Monte-Carlo Tree Search and Computer Go

Keh-Hsun Chen, Dawei Du, and Peigang Zhang

Abstract. The difficulty in positional evaluation and the large branching factor have made Go the most challenging board game for AI research. The classic full-board game-tree search paradigm has been powerless on Go even though this classic paradigm has produced programs with expert level performances in many other games. Three decades of research on knowledge and search did not push computer Go above intermediate amateur level. The emerging Monte-Carlo Tree Search (MCTS) paradigm is bringing an exciting breakthrough in computer Go toward challenging human experts, especially on smaller Go boards. This chapter gives an overview of both classical and MCTS approaches to computer Go.

Keywords: Computer Go, Monte-Carlo Tree Search, UCT Algorithm, Simulations, Pattern Mining, Go Tactic Problems, Genetic Algorithm.

1 Introduction

The difficulty in positional evaluation and the large branching factor have made Go the most challenging board game for AI research. The classic full-board game-tree search paradigm has been powerless on Go even though it has produced programs with expert level performances in many other games such as chess, checkers, Othello, Shoji, & Chinese Chess and it has solved some nontrivial games such as checkers.

Go has the highest state-space complexity & game-tree complexity among all popular board games. For decades, computer Go researchers and programmers couldn't advance the playing strength of computer Go programs to above intermediate human amateur level. Recent development in Monte-Carlo Tree Search (MCTS) and UCT algorithm changed outlook of computer Go completely. Now computer Go programs based on these new techniques can challenge advanced

Keh-Hsun Chen, Dawei Du, and Peigang Zhang
Department of Computer Science, University of North Carolina at Charlotte
Charlotte, NC 28223, USA
e-mail: chen@uncc.edu, ddu@uncc.edu, pzhang1@uncc.edu

Z.W. Ras and W. Ribarsky (Eds.): Advances in Information & Intelligent Sys., SCI 251, pp. 201–225.
springerlink.com

dan-level amateur players. And there is hope that one day Go programs may be able to rival or surpass professional Go experts. At least on small Go boards, such as 9x9, machine is starting to hold its own against professional players. This is beyond wildest dream of the computer Go community just a few years ago.

We shall introduce the game Go and its role in machine intelligence in Section 2, and then discuss the history of computer Go and techniques used before the recent breakthrough with MCTS in Section 3. Section 4 gives an overview of MCTS and the UCT Algorithm. We introduce Monte-Carlo simulations in Go in Section 5. We discuss the essential Go domain knowledge needed for simulations in Section 6. We present pattern mining to establish default urgencies for moves in simulations in Section 7. We discuss some key enhancements to MCTS in Section 8. We describe solving Go tactic problems using MCTS in Section 9. A novel approach to tune the parameters in MCTS is presented in Section 10. Concluding remarks are given in Section 11.

2 Go and Machine Intelligence

Go is a board game invented four thousand years ago in China. This game is very popular in Japan, China, and Korea. There are estimated 30 million players worldwide. It is gaining popularity in western world in recent decades.

The rules of the game Go are deceptively simple. It is played on a 19X19 grid using black and white stones. There are two players, one uses black stones and the other uses white stones. They alternately place their stones one at a time onto some empty board intersection points. Unlike a chess piece, a stone never moves, but it may disappear from the board. This disappearance, called captured, occurs when a block of stones looses all its liberties - that is, when it is completely surrounded by the opponent's stones. With the exception of Ko and stone suicide, which cause a previous full board configuration to reoccur and are forbidden, every empty grid point is a legal position for the next move. The objective of the game is to secure more grid points, called territory, than the opponent. Go, just like chess, is a two-person perfect information game. For more detailed descriptions of the game and the rules of Go, readers can visit American Go Association web site. Its home page address is http://www.usgo.org/.

A Go game normally runs over 200 moves. Each turn offers about 250 choices of legal plays on average. Because of this high branching factor and difficult positional understanding, Go is considered a most challenging game for the machine. It provides an excellent model for machine intelligence research. Theoretically, *N*-by-*N* Go has been proved to be P-space hard and exponential-time complete.

Games of strategy such as chess, Chinese chess and Go are generally good models for machine intelligence research, since

1) The rules of the games are clear enough to provide an unambiguous background,

2) The possible developments in the games are complex enough to warrant the investigation of AI techniques, and

3) The progress/accomplishment of computer systems for playing the games can be measured objectively by observing their performance against ranked human players.

Go is an excellent model for machine intelligence research, because

4) The size of the Go board and the nature of the Go game prohibit the use of classic game-tree search paradigm of chess and other games:

a. Go game tree has extremely high branching factor - about 250 on average considering all legal moves.

b. The static evaluation is very hard to achieve reasonable degree of accuracy for Go.

Hans Berliner has suggested that Go is the task par excellence for AI.

3 Computer Go before MCTS

The research of computer Go started around 1970 with the PH.D. dissertations of Zobrist at University at Wisconsin and Ryder at Stanford University. Zobrist's program relied on pattern recognition, Ryder's program depended on static analysis. Both of their programs played at total novice level. During 1972-79, Reitman and Wilcox, supported by NSF, build a Go program based on human player models- trying to capture human player's perception of the Go board as the basis of move selection. The playing strength was about 17th kyu. The Japanese 5th Generation Project included a computer Go project, which used massive parallelism and involved many computer scientists and Go experts. But the result was disappointing, it was weaker than existing top Go programs at that time. In 1992, GoIntellect, developed by the first author of this chapter, defeated the PC version of Go Generation at the final match of the Ing Cup to win its first world championship.

Go programmers have found that understanding Go positions is extremely hard for a machine [7]. Static evaluation of Go-board configurations is essentially impossible: it is hard to achieve any reasonably high degree of accuracy on regular basis (except for near endgames and very calm positions). Moreover, the dynamic evaluation and the positional understanding problem create even more difficult challenges for computer Go [9].

Go is a territorial game. A Go program looks at black and white stones scattered on a 19-by-19 grid. It needs to figure out what each side's territories and potential territories are, so that it can make intelligent move decisions. There is a huge gap between the two ends: the board configuration and the move decision. It is logical to use a hierarchical model creating intermediate steps and knowledge structures to bridge the gap. A typical hierarchical model consists of something similar to the following five layers: Stones – Blocks – Chains – Groups – Territories.

At the Stones level, a program needs to know the current stone distribution on the board and the history of the move sequence leading to that position. Data structures of arrays, stacks, and trees are commonly used for this level.

A block, also called a string, is a set of adjacent stones of one color. According to the rules of Go, stones of a block are captured in unison when the block loses

all its liberties (empty adjacent points). If we view a board configuration as a graph, with stones as nodes and any two vertically-adjacent or horizontally-adjacent stones of the same color determining an edge, then a block is a component of this graph. A depth-first search-based graph-component algorithm can be used to identify blocks efficiently.

A chain is a collection of inseparable blocks of the same color. Heuristics, pattern matching, and search are used to recognize the connectivity of two blocks.

A group is a strategic unit of an army of stones. It consists of one or more chains of the same color plus the dead opponent blocks, called the prisoners. The chains and dead blocks are connected through empty points[1] that have an influence above a certain threshold. For details, we refer to [7, 8, 9].

Territory can be estimated by measuring interior spaces (spaces surrounded by grid points belonging to the same group) and prisoners (dead opponent stones), with an adjustment for the safety of the group.

Move decision strategies can be roughly classified into four paradigms: static analysis, try and evaluate, global selective search, and incentive/temperature approximation [9].

3.1 Static Analysis

Programs of the static-analysis paradigm do not perform any global search, but may perform various goal-oriented local searches, such as capturing, connection, life and death. Since these local searches do not need to be performed for each node in a global search tree, they tend to be done more thoroughly. Each program has its own set of move generators to suggest candidate choices for global-move selection. The programs DRAGON, EXPLORER, and FUNGO are in this category. DRAGON uses a priority scheme. It divides all possible moves into 13 priority categories: capturing/escaping, urgent pattern, joseki, …, small yose. It selects the move in the highest non-empty priority category with the biggest move value provided by the related move generator(s). EXPLORER adds the values from all move generators for each point and then selects the point with a maximum sum to play. FUNGO uses the maximum value over all move generators for each point. It then selects 18 points with the highest values in order to perform more sophisticated and time-consuming tasks for the final move selection.

3.2 Try and Evaluate

In the try-and-evaluate paradigm, first candidate moves are generated and then a thorough evaluation is performed with each candidate move tested in turn on the board. The one with highest evaluation will be chosen. GNUGO, GO4++, MANY FACES use this strategy. GNUGO's move generators do not assign valuations but rather move reasons. The actual valuation of the moves is done by a single module. GNUGO is developed by many people jointly. Its source code is in the public domain: http://www.gnu.org/software/gnugo/devel.html.

[1] They are called the spaces of the group.

GO4++ uses pattern matching to generate a large number of candidate moves, about 40 or so, together with a ranking. A thorough evaluation based on a connectivity probability map is done on each move candidate. MANY FACES performs a quiescence search for the evaluation of each candidate move. The search result is modified by the estimate of the opponent's gain if the opponent is playing locally first. Move generators in MANY FACES generate (reason, value) pairs for each candidate move. The maximum value is taken in a reason category for each point and values are added over different categories for a move point.

3.3 Global Selective Search

Many programs perform some variation of alpha-beta look-ahead with a heuristically-selected small set of move candidates at each non-terminal node. The mini-max back up determines the move and scoring estimate. Programs using this strategy include GO INTELLECT, SMARTGO, INDIGO, GO MASTER, JIMMY. GO INTELLECT uses quiescence search modified by urgency. SMARTGO uses iterative deepening and widening. INDIGO performs two separate global searches, one with urgent moves and the other with calm moves. GOMASTER converts everything to points in the evaluation including influence, thickness, sente, gote, … etc. JIMMY performs global selective search using B*-type upper and lower bounds associated with each move candidate.

3.4 Incentive/Temperature Approximation

Programs using the incentive/temperature approximation paradigm consider the consequences of each side playing first in a local situation. They include HANDTALK, GOEMATE, WULU, HARUKA, KCC IGO, GOLIATH, GOSTAR, GOLOIS, and STONE. The programs HANDTALK, GOEMATE, and WULU use the sum of the move values of both sides modified by "move efficiency" to decide on the move to play. Local searches are performed but no global look-ahead. HARUKA performs 1 to 4 general local searches, called main searches, each with about an 10*10 scope, a search depth of 3 to 5 plies, and a width of 6 to 9 moves. KCC IGO first identifies critical areas, then performs local searches with candidate moves mostly taken from pattern matching. GOLIATH performs two local searches for each critical area, one for each side playing first. The biggest difference on the results of the two searches determines the move selection. Candidate moves are from pattern libraries with patterns represented by bit strings (Boon, 1989). STONE contains a set of tactic move generators and a set of position move generators. Each move generator has a temperature predictor and a move searcher. A move generator may find itself applicable to one or more (or none) sub-games (sub-regions). For each applicable sub-game, the temperature predictor heuristically produces a maximum temperature (upper bound) and a minimum temperature (lower bound). For the sub-game with highest maximum temperature, the associated move searcher will be invoked. The move searcher will find the best move for the sub-game and the temperature of the sub-game, which will replace the

sub-game's old maximum and minimum temperatures. This process is repeated until a lower bound is greater than or equal to the rest of the upper bounds and the best move of that hottest sub-game will be chosen.

The above techniques have advanced computer Go from total novice level to intermediate amateur level over a 30 years time frame but it was very difficult to make further advancement until MCTS paradigm emerged about 3 years ago [24, 15, 25, 18].

4 UCT Algorithm and MCTS

Using Monte-Carlo techniques in a Go program has a 15 year history [3, 2]. The recent invention of UCT (Upper Confidence Bounds applied to Trees) algorithm [24] enabled best-first tree search based on the results of dynamic simulations, which brought a breakthrough in computer Go. At Computer Olympiad 2007, all MCTS programs finished ahead of all traditional knowledge and search based programs in the 9x9 Go tournament and MCTS programs also had top two finishes in the 19x19 Go tournament. In Computer Olympiad 2008, almost all participating programs were MCTS-based. We could hardly see any traditional Go programs at recent computer Go tournaments.

At the heart of this new generation of Monte-Carlo Go programs is the UCT algorithm, which guides the program toward winning moves based on statistical results of simulations (pseudo random games) played at selected nodes of a dynamically growing MC Search Tree.

UCT algorithm efficiently balances exploitation of the most promising moves and exploration of the most uncertain moves. It is an excellent tool for MCTS. We outline the UCT algorithm as implemented in Go Intellect below (see Algorithm 1) [13].

The root of the MC tree represents the current Go board configuration. We assume that the MC tree is already in the memory. If not, a MC tree with just the root node is created.

```
While (more simulations to go) {
      current_node = root of MC tree;
      if (All legal moves from the position have been generated or maximum
   number of children allowed is reached)
            Advance to the child_i maximizing
            r_i + sqrt( log(p)/(5*n_i));
            where r_i is the winning rate of move_i, p is the number of
         simulations passing through the current_node, n_i is # of simulations
         passing through child_i.
      else {
            Add a new child to MC tree for the move with the next highest
         urgency value;
            Play a random continuation (simulation game) to game-end with
         move selection probabilities proportional to move urgency values;
            Score the new node with the win/loss result of the simulation;
```

```
            Update the number of games passed through by adding 1 to all
        ancestors;
            Update the number of wins by adding 1 for every other ancestor
        sharing the success;
    }
}
```

Algorithm 1. The UCT algorithm as implemented in 9×9 GOINTELLECT 2007.

The main while loop can be controlled by a limit on the number of simulations, a limit on the time allocation, and/or some other criteria. After the simulations, the child of the root with the highest winning rate and exceeding a threshold of minimum number of simulation games is selected for move decision. A more detailed UCT algorithm description for the game Go can be found in [25, 18, 5].

5 Simulations

Both the quality and the quantity of the simulations profoundly affect the playing strength of a UCT-MC Go program. The quality of the simulations helps to produce more accurate evaluations and the quantity of simulations allows MC tree to grow/see deeper. Here the quality means ability to reflect the merit of the position, which does not mean the simulator itself is a better stand along player. Only light domain knowledge computable in a comparable time to random-move generation and execution may be beneficial in the simulations [13].

Each move is associated with a non-negative integer urgency value, which is used in weighted random-move generation with the urgency as the weight. We use the knowledge to alter the urgencies of candidate moves, which normally have default urgency 10 (this allows us to decrease urgency of a particular move to a lower integer if so desired). A move with higher urgency has higher probability to

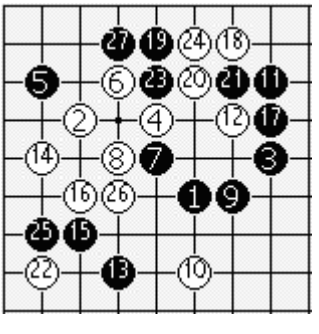

Fig. 1 A semi-random (partial) game.

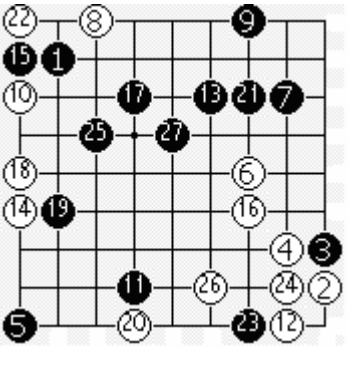

Fig. 2 A pure-random (partial) game.

be selected by the weighted random-move generator for the simulation. We call such knowledge-guided random games semi-random games. Figures 1 and 2 compare a typical semi-random game with a pure-random game. Experienced Go players can see that semi-random games are much more reasonable and will carry better feedback than pure-random games in MC simulations.

We keep the MC tree in memory at all times until the game is over. When either side makes an actual move, the corresponding sub-tree from the previous search is used as the starting point for the next search. A significant amount of simulations can be reused this way.

6 Essential Knowledge in Simulations

Experiments show that MCTS with knowledge guided simulations performed much better than MCTS with pure random simulations but too much knowledge used in simulation makes the MCTS weaker than just using small amount of basic key knowledge. Simulation player which is a stronger stand alone Go player is not necessarily producing a stronger MCTS based Go program. In this subsection, we shall present knowledge items in Go which are essential and beneficial to simulations in MCTS found from extensive experiments and testing as reported in [13]. We adopt the following knowledge items by adjusting urgency values of the board points involved in the simulations.

Avoidance of Filling Solid Eyes - A simulation routine in MC Go must know to avoid filling a solid eye otherwise a simulation could go on endlessly. We set the urgency of a solid eye point to 0.

Neighborhood Replacement - While the program checks for solid eyes and finds an empty point is not a solid eye but it has a heavy concentration of neighbors and

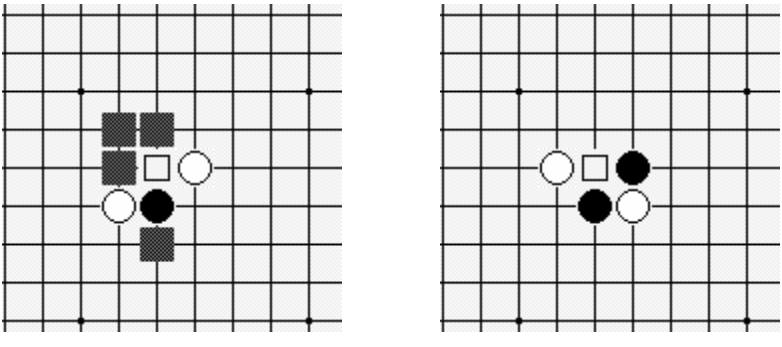

Fig. 3 The left is a blocking pattern and the right is a cutting pattern. The square marks the pattern move (currently empty). White represents ToPlay color and black represents the Opponent color. Dark shade square represents empty or Opponent.

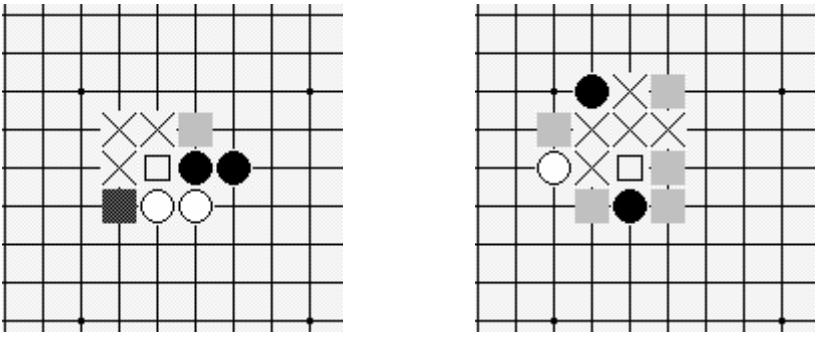

Fig. 4 The left is an extending pattern and the right is a separating pattern. The X is required to be empty. The light shade represents empty or ToPlay.

diagonals of stones of same color; in this case the simulation uses the point's empty neighbor/diagonal as a substitution for the point itself in random move generation, which can make eye producing/breaking moves.

Capture/Escape – Moves to rescue stones under Atari and move to capture opponent stones should get high urgencies in the simulations. We can generalize this principle to liberties of blocks with few liberties. Pseudo ladder scan, [13], can be performed to avoid producing an unsuccessful ladder chase in a simulation.

Patterns - We match patterns only on the surroundings of the last random move to save time. The emphases are on blocking such as in Figure 3 left, cutting such as in Figure 3 right, extending such as in Figure 4 left, and separating such as in Figure 4 right.

7 Pattern Mining and Default Urgencies

We perform offline pattern mining from forty thousand professional Go game records to create libraries of weights/urgencies of all local patterns in Go under a given restricted template [14]. Urgency estimates of all 3x3 patterns were uncovered in our study, and similar approach can be used for other pattern templates. Since these weights are to be used in Monte-Carlo simulations to improve the performance of MC Go programs, the pattern matching speed needs to be extremely fast. We use a direct indexing approach to access the weight information. The surrounding index of each board point is updated incrementally as moves are executed. We first computed the adoption ratios of surrounding indices via stepping through all forty thousands professional 19x19 Go game records. These pattern adoption ratios, taking rotations and symmetries into consideration, serve as the initial pattern urgency estimates. The pattern urgencies are computed through

additional iterations of non-uniform adoption rewards based on the urgency estimation values of the previous iteration until the sequence of values converge [14]. We describe the pattern mining process in the following subsections.

7.1 Surrounding Index

We shall use two bits to code the contents of a board point: empty (00), black (01), white (10), or border (11). The immediate surrounding pattern of a board point is a sequence of 8 2-bit codes from the north neighbor, the northeast diagonal, the east neighbor, …, to the northwest diagonal which can be coded as a 16-bit binary string. After initialization, the surrounding indices of all board points can be updated incrementally as moves are executed. When a stone is added, it only affects the surrounding indices of its 8 immediate neighbors. For each of the 8 neighbors, the updating involves changing certain 2 0-bits (empty) to the code for the new stone color. Similarly, when a stone is removed, certain 2 bits coding the old color will be reset to 00. We are only interested in the surrounding indices of empty board points, but we have to keep track the surrounding indices of all board points since stones could be captured and their points become available empty points thereafter. The above updating can be implemented in the program's execute move and undo move routines efficiently with minimal tax, 8% in GoIntellect case, on processing time. The above indexing framework can be extended to any local pattern template. For example, we have also created 24-bit surrounding indices for Manhattan distance 2 or shorter neighbors. The time cost increases only linearly to the template size in the incremental updating. The resource bottleneck is the exponential memory space requirement, since we need to keep track arrays of size 2^{2s} where s is the number of neighboring points in the template. In the next subsection, we discuss calculating adoption ratios for surrounding indices.

7.2 Adoption Ratios

The first step of our pattern mining is to obtain the adoption ratios for each possible surrounding index (local pattern) from professional Go game records. We use two arrays NuOccurrences[ ] and NuAdoptions[ ], both indexed by surrounding index and initialized to 0. We put all game records to be processed in one directory, then load and process them one at a time. For each game record, we step through the moves played one move at a time with surrounding indices automatically updated. We compute the initial adoption ratios from black's point of view. So if black is to play, we increment NuOccurrencse[i] for all surrounding indices i of legal points of the current board configuration by 1 and increment NuAdoptions[j] by one where j is the surrounding index of the move chosen in the record. If white is to play, all the indices go through a procedure flipBW, which produces the corresponding surrounding index with black and white flipped. We do not count into the adoption ratio statistics if the chosen move is a capturing move, atari move, or an extension or connection move after a block being ataried. Since those moves are played not because the urgency of the local pattern. Our MC simulation procedure treats capture/escape separately assigning very high urgency values.

Each pattern has 8 equivalent patterns under rotations and symmetries (not counting the color flipping). We use a loop to add counts from equivalent patterns together to be shared by all patterns (surrounding indices) in the same equivalent class. The adoption ratio of a pattern is calculated as the number of adoptions divided by the number of occurrences of the corresponding surrounding index.

7.3 Pattern Weights

A pattern (surrounding index) with high adoption ratio is not necessarily an urgent pattern. It could occur when the competing patterns on the board were all weak, it registered many adoptions but may not be really urgent. On the other hand, when the board has several urgent patterns occurring, we want to award the adopted pattern higher credit for "beating" those tough competitions. So we do a second pass, this time for each pattern selected to play according to the game record is credited with the adoption rations of those other patterns occurred on the board configuration. At the end of the pass, share weights within an equivalent class of patterns, then we normalize the new weights/urgencies by multiplying 8*total regular moves played in all the games divided by total weights. Finally each weight is divided by its own occurrence count. The resulting weights are more meaningful than the original ones. We repeat this process many times until the weights converge, i.e. they don't change much from one iteration to the next. We view a weight distribution over all surrounding indices as a point in 2^{2s} dimensional space. When the distance d = $\sqrt{\sum_i \left(w_i - w_i^{'}\right)^2}$ for two consecutive passes less than a threshold (0.01), the weights are considered stabilized. We apply this method to a collection of forty thousand professional games. It took 9 iterations to converge. Each iteration took about 20 minutes.

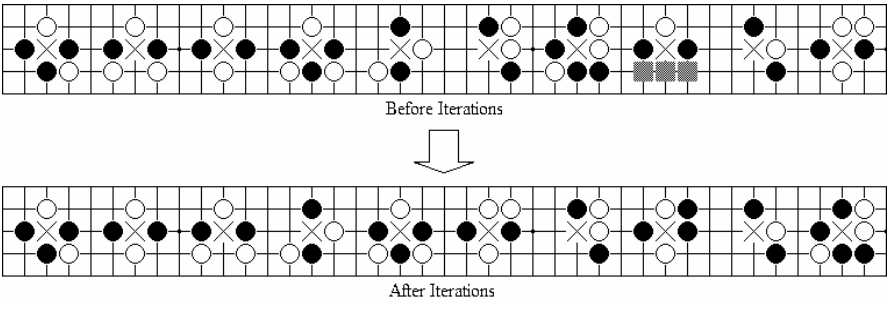

Fig. 5 The 10 patterns with highest weights/urgencies based on the 8 immediate surrounding alone. We list a representative from an equivalent class. All patterns are 3x3 with Black to play in the center X.

7.4 Top Patterns

Our pattern mining discovers the weights of all 64K surrounding indices (all possible patterns of the 3x3 template). Fig. 5 shows the top 10 patterns of highest weights. The first row patterns are ordered by adoption ratios in the professional games collection. The second row patterns are ordered by final converged weights. Only one representative is listed from an equivalent pattern group.

The learned weights provide excellent default urgency values of empty board points. This approach significantly improved the quality of random simulations and the playing strength of our MC UCT-algorithm based Go programs. Testing result showed that it increased the winning rates of GoIntellect against GnuGo on 9x9 games by about 10% [14].

8 Enhancements to MCTS

The pattern mining described in the last section is an off-line learning mechanism, which provides fast pattern knowledge. The UCT Algorithm is a dynamic on-line learning procedure on the merits of game configurations represented by the nodes of Monte-Carlo Search Tree (MCST). MCTS can be further enhanced by the following techniques.

All Moves as First Heuristic
Assume we have a simulation move sequence m_1, m_2, …, m_k,…, m_n with result r (win or loss). Then m_k, m_2,…, m_{k-1}, m_1, m_{k+1},…, m_n is also a simulation sequence with same result r, providing m_1 and m_k are moves of the same color. This is called "All Moves as First Heuristic" (AMAF). So every time we perform a simulation at a leaf node of MCST, we can update AMAF statistics for every other ancestor node, on its child node with move m1, along the path from the leaf to the root. Of course, the swapped move sequence may not be a sensible sequence or even legal at all. AMAF statistics accumulate very fast - for each simulation, we get many AMAF-simulations. We can use AMAF statistics to help guiding the MCTS selection process when the number of actual simulations is low. It is also called RAVE (Rapid Action Value Estimates) [17, 6]

Progressive widening
If we are to generate all legal successors for a node in MCTS, then the search depth will suffer yielding inferior playing strength. So for each node, we allow a small number of initial successors to be generated based on move urgencies, then gradually add additional successors one at a time when the number of simulations exceeds various thresholds, which typically is an exponential function of the number of simulations passed through the node [15, 5].

Parallelism
Since simulations can be performed independently, MCTS can take advantage of parallel processing on multi-core machines and/or clusters. We will not get into the details here. Interested readers can refer to [20].

9 Solving Tactic Problems with MCTS

Monte Carlo evaluation and move decision with UCT [24] (MC-UCT) algorithm has shown its ability and efficiency in searching Go game trees (global MC-UCT algorithm). It has produced much stronger Go playing programs compared to traditional knowledge and search based programs [18].

When facing a standard 19x19 Go board, human players usually identify and solve a number of local tactic problems [9] such as capturing [11, 27], connection [12], life and death [8] as key information in making the final move decision. Solving Go tactic problems is an important step towards building a strong 19x19 Go playing program.

In this section, MC-UCT algorithm is used to solve Go Tactic (capturing as an example) problems.

The structure and extensions of MC-UCT algorithm for solving Go Tactic (capturing) problems are discussed. Experimental results show that MC-UCT algorithm is very effective in producing a strong Go capturing problem solver. It outperforms traditional techniques such as αβ search [22] and proof number search [1] based Go capturing routines.

9.1 Go Tactic Problems

Go tactic problems such as capturing, connection, and life and death problems are a set of important problems in Go.

When facing a Go board status, human players usually will perform a lot of tactical calculations as well as global considerations. It is believed that global search and local tactical search should be combined to achieve a strong computer Go program, especially at 19 x 19 boards.

The goal of capturing is to find out if a block of stones can be captured or escaped and to find the best move sequence to accomplish this goal. Capturing calculation is basic and essential to computer Go programs. Capturing algorithm can be used in post-game capturing problem solving or embedded in actual game engines to do real time capturing calculations.

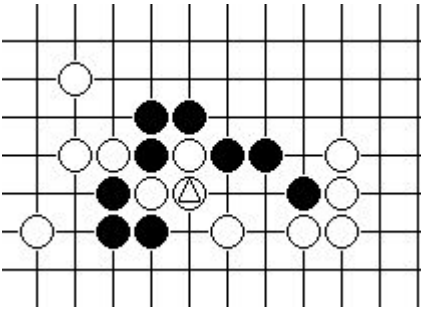

Fig. 6 Go Tactic Problem (Capturing Problem)

Using traditional game tree search techniques, some progress have been achieved to provide strong tactical problem solvers [27]. But traditional game tree search techniques failed to some hard problems (such as some hard capturing problems, unbounded life and death problems).

9.2 Difference with Standard MC-UCT Algorithm

Tactic MC-UCT algorithm is based on standard (global) MC-UCT algorithms. Compared with global MC-UCT algorithm, the basic differences of capturing specific MC-UCT algorithm include local search region, terminal node tag and Go capturing specific MC random simulation. These differences are discussed in the following three subsections.

9.2.1 Local Search Region

Global MC-UCT algorithm usually treats all legal moves as candidate moves except those that fill their own solid eyes. For capturing problems, it is not efficient to consider all legal moves. Candidate moves come from legal moves inside the local search region except those moves that fill their own solid eyes. The search region may dynamically grow as explained below (see figure 7).

- Initial candidate move region
- Target block
  First liberties, second liberties and third liberties
- Adjacent opponent of target block
  First liberties and second liberties
- Floating Search Region
  After a move (for both UCT tree and Monte Carlo simulation) this move's adjacent points are added to the candidate move list

9.2.2 Terminal Node Tag

Global MC-UCT algorithm usually treats a UCT tree node as a terminal node if no more legal moves exist except those that fill their own eyes. For Go capturing

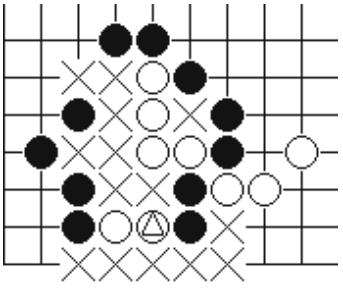

Fig. 7 Local search region (the triangle marked block is the target to capture, X marked positions are candidate moves inside local search region)

specific UCT tree, it is important to put a terminal node tag for those nodes with an already known definite value and store the value in the node for later use. There is no need to do a MC evaluation for those nodes. To decide whether a node is a terminal node or not, some heuristics such as ladder status and the number of liberties are used. In our implementation, at the time the node is created, some heuristics are used to determine whether this node is a terminal node or not. If it is a terminal node, the value will be stored and a terminal node tag will be set. There will be no more expansion from this node. When the UCT tree search visits this node at future time, the value will be returned directly.

9.2.3 Go Capturing Specific MC Simulation

For MC capturing simulations, two kinds of stop criteria are used.

- If the target block is removed from board, return captured.
- If the number of liberties of the target block exceeds a given threshold or the number of moves from start board to current board exceeds a given threshold, return escaped.

About the urgency of a move, the move urgency of global MC-UCT algorithm is inherited [15] and the capturing specific move urgency which mainly considers the liberty status around the target block is used also.

- Urgencies of the target block's liberties: Urgency_FirstLib = 400, Urgency_SecondLib = 100, Urgency_ThirdLib = 25
- Urgencies of the liberties of the adjacent opponents of the target block: Urgency_OppFirstLib = 50, Urgency_OppSecondLib = 12
- Other urgencies: Urgency_OppOppFirstLib = 25, Urgency_NewFirstLib = 11, Urgency_ProtectedLib = 11
  These urgencies come from heuristics and can be optimized in the future.

9.3 Some Extensions to Standard MC-UCT Algorithm

The extensions of standard MC-UCT algorithm are discussed in this section. These extensions can be used to enhance a global MC-UCT algorithm also. Some test results of these extensions can be found at the next section.

9.3.1 UCT Node Weight and Adjustment by Heuristics

The UCT node weight is determined by domain specific heuristics and is used to adjust the value of the node.

$$\text{node.outputValue} = \text{node.value} \times \text{node.weight}$$

The outputValue is used in SelectByUCB function to replace the value property.

If the move can be ladder captured, the weight is set to 0.99; if the move can ladder capture the last opponent move, the weight is set to 1.01; in any other situations the weight is set to 1.0.

The idea is giving a little bonus to those nodes with good heuristic values so that they get better chance to be explored and giving a little penalty to those nodes with bad heuristic values so that they get less chance to be selected. Big bonus and big penalty such as 1.05 and 0.95 were tested, but the results were not promising.

Sometimes standard global UCT search will choose some useless moves that only cause losing of ko-threats, this extension can ease the problem to some degree.

9.3.2 Greedy Mode in UCT Tree Search

From a given intermediate node, standard global UCT tree search usually explores every child node at least once then focuses on some branches to explore. A greedy mode UCT tree node exploration extension is introduced. It puts more emphasis on exploitation. Experimental results show that it makes the Go capturing oriented UCT tree search more efficient. This extension contains several parts:

1. At the time creating all the child nodes of a parent node, all the child nodes will be ordered by heuristic values (mainly urgencies) and they will be explored in this order.
2. From a given parent node, if it can find a child node with win rate exceeding a given threshold, it will explore from this node even if some of this child nodes' sibling nodes have never been explored.

9.3.3 Use Pseudo Ladder at MC Simulation

At standard MC simulations, higher urgency is usually put on the single liberty of one-liberty blocks [15]. Using pseudo ladder extension, more urgency is put on those liberties of the two-liberty blocks if those liberties are responsible for the ladder capture of those blocks. The pseudo ladder capture is used to calculate ladder status. It can deal with over 99% ladder problems and use very little time compared with a regular ladder solvers. This extension costs 30%-40% extra time to perform a MC simulation. Experimental results show that it significantly improves the performance of the Go capturing MC-UCT algorithm; the extra time is well spent.

9.4 Result Analysis

The problems in Kano book [21] series 3 and 4 is used as the test problem sets. The tests were run on a 3.0 GHz P4. GoGui Gtpregress was used to do the test.

We experimented with up-to 500k simulations. On each problem an early stop is allowed (at least 3k simulations) if the win rate of the root node exceeds a given threshold (depend on the number of simulations). When the search stops, the best child move of the root node and the win rate of root node will be returned as the result. If the best child move is the correct answer and the win rate of root node exceeds the given threshold (a given confidence), the problem is considered to be correctly solved.

Essentially, MC-UCT algorithm is a stochastic algorithm. Compared with tradtional game tree search algorithms which can give sure answer (capture, escape or

Table 1 Results of different algorithms to solve capturing problems in Kano book 3

	Solved problems	Unsolved problems	Average time(ms)	Average nodes explored	Average search depth
αβ no transposition table	52	9	5502	28746	5.6
Aβ with transposition table	53	8	2785	13187	5.8
Proof number search(pn)	55	6	2148	18607	7.2
pn+ (with heuristic)	55	6	1303	10931	7.0
df-pn	55	6	2259	5792	7.4
df-pn+	55	6	1648	2175	6.7
MC-UCT	60	1	5410	15000	N/A

The average time in table 1 is the the average time cost for solved problems.

Table 2 Compare test results of MC-UCT algorithm with and without the extensions

	Unsolved book 3	Total time book 3 (s)	Unsolved book 4	Total time book 4 (s)	Total unsolved	Total time (s)
No node weight	2	308	8	490	10	798
No greedy mode	1	309	6	550	7	859
No pseudo ladder	8	250	15	472	23	722
With all extension	1	330	6	419	7	749

Table 3 Back up heavily while reaching terminal nodes

K	Unsolved book 3	Total time book3 (s)	Unsolved book 4	Total time book 4 (s)	Total unsolved	Total time (s)
1	2	251	7	438	9	689
2	0	496	8	1001	8	1497
4	1	436	7	367	8	803
8	0	235	6	464	6	699
16	2	546	8	324	10	870
32	2	281	6	368	8	649
64	1	528	10	396	11	924
128	4	235	12	265	16	500
256	3	308	9	267	12	575
512	7	339	13	352	20	691
1024	5	322	17	214	22	536

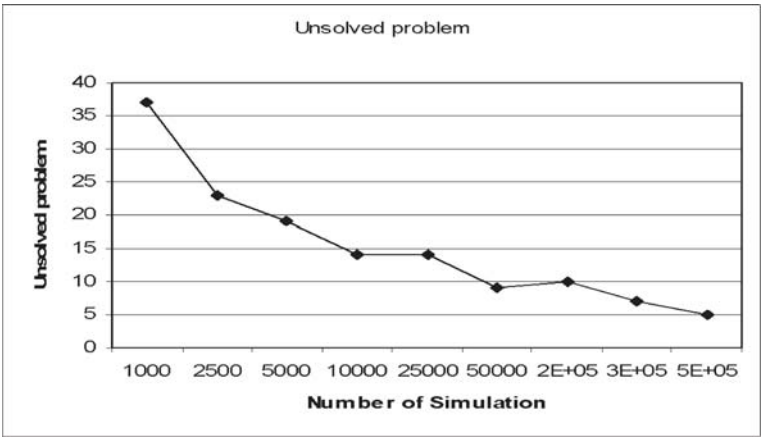

Fig. 8 Total number of unsolved problems with simulation increases

unknown), MC-UCT algorithms's results are based on confidence (the given threshold that is mentioned at last paragraph) which is defined by heuristic.

Kano book 3 contains 61 capturing problems: the average number of un-solved problems is 1 and the average total time used to do all the 61 problems is 330 seconds - about 5 seconds per problem (table 1).

Kano book 4 contains 51 capturing problems: the average number of un-solved problems is 6 and the average total time used to do all the 51 problems is 419 seconds - about 8 seconds per problem (table 2).

One note about the experimental results: the MC-UCT algorithm is a stochastic algorithm, thus each test was performed 3 times, the mean value was listed as the result; the results are quite stable, for example, the standard deviation of the number of unsolved problems is no more than 1.

The result shows that the MC-UCT algorithm outperforms other game tree search algorithms [4]. It also confirms that the more simulations the algorithm performs, the better the result (figure 8).

9.4.1 Comparing with Other Game Tree Search Method

For the 61 problems in Kano book 3, Using αβ and proof number search and each problem given up-to 200 seconds, 6 problems are still un-solved [27] (table 1). These algorithms are local search also and use much complex heuristics to generate candidate moves and evaluate terminal nodes.

The best result using traditional search methods on solving the 51 capturing problems in Kano book 4 left 11 problems un-solved, where each problem was given up-to 200 seconds [11].

9.4.2 Test Result without Extensions

Table 2 shows that pseudo ladder largely increases the ability of Go capturing computation. Using node weight, UCT improves the correctness. The greedy mode saves 13% time.

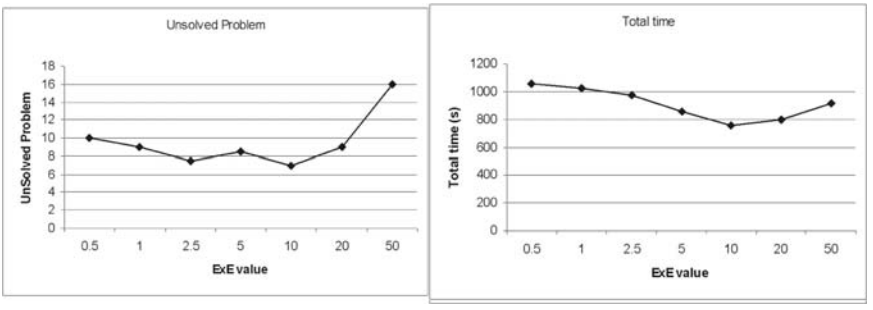

Fig. 9 Correctness and efficiency with different ExEValue

9.4.3 ExEValue

When computing the UCTValue in equation (2.1), an ExEValue constant (exploitation and exploration value) is needed. A series tests to determine the best ExEValue constant was performed. It shows that 10 is the best choice for both correctness and efficiency (Figure 9).

9.4.4 Back up Heavily When Reaching Terminal Nodes

A terminal node (with a terminal node tag) stores a definite value while MC evaluation returns a value with uncertainty. It may be possible to accelerate the search process by back up heavily when the search process reaches a terminal node. A series of tests were performed; at the time the search process reaches a terminal node we treat it as k-fold backup (backup like k simulations, the number of visit increment by k and the value increment by 0 or k). The test result shows that 8-fold backup is the best (table 3).

9.5 Discussion

Experimental results are very promising. They outperform traditional game tree search algorithms such as αβ and proof number search. And standard MC-UCT algorithm is enhanced by some extensions to improve the performance.

One drawback of tactic MC-UCT algorithm is that compared with traditional game tree search algorithms which can give sure answer (capture, escape or unknown); MC-UCT algorithm's results are based on confidence which is defined by heuristic - caused by the stochastic nature of the algorithm.

10 Tuning Parameters with a Confidence-Bounds-Guided Genetic Algorithm

A MCTS-based Go program has many critical parameters to tune [6]. These parameters interact with one another. A programmer's intuition can, at the best, only provide a rough guess of proper parameter values. GoIntellect contains ten key

parameters to be tuned. They include the constant in UCB formula, the rate of MCST widening, the relative weights of RAVE, heuristic knowledge, & simulation win rate, the basic weight of capturing/escaping,…. It is very difficult to hand tune this large set of interacting parameters. In this section, we shall describe a parameter-tuning genetic algorithm (GA) guided by upper confidence bounds of winning rates as the fitness criterion. We used a cluster of 100+ PCs to run our parameter tuning task. One of the PCs plays the role of the master and the rest PCs play the role of the slaves. Each PC keeps its own pool of chromosomes. The master will collect the statistics, generate and distribute pools of chromosomes for the next generation until the top chromosomes stabilize or produce satisfactory winning rates on numbers of games above a required threshold. The master also does a slave's job when it is idle. We used the algorithm to tune parameters of 9x9 GoIntellect with 3K simulations per move (GI3K) using GnuGo 6.0 level 10 as testing opponent and got very good results.

10.1 Genetic Algorithm in a Slave Pool

In this subsection, we discuss the details of the genetic algorithm in each of the slave pools.

10.1.1 Coding of the Parameters

The i^{th} chromosome in j^{th} slave pool can be represented by a 3-tuple:

$$N_{i,j} = \left(V_{i,j}, w_{i,j}, s_{i,j}\right)$$

where $V_{i,j}$ is a vector of parameter values (a_1, a_2 … a_n) for the n parameters to be tuned, $w_{i,j}$ is the winning rate of the parameter setting $V_{i,j}$ in pool j for GI3K against GnuGo, and $s_{i,j}$ is the total number of testing runs for chromosome i in pool j against GnuGo. Only the part $V_{i,j}$ is subject to evolution. $w_{i,j}$ and $s_{i,j}$ are result statistics of the testing runs.

The fitness function of a chromosome should be the accurate winning rate of the parameter settings.

$$ft_{org}\left(N_{i,j}\right) = \overline{w_{i,j}}$$

But obtaining the accurate winning rate of each chromosome needs a large number of testing games on it. It costs too much time and computing resources, so we adopted upper confidence bound of the winning rate instead. Thus our fitness function used is:

$$ft_{ucb}\left(N_{i,j}\right) = \begin{cases} w_{i,j} + \alpha * \sqrt{\dfrac{\lg\left(T_j\right)}{s_{i,j}}} & s_{i,j} > 0 \\ \beta > 0 & s_{i,j} = 0 \end{cases}$$

Where α and β are constants, T_j is the total number of times of all chromosomes selected from pool j and $N_{i,j}$, $w_{i,j}$ and $s_{i,j}$ are as before. The algorithm selects the chromosome whose fitness is highest in the pool at each turn.

10.1.2 Algorithm Description

As figure 10 shows, the GA algorithm in a slave pool is composed of 5 steps:

Step 1: Get the population pool from last iteration.
Step 2: Select a chromosome vector from population pool.
Step 3: Load parameters into GI3K and run a testing game.
Step 4: Update population.
Step 5: go to step 2 until the total number of runs in this pool exceeds the limits.

In step 3, we invoke GI3K to load parameter vector and play one game against GnuGo. There are two major tasks of population updating in the step 4. The first is to re-compute all parameters in population pool which need updating

$$T_j = T_j + 1; \quad s_{i,j} = s_{i,j} + 1;$$

$$w_{i,j} = \begin{cases} \dfrac{w_{i,j} * (s_{i,j} - 1) + 1}{s_{i,j}} & if \quad GI3K \quad won \\ \dfrac{w_{i,j} * (s_{i,j} - 1)}{s_{i,j}} & otherwise \end{cases}$$

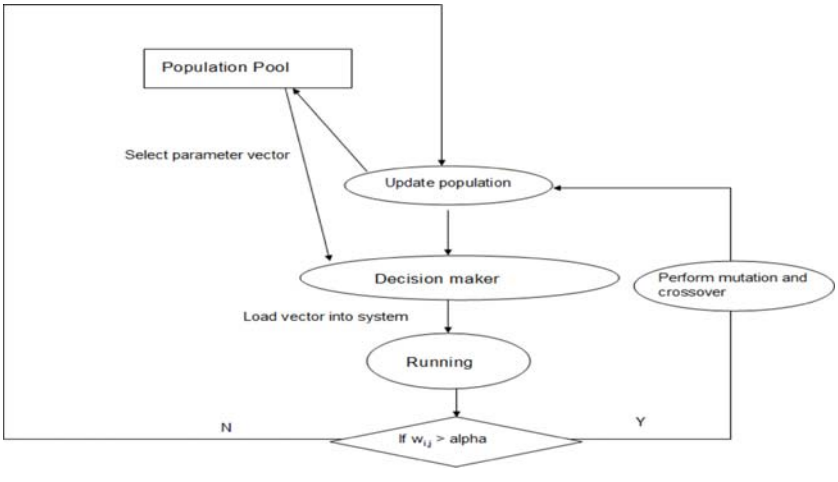

Fig. 10 Flowchart of GA in a slave pool

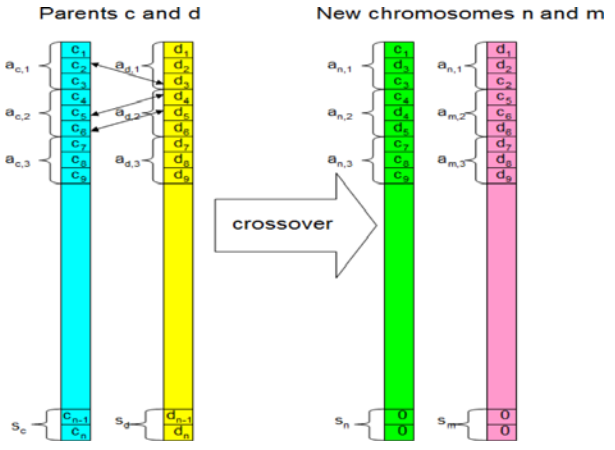

Fig. 11 Generate two new offspring from parents

Secondly, the system applies two variation operators Crossover and Mutation that generate new novel chromosomes. Crossover builds two offsprings from two parent chromosomes and mutation generates one offspring from one parent. As illustrated in Fig. 11, Crossover creates 2 new chromosomes by exchanging values of different bits in the same parameter and resetting $s_{i,j}$ to 0. Chromosomes of parents are mixed together at a rate determined by the crossover rate. This is performed iteratively until new chromosomes are generated. In Mutation, small random changes are made to the selected chromosome to generate a new chromosome. Whether choosing to perform Crossover, Mutation, or both, or just doing re-computing is determined by current winning rate of the selected chromosome plus a random factor.

10.2 General Framework

The general framework can be depicted as in Fig. 12. The iteration consists of 4 steps as follows:

Step 1: Perform GA in each of the slave pools
Step 2: Compute new population in the master pool
Step 3: Assign new population to each of the slave pools
Step 4: Go to Step 1 until convergence

The system consists of many slave pools, each of which performs GA on its own population set, and a master pool, which gathers populations from slave pools, combine the statistics of same chromosomes, make surviving decisions, and then re-distribute survivors back to individual slave pools. First, the system applies GA algorithm described in section 10.1.2 on each slave pool. Then the master pool gathers information from all slave pools and combines all chromosomes gathered

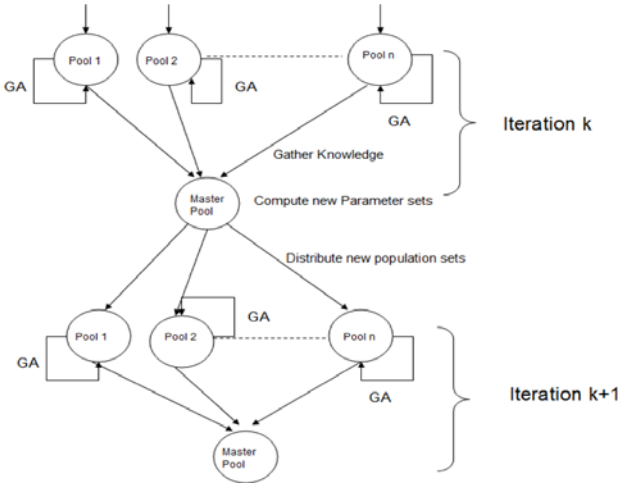

Fig. 12 Flow chart of general framework

from each slave pool with the same parameter vectors. Let $N_{i,master}$ be the i^{th} chromosome in the master pool. For each slave pool j containing the same chromosome, we update

$$N_{i,master} = \left(V_{i,master}, w_{i,master}, s_{i,master}\right)$$

$$w_{i,master} = \frac{\left(w_{i,master} * s_{i,master} + w_{i,j} * s_{i,j}\right)}{\left(s_{i,master} + s_{i,j}\right)} \qquad where \quad V_{i,master} = V_{i,j}$$

$$s_{i,master} = s_{i,master} + s_{i,j} \qquad where \quad V_{i,master} = V_{i,j}$$

The master pool will preserve all chromosomes whose lower confidence bound of fitness ft_{lcb} are higher than a threshold. And ft_{lcb} is defined as:

$$ft_{lcb}\left(n_{i,j}\right) = w_{i,j} - \alpha * \sqrt{\frac{\lg\left(T_j\right)}{s_{i,j}}} \qquad s_{i,j} > 0$$

Termination occurs when the GA converges to a solution that satisfies the target impedance requirements. To check whether the system converges, we sort all the chromosomes by ft_{lcb} in master pool. When top chromosomes are the same as the top chromosomes in last generation, we consider the process converged.

The learned parameter settings in a MCTS-based Go program with low number of simulations per move do not necessarily work well if the number of simulations per move is significantly increased. So additional tuning will be needed for a program version using large number of simulations before a move decision. Nevertheless, the parameter setting learned from a low setting can be a good starting point for further tuning at the high settings.

11 Concluding Remarks

MCTS paradigm can be used for complex systems or tasks for which the evaluation is too hard to achieve a reasonable degree of accuracy, since it bypasses the need of intermediate evaluations until the situation in the model becomes calm and can be properly understood. This makes MCTS a very powerful paradigm.

References

1. Allis, L.V., van der Meulen, M., van den Herik, H.J.: Proof-number search. Artificial Intelligence 66(1), 91–124 (1994)
2. Bouzy, B.: Associating domain-dependent knowledge and monte carlo approaches within a go program. In: Chen, K. (ed.) Information Sciences, Heuristic Search and Computer Game Playing IV, vol. 175, pp. 247–257 (2005)
3. Brügmann, B.: Monte Carlo Go (1993), http://www.ideanest.com/vegos/MonteCarloGo.pdf
4. Cazenave, T.: Abstract Proof Search. In: Marsland, T., Frank, I. (eds.) CG 2001. LNCS, vol. 2063, pp. 39–54. Springer, Heidelberg (2002)
5. Chaslot, G.M.J.B., Winands, M.H.M., Uiterwijk, J.W.H.M., van den Herik, H.J., Bouzy, B.: Progressive strategies for monte-carlo tree search. In: Wang, P., et al. (eds.) Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007), pp. 655–661. World Scientific Publishing Co. Pte. Ltd, Singapore (2007)
6. Chatriot, L., Gelly, S., Hoock, J.B., Perez, J., Rimmel, A., Teytaud, O.: Including expert knowledge in bandit-based Monte-Carlo planning with application to Computer Go. In: The 10th European Workshop on Reinforcement Learning (EWRL 2008), Lille, France (2008)
7. Chen, K.: Heuristic Programming in Artificial Intelligence. In: Levy, D., Beal, D. (eds.) Group Identification in Computer Go, pp. 195–210. Ellis Horwood (Fall 1989)
8. Chen, K., Chen, Z.: Static Analysis of Life and Death in the game of Go. Information Sciences 121(1-2), 113–134 (1999)
9. Chen, K.: Computer Go: Knowledge, Search, and Move Decision. ICGA Journal 24(4), 203–215 (2001)
10. Chen, K.: Maximizing the Chance of Winning in Searching Go Game Trees. Information Sciences 175, 273–283 (2005)
11. Chen, K., Zhang, P.: A New Heuristic Search Algorithm for Capturing Problems in Go. ICGA Journal 29(4), 183–190 (2006)
12. Chen, K.: Connectivity in the Game of Go. New Mathematics and Natural Computation 2(2), 147–159 (2006)
13. Chen, K., Zhang, P.: Monte-Carlo Go with Knowledge-guided simulations. ICGA Journal 31(2), 67–76 (2008)
14. Chen, K., Du, D., Zhang, P.: A Fast Indexing Method for Monte-Carlo Go. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 92–101. Springer, Heidelberg (2008)
15. Coulom, R.: Efficient selectivity and backup operators in monte-carlo tree search. In: Ciancarini, P., van den Herik, H.J. (eds.) Proceedings of the 5th International Conference on Computers and Games, Turin, Italy (2006)

16. Coulom, R.: Computing Elo Ratings of Move Patterns in the Game of Go. In: van den Herik, H.J., Uiterwijk, J.W.H.M., Winands, M.H.M., Schadd, M.P.D. (eds.) Proceedings of the Computers Games Workshop 2007 (CGW 2007), pp. 113–124 (2007)
17. Gelly, S., Silver, D.: Combining online and offline knowledge in uct. In: ICML 2007: Proceedings of the 24th international conference on Machine learning, pp. 273–280. ACM Press, New York (2007)
18. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modifications of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA (2006)
19. Gelly, S., Silver, D.: Combining Online and Offline Knowledge in UCT. In: Ghahramani, Z. (ed.) Proceedings of the International Conference of Machine Learning (ICML 2007), pp. 273–280 (2007)
20. Gelly, S., Hoock, J.B., Rimmel, A., Teytaud, O., Kalemkarian, Y.: The parallelization of Monte-Carlo planning. In: Proceedings of the 4th International Conference on Informatics in Control, Automation and Robotics (ICINCO), Madeira, Portugal, pp. 198–203 (2008)
21. Kano, Y.: Graded Go Problems For Beginners, Intermediate Problems, vol. 3. Kiseido Publishing Company (1987) ISBN 4-906574-48-3
22. Knuth, D., Moore, R.: Analysis of Alpha-Beta Pruning. Artificial Intelligence 6(4), 293–326 (1975)
23. Kierulf, A., Chen, K., Nievergelt, J.: Smart Game Board and Go Explorer: A study in software and knowledge engineering. Communications of ACM 33(2), 152–166 (1990)
24. Kocsis, L., Szepesvári, C.: Bandit Based Monte-Carlo Planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
25. Wang, Y., Gelly, S.: Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In: IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii, pp. 175–182 (2007)
26. Zhang, P., Chen, K.: Monte-Carlo Go Tactic Search. New Mathematics and Natural Computation Journal 4(3), 359–367 (2008)
27. Zhang, P., Chen, K.: Using Different Search Algorithms to Solve Computer Go Capturing Problems. In: Proceedings of 2006 Chinese Computer Games Conference, pp. 55–61 (2006) (in Chinese)