $e \leftarrow$ BANDIT$(E(S))$                    *Descend*
  **while** $e \in \hat{E}$ & $E(head(e)) \neq \emptyset$
  **do**
    $e \leftarrow$ BANDIT$(E(head(e))$
  **end while**
  $v \leftarrow head(e)$
  **if** $v \notin \hat{G}$ or $e \in \hat{G}$ **then**
    add $v$ and $e$ to $\hat{G}$
    $e \leftarrow$ RANDOM$(E(v))$                 *Evaluate*
    **while** $E(head(e)) \neq \emptyset$ **do**
      $e \leftarrow$ RANDOM$(E(head(e))$
    **end while**
    $w \leftarrow head(e)$
    **if** $f(w) > f(w_{\text{best}})$ **then**
      $w_{\text{best}} \leftarrow w$
    **end if**
    $r \leftarrow f(w)$
    add $r$ to $R(v)$                        *Backpropagate*
    **for** $a$ ancestor of $v$ in $\hat{G}$ **do**
      add $r$ to $R(a)$
    **end for**
  **else**
    **for** $a$ ancestor of $v$ in $\hat{G}$ **do**
      mark $a$
    **end for**
    add $e$ to $\hat{G}$
    **for** $a$ ancestor of $v$ in $\hat{G}$ **do**
      **if** $a$ is marked **then**
        unmark $a$
      **else**
        add all $R(v)$ to $R(a)$
      **end if**
    **end for**
  **end if**
**end while**
**return** $w_{\text{best}}$
```



(a)

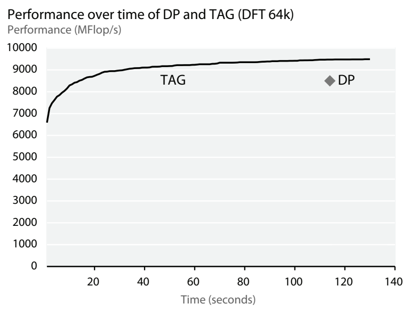


(b)

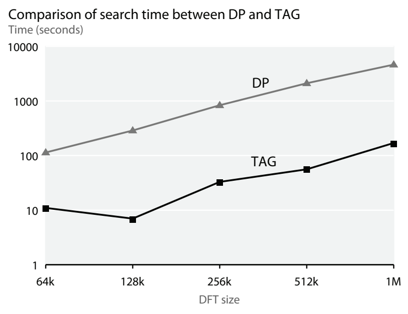


(c)

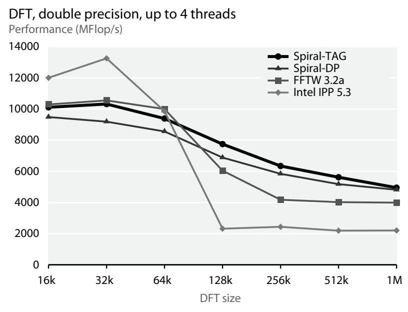
Figure 5.7: (a) Parameters for TAG are optimized on $\mathbf{DFT}_{2^{19}}$. (b) Mean performance (and standard error of the mean) for DP and Monte-Carlo on $\mathbf{DFT}_{2^{18}}$. Data is averaged over 100 runs. (c) Standard deviation on the same experiment.



(a)



(b)



(c)

Figure 5.8: (a)Average performance of TAG compared with DP on a single problem size. (b)Search time of TAG and DP to achieve the same performance on different libraries. (c)Comparison with different FFT libraries.

Chapter 6

Conclusion

MCTS is a new type of algorithm which achieves great results in the case of observable problems with high dimension. Some works have even been done in the non-observable case in [Rolet et al., 2009]. It has been proved to be successful in particular for the game of Go.

In chapter 2, we presented the study of the parallelization of the algorithm. While the numerical results are good, they are somewhat disappointing because the bias inherent to the method is not reduced by the parallelization. Therefore, it does not bring so much improvement against humans. The proposed parallel version of the MCTS algorithm is completely independent from the application to the game of Go. The fact that an efficient parallelization is possible is an important feature of this algorithm. Today, multi-cores computers are becoming the norm and algorithms not able to take advantage of it will have some serious trouble to be competitive in the future.

In chapter 3, we presented a generic way of constructing an opening book for a MCTS algorithm. This method combined with analyzes from expert leads to great successes in 9x9 Go. MoGo won the 9x9 KGS tournament of September 2009, with 3 victories against Zen, the second program. For one of the victory, MoGo played 7 moves in the opening and only 4 moves outside before Zen resigned. This method can also be applied to other games and to construct opening book for other algorithms.

In chapter 4, we introduced different modifications of the Monte-Carlo part of the algorithm. This leads to great improvements for MoGo thanks to (i) inclusion of expert knowledge, like for the Nakade problem, and yet more with (ii) careful analysis of diversity loss, like for the so-called "fill board" modification of MoGo. The basic version of the MCTS algorithm is independent from the domain of application. This is an advantage because it can be applied to a lot of situations. However, very often, it already exists some knowledge and not being able to use it would be a serious limitation.

By proposing several generic ways of adding expert knowledge to the MCTS algorithm, we give the possibility to use it even in situation where experts have already found useful information.

In chapter 5, we proposed an application to grammars with a real-world application in which a variant of MCTS, namely our TAG algorithm, clearly outperformed a baseline method which was heavily optimized by teams of engineers. It shows that the algorithm is not limited to the use of the classical k-armed bandit formula. It also shows that the algorithm is not restricted to games and to artificial benchmarks.

The main difference with algorithms like Minimax and AlphaBeta is that the search is much more in depth. This allows the exploration of a space of higher dimension. This principle is not new as we find the same idea in variants of AlphaBeta like iterative deepening.

The MCTS algorithm has also some serious limitations as the study on the game of Go has shown. For example, it has some trouble to analyze local situations where a lot of moves are equivalent. This is the case for the problem of semeai in Go (see chapter 4). This task is easy for a human because he can see the equivalence and generalize the solution. But MCTS has to consider every variation in order to take the correct decision. As this is impossible because of the number of possibilities, the situation will be wrongly evaluated for a non negligible percentage of times. Solving this problem will be the key to improve the performance of MCTS algorithms in the near future.

An other improvement of the algorithm would be to find a way to handle partially observable states. In many applications, some elements needed to describe the situation are unknown. For example in the case of Poker, the cards of the opponent are needed to know which actions are possible for him. Or in the case of taking decision in an environment where some random elements will affect the situation. The basic version of MCTS can not be applied directly to those cases. If an efficient solution is found to improve the MCTS algorithm to solve those problems, it could be applied to a much larger range of applications.

Appendix A

Commentaries on Games played by MoGo in Taiwan

A.1 Introduction

In order to promote and strengthen Computer Go programs and to advocate the research, development and application of related fields, Chang Jung Christian University (CJCU), National University of Tainan (NUTN) and the Taiwanese Association for Artificial Intelligence (TAAI) collectively hosted the "2008 Computational Intelligence Forum and World 9x9 Computer Go Championship". This event was conducted in Taiwan to fulfill the purpose "Enjoy Learning in the Digital Life through Computer Go playing." The activities were composed of two main parts: "Computational Intelligence Forum" and "World 9x9 Computer Go Championship". The World 9x9 Computer Go Championship was divided into two sections. Section A was computer program competitions, won by MoGo with no defeat, while section B was human versus computer competitions. The organizers hope through the championship, the fun in Go playing by human interaction with computer programs can be promoted, as well as stimulating the development and research in Computer Go.

In this chapter, we focus on the invited games for "Taiwanese Go players vs. computer program MoGo" held at NUTN in Taiwan. Several Taiwanese Go players, including one 9-Dan Professional (9P) Go player and eight amateur Go players ranging from 1 Dan (1D) to 6D, were invited by NUTN to play with MoGo from August 26 to October 4, 2008. In particular, Mr. Jun-Xun Zhou, a 9P Go player, played 9x9 and 19x19 games with MoGo running on a supercomputer with 800 CPUs, through KGS Go server on September 27, 2008. Mr. Zhou is the strongest Go player in Taiwan. He won the 2007

Title	Name	Age	Sex	Dan Grade
Mr.	Jun-Xun Zhou	28	Male	9D Professional
Mr.	Biing-Shiun Luoh	45	Male	6D Amateur
Prof.	Shang-Rong Tsai	55	Male	6D Amateur
Mr.	Cheng-Shu Chang	50	Male	6D Amateur
Prof.	Cheng-Wen Dong	70	Male	5D Amateur
Child	Yu-Shu Huang	12	Female	4D Amateur
Child	Yu-Xin Wang	11	Male	3D Amateur
Mr.	Wen-Tong Yu	50	Male	3D Amateur
Child	Sheng-Yu Tang	10	Male	2D Amateur

Table A.1: Profiles of all the Go players competing with MoGo.

Game Board	Komi	Time per side (min)
9x9	7.5 (unless otherwise stated)	30
19x19	7.5	45

Table A.2: Parameters of the games.

World LG-Cup. MoGo lost three games against Mr. Zhou, including two 9x9 games and one 19x19 game with 7 handicap stones. There was also one 19x19 game with 7 handicap stones and 45 minutes per side in the "Mr. Zhou versus MoGo" contest. MoGo had a very favorable situation in the first 9x9 game but made a big mistake and lost. The invited eight amateur Go players in Taiwan included the retired professor of NUTN (Prof. Dong, 70-years-old, 5D), the CIO of a software company (Mr. Chang, 50-years-old, 6D), the Chief Referee of this championship (Prof. Tsai, 55-years-old, 6D), two teachers of Tainan's Go Association (Mr. Luoh, 45-years-old, 6D, and Mr. Yu, 50-years-old, 3D), and three children of Tainan's Go Association (12-years-old, 4D, 11-years-old, 3D, and 10-years-old, 2D). The results revealed that MoGo might reach about 3D based on the amateur Taiwanese scale.

A.2 Game Results of MoGo vs. Human Players in Taiwan

The profiles of all the Go players competing with MoGo in Taiwan are listed in Table A.1. The Chinese rule was adopted and the related parameters of the game are listed in Table A.2.

During the tournament, MoGo ran respectively on the DELL PowerEdge

R900 machine with 16 cores and on the Supercomputer "Huygens" provided by the Dutch research organizations SARA and NCF. MoGo was allowed to use at most 25 out of the 104 nodes of the Supercomputer, i.e., 800 cores at 4.7GHz, with a floating point processing power of 15 Teraflop (more than 1000 times Deep-Blue). The game was played through the KGS Go server platform when MoGo ran on the Huygens cluster with different numbers of cores. Table B.1 and A.4 lists the related information and the results of the games that MoGo played against nine Taiwanese Go players in the tournament. The first column shows the game number and the second column denotes the performance of MoGo. The performance is represented by XD+ or XD- with X = L - H, where L is the rank of a player and H is the handicap level. If MoGo won, then its performance is XD+, otherwise its performance is XD-. The level Xkyu corresponds to -(X-1) Dan; the higher the Dan number, the stronger the player. The 9P Mr. Zhou (9Dan on pro scale) is assumed to be equivalent to 10D (10Dan on amateur scale). We also invited Mr. Biing-Shiun Luoh, who is a Go teacher with 6D amateur, and Professor Shang-Rong Tsai to comment on some game results. Their comments on 9x9 and 19x19 games are stated in the next two subsections, respectively.

A.2.1 Comments on 9x9 games

The games No. 15 and No. 16 are 9x9 and very interesting. In the two games, MoGo played with the 9P Go player, Mr. Zhou. The boards for these two 9x9 games are shown in Fig. A.1 and Fig. A.2, respectively.

According to Professor Tsai's and Mr. Zhou's comments, the game No. 15 was worth studying because MoGo might have had a chance to win. But unfortunately, MoGo was tricked by Mr. Zhou with White 20 to lose the game. Mr. Zhou analyzed that if the time per side could be lengthened, then MoGo would take much more advantage, which can be revealed in Fig. A.3. It indicates that the probability of playing the good move (E9) instead of one of the two bad moves was below 50% since 5 minutes cores of computation, which are quickly reached with the parallelization. The probability of playing D8, E2 or E9 depends on the computational effort. The bad moves D8 and E2 are probably played when MoGo has little time, and the probability of E9 increases with increasing computational effort.

9x9 Go is the first field invaded by the MCTS methods. However, the playing results in Taiwan were not good against top-level human players, as MoGo lost most of his 9x9 games: two games against Mr. Zhou, three games against Mr. Luoh. However, the komi has been modified in some games so that MoGo did not play the komi for which its openings were optimized. Besides, the first game against Mr. Zhou was difficult. The professional

No	Performance	Date	Setup	opponent
1	9x9 5D+	08/26/2008	9x9	5D
2	1kyu+	08/26/2008	19x19 H5	5D
3	9x9 6D+	08/26/2008	9x9	6D
4	9x9 6D-	08/26/2008	9x9	6D
5	2 kyu+	09/24/2008	19x19 H6	5D
6	1D+	09/24/2008	19x19 H4	5D
7	1D+	09/25/2008	19x19 H4	5D
8	1D+	09/25/2008	19x19 H4	5D
9	1D+	09/25/2008	19x19 H4	5D
10	9x9 6D-	09/25/2008	9x9	6D
11	9x9 6D-	09/25/2008	9x9	6D
12	2D-	09/25/2008	19x19 H4	6D
13	1D+	09/27/2008	19x19 H5	6D
14	1D+	09/27/2008	19x19 H5	6D
15	9x9 10D-	09/27/2008	9x9	9P
16	9x9 10D-	09/27/2008	9x9	9P
17	3D-	09/27/2008	19x19 H7	9P
18	3D+	10/02/2008	19x19	3D
19	2D+	10/02/2008	19x19 H4	6D
20	1D-	10/03/2008	19x19 H5	6D
21	1D-	10/03/2008	19x19 H5	6D
22	4D+	10/04/2008	19x19	4D
23	3D+	10/04/2008	19x19	3D
24	9x9 2D+	10/04/2008	9x9	2D

Table A.3: Related information and results of the games that MoGo played against humans in the tournament.

No	Environment	White	Black	Result
1	Huygens with 150CPUs	MoGo	Prof. Dong	W+0.5
2	Huygens with 150CPUs	Prof. Dong	MoGo	B+0.5
3	Huygens with 150CPUs	Prof. Tsai	MoGo	B+Resign
4	Huygens with 150CPUs	MoGo	Mr. Luoh	B+Resign
5	R900 machine	Prof. Dong	MoGo	B+Resign
6	R900 machine	Prof. Dong	MoGo	B+Resign
7	R900 machine	Prof. Dong	MoGo	B+Resign
8	R900 machine	Prof. Dong	MoGo	B+Resign
9	R900 machine	Prof. Dong	MoGo	B+0.5
10	Huygens with 320CPUs	Mr. Luoh	MoGo	W+Resign
11	Huygens with 320CPUs	MoGo	Mr. Luoh	B+Resign
12	Huygens with 320CPUs	Mr. Luoh	MoGo	W+Resign
13	Huygens with 480CPUs	Prof. Tsai	MoGo	B+1.5
14	Huygens with 480CPUs	Mr. Chang	MoGo	B+1.5
15	Huygens with 800CPUs	Mr. Zhou	MoGo	W+Resign
16	Huygens with 800CPUs	MoGo	Mr. Zhou	B+Resign
17	Huygens with 800CPUs	Mr. Zhou	MoGo	W+Resign
18	R900 machine	MoGo	Mr. Yu	W+11.5
19	R900 machine	Mr. Luoh	MoGo	B+7.5
20	R900 machine	Prof. Tsai	MoGo	W+Resign
21	R900 machine	Prof. Tsai	MoGo	W+Resign
22	R900 machine	Child Huang	MoGo	B+0.5
23	R900 machine	Child Wang	MoGo	B+2.5
24	R900 machine	Child Tang	MoGo	B+0.5

Table A.4: Related information and results of the games that MoGo played against humans in the tournament.

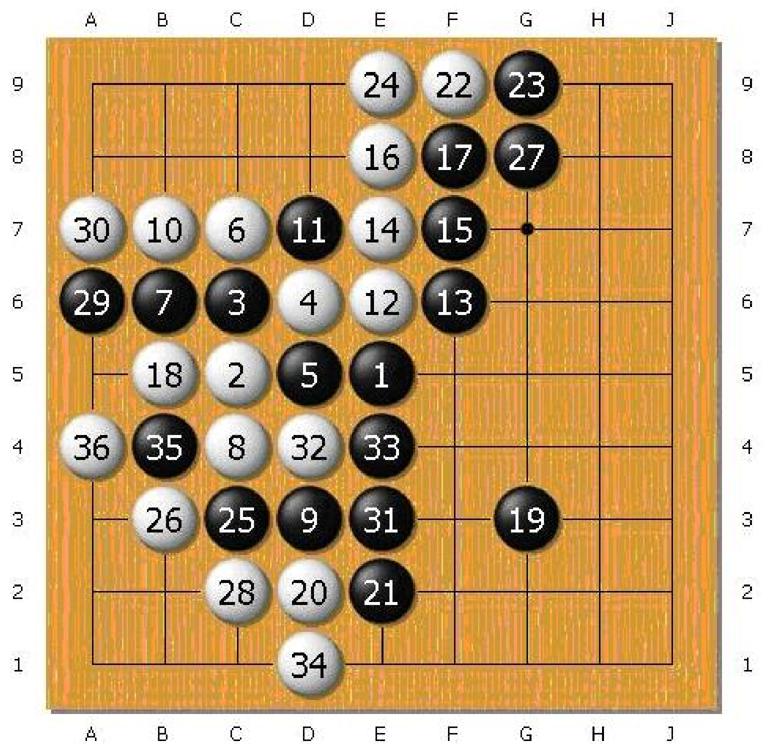


Figure A.1: Game No. 15. 9x9 game played against Mr. Zhou (9D Pro). MoGo was black and lost the game. The komi was 6.5, whereas MoGo had hard-coded first moves for komi 7.5. Comments by Prof. Tsai: MoGo was black. 20 was a good move of White (Mr. Zhou): black answered 21 E2 whereas E9 would lead to a win for black. Comments from Prof. Luoh: MoGo could also play C3 as a reply to D2. A posteriori analysis by MoGo on the situation after move 20: (1) MoGo inferred that it was likely to win with the move E9 (65% probability of winning, estimated after a few seconds of thinking) (2) MoGo did not see clearly that E2 was a bad move (MoGo stayed a long moment and got an estimate of roughly 50%), (3) MoGo was likely to play the good move E9, but could also play move D8 (loosing move) or E2. The probability of a good move increases with the computational effort, which can be observed in Fig. A.3.

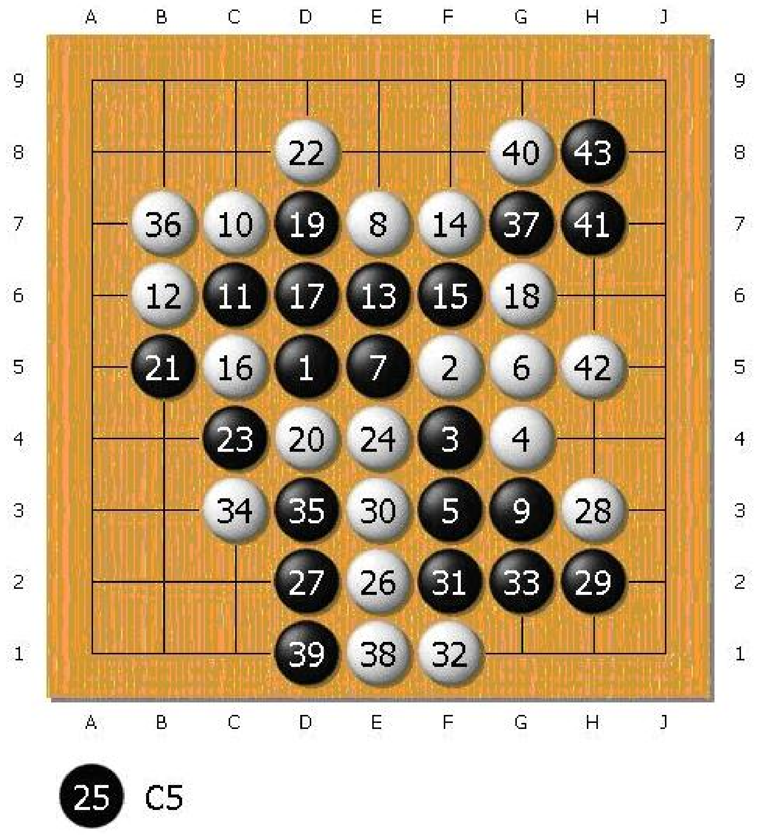


Figure A.2: Game No. 16. 9x9 game played against Mr. Zhou (9D Pro). MoGo was white and lost the game. The komi was 6.5, whereas MoGo had hard-coded first moves for komi 7.5. Comments by Prof. Tsai: White (MoGo) played a bad move 16 (C5). A posteriori analysis by MoGo on the situation before move 16 (C5): With the limited time per move, MoGo was likely to play the bad move C5 with 50% probability, and play G6 with the other 50% probability. Interestingly, MoGo, when playing C5, was aware of the fact that this move did not lead to a good situation. It, however, did not find a move with a better probability of winning. Some tricks like distributing the computational power on several moves when the situation seemed to be very good might be a good idea.

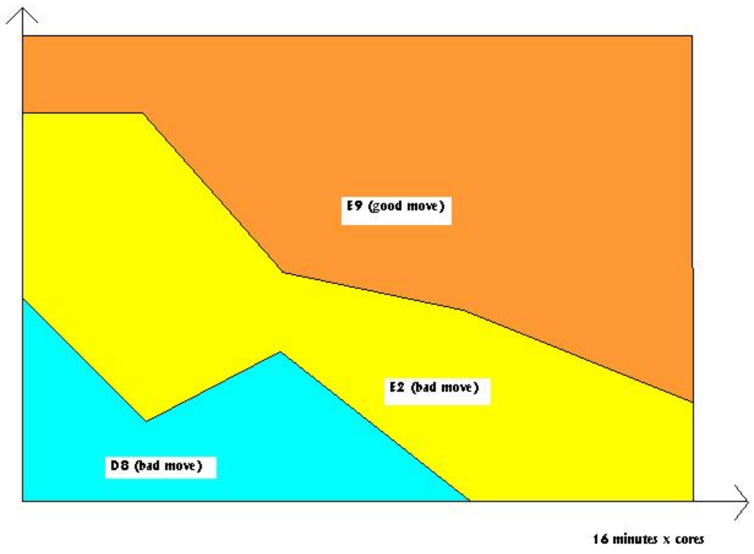


Figure A.3: Probability of playing the good move (E9).

player estimated during the game that MoGo was likely to win the game, before its big mistake. Nonetheless, MoGo won one out of the two games against 6D Professor Tsai. Figure A.4 and Fig. A.5 show the results of the game No. 3 and game No. 4, respectively. Figure A.6 shows the result of the game No. 10.

A.2.2 Comments on 19x19 games

In this subsection, we will discuss the properties of MoGo on 19x19 games. We focus on the following four features: (i) the main weakness of MoGo, namely corners, (ii) the scaling with time, (iii) the behavior in handicap games, and (iv) the main strength of MoGo in contact fights.

Weakness in the corners

The weakness in the corner appeared very clearly in the game against Mr. Zhou, in which MoGo lost his advantage in all the corners. Figures A.7 and A.8 display the results of the games No. 17 and No. 21, respectively. The game No. 17 with seven handicap stones clearly shows a strong weakness of the program. That was, life and death conditions in the corners could not be correctly judged by MoGo. The reason was that the Monte-Carlo simulator did not properly estimate “Semeai” situations. “Semeai” situations are the situations which involve a different way of reasoning based on counting

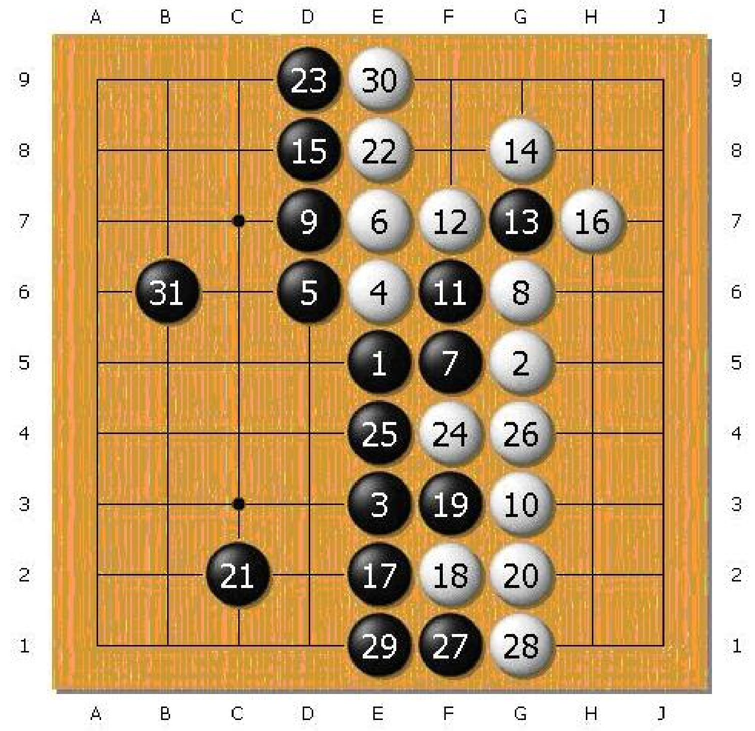


Figure A.4: Game No. 3. 9x9 game won by MoGo (Black) against Prof. Tsai (6D) Comments by Prof. Tsai: With Black (MoGo) playing good moves 11, 13 and 15, MoGo shows a good yose technique. Therefore, Black gets yose at 17.

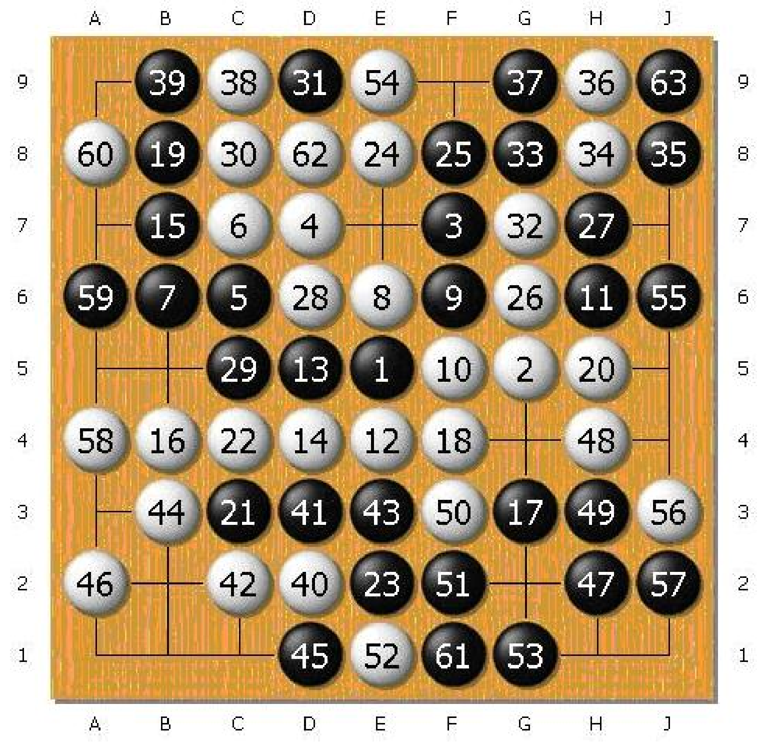


Figure A.5: Game No. 4. 9x9 game lost by MoGo (White) against Mr. Luoh (6D) Comments by Prof. Tsai: This game focused on complex fights. Therefore, there were so many variations in the game that it was difficult to analyze.

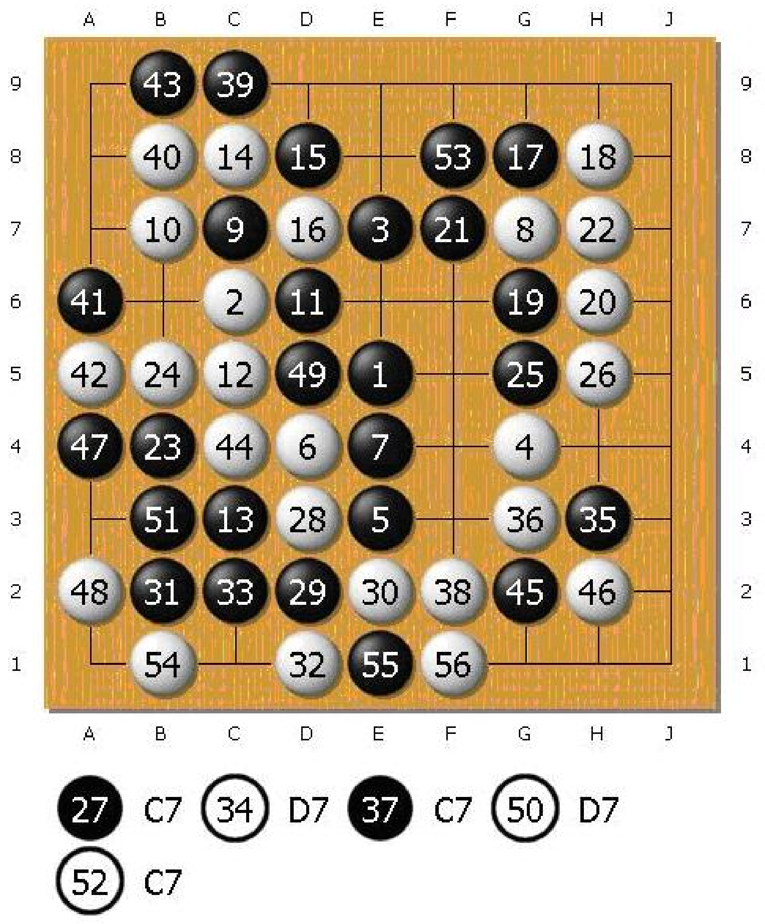


Figure A.6: Game No. 10. 9x9 game lost by MoGo (Black) against Mr. Luoh (6D) Comments by Mr. Luoh: If MoGo (Black) had played 37 at G2 instead of C7, Mr. Luoh (White) would have played at H4. In response to it, if Black had answered 46, Black would have won the game. But unfortunately, Black played 37 at C7 not G2, so Black lost the game.

liberties of groups. Other games, such as game No. 21, shown in Fig. A.8, illustrate the same weakness.

Scaling with time

It is well known that MoGo needs time to reach its best level. In particular, the 8P Go player, Myung-Wan Kim, could win against MoGo with 11 handicap stones by a setting a short time limit. In this game, the MoGo ran on the supercomputer Huygens with only 45 minutes per side. On the other hand, MoGo could win against Mr. Kim at the 2008 US Congress in Portland with 9 handicap stones for 90 minutes per side. In addition, Mr. Kim commented that MoGo could also probably win with 8 handicap stones only. Although humans also become stronger with a longer period of time, the results of Table IV below show that the improvement for humans is not so significant as for computers. But humans can spend a long time on some important moves whereas Monte-Carlo Tree Search programs usually spend a very regular time on each move. Criteria for deciding that spending more time on the current situation are necessary.

MoGo in handicap games

MoGo, as well as other Monte-Carlo Tree Search algorithms, is based on the best-first search. Hence, in the beginning of very strong handicap games, MoGo just studies a few moves and keeps simulating one of them to ensure that they keep a high probability of winning. As all moves have a high probability of winning at the beginning, for the underlying assumption of equal strength between the two players (essential assumption in the MCTS algorithms), MoGo keeps simulating only these initial moves. As a consequence MoGo plays the first moves almost randomly. This can be contrasted with the case of an equilibrated situation without handicap, in which MoGo will spend a lot of time on various moves until MoGo finds a move with a higher probability of winning. Interestingly, the same situation happens in games in which MoGo has the advantage (see the comments of the first 9x9 game lost against M. Zhou in Fig. A.1). This is clearly illlustrated by the successes of MoGo in non-handicap games, which are shown in Figs. A.9, A.10 and A.11, and by the statistics in section A.2.3.

Strength of MoGo in contact fights

It is well known that MCTS algorithms have a very strong ability in local fights. Figures A.12, A.13, and A.14 illustrate it.

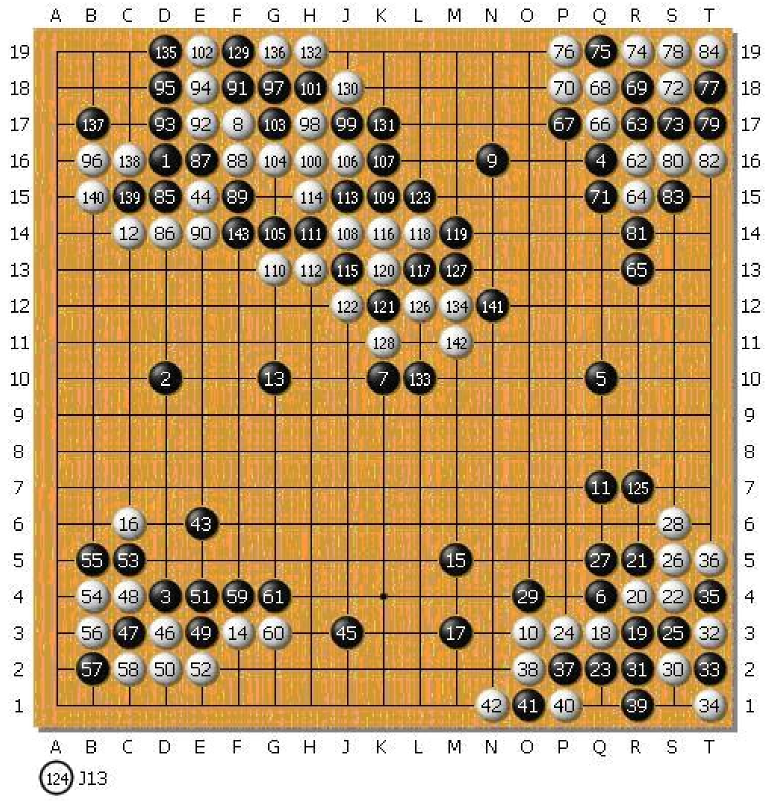


Figure A.7: Game No. 17. Game against Mr. Zhou (6D), with 7 handicap stones. MoGo was Black and lost the game. Comments by Prof. Tsai: This game was a bad game which MoGo played in this championship. White (Mr. Zhou) profited from the four corners which meant that MoGo might not be good at processing the corners in the game. So, White quickly won the game after Black (MoGo) lost points at the four corners.

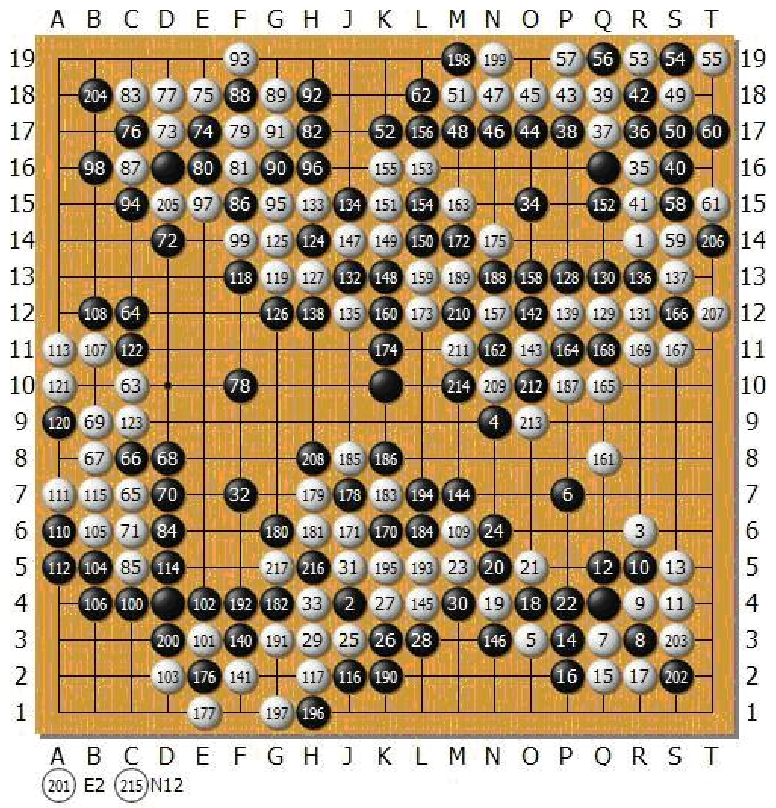


Figure A.8: Game No. 21. Game against Prof. Tsai (6D), with 5 handicap stones. MoGo was Black and lost the game. Comments by Prof. Tsai: In this game, Black (MoGo) made a mistake on the right upper corner so that Black lost the game. After playing with Black for some games, Prof. Tsai thought that Black had made such a mistake many times. This was not a good move.

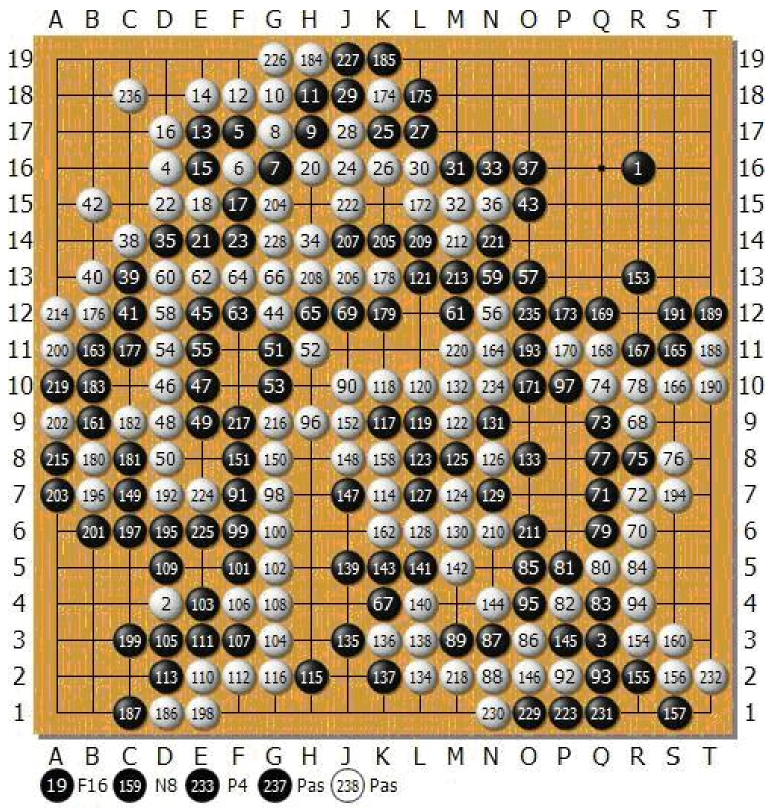


Figure A.9: Game No. 18. Game against Mr. Yu (3D), without handicap stones. MoGo was White and won the game.

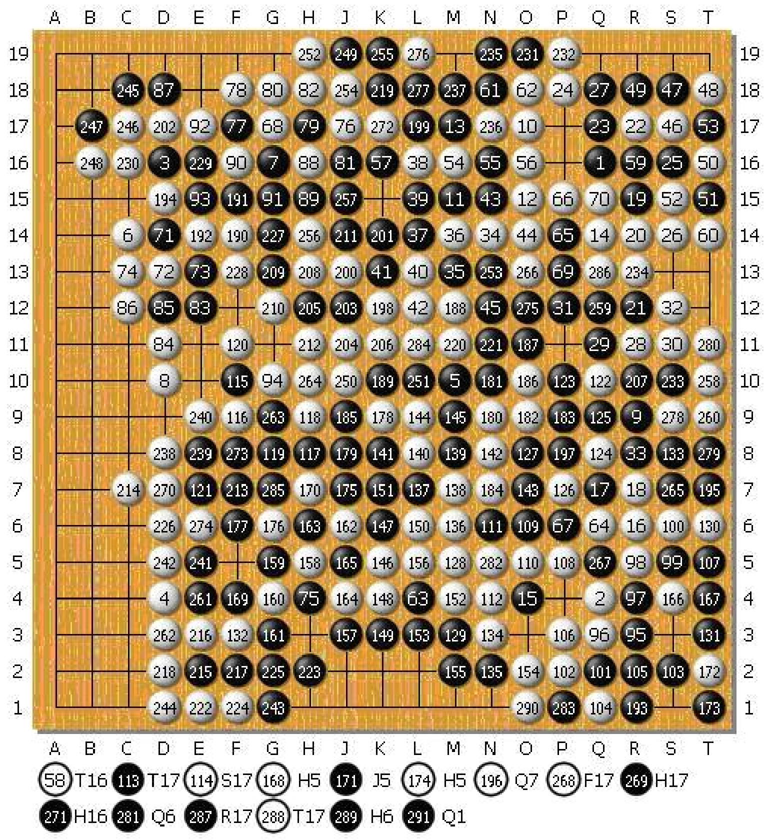


Figure A.10: Game No. 22. Game against Child Huang (4D), without handicap stones. MoGo was Black and won the game. A big fight was important in the game, which was favorable for MoGo.

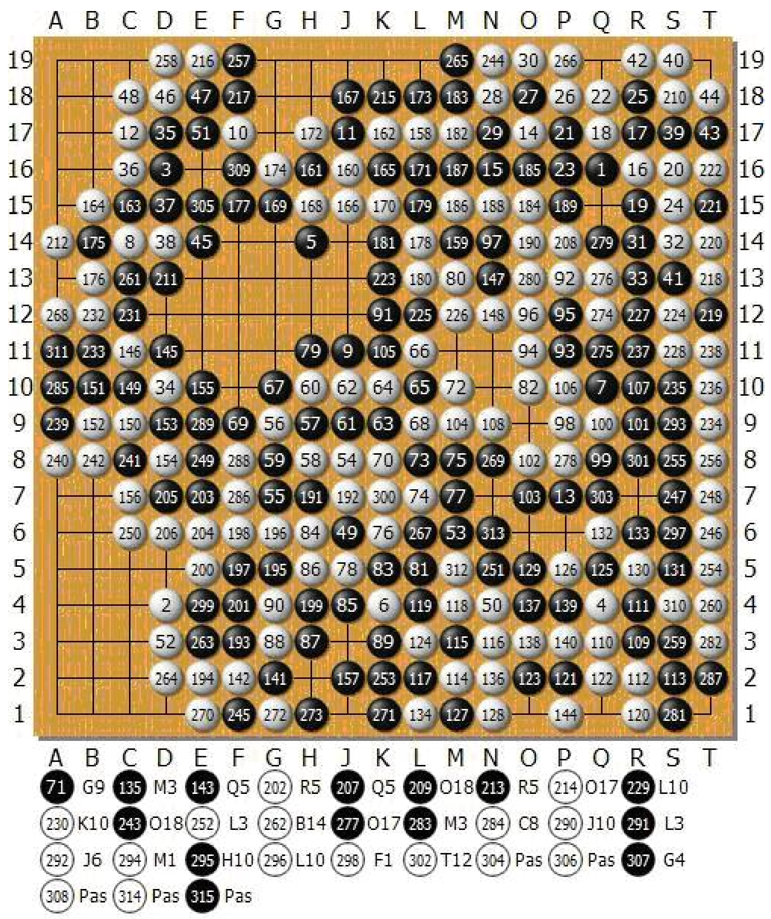

71 G9 135 M3 143 Q5 202 R5 207 Q5 209 O18 213 R5 214 O17 229 L10
230 K10 243 O18 252 L3 262 B14 277 O17 283 M3 284 C8 290 J10 291 L3
292 J6 294 M1 295 H10 296 L10 298 F1 302 T12 304 Pas 306 Pas 307 G4
308 Pas 314 Pas 315 Pas

Figure A.11: Game No. 10. Game against Child Wang (3D), without handicap stones. MoGo was Black and won the game.

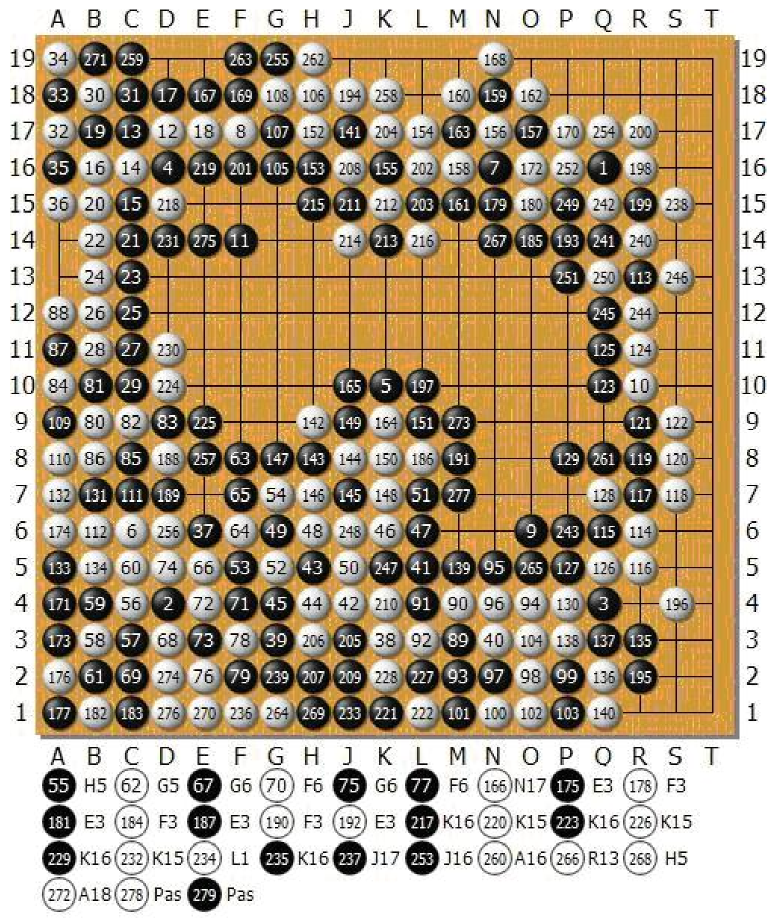


Figure A.12: Game No. 13. Game against Prof. Tsai (6D), with 5 handicap stones. MoGo was Black and won the game. Comments by Prof. Tsai: Originally, White (Prof. Tsai) should have had a great chance to win in the middle of the game. When White played 142 to attempt to break into Black's territory, Black played 143 and 145 to cut White's stones. Meanwhile, White made another mistake. That is, White played a bad move at 146. If White had played 146 at G8 instead of H7, White would have successfully intruded into Black's territory to get more than 10 points. From the board of this game, Prof. Tsai said that MoGo had a good performance on the contact fight. In spite of the fact that Black (MoGo) also lost some points at the corners in this game, White ended up losing the game because of this vital mistake. Another key point of this game was the ko fight at 156 and 157. From the result of ko fight, Black also performed ko fight well in this game.

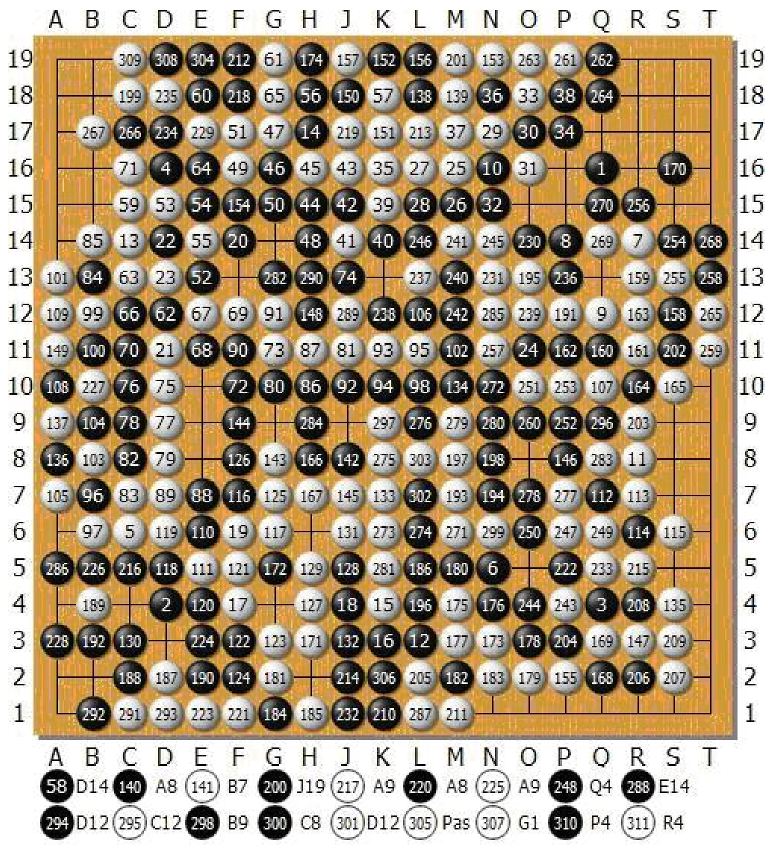

58 D14 140 A8 141 B7 200 J19 217 A9 220 A8 225 A9 248 Q4 288 E14
294 D12 295 C12 298 B9 300 C8 301 D12 305 Pas 307 G1 310 P4 311 R4

Figure A.13: Game No. 12. Game against Mr. Luoh (6D), with 4 handicap stones. MoGo was Black and lost the game. Comments by Prof. Tsai: Black (MoGo) played well in the beginning of the game, but made big mistakes at the end of the game. The key point of this game was at the fight located at the left side. Most of the time, Black played reasonable moves, but critical mistakes, such as Black 68, caused Black to lose the game. Anyway, Black had a good performance in this game.

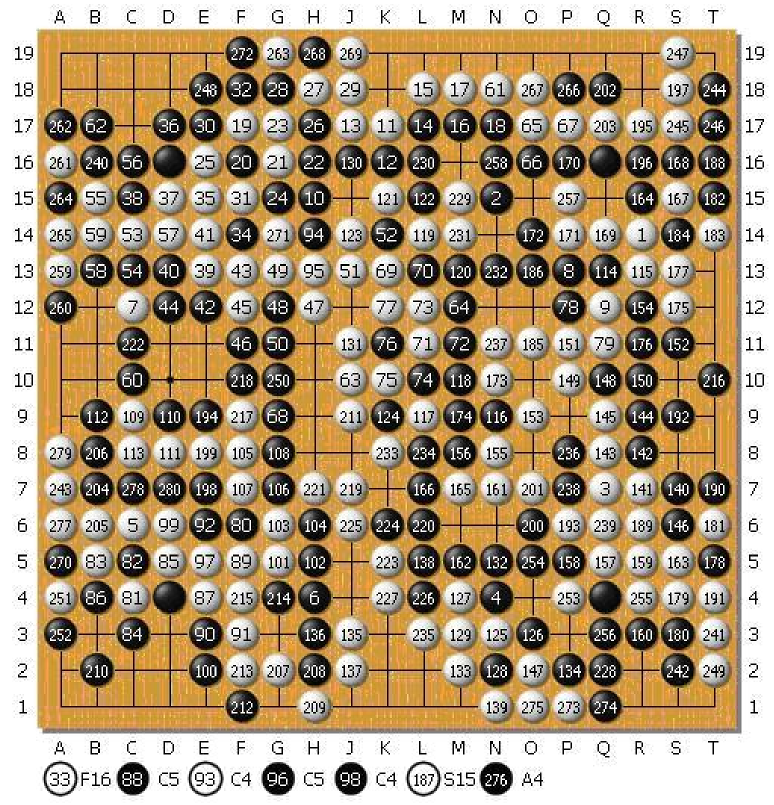


Figure A.14: Game No. 19. Game against Mr. Luoh (6D), with 4 handicap stones. MoGo was Black and won the game. Comments by Mr. Luoh: Black (MoGo) played the locally optimal move when White (Mr. Luoh) played 25, which caused White to play in a difficult situation later. Therefore, Black won the game. From the result of this game, Mr. Luoh said that Black could not only detect the locally optimal move but also had a strong center territory performance. But, Black performed poorly in managing the edges and corners of the board.

A.2.3 Numerical analysis of performance

A classical formula of likelihood used in the IGS rating system (http://www.pandanet.co.jp/English/ratingsystem/) estimates the level by likelihood maximization, based on the following formulas. The probability of losing against the L Dan player with H handicap stones if one's level is B, is estimated by the following evaluation:

- Effective advantage of the opponent $A = L - H - B$;
- $Likelihood = 1 - (3/4)^{2A}/2$ if $A > 0$;
- $Likelihood = (3/4)^{2A}/2$ if $A < 0$.

MoGo's level can then be evaluated by maximizing the overall likelihood, i.e. the product of all likelihoods. The following results can be obtained.

- (i) MoGo's level against all games was evaluated as a bit less than 2D (1.6D). It can further be divided into two kinds: - Games played by the R900 machine (16 cores, 3GHz): 2.5D, and - Games played by the Huygens cluster: 1.7D. This is quite surprising at first view, as the Huygens cluster is a powerful machine and the speed-up is very good. The reason is that Huygens has been devoted to the games against Mr. Jun-Xun Zhou and 6D players, with a large number of handicaps, and thus has low performance in the framework. This could be contrasted with the fact that the R900 machine was tested against players in the range of 1D to 4D. This effect introduces a much stronger bias than the computational power. In particular, strong players always defeated MoGo in corners, whatever may be the handicap and this will not be solved by computational power.

- (ii) MoGo's level against the games with at most 4 handicap stones (as it is known that MCTS algorithms do not handle handicap properly) was evaluated as 5D (5.3D). This is surely too high as an estimate. The won game against a 4D player involved big fights, a situation which was quite favorable to MoGo. Besides, MoGo might have lost in other situations - changing just one game would have a big impact on the estimate, due to the small number of games. We conclude that MoGo is estimated to have a 2D-3D with (i) good skills for fights (ii) weaknesses in corners (iii) weaknesses in favorable situations as in handicapped games.

A.3 Conclusion

In this chapter, the advances in computational intelligence of MoGo are revealed from Taiwan's Computer Go tournaments. MoGo is introduced through the Monte-Carlo player, the formulas used for biasing the Monte-Carlo simulator, and the discussion over the parallelization. According to the comments made by the Go players, MoGo performed well in fighting and surrounding center territory. In addition, MoGo is very intelligent and its capability is beyond their expectations. Therefore, MoGo should be around 1P professional and 2D-3D amateur for 9x9 and 19x19 games in Taiwan, respectively. However, according to the Go players' comments, MoGo still has some drawbacks such as the skill in corners and edges, as the Monte-Carlo player does not solve these issues.

Appendix B

Commentaries on Games played by MoGo in Jeju

B.1 Introduction

During IEEE Fuzz, in Jeju Island, games were played between four of the current best programs against a top level professional player Chou-Hsun Chou and a high-level amateur Shen-Su Chang, with in particular the first win of a computer program (the Canadian program Fuego) against a 9p player in 9x9 as white. On the other hand, none of the program could win against Chou-Hsun Chou in 19x19, in spite of the handicap 7, showing that winning with handicap 7 against a top level player is still almost impossible for computers, in spite of the win by MoGo a few months ago with handicap 7.

Chou-Hsun Chou is a top level professional player born in Taiwan. He became professional in 1993 and reached 7P in 1997 and 9p in 1998. He won the LG Cup in 2007, beating Hu Yaoyu 2 to 1.

Shen-Su Chang is a 6D amateur from Taiwan.

B.2 Results and comments

The overall results are presented in Table B.1 and discussed in the rest of this chapter. All comments around the game of Go are given by experts: Chun-Hsun Chou 9P, Shen-Su Chang 6D, Shi-Jim Yen 6D, Shang-Rong Tsai 6D. The ability of MCTS for fights is illustrated in section B.2.1. The 9x9 opening books are discussed in section B.2.2. The weaknesses in corners are discussed in section B.2.3. The aggressivity of bots is discussed in section B.2.4. The weakness in semeais and in seki, probably the current most important weakness, is discussed in section B.2.5.

No	Rank	Date	Setup	White	Black	Result
Scheduled games						
1	9P	08/21/2009	19x19 H7	Mr. Chou	Many Faces of Go	W+Res.
2	6D	08/21/2009	19x19 H4	Mr. Chang	MoGo	W+Res.
3	9P	08/21/2009	9x9	MoGo	Mr. Chou	B+Res.
4	6D	08/21/2009	9x9	Many Faces of Go	Mr. Chang	B+6.5
5	9P	08/21/2009	9x9	Mr. Chou	MoGo	W+Res.
6	6D	08/21/2009	9x9	Mr. Chang	Many Faces of Go	W+Res.
7	9P	08/22/2009	9x9	Fuego	Mr. Chou	W+2.5
8	6D	08/22/2009	9x9	Zen	Mr. Chang	W+Res.
9	9P	08/22/2009	9x9	Mr. Chou	Fuego	W+Res.
10	6D	08/22/2009	9x9	Mr. Chang	Zen	B+Res.
11	9P	08/22/2009	19x19 H7	Mr. Chou	Zen	W+Res.
12	6D	08/22/2009	19x19 H4	Mr. Chang	Fuego	B+Res.

Table B.1: Overview of results; games played during IEEE-Fuzz at Jeju Island, Korea.

B.2.1 Ability for fights

MCTS/UCT algorithms are known for being very strong in killing. This is illustrated in the game won by Zen as white against Shen-Su Chang 6D (Fig. B.1, left).

B.2.2 9x9 opening books

We distinguish below handcrafted opening books and self-built opening books.

Handcrafted opening books

Fuego's opening book is handcrafted; nonetheless, Fuego plays a bad move very early, namely the "kosumi" (move 3, Fig. B.2, left). This move was supposed to be good with a komi of 6.5 but is not aggressive enough with a komi of 7.5. Kosumis (diagonal move), according to [Audouard et al., 2009], are very often bad moves in the beginning of a 9x9 game. On the other hand, Fuego won as white with good opening moves (only 3 moves in the opening book), see Fig. B.2 (right).

Opening moves by Zen were all good in 9x9 according to experts; Zen won one game as black and one game as white against Shen-Su Chang 6D (Fig. B.1). There were very few moves in the opening book.

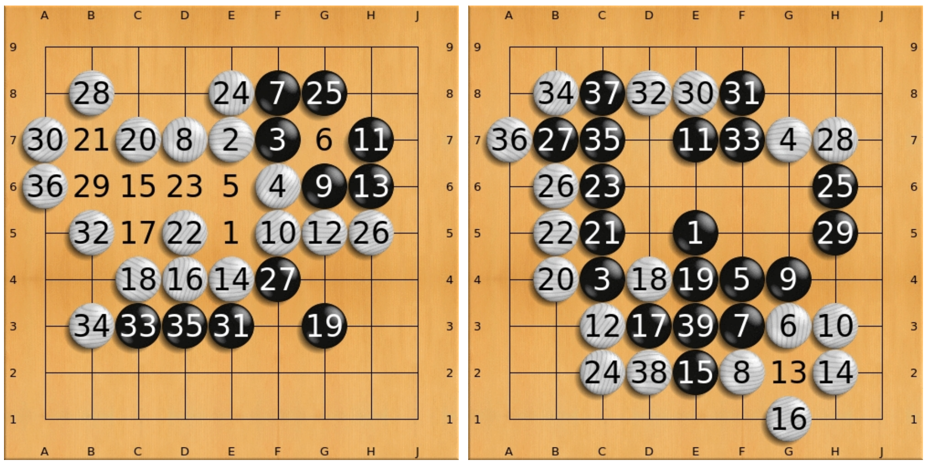


Figure B.1: Left: game won by Zen as white against Shen-Su Chang 6D; black made a mistake (move 29 at B6 instead of B4), immediately punished by White killing E5. Right: game won by Zen as black against Shen-Su Chang 6D. In both cases, Zen had good opening moves. As black, Zen had a big moyo.

Self-built opening books

MoGo has a huge opening book built on a cluster [Audouard et al., 2009]. However, the two openings (black and white) contained mistakes which were exploited by Chun-Hsun Chou 9P, who won both as black and as white against MoGo (Figure B.3).

B.2.3 Weaknesses in corners

It is known that MCTS algorithms have a bad strategy, as they try to develop a big moyo instead of focusing in corners; this has been related to cosmic go. In 9x9, having a big moyo can be efficient, as in e.g. Fig. B.1 (right) where Zen, with a big moyo only, wins the game as black. On the other hand, in 19x19, protecting the moyo is very difficult, and it is therefore often preferable to take care of corners.

For example, ManyFaces lost against Chun-Hsun Chou 9P in spite of handicap 7 with 4 corners taken by the pro, and then the moyo also invaded (N15, N11 at least can have access to the moyo, Fig. B.4, left). Zen and MoGo lost against Chun-Hsun Chou 9P with the same settings. Shen-Su Chang won his games with H4, except the one against Fuego (Fig. B.4, right) in which he made a mistake and could not invade the moyo.

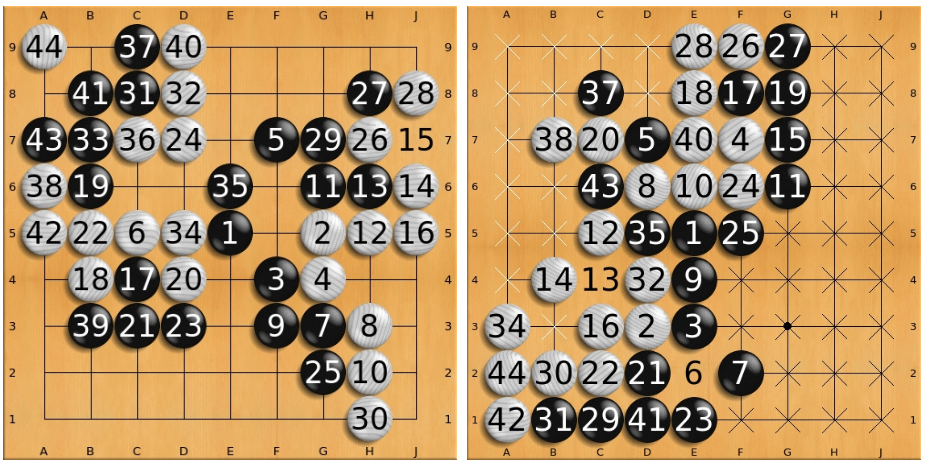


Figure B.2: Left: game won as white by Chun-Hsun Chou 9P against Fuego. Move 3 (handcrafted move from the opening book) is a kosumi and is considered to be bad in early 9x9 game. Right: game won as white by Fuego against Chun-Hsun Chou 9p; according to experts the opening by Fuego was good.

B.2.4 Robots are too aggressive

It is often said that MCTS programs are quite efficient for killing, but that they are too confident on their ability to kill. This is confirmed in *e.g.* Fig. B.6.

B.2.5 Weaknesses in semeais and sekis

MCTS programs are known for being weak in semeais; this is also true for sekis.

Figure B.2, where Fuego made a mistake in the opening, is also an example of semeai, as B8 could only live by killing A5; however there are much more liberties for white which easily kills B8 by nakade.

Figure B.7 (left) shows an example in which a seki was used by the human for winning as black against ManyFaces in 9x9. Fig. B.7 (right) shows an example in which the human won by semeai against ManyFaces, also in 9x9.

Figure B.5 (left) shows that Zen lost a semeai in the upper-right corner, and Fig. B.5 (right) shows that MoGo lost a semeai in the upper right corner, and only understood it when the situation was completely clarified by the pro.

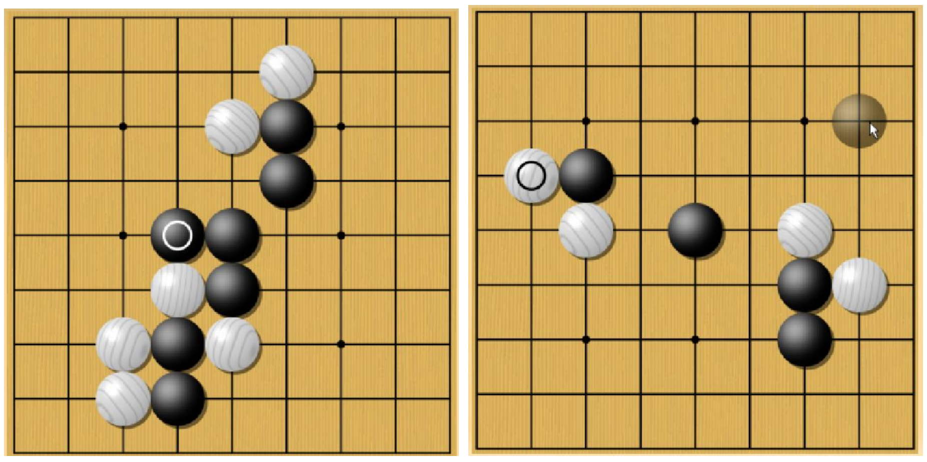


Figure B.3: Situation at the end of MoGo's opening book as white (left) and black (right). According to Chun-Hsun Chou 9P, the situation at the end of the opening book (the two situations presented here) was bad.

B.3 Conclusions

During IEEE Fuzz in Jeju Island, Fuego won the first ever victory of a computer against a top pro in 9x9 with komi 7.5 as white. Komi should be smaller according to the experts, if we want the setting to be fair; this would have a big impact on the opening book. The 9x9 opening books could easily be made stronger with the help of high-level players; current handcrafted opening books are too short, and automatically built opening books contain errors. Humans suggest 13x13 as a future challenge, and also consider that ensuring a win with handicap 7, from the current strength of programs, should be possible by including a big fuseki database.

Technically speaking, semeais and sekis are still poorly analyzed by MCTS, in spite of many research on criticality [Coulom, 2009] and introduction of tactical solvers [Cazenave and Helmstetter, 2005]. Also, MCTS programs are too much interested in the moyo and neglect the corners. There's no learning from one branch of the tree search to another, and no learning on the Monte-Carlo part in current programs.

It is interesting to point out the tools that were used also in other successful applications of MCTS/UCT. UCT is the most classical formula used in one-player applications ([Auger and Teytaud, pted, Rolet et al., 2009] for non-linear optimization and active learning respectively), but there are other bandit rules also ([De Mesmay et al., 2009] for optimization on grammars, using max-bandits). There are plenty of applications to other games; for Havannah (a game which is specially dif-

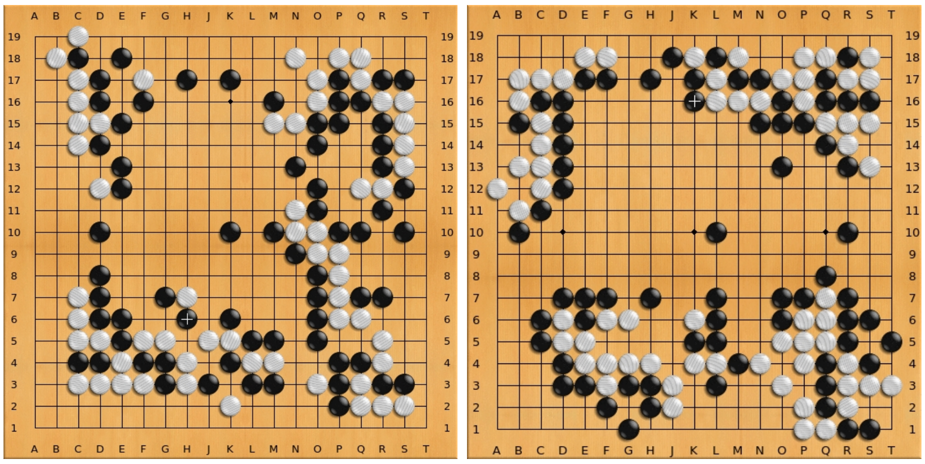
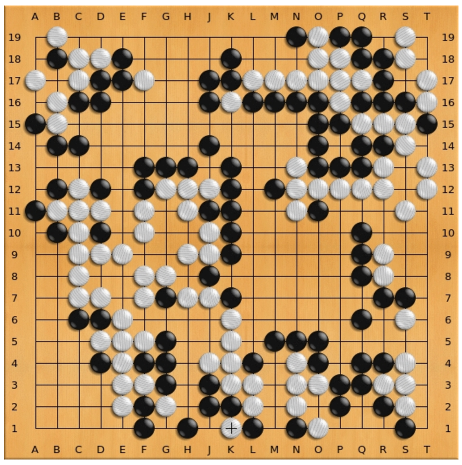


Figure B.4: Left: ManyFaces was black, handicap 7, against Chun-Hsun Chou 9P and lost with the 4 corners taken by the pro; the pro also invaded the moyo. Right: Fuego was black, H4, against Shen-Su Chang 6D. White was in very good situation on the picture, but played a bad move instead of L15 which would invade the moyo and win. Fuego could keep the moyo and therefore won.

ficult for computers and for which the RAVE heuristic is highly efficient [Teytaud and Teytaud, 2009]), general game playing [Sharma et al., 2008]; multiplayer games [Sturtevant, 2008] and in particular multiplayer Go [Cazenave, 2008]. It has been shown that for sudden-death games there are fruitful possible modifications [Winands et al., 2008], and for partially observable games like phantom-Go heuristic adaptations have been proposed [Cazenave, 2006, Cazenave and Borsboom, 2007] - a principled application to the partially observable case has been proposed in [Rolet et al., 2009] but it is deeply limited to one-player applications.



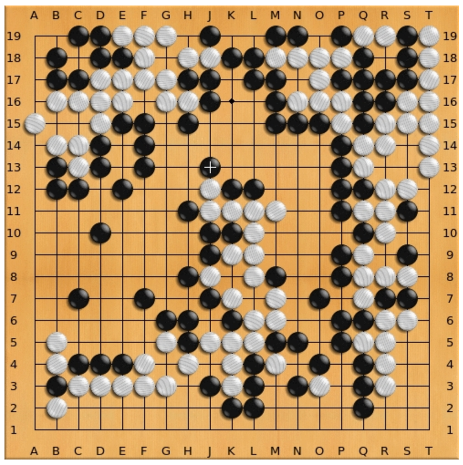


Figure B.5: Left: Zen was black, handicap 7, against Chun-Hsun Chou 9P and lost with 3 corners taken by the pro (white stones on the bottom right are dead); the pro also invaded the moyo. The situation was good for black at move 65 but after that Zen made some mistakes by not defending the corners which caused the lost. Right: MoGo was black, H7, against Chun-Hsun Chou 9P; as in other 19x19 games, the pro takes most of the corners, invades the moyo by playing K4 and wins. MoGo could have prevented the invasion by playing K4 itself instead of F13.

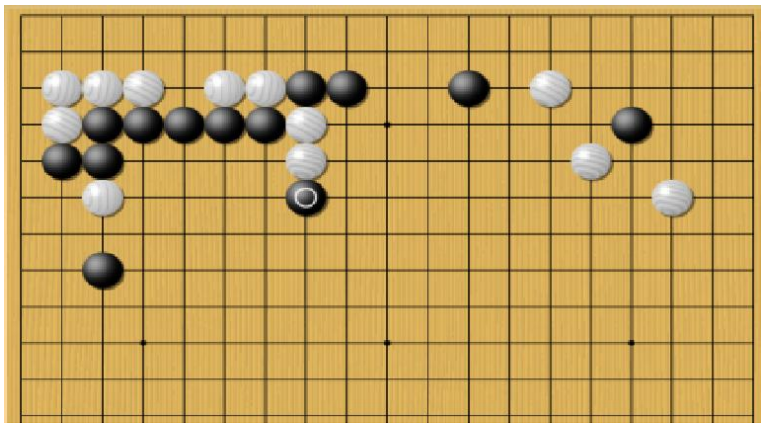


Figure B.6: MoGo is playing as black against Shen-Su Chang with H4. MoGo plays the circled black stone, trying to kill the two white stones; this was impossible, and as MoGo was keeping trying to kill white, it lost the upper center part of the goban and lost.

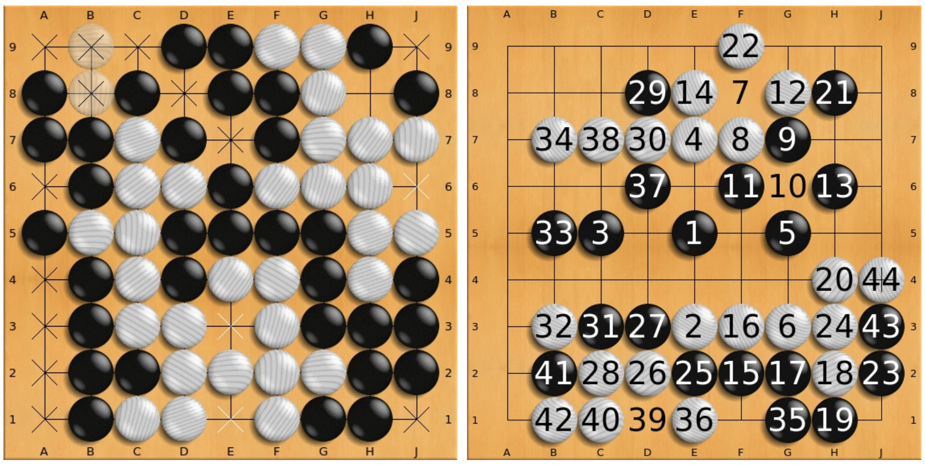


Figure B.7: Left: ManyFaces plays as white and has two groups alive; nonetheless, black wins thanks to the seki in the upper right corner (the two black stones are alive). Right: ManyFaces plays as black and looses by semeai in the lower part. In both cases, ManyFaces was playing against Shen-Su Chang 6D.

Bibliography

[Agrawal, 1995] Agrawal, R. (1995). The continuum-armed bandit problem. *Society for Industrial and Applied Mathematics J. Control Optim.*, 33(6):1926–1951.

[Amdahl, 1967] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA. ACM.

[Audouard et al., 2009] Audouard, P., Chaslot, G., Hoock, J.-B., **Rimmel, Arpad**, Perez, J., and Teytaud, O. (2009). Grid coevolution for adaptive simulations; application to the building of opening books in the game of Go. In *EvoGames*, Tuebingen Allemagne. Springer.

[Auer et al., 2002] Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2/3):235–256.

[Auger and Teytaud, pted] Auger, A. and Teytaud, O. (Accepted). Continuous lunches are free plus the design of optimal optimization algorithms. *Algorithmica*.

[Banks and Sundaram, 1992] Banks, J. S. and Sundaram, R. K. (1992). Denumerable-armed bandits. *Econometrica*, 60(5):1071–96. available at http://ideas.repec.org/a/ecm/emetrp/v60y1992i5p1071-96.html.

[Berry et al., 1997] Berry, D. A., Chen, R. W., Zame, A., Heath, D. C., and Shepp, L. A. (1997). Bandit problems with infinitely many arms. *Ann. Statist.*, 25(5):2103–2116.

[Bertsekas, 2009] Bertsekas, D. P. (2009). Neuro-dynamic programming. In *Encyclopedia of Optimization*, pages 2555–2560.

[Bouzy, 2005] Bouzy, B. (2005). Associating domain-dependent knowledge and monte carlo approaches within a go program. In Chen, K., editor, *Information Sciences, Heuristic Search and Computer Game Playing IV*, volume 175, pages 247–257.

[Bouzy and Chaslot, 2005] Bouzy, B. and Chaslot, G. (2005). Bayesian generation and integration of k-nearest-neighbor patterns for 19x19 go. In *G. Kendall and Simon Lucas, editors, IEEE 2005 Symposium on Computational Intelligence in Games, Colchester, UK*, pages 176–181.

[Bouzy and Helmstetter, 2003] Bouzy, B. and Helmstetter, B. (2003). Developments on monte carlo go. In *Advances in Computer Games 10.*

[Bruegmann, 1993] Bruegmann, B. (1993). Monte-carlo go. available at http://www.cgl.ucsf.edu/go/Programs/Gobble.html.

[Buro, 2001] Buro, M. (2001). Toward opening book learning. In *Machines that learn to play games*, pages 81–89, Commack, NY, USA. Nova Science Publishers, Inc.

[Cazenave, 2006] Cazenave, T. (2006). A phantom-go program. In *Advances in Computer Games*, pages 120–125.

[Cazenave, 2008] Cazenave, T. (2008). Multi-player go. In *Computers and Games*, pages 50–59.

[Cazenave, 2009] Cazenave, T. (2009). Nested monte-carlo search. *International Joint Conferences on Artificial Intelligence*, pages 456–461.

[Cazenave and Borsboom, 2007] Cazenave, T. and Borsboom, J. (2007). Golois wins phantom go tournament. *International Computer Games Association Journal*, 30(3):165–166.

[Cazenave and Helmstetter, 2005] Cazenave, T. and Helmstetter, B. (2005). Combining tactical search and monte-carlo in the game of go. *IEEE CIG 2005*, pages 171–175.

[Cazenave and Jouandeau, 2007] Cazenave, T. and Jouandeau, N. (2007). On the parallelization of UCT. In *Proceedings of Computer Games Workshop 2007*, pages 93–101.

[Chaslot et al., 2009] Chaslot, G., Fiter, C., Hoock, J.-B., **Rimmel, Arpad**, and Teytaud, O. (2009). Adding expert knowledge and exploration in Monte-Carlo Tree Search. In *Advances in Computer Games*, Pamplona Espagne. Springer.

[Chaslot et al., 2007] Chaslot, G., Winands, M., Uiterwijk, J., van den Herik, H., and Bouzy, B. (2007). Progressive strategies for monte-carlo tree search. In Wang, P. et al., editors, *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 655–661. World Scientific Publishing Co. Pte. Ltd.

[Chaslot et al., 2008] Chaslot, G., Winands, M., and van den Herik, H. (2008). Parallel monte-carlo tree search. In van d. Herik., H., X.Xu, Ma, Z., and Winands, M., editors, *Proceedings of the Conference on Computers and Games 2008 (CG 2008)*, volume 5131 of *Lecture Notes in Computer Science*, pages 60–71. Springer, Berlin Heidelberg.

[Chatriot et al., 2008] Chatriot, L., Fiter, C., Chaslot, G., Gelly, S., Hoock, J.-B., Perez, J., **Rimmel, Arpad**, and Teytaud, O. (2008). Combiner connaissances expertes, hors-ligne, transientes et en ligne pour l'exploration Monte-Carlo. *Revue d'Intelligence Artificielle.*

[Cicirello and Smith, 2005] Cicirello, V. and Smith, S. (2005). The max k-armed bandit: A new model for exploration applied to search heuristic selection. In *Proc. 20th National Conf. on Artificial Intelligence (AAAI)*, pages 1355–1361.

[Collet et al., 2000] Collet, P., Lutton, E., Raynal, F., and Schoenauer, M. (2000). Polar ifs + parisian gp = efficient inverse ifs problem solving. *Genetic Programming and Evolvable Machines*, 1(4):339–361.

[Coquelin and Munos, 2007] Coquelin, P.-A. and Munos, R. (2007). Bandit algorithms for tree search. In *Proceedings of Uncertainty in Artificial Intelligence 2007.*

[Coulom, 2006] Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy.*

[Coulom, 2007] Coulom, R. (2007). Computing elo ratings of move patterns in the game of go. In *Computer Games Workshop, Amsterdam, The Netherlands.*

[Coulom, 2009] Coulom, R. (2009). Criticality: a monte-carlo heuristic for go programs. Invited talk at the University of Electro-Communications, Tokyo, Japan.

[Dani and Hayes, 2006] Dani, V. and Hayes, T. P. (2006). Robbing the bandit: less regret in online geometric optimization against an adaptive adversary. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 937–943, New York, NY, USA. ACM Press.

[De Mesmay et al., 2009] De Mesmay, F., **Rimmel, Arpad**, Voronenko, Y., and Püschel, M. (2009). Bandit-Based Optimization on Graphs with Application to Library Performance Tuning. In *International Conference on Machine Learning*, Montréal Canada.

[Donninger and Lorenz, 2006] Donninger, C. and Lorenz, U. (2006). Innovative opening-book handling. In *Advance in Computer Games*, pages 1–10.

[DP Landau, 2005] DP Landau, K. B. (2005). *A guide to Monte Carlo simulations in statistical physics.*

[Drake and Chen, 2008] Drake, P. and Chen, Y.-P. (2008). Coevolving partial strategies for the game of go. In *International Conference on Genetic and Evolutionary Methods.* CSREA Press.

[Frigo and Johnson, 2005] Frigo, M. and Johnson, S. (2005). The design and implementation of FFTW3. *Proc. IEEE*, 93(2):216–231.

[Gelly et al., 2008] Gelly, S., Hoock, J.-B., **Rimmel, Arpad**, Teytaud, O., and Kalemkarian, Y. (2008). On the Parallelization of Monte-Carlo planning. In *International Conference on Informatics in Control, Automation and Robot*, Madeira Portugal.

[Gelly and Silver, 2007] Gelly, S. and Silver, D. (2007). Combining online and offline knowledge in uct. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 273–280, New York, NY, USA. ACM Press.

[Hussain et al., 2006] Hussain, Z., Auer, P., Cesa-Bianchi, N., Newnham, L., and Shawe-Taylor, J. (2006). Exploration vs. exploitation challenge. *Pascal Network of Excellence.*

[Johnson and Püschel, 2000] Johnson, J. and Püschel, M. (2000). In search of the optimal Walsh-Hadamard transform. In *Proc. Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, volume 6, pages 3347–3350.

[Kato and Takeuchi, 2008] Kato, H. and Takeuchi, I. (2008). Parallel monte-carlo tree search with simulation servers. In *13th Game Programming Workshop (GPW-08).*

[Kocsis and Szepesvari, 2005] Kocsis, L. and Szepesvari, C. (2005). Reduced-variance payoff estimation in adversarial bandit problems. In *Proceedings of the ECML-2005 Workshop on Reinforcement Learning in Non-Stationary Environments.*

[Kocsis and Szepesvari, 2006] Kocsis, L. and Szepesvari, C. (2006). Bandit-based monte-carlo planning. In *European Conference on Machine Learning 2006*, pages 282–293.

[Korf, 1985] Korf, R. (1985). Depth-first Iterative Deepening: an Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97–109.

[Korf and Chickering, 1994] Korf, R. and Chickering, D. (1994). Best-first search. *Artificial Intelligence*, 84:299–337.

[Korpaas et al., 2003] Korpaas, M., Holen, A. T., and Hildrum, R. (2003). Operation and sizing of energy storage for wind power plants in a market system. *International Journal of Electrical Power & Energy Systems*, 25(8):599–606.

[Krauth, 2006] Krauth, W. (2006). *Algorithms and Computations.* Oxford University Press.

[Lai and Robbins, 1985] Lai, T. and Robbins, H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6:4–22.

[Lee et al., 2009a] Lee, C.-S., Mei Hui, W., Hong, T.-P., Chaslot, G., Hoock, J.-B., **Rimmel, Arpad**, Teytaud, O., and Kuo, Y.-H. (2009a). A Novel Ontology for Computer Go Knowledge Management. In *IEEE FUZZ*, Jeju Corée, République de.

[Lee et al., 2009b] Lee, C.-S., Wang, M.-H., Chaslot, G., Hoock, J.-B., **Rimmel, Arpad**, Teytaud, O., Tsai, S.-R., Hsu, S.-C., and Hong, T.-P. (2009b). The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in games.*

[Metropolis and Ulam, 1949] Metropolis, N. and Ulam, S. (1949). The monte carlo method. In *Journal of the american statistical association*, volume 44.

[Müller, 2002] Müller, M. (2002). Computer go. *Artificial Intelligence*, 134(1-2):145–179.

[Nagashima et al., 2006] Nagashima, J., Hashimoto, T., and Iida, H. (2006). Self-playing-based opening book tuning. *New Mathematics and Natural Computation (NMNC)*, 2(02):183–194.

[Ong et al., 2007] Ong, C., Quek, H., Tan, K., and Tay, A. (2007). Discovering chinese chess strategies through coevolutionary approaches. In *IEEE Symposium on Computational Intelligence and Games*, pages 360–367.

[P Boyle, 1997] P Boyle, M Broadie, P. G. (1997). Monte carlo methods for security pricing. In *Journal of Economic Dynamics and Control*.

[Pearl, 1984] Pearl, J. (1984). *Heuristics. Intelligent search strategies for computer problem solving*. Addison-Wesley.

[Plaat et al., 1996] Plaat, A., Schaeffer, J., Pils, W., and de Bruin, A. (1996). Best-first fixed depth minimax algorithms. *Artificial Intelligence*, 87:255–293.

[Püschel et al., 2005] Püschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., et al. (2005). SPIRAL: Code generation for DSP transforms. *Proc. of the IEEE*, 93(2):232– 275.

[Quante et al., 2009] Quante, R., Fleischmann, M., and Meyr, H. (2009). A stochastic dynamic programming approach to revenue management in a make-to-stock production system. Research Paper ERS-2009-015-LIS RevisionDate: 2009-07-29, Erasmus Research Institute of Management (ERIM), ERIM is the joint research institute of the Rotterdam School of Management, Erasmus University and the Erasmus School of Economics (ESE) at Erasmus University Rotterdam.

[Ralaivola et al., 2005] Ralaivola, L., Wu, L., and Baldi, P. (2005). Svm and pattern-enriched common fate graphs for the game of go. In *Proceedings of European Symposium on Artificial Neural Networks 2005*, pages 485–490.

[Rolet et al., 2009] Rolet, P., Sebag, M., and Teytaud, O. (2009). Optimal active learning through billiards and upper confidence trees in continous domains. In *Proceedings of the European Conference on Machine Learning*.

[Sharma et al., 2008] Sharma, S., Kobti, Z., and Goodwin, S. (2008). Knowledge generation for improving simulations in uct for general game playing. pages 49–55.

[Siu et al., 2001] Siu, T. K., Nash, G. A., and Shawwash, Z. K. (2001). A practical hydro, dynamic unit commitment and loading model. *Power Systems, IEEE Transactions on*, 16(2):301–306.

[Stockman, 1979] Stockman, G. (1979). A minimax algorithm better than Alpha-Beta ? *Artificial Intelligence*, 12:179–196.

[Streeter and Smith, 2006] Streeter, M. J. and Smith, S. F. (2006). A simple distribution-free approach to the max k-armed bandit problem. In *Principles and Practice of Constraint Programming (CP)*, pages 560–574.

[Sturtevant, 2008] Sturtevant, N. R. (2008). An analysis of uct in multi-player games. In *Computers and Games*, pages 37–49.

[Teytaud and Teytaud, 2009] Teytaud, F. and Teytaud, O. (2009). Creating an Upper-Confidence-Tree program for Havannah. In *Advances in Computer Games 12*, Pamplona Espagne.

[Teytaud and Fournier, 2008] Teytaud, O. and Fournier, H. (2008). Lower bounds for evolution strategies using vc-dimensio n. In *Parallel Problem Solving From Nature*, pages 102–111.

[Tromp and Farnebäck, 2006] Tromp, J. and Farnebäck, G. (2006). Combinatorics of go. In *Proceedings of 5th International Conference on Computer and Games*, Torino, Italy.

[Voronenko, 2008] Voronenko, Y. (2008). *Library Generation for Linear Transforms*. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University.

[Voronenko et al., 2009] Voronenko, Y., de Mesmay, F., and Püschel, M. (2009). Computer generation of general size linear transform libraries. In *Int. Symp. on Code Generation and Optimization (CGO)*.

[Wang et al., 2008] Wang, Y., Audibert, J.-Y., and Munos, R. (2008). Algorithms for infinitely many-armed bandits. In *Advances in Neural Information Processing Systems*, volume 21.

[Wang and Gelly, 2007] Wang, Y. and Gelly, S. (2007). Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, pages 175–182.

[Winands et al., 2008] Winands, M. H. M., Björnsson, Y., and Saito, J.-T. (2008). Monte-carlo tree search solver. In *Computers and Games*, pages 25–36.

[Zobrist, 1990] Zobrist, A. (1990). A new hashing method with application for game playing. *International Computer Chess Association Journal*, 13(2):69–73.