Monte Carlo Tree Search for Go

Author: *Thomas Ager*

Supervisor: *Steven Schockaert*

Moderator: *Paul Rosin*

Module Number: *CM3203*

Module Title: *Individual Project*

Credits Due: *40*

Abstract

This report details an investigation into the Monte Carlo Tree Search (MCTS) algorithm applied to Go. It discusses the background of the MCTS algorithm, and details an implementation of the MCTS algorithm in Java. It ends with a statistical analysis of the different features discussed in the background and implementation. This is the first Java MCTS Go player released open-source, and the first report to detail the statistics behind every feature chosen for the final implementation.

Contents

Introduction

Go is a skill-based game in which two players take turns putting black and white stones on a board with the objective of surrounding vacant areas. Well known for its strategic diversity and tactical complexity, Go has been a staple game across Asia for centuries. Its rich history of study, endless potential for improvement, and a varied history of attempted solutions has elevated the task of creating a computer Go player that can rival top human players into a grand challenge of Computer Science.

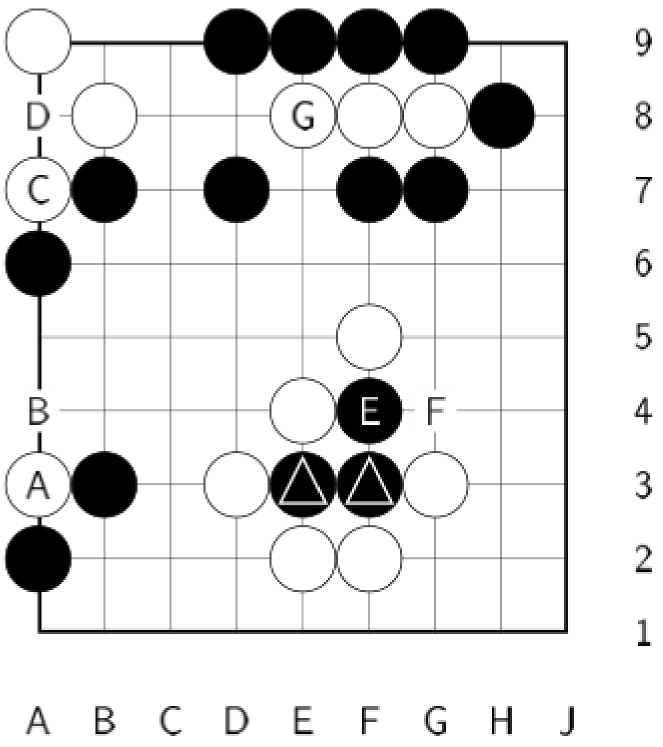
Before the Monte Carlo Tree Search (MCTS) algorithm, no attempted solutions were met with success above that of an amateur player. This report takes on the task of investigating the benefits and limitations of MCTS, discussing the effectiveness of different modifications to this approach, as well as a statistical analysis of every feature included in the project.

To begin, this report discusses the groundwork of research that lead up to the modern MCTS algorithm, as well as detail about the algorithm itself. Once MCTS has been introduced, different enhancements featured in the latest and greatest MCTS Go players are covered, followed by how some of these enhancements were developed for our own open source Java implementation. The report then moves onto a statistical analysis of every feature in the implementation, with analytical investigation into finding the optimal values for those features. Finally, the report concludes with a discussion about the overall effectiveness of the current implementation, and a hypothetical discussion about the direction of possible future research.

The Game of Go

The Challenge of Go comes from the way its simple rules are utilized. The strategic and tactical possibilities of Go are endless, allowing the game to accommodate any playstyle and any development. Players are skilled not purely because they have learned the rules and applied them, but because Go allows players to make use of their unique intuition insights and apply their personality to create a unique playstyle that can defeat their opponent.

Starting on an empty board, players place take turns placing stones on the board to surround vacant areas, called territory. Stones are placed on the intersections of the lines, called points, with each player placing a single stone on a vacant point to complete their turn. The colours of the stones are black and white, with black going first. There are three popular board grid sizes, 9x9, 13x13, and 19x19, and each of these board types give the option to handicap players by giving their opponent up to nine stones at the start of the game. The game is played until both players pass their move or one resigns completely. There are a few different types of scoring. The scoring that we will be used in this report is Tromp-Taylor scoring, where each player receives a point for each vacant point inside their own territory, and one point for every stone of their colour. The player with the most points at the end of the game wins.



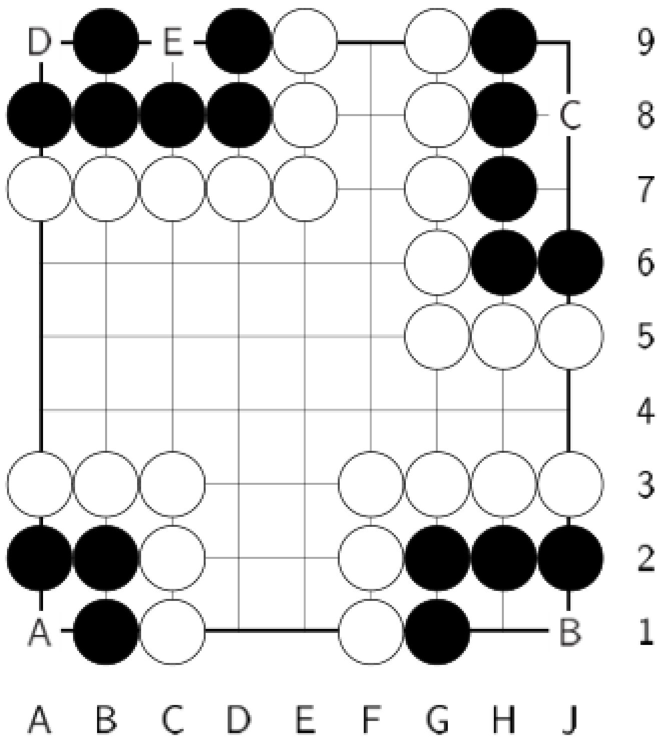
An example figure showing various Go situations, from [Bau11]

Playing the same colour stone vertically or horizontally adjacent to another stone makes both stones part of a single group, which can then be used to capture territory or enemy stones.

Free spaces next to stones are called liberties. Capturing opponent stones is done by surrounding groups so that the group no longer has any liberties, in which case the opponent stones are removed from the board. A stone or group of stones with a single liberty is in Atari, and self-Atari is known as one of the worst moves that a player can do. However, Go games are varied enough that like all generally bad moves, self-Atari moves are sometimes necessary.

The ko-rule does not allow a player to instantly re-capture a single stone once it has been surrounded, forcing players to play tactically around waiting for a move before they are able to capture the stone back.

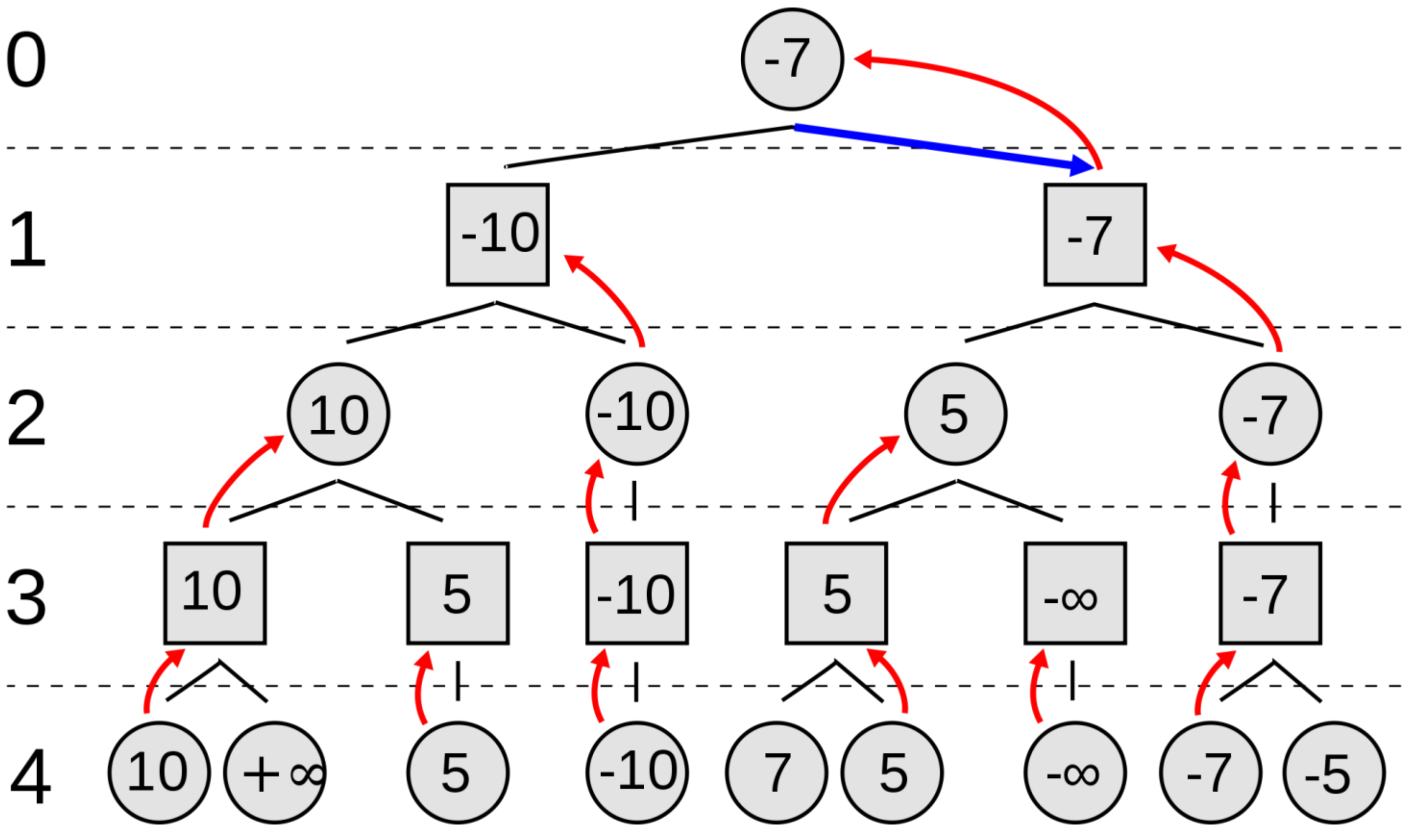
An eye is described as a vacant space is surrounded by a group of a same-colour stones. Eyes are one of the most important tactical concepts for Go. Playing within opponent's eyes without capturing the stones around the eye instantly removes the stone being played. This move is considered illegal. This means that if a group has two eyes, then it is impossible to capture. Finding a way to give a group of stones an eye or two eyes is known as a life and death problem.



An example figure showing a life and death situation, from [Bau11]

Computer Go

There have been many different approaches to solving the game of Go. The most successful algorithm out of the classical Go techniques is A-B pruning/minimax, with efforts culminating in GNUGo [BFB+]. A minimax algorithm is an algorithm that computes the value of possible moves for both you and your opponent, then makes decisions based on what move will give your opponent the lowest possible value option for their move. This can be represented as a tree, with every node being a move that can be taken from the parent move. In GNUGo [BFB+] this was combined with alpha-beta pruning, an algorithm that prunes tree branches that cannot possibly influence the final decision.

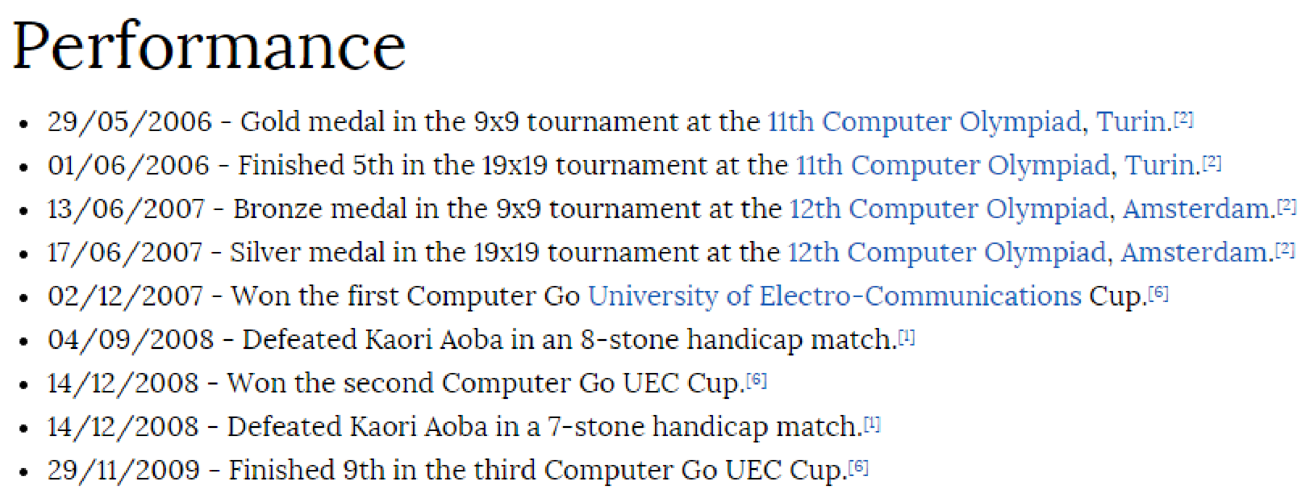


A minimax tree, from url: http://upload.wikimedia.org/wikipedia/commons/thumb/6/6f/Minimax.svg/2000px-Minimax.svg.png.

Early on in Go, an algorithm was created for use in pattern-based play [B0090], which matched 5x5 windows with patterns that could be determined with a generated pattern database [BC05]. However, this approach had required an immense amount of domain knowledge and a gigantic amount of patterns to be effective. Despite the classic approaches and other popular alternative approaches like neuron networks [Enz96] falling short of the mark, some sub problems of Go were solved. Solvers of isolated life and death problems are now capable of solving that are considered of professional-level difficulty [Wol07].

Despite the monumental amount of effort invested in domain-knowledge techniques, The thrust of a good deal of Computer Go research is now to determining which variations and enhancements of the MCTS algorithm are best suited for Go. [Iee12] GNUGo, considered the strongest of the classic computer Go programs, has not been competitive since 2008 [BFB+]. The MCTS approach is the only approach to come close to defeating professional players any board above 8x8. In 2008 MoGo [Gel+06] managed to take a game from professional Go Player Cătălin Ţăranu, 5th dan pro, on a

9x9 board. In 2013, Crazy Stone beat Yoshio Ishida, professional 9 dan player with a four stone handicap [1]. The future is promising for MCTS Go players.



Performance

- 29/05/2006 - Gold medal in the 9x9 tournament at the 11th Computer Olympiad, Turin.[2]
- 01/06/2006 - Finished 5th in the 19x19 tournament at the 11th Computer Olympiad, Turin.[2]
- 13/06/2007 - Bronze medal in the 9x9 tournament at the 12th Computer Olympiad, Amsterdam.[2]
- 17/06/2007 - Silver medal in the 19x19 tournament at the 12th Computer Olympiad, Amsterdam.[2]
- 02/12/2007 - Won the first Computer Go University of Electro-Communications Cup.[6]
- 04/09/2008 - Defeated Kaori Aoba in an 8-stone handicap match.[1]
- 14/12/2008 - Won the second Computer Go UEC Cup.[6]
- 14/12/2008 - Defeated Kaori Aoba in a 7-stone handicap match.[1]
- 29/11/2009 - Finished 9th in the third Computer Go UEC Cup.[6]

[1] A performance list gathered from https://en.wikipedia.org/wiki/Crazy_Stone_(software).

Theory

Monte Carlo

Monte Carlo is a semi-random approach that makes use of randomly simulating games according to a policy, and then updating statistics

Monte Carlo approaches use the results of semi-random simulations to determine statistics about the possible solutions to the problem. The first Monte Carlo approaches were not successful [Bru93], and further research did not show much of an improvement [BH03]. Monte Carlo simulations were used to determine the next possible moves, rating the move high if the random simulation was not likely to play a move and low if the random simulation was likely to play the move. This was good at finding strategic moves that set up a large attack or enclosed a chunk of territory, but suffered problems in tactical situations. [Bau11]

Multi-armed Bandit Problem

The multi-armed bandit problem is a decision making problem based on the idea of slot machines named one-armed bandits, where the player inserts a coin and then receives a reward from the bandit according to a fixed probability distribution. In the multi-armed bandit problem, there are multiple arms each with different fixed probability distributions. In order to maximize the reward, arms should be chosen in a way that maximizes the possible gain. These estimations can then be used to choose the bandits that are most likely to lead to the total highest reward. [LR85], [KS06] and [GW06] define the K-armed bandit as a sequence of random rewards $X_{it} \in [0,1], \mathrm{i} = 1,\ldots, K, t \in \mathrm{N}$ where each arm has an index i and t denotes successive plays of the arm. [Bau11]

The most effective algorithms for solving this problem make use of the idea of balancing exploration and exploitation. Exploring the environment to find potentially profitable actions, and exploiting the current highest reward from the best arm. Regret is the term used for the expected loss due to not playing the best arm. [Iee12] defines the regret R_n as such: let μ_i denote the estimated expectation of arm i, $\mu*$ denote the expectation of the optimal arm, and $T_i\ (n)$ the number of times arm i has been played after the first n plays:

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

$$\mu^* = \max i\ \mu_i$$

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

UCB

A bandit expectation policy π is a function that selects the next arm to be played based on previous rewards, designed to minimize the amount of regret [Bau11]. Lai and Robbins [LR85] show that there is no policy with a regret that grows slower than *O(log n*). If a policies regret is proven to grow at *O(log n)*, then it is said to solve the multi-armed bandit problem. Lai and Robbins a policy that solved the multi-armed bandit problem, making use of hard to compute upper confidence indices. Following this, a family of algorithms called UCB was proposed by [ABF02] that expressed the upper confidence index in a simpler way, finally resulting in an extension of the UCB policies to fit the sequential, tree-based planning required for Artificial Intelligence applications [KS06], which has been used in top Go programs MoGo [Gel+06], and [Enz+10], as well as many others.

UCB1

UCB1 is one of the policies introduced by [KS06] that solve the multi-armed bandit problem. It calculates uncertainty relative to the amount of visits to the node by maintaining the number of times an arm has been played in addition to the empirical mean. This means that arms that it will visit low-expectation arms if others are too uncertain.

UCB-TUNED

UCB1-TUNED is an algorithm introduced by [ABF02] that has no theoretically proven guarantees for its regret bound, but has been shown through research to vastly outperform UCB1 [Gel+06] [ABF02]. It uses variance for its upper confidence bound, calculated using every individual reward from a bandit [2]. In [ABF02], UCB-Tuned is defined as:

$$V_j(s) \overset{\text{def}}{=} \left(\frac{1}{s}\sum_{\tau=1}^{s} X_{j,\tau}^2\right) - \bar{X}_{j,s}^2 + \sqrt{\frac{2\ln t}{s}}$$

$$\bar{X}_j + \sqrt{\frac{\log n}{T_j(n)} \min\{1/4, V_j(T_j(n))\}} .$$

Monte Carlo Tree Search

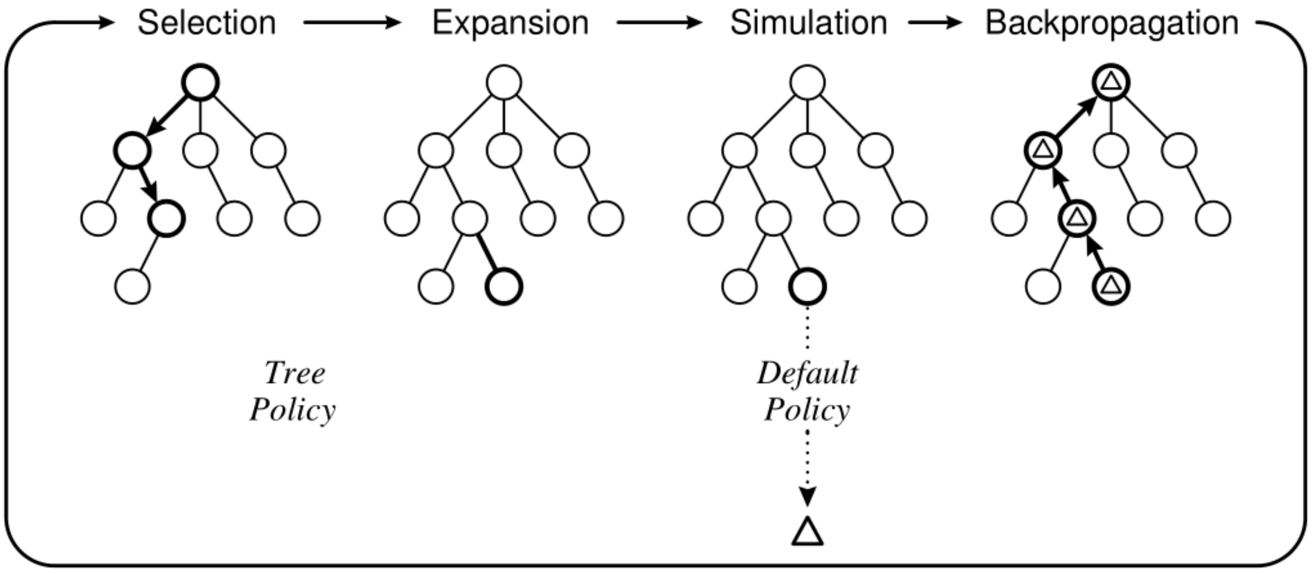
Monte Carlo Tree Search (MCTS), first introduced as Upper Confidence Tree (UCT) in and proven to converge on the minimax action value function by [KS06]. The MCTS tree is simple, every node is a state of the game, and the children of those nodes are the next possible moves. MCTS treats each state of the tree as a multi-armed bandit, using a UCB policy to calculate the value for each state. To obtain the values to calculate the UCB for each node, semi-random Go games are played out and the amount of visits to that node are recorded.

MCTS has many benefits. Its tree is built online and can be stopped anytime to choose a result that improves as more time is spent. It is highly memory efficient, growing asymmetrically based on the prevailing direction of the search, and as the values of intermediate states do not have to be evaluated as in depth-limited minimax search, only simple domain knowledge is required to make a big impact.

The algorithm develops the tree until it has reached the end of its predefined budget, typically an amount of seconds or an amount of simulations. It is first initialized by a single root node, which represents the state that we are looking to find a decision for. [Iee12] Outlines four steps to developing the tree:

1. **Selection**: The tree is descended according to a selection policy until a non-terminal node with unexpanded children is found.
2. **Expansion**: The child nodes are expanded based on the pruning policy.

3. **Simulation**: A simulation is run from the new node(s) according to the simulation policy to produce an outcome.
4. **Backpropagation**: The simulation result is backed up through the selected nodes to update their statistics.



MCTS Diagram, From [Iee12]

Once the budget is reached, a move is selected from the root node according to a move selection policy, typically the most simulated node. In [Iee12], there are four criteria for selecting the winning action:

1) Max child: Select the root child with the highest mean reward.
2) Robust child: Select the most visited root child.
3) Max-Robust child: Select the root child with both the highest visit count and the highest reward. If none exist, then continue searching until an acceptable visit count is achieved [70].
4) Secure child: Select the child which maximises a lower confidence bound

All Moves As First (AMAF)

Introduced before MCTS [BH03], the All Moves As First (AMAF) heuristic is the underlying idea that there can be one value for each move, regardless of when it is played. This allows information about moves to be shared across every situations subtree, updating a significant amount of nodes for a single result.

Rapid Action Value Estimation (RAVE)

The RAVE algorithm proposed by [GS07], gave MCTS a major leap in strength by combining Monte-Carlo tree search with the all-moves-as-first heuristic. The basic idea of RAVE is to generalise over subtrees by replacing Monte Carlo values in the tree with RAVE values. The assumption is that the value of a move in a situation will be similar to all states within the subtree of that situation. Thus, the value of a move is estimated using all relevant simulations, regardless of exactly when the move is played. For every position in the simulation that is represented in the tree, and for every subsequent move of the simulation the AMAF value of is updated according to the simulation outcome [GS07].

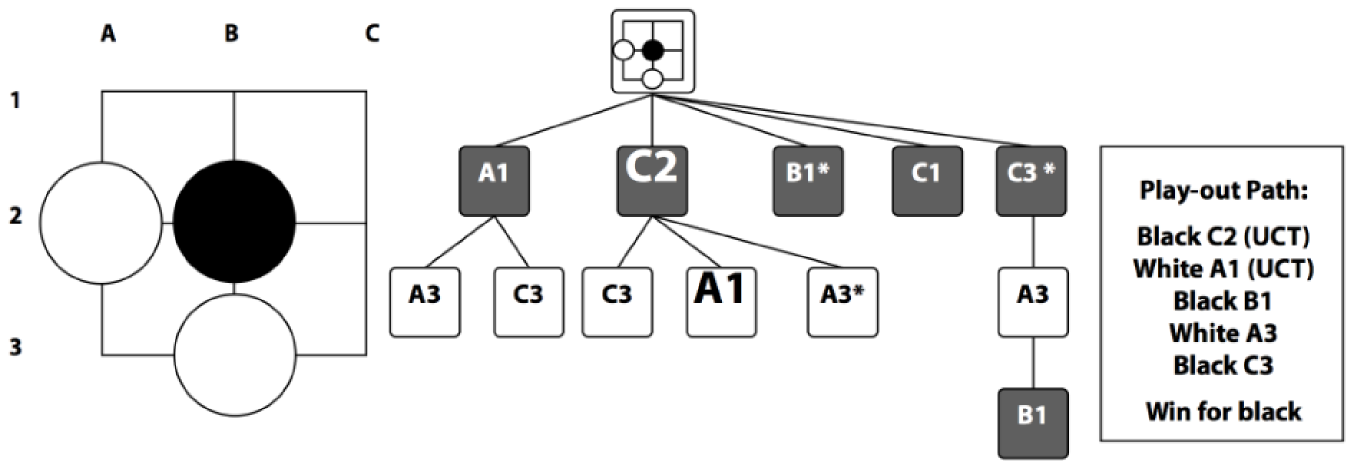


Fig. 4. The All Moves As First (AMAF) heuristic [101].

AMAF Example diagram, from [Iee12]

MC-RAVE

Despite RAVE creating large amounts of information to be shared early on in the development of the tree, drastically reducing the need for explicit exploration [GS11], the idea that the value for a move is the same no matter what time it's played can be proved wrong fairly trivially. To counter this, both the Monte Carlo and the RAVE statistics are stored, and both are calculated when requested. Then, both of the calculated values are balanced based on the amount of simulations for the node, giving more value to RAVE if the simulation count is lower, and more value to the Monte Carlo value if the simulation count is higher.

Prior Values

The UCB1 policy requires nodes to be initialized with a value. Generally, this is done by playing a random simulation for each new node [Iee12]. Because of the additional time taken to play simulations for every node, as well as the possibility that a large amount of nodes initialized will never need consideration for playing, we have opted for a technique called First Play Urgency (FPU) instead.

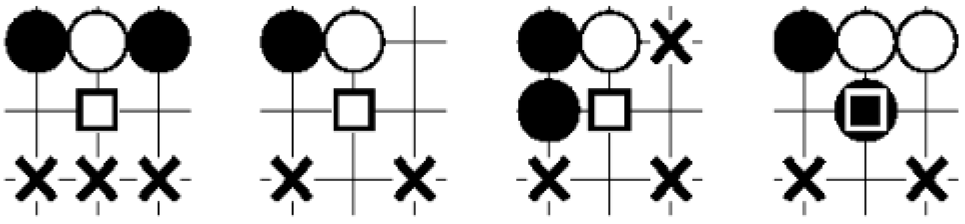
FPU is a technique used by a few Go players [Gel+06] to allow their policy to explore new nodes more often. It assigns a high predefined value to every new node, which can be adjusted higher to increase exploration and lower to decrease it. In our tests [3] we found that FPU handily defeated other approaches.

There are two other ways to initialize prior values: Applying a ranking function and pruning all children that are below a certain value threshold, and adjusting the expectation of nodes through virtual simulations based on heuristic rules. One advantage of the latter approach is that the bonus is devalued as more simulations are developed, reducing the overall impact of the virtual simulations. This technique was used in both Fuego [Enz+10] and Pachi [Bau11].

Simulation Policy

In the general MCTS algorithm, the simulation policy picks random move after random move until there are no more moves to play. Although this can give some effective results for some applications [Iee12], in order for meaningful game-related results the simulation policy must use domain knowledge heuristics to play moves [Gel+06]. The most common simulation policy for Go is playing a game from start to a pre-determined finishing point, usually when there are no more moves to play apart from within friendly eyes. This approach works, but does not play tactically very well.

This limitation can be countered by using heuristics to determine which move should be played. MoGo [Gel+06] and Fuego [Enz+10] uses a set of heuristic rules in a fixed-order. These rules are simple principles of how to tactically play Go, in MoGo [Gel+06] and Fuego [Enz+10], a set of 3x3 patterns are matched on the spaces around the last move. These patterns are flipped, rotated and the colours of the stones within are stopped in order to find a move, and if a move is not found then the next heuristic rule is attempted. In Fuego, this is checking if an opponent can be put into Atari, or if an opponent can be captured.



Hane 3x3 patterns, as described by MoGo [Gel+06]

Other players use a probability distribution policy [Cou07] where all available moves are assigned an expectation as a product of evaluations by individual heuristics matching the particular move and its surrounding spatial patterns. The move is then given a probability to be played in the simulation. Pachi [Bau11] uses a similar probability function for their fixed-order methodology, giving each rule a certain probability to be played.

Despite heuristics being effective at improving the playing ability of MCTS, the implementation of a heuristic will always result in a bias that ends up playing a specific case wrong. This can be a serious problem when playing against an opponent that can recognize and exploit a weakness. An example of an approach in AI that takes advantage of this is YoGo's opening book, which contains a large number of positions designed by professional 6 Dan Yu Ping that are often misplayed by current MCTS [Enz+10].

Binary Scoring

The scoring of a simulation game can be returned as the real score of that game or a more simplified, binary score [4], where 1 is returned for a win and 0 is returned for a loss. This decision gives a more robust playstyle that avoids preferring moves that have just a small chance of a large score swing, instead aiming to obtain a win no matter how many overall points are in the result. The binary score can be adjusted to create more humanlike behaviour by returning half of a win (0.5) if the score is near-even. Additionally, small bonuses for shorter playouts and higher scores can be given like in Fuego [Enz+10]. For our approach, we adopted the humanlike even-scoring but not the bonuses for shorter playouts and higher scores, in order to keep the implementation focused on scoring a win by any means necessary rather than doing it quickly or with a larger score.

Setup

The idea behind the setup of the implementation was to make use of already existing software and players in order to maximize the time spent investigating and statistically analysing MCTS. This resulted in less time spent implementing features, and more time spent understanding them. There is already a well-established set of tools for developing a Go AI. GTP, a command protocol used to interact with well-known tools, and GoGUI, an open-source GUI that can receive GTP commands to play out AI games.

In order to get a solid base for the board representation, I used Vegos, an oldschool Monte Carlo implementation that uses simulating annealing [1] to find a solution. Vegos had all the tools I needed for the baseline of developing an MCTS application [2]. The implementation was developed on that base using the Java programming language, as at the time of writing this report there are no MCTS Go players available in Java. I also used the ideanest and the Apache Commons for some trivial functionality implementations.

[1] Vegos Simulated Annealing report url: http://www.ideanest.com/vegos/LasVegasGo.pdf

[2] Vegos site with source code url: http://www.ideanest.com/vegos/

Implementation

In this section, I will be discussing how the final software works with every option that resulted in a higher win rate when used. The actual implementation has complete configuration options, but for the sake of brevity the implementations of alternative options will not be covered.

The configuration for the software was done on a player-by-player basis, with each player requiring an input of a ruleset which includes information about every feature. This ruleset is then used during the tree development and the move selection to make decisions appropriate to what the user has specified. Below are the functional features that can be initialized for the configuration.

```
/* set the values for different features */
this.nodeRuleSet = new Configuration(binaryScoring, uct, amaf, rave,
openingBook  initialWeight, aAmafWeight, raveWeight, raveSkip, bonusFpu,
firstPlayUrgency, bonusPatterns, bonusAvoidEyes, explorationWeight,
simulateAvoidEyes, simulatePatterns, simulateMercyRule, varySimEyes,
varySimPatterns, pickRobust, pickMax, pickSecure, pickMaxRobust, clearMemory,
pruneNodes, developPruning, ucb, simpleUcb, singleLogUcb, ucbTuned,
evenScoring);
```

The following are the options that we chose based on our theoretical and experimental research. Every option has been justified by an increased winrate or a significant difference in playstyle.

The configuration for the software was done on a player-by-player basis, with each player requiring an input of a ruleset which includes information about every feature. This ruleset is then used during the tree development and the move selection to make decisions appropriate to what the user has specified. Below are the functional features that can be initialized for the configuration.

```
/* set the values for different features */
Configuration(binaryScoring = true, rave = true, openingBook = true, raveWeight
= 1000, raveSkip = 20, firstPlayUrgency = 0.9, bonusPatterns = 50,
bonusAvoidEyes = -5000, simulateAvoidEyes = true, simulatePatterns = true,
simulateMercyRule = true, varySimEyes = 0.05, varySimPatterns = 0.3, pickRobust
= true, clearMemory = true, pruneNodes = 2, ucbTuned = true, evenScoring = 10);
```

Vegos

In Vegos, an engine uses a player to generate moves using the AI modules, then that move is translated into a GTP command using the GTP modules. For my implementation, I used the exact same setup. The board in vegos is represented as an array of points, each with the property of EMPTY, BLACK, WHITE, or OUT_OF_BOUNDS. The game used Tromp-Taylor Rules, and a more restrictive Semi-Primitive game type [1] disallows playing in eyes.

[1] This type was renamed to SimulateGame, as I exclusively used it for simulations.

MCTSEngine

The MCTSEngine is a modified version of the Vegos Engine, with the simulated annealing specific classes removed. The two most important classes are the ones that handle the opponents moves, play(move), and the AI player's moves genmove(color). Play works first by checking if the game is over or the colour doesn't match the one on the pre-built game. If this passes then the move is played using the game class by converting the GTP command into a position on the board.

```
public void play(Move move) {

  /* if the game is over or it isn't our move */
  if (game.isOver() || move.color != game.getNextToPlay()) {
    // play out of order, simulate a pass
    boolean r = game.play(Game.MOVE_PASS);
    assert r;
  }
  assert move.color == game.getNextToPlay();
  boolean r = game.play(move.vertex.toPosition(game.getGrid()));
  game.recordMove(move.vertex.toPosition(game.getGrid()));
}
```

Genmove works by checking if the player has been initialized, and if it hasn't then starting off the variables for the player-specific game. The updated game is then set in the player, ensuring that the new game state is used for the next tree development. Then, if the game is over, the player passes with a -1 move, and if it isn't over then the a request is sent to the player to get the move. Once the move is returned, it is converted to a vertex and sent back as a GTP command.

```
public Vertex genmove(Color color) {
  Player player = players[color.getIndex()];
  if (player.getPlayingColor() != color) {
    // bring player into the game
    player.startGame(game, color);
  }
  player.setGame(game);
  int move;

  if (game.isOver()) {
    move = -1;
  } else {
    move = player.playMove();
  }
  return Vertex.get(move, game.getGrid());

}
```

Opening Book

The opening book is imported from a data file provided by Fuego [Enz+10], consisting of a sequence of lines, each one containing a move to be played, information about the board size, and the moves that need to be played in order for that opening book move to be matched. The data is imported into list of integer arrays, with each integer within the arrays containing a single move. The lists are split into two sets of integer arrays, the moves to take once a pattern is matched, and the pattern to match to the moves played.

```
/* For every line in the opening book */
for(String line: FileUtils.readLines(new File("D:/#Work/Learning/Third Year
Project/Project/book.dat"))) {
    /* If the line is not empty */
    if(line.startsWith(sideSize)) {

      /* Split it into the moves to match and the moves to play */
      String[] pipeSplit = line.split("\\|");
      String[] tempBeforeMoves = pipeSplit[0].substring(2).split("\\s");
      if(tempBeforeMoves == null || tempBeforeMoves.length == 0) {
        tempBeforeMoves = new String[1];
        tempBeforeMoves[0] = "";
      }
      String[] tempMovesToTake = pipeSplit[1].split("\\s");
      int[] beforeMoves = new int[tempBeforeMoves.length];
      int[] movesToTake = new int[tempMovesToTake.length];

      /* If there are no moves to match, set a value to recognize that */
      if(tempBeforeMoves[0].isEmpty()) {
        beforeMoves[0] = -1;

      /* If there are moves to match, convert them into points on the board */
      } else {
        for(int i=0;i<tempBeforeMoves.length;i++) {
          String[] splitBeforeMoves =
tempBeforeMoves[i].split("(?<=\\D)(?=\\d)|(?<=\\d)(?=\\D)");
          int move = new Vertex(convertLetterToInt(splitBeforeMoves[0]),
Integer.parseInt(splitBeforeMoves[1])).toPosition(game.getGrid());
          beforeMoves[i] = move;
        }
      }
      if(tempMovesToTake != null && !tempMovesToTake[0].isEmpty()  ) {
        for(int i=0;i<tempMovesToTake.length;i++) {
          String[] splitMovesToTake =
tempMovesToTake[i].split("(?<=\\D)(?=\\d)|(?<=\\d)(?=\\D)");
          int move = new Vertex(convertLetterToInt(splitMovesToTake[0]),
Integer.parseInt(splitMovesToTake[1])).toPosition(game.getGrid());
          movesToTake[i] = move;
        }
      } else {
        String removeWhiteSpace = pipeSplit[1].replaceAll("\\s+","");
        String[] splitMovesToTake =
removeWhiteSpace.split("(?<=\\D)(?=\\d)|(?<=\\d)(?=\\D)");
        int move = new Vertex(convertLetterToInt(splitMovesToTake[0]),
Integer.parseInt(splitMovesToTake[1])).toPosition(game.getGrid());
        movesToTake[0] = move;
      }

      /* And add the array of ints to the list of arrays */
      this.beforeMoves.add(beforeMoves);
      this.movesToTake.add(movesToTake);
    }
  }
}
```

Then, the values in the arrays are rotated, flipped and mirrored to form a set of new arrays.

```java
public int[] modifyArray(int[] array, int type) {
  int[][] rotate = Game.rotate(convertArrayToGrid(array));
  switch(type) {
  case 0: return array;
  case 1: return convertGridToArray(rotate, array.length);
  case 2: return convertGridToArray(Game.rotate(rotate), array.length);
  case 3: return convertGridToArray(Game.rotate(Game.rotate(rotate)),
array.length);
  case 4: return
convertGridToArray(Game.verticalFlip(convertArrayToGrid(array)), array.length);
  case 5: return
convertGridToArray(Game.horizontalFlip(convertArrayToGrid(array)),
array.length);
  }
  return null;
}
```

It is these new arrays, as well as the initial arrays, that are tested to match the board. If they match, then the move is played and the opening book method returns success. A value is set if there are more than one moves to play.

```java
public boolean playOpeningBookMove(int[] mostRecentMoves) {
  move = 0;
  boolean broken = false;

  /* if no moves have been played */
  if(mostRecentMoves[0] == 0) {
    /* find a move without beforemoves */
    for(int i = 0; i < beforeMoves.size(); i++) {
      if(beforeMoves.get(i)[0] == -1) {
        /* and play it */
        move = movesToTake.get(i)[0];
        System.out.println(move);
        return true;
      }
    }
  }

  /* if there were moves already recognized and they weren't all equal */
  if(movesLeft > 0) {
    /* get the next move and decrement */
    int tempMove =
movesToTake.get(moreMovesToPlay)[movesToTake.get(moreMovesToPlay).length -
movesLeft];
    if(!isContainedInArray(mostRecentMoves, tempMove)) {
      move = tempMove;
      movesLeft = movesLeft - 1;
      return true;
    }
  }
  List<int[]> beforeMoves;
  List<int[]> movesToTake;

  /* for every type of modification to the moves */
  for(int j=0;j<6;j++) {
    /* modify the beforeMoves and movesToTake */
    beforeMoves = modifyList(this.beforeMoves, j);
    movesToTake = modifyList(this.movesToTake, j);
    /* for every line */
    for(int i = 0; i < beforeMoves.size(); i++) {
      /* if there are more moves than the moves that are being checked then
skip this line */
      if(beforeMoves.get(i).length == movesTaken) {
        broken = false;
        /* if any of the opening moves last moves match the last move */
        if(beforeMoves.get(i)[beforeMoves.get(i).length - 1] == game.getMove(0))
{
          /* check if the whole sequence matches, if the length is larger than
one */
          for(int n=0;n<beforeMoves.get(i).length;n++) {

            /* if the element in the beforemoves 0-n doesn't match the move -
beforeMoves.length ago, with -1 to accommodate for the way that getmove
functions */

            if(beforeMoves.get(i)[n] != mostRecentMoves[n]) {
              /* there's no need to look at any other moves, so just move onto
the next line */
              broken = true;
              break;
            }
            if(broken) {
              continue;
            }
          }

          /* if a move set matched, the move is equal to the matching move to
take */
          int tempMove = movesToTake.get(i)[0];
          if(!isContainedInArray(mostRecentMoves, tempMove)) {
            move = tempMove;

            /* if there's more moves to play for this set, then set the value
equal to the iteration */
            if(movesToTake.get(i).length > 1) {
              moreMovesToPlay = i;

              /* and set the amount of moves left to go through equal to the
length -1 */
              movesLeft = movesToTake.get(i).length -1;
            }
            return true;
          }
        }
      }
    }
  }
  return false;
```

```
}
```

The Player

The player, named MCTSPlayer makes use of three methods and an initializer method. setGame(), which updates the game state, endGame() which resets game-related values, and playMove() which goes through a sequence of events in order to play a move on the board, and return that move to the engine. setGame() and endGame() exist so that the game state can be updated from the engine after an opponent's move has been played, playMove is setup as a single method so that multiple different attempts at getting the move can be tried by accessing other classes, based entirely on the configuration. This means that the methodology for obtaining the move is determined by the player, which is an important distinction to make when considering that each player can have a different configuration setup.

Playing Moves

First, the player checks if they are being beaten to a degree that it would be impossible to recover.

```
/* If it is lategame and the opponent has a much higher score than us, just
give up */
if(movesTaken>=((game.getSideSize()*game.getSideSize()) -1) && game.mercy()) {
  /* just end the game */
  return -2;
}
```

Then, if there exists a move to record, meaning that the opponent has played a move, then record it in the most recent moves. Then, based on the most recent moves check if it is possible to use a move from the opening book, and if it is possible then the move is played.

```
/* If using an opening book and no moves have been taken */
if(game.getMove(0) != -3 && openingBook.movesTaken < 15) {
      /*Add a first move and increase the amount of moves taken in the
openingBook*/
  firstMoves[openingBook.movesTaken] = game.getMove(0);
  openingBook.movesTaken++;
}
/* If there are still possible moves left to be played, and the opening book
matches the moves played */
if(openingBook.movesTaken < 15 &&
openingBook.playOpeningBookMove(mostRecentMoves)) {

  /* Set the move to the opening book move */
  move = openingBook.move;

  /* Play the move on the board for this player */
      game.play(move);

      /* And add that move to the mostRecentMoves */
      mostRecentMoves[openingBook.movesTaken] = move;

      /* Incrementing the amount of moves taken in the opening book,    and
setting the initrootNode variable to false, meaning that there is not a tree
that has been built because an opening book move has been taken. This triggers
code below to recreate the rootNode from scratch. */
      openingBook.movesTaken++;
      initRootNode = false;

}
```

If that's not possible and a root node does not already exist then it creates a root node.

```
/* Check if the rootNode requires initialization */
if(!initRootNode) {

  /* If it does, create a TreeNode that represents the players current position
   * And set the initRootNode value to true, meaning that the rootNode can be
used
   * Later to re-use subtrees. */
  initRootNode = true;
  UCB ucbTracker = new UCB(nodeRuleSet);
  rootNode = new TreeNode(game, game.getMove(0), 0, side, nodeRuleSet,
ucbTracker);
}
```

If a root node does exist, it searches through its children to find the node that matches the opponent's most recent move, and re-uses the subtree developed from that node.

```
/* If the rootNode doesn't require initialization, then find the new root node
 * Out of the old root nodes children, by matching the move taken from that
node To the last move played by the opposing player.
 */
} else {
  TreeNode oldRootNode = rootNode;

  /* The root node is equal to the child that matches the last move played in
the game. The game knows which move was played because setGame was called
before a move was requested by the engine.
   */
    rootNode = rootNode.getChild(game.getMove(0));

    /* Once the old root node has been discarded, there is no need to store any
of its values or its children in memory. This algorithm iterates over the
entire subtree, setting every child in the subtree of the root node that is not
the new root node to null. */
    oldRootNode.clearParentMemory(rootNode, oldRootNode.getChildren());
}
```

Once a root node is initialized or selected for re-use, the tree is developed from that root node.

```
/* If the player wants to make use of time */
if(time > 0) {

  /* Initialize a timer */
  ElapsedTimer t = new ElapsedTimer();

  /* Until that timer has elapsed its predefined limit */
  while(t.elapsed() < time) {
        /* Develop the tree from the root node */
    rootNode.developTree();
     }
}

/* If the player wants to use the amount of iterations/simulations, then
 * just run developtree the amount of times they've determined.
 */
if (iterations > 0) {
  for(int i=0;i<iterations;i++) {
    rootNode.developTree();
     }
}
```

Once the tree is developed, the highest value node is taken from the children of the root node, based on the criteria detailed in the configuration.

```
/* Get the highest value move from the root node, based on the criteria set out
in the config */
move = rootNode.getHighestValueMove();

/* Play the move on the board for this player */
game.play(move);
```

The opening book moves are updated, and the child of that move is then found so it can be used to find the subtree for the opponents move later.

```
/* If the opening book still has moves to play, then add this move to the moves
played and increment the amount of moves taken on the openingBook instance. */
if(openingBook.movesTaken < 15) {
  mostRecentMoves[openingBook.movesTaken] = move;
  openingBook.movesTaken++;
}
/* If re-using the subtree, then the child node of the current root must be
navigated to In order to get the child node of that node once the opponent
takes their move */
TreeNode oldRootNode = rootNode;
rootNode = rootNode.getChild(game.getMove(0));
```

Finally, the move is returned, and the memory of the parent that is going to be discarded is removed.

```
oldRootNode.clearParentMemory(rootNode, oldRootNode.getChildren());
/* Collect all of the parent memory that has been cleared */
System.gc();

/* And return the move to the engine, so it can be converted into GTP */
return move;
```

Developing the Tree

Tree development is done in four stages [1]. First, the tree navigates to a leaf node, initializing a list of all visited nodes and adding the root node to the visited list.

```java
/* create a list of all visited nodes to backpropogate later */
List<TreeNode> visited = new LinkedList<TreeNode>();

/* set the current node to this node, and add it to the visited list */
TreeNode cur = this;
visited.add(this);

while (!cur.isLeaf()) {
  /* follow the highest uct value node, and add it to the visited list */
  cur = select(cur.getChildren());
  visited.add(cur);
}
```

Then, the leaf node is expanded.

```java
/* at the bottom of the tree expand the children for the node, including a pass
move but only if it hasn't been expanded before */
cur.expand();
```

And if the leaf node was non-terminal, a random child is selected. This is because first play urgency is being used, meaning that all of the newly expanded nodes will have an equal value.

```java
/* get the best child from the expanded nodes, even if it's just passing */
TreeNode newNode = null;
if(cur.children.size() > 0) {
  newNode = cur.selectRandom();
  visited.add(newNode);
} else {
  newNode = cur;
}
```

Once a random child node has been selected, the node is simulated.

```java
/* simulate from the node, and get the value from it */
double value = simulate(newNode);
```

And that value is backpropagated across all visited nodes.

```java
/* backpropogate the values of all visited nodes */
for (TreeNode node : visited) {

  /* type 0 for just uct updating */
    node.updateStats(0, value);

}
```

Finally, the RAVE statistics are updated for the parents subtree.

```java
/* based on the amaf map of the node that was just simulated */
updateStatsAmaf(newNode.amafMap, cur.getChildren(), value);
```

[1] The basic tree development algorithm was based on the http://mcts.ai/ algorithm.

Selection Policy

Nodes are selected across all of the current nodes children. First, the Rave Skip heuristic is checked. If it hasn't reached the predefined value, then the value is retrieved normally. If Rave Skip activates, then rave is disabled before retrieving the uct value.

```
/* if the amaf skip counter has reached the amount of times to wait until
skipping amaf */
if(raveSkipCounter < nodeRuleSet.raveSkip) {

  /* get the uct value using standard rules */
  uctValue = ucbTracker.ucbValue(this, c);

  /* and increment the counter */
  raveSkipCounter++;

} else if(raveSkipCounter == nodeRuleSet.raveSkip) {

  nodeRuleSet.rave = false;
  uctValue = ucbTracker.ucbValue(this, c);
  nodeRuleSet.rave = true;

  /* and set the counter to 0 */
  raveSkipCounter = 0;
}
```

Then, if the uct value is larger than the current best value, the current child is selected as the highest value node and the best value is updated.

```
/* if the uctvalue is larger than the best value */
if (uctValue > bestValue) {

  /*the selected node is that child */
    selected = c;

    /* and the best value is the current value */
    bestValue = uctValue;

}
```

The weighting between the MC value and the RAVE value is calculated using the weighting formula introduced in the theory section.

```
double weight = Math.sqrt(nodeRuleSet.raveWeight / (tn.nVisits[0] +
nodeRuleSet.raveWeight * tn.nVisits[0] + nodeRuleSet.raveWeight * tn.nVisits[0]
+ nodeRuleSet.raveWeight));

return (weight * calculateUctValue(1, parent, tn)) + ((1 - weight) *
calculateUctValue(0, parent, tn));
```
The UCT values are calculated using the UCB-TUNED algorithm.

The UCT values for MC and RAVE use the UCB-Tuned algorithm. If this is the node has not been visited before, then the first play urgency value is returned instead.

```
private double calculateUctValue(int type, TreeNode parent, TreeNode tn) {
  if(nodeRuleSet.firstPlayUrgency > 0 && tn.nVisits[0] == 0) {
    /* random values added to the end to break random ties */
    return nodeRuleSet.firstPlayUrgency + r.nextDouble() * epsilon;
  }
  double mean = tn.totValue[type] / (tn.nVisits[type] + epsilon);
  return mean + getBonus(tn) * (Math.sqrt((Math.log(parent.nVisits[type]+1)) /
(tn.nVisits[type] + epsilon )) * getVariance(type, mean, parent, tn)) +
r.nextDouble() * epsilon;
}
```

The variance is calculated using the recorded individual simulation rewards and the mean calculated earlier.

```
public double getVariance(int type, double mean, TreeNode parent, TreeNode tn)
{
  double value = 0.25;
  /* estimate of the variance + sqrt(2 log(n) / n)
   * variance computed by subtracting the mean from each value of the reward,
and then getting the mean
   * of those subtracted values squared*/
  double variance = 0;
  if(type==0) {
    for(double result : tn.simulationRewards) {
      variance = variance + Math.sqrt((result - mean));
    }
  } else {
    for(double result : tn.amafRewards) {
      variance = variance + Math.sqrt((result - mean));
    }
  }
  variance = variance / (tn.nVisits[type]);
  variance = variance + Math.sqrt(2*Math.log(parent.nVisits[type]+1) /
tn.nVisits[type]);
  if(variance < value)
    value = variance;
  /* if overall V is less than 0.25, return 0.25 */
  return value;
}
```

Expansion

For our expansion algorithm, we used the same dynamic tree technique as MoGo [Gel+06] and Pachi [Bau11]. If the amount of visits is less than a predefined amount, then the nodes aren't expanded. This stops expansion of nodes that will never be used. If the node does have enough visits, all of the empty points on the board are set to a list of positions, and then a move is attempted in each one. If the move cannot be played, then it is not added as a child.

```
if(nVisits[0] >= nodeRuleSet.pruneNodes) {

  /* get all of the empty points on the board */
  PositionList emptyPoints = currentGame.board.getEmptyPoints();
  int emptyPointsSize = emptyPoints.size();
  int childrenCounter = 0;
  /* for every empty point on the board */
    for (int i=0; i<emptyPointsSize; i++) {

      /* just try and play on it normally, ensuring the rules of the game are
followed */
      Game normalGame = currentGame.duplicate();
      boolean canPlayNormal = normalGame.play(emptyPoints.get(i));

      /* checking if it is possible to play that point, checking if we're
playing into an eye */
      if(canPlayNormal) {

        /* create a new child for that point */
        TreeNode newChild = new TreeNode(normalGame, emptyPoints.get(i),
raveSkipCounter, playerColor, nodeRuleSet, ucbTracker);

        /* and add it to the current nodes children */
        children.add(newChild);
        childrenCounter++;
      }

    }
```

Once all of the children have been expanded, it is then decided whether a pass move should be added to the children. In Go, a pass move should only be used if the game is even and near the end of the game, so the game can finish. This has been accommodated by checking how many empty spaces there are left. The pass move is only added if there are very few or there were no other children.

```
/* passing isn't something that should be done unless requesting an end to the
game meaning that there is a clear winner, one way or another. with this in
mind, passes are only added to the tree when there are few spaces on the board
left
 */
if(emptyPointsSize <currentGame.getSideSize()*2 || childrenCounter == 0) {
  /* add a pass move as well as playing on every allowable empty point */
    Game passGame = currentGame.duplicate();
    passGame.play(-1);
    TreeNode passChild = new TreeNode(passGame, -1, raveSkipCounter,
playerColor, nodeRuleSet, ucbTracker);
    /* and add it to the current nodes children */
    children.add(passChild);
}
```

Simulation

The simulation policy is split into two parts. The move selection for the simulation using a simulation player, and the collection of values. First of all, a simulation player is created using the rules set out in the configuration. Then, a duplicate of the game is created to play moves on.

```
/* create a duplicate of the game */
TreeNode simulateNode = tn;
Game simulateGame = simulateNode.currentGame;
SimulateGame duplicateGame = simulateGame.semiPrimitiveCopy();
```

```
/* initialize the game using the duplicate */
randomPlayer.startGame(duplicateGame, null);
```

Then, until the game is over a map of values representing where the simulation player placed stones is created. This is used in RAVE to determine which children of the subtree should be updated.

```
/* until the simulation has finished, play moves */
while(!duplicateGame.isOver()) {

  /* get the move, and play on the board */
  int move = randomPlayer.playMove();

  /* if the move isn't a pass */
  if(move > -1) {

    /* set the last move takens colour on the amaf map */
    tn.amafMap[move] = randomPlayer.game.getNextToPlay().inverse();
  }
}
```

Finally, the score is determined using Tromp Taylor Rules and translated into binary scoring, with a 10 point variance used to decide whether or not it is an even game.

```
/* get the score for the players color, from the perspective of black */
float score = duplicateGame.score(playerColor);
/* if we want the score from the perspective of white */

/* if we are using even scoring, and the scores end similarly */
if(score < 10 && score > -10) {
  return 0.5;
}
/* return 0 for loss, 1 for win */
if(score > 0)
  return 1;
return 0;
```

The moves chosen in the simulation are in the same fixed-order set out in Fuego, in order to maximize the possibility that a move will be played that uses our pattern matching algorithm. First the player checks if it should surrender.

```
/* get the last move of the game */
int lastMove = game.getMove(0);
/* if the opposing side has more than 30% of the board in captured pieces */
if(simulateMercyRule && movesTaken >= ((game.getSideSize() *
game.getSideSize()) -1) && game.mercy()) {
  /* just end the game */
  return -2;
}
```

Then, based on a certain set variance, the game attempts to match a pattern around the last move.

```
if(simulatePatterns && Math.random() > varySimPatterns) {
  /* if the opponents move matches any mogo patterns */

  if (game.matchPattern(lastMove)) {

    /* if it returns true, that means a random mogo pattern was picked and
played, so return that move*/
    return game.getMove(0);
  }
}
```

This pattern matching is based on the details set out by MoGo [Gel+06]. There are a set of 3x3 patterns that are flipped, rotated, and mirrored. If all of these options fail, then the stones are swapped. This is the only part of the Game implementation that I changed for my development process. The algorithm works by creating a series of 3x3 grids based on the positions surrounding the last move. Then, all of those grids are flipped, rotated and mirrored, then tested to match against the MoGo patterns. If this doesn't work, then the MoGo pattern colours are swapped.

```
public boolean matchPattern(int z) {
  /* Get all of the positions surrounding the current position */
  int[][] surroundingPositions = populateSurroundingPositions(z);
  /* For every surrounding position */
  for(int i=0;i<3;i++) {
    for(int j=0;j<3;j++) {
      /* If that position can be played */
      if(play(surroundingPositions[i][j])) {
        /* Collect all of the grids with each position of the surrounding
positions at the center */
        int[][] patternMatchGrid =
populateSurroundingPositions(surroundingPositions[i][j]);
        /* And create an array of all the pattern types, flipping, rotating,
mirroring */
        int[][][] patternTypes = populatePatternTypes(patternMatchGrid);
        /* For every pattern type */
        for(int k=0;k<patternTypes.length;k++) {
          /* That matches the patterns using a simple loop */
          if(matchesPatterns(patternTypes[k])) {
            /* Return true */
            return true;
          }

        }
        /* If none match, then swap all the pattern colours and try to match the
default option */
        swapPatternColors();
        if(matchesPatterns(patternTypes[0])) {
          swapPatternColors();
          return true;
        }
        swapPatternColors();
      }
    }
  }
  return false;
}
```

The flips are done using simple loops. These are the same algorithms used in the opening book.

```
public static int[][] rotate(int[][] grid) {
    int[][] out = new int[grid.length][grid.length];

    for (int i = 0; i < grid.length; ++i) {
        for (int j = 0; j < grid.length; ++j) {
         out[i][j] = grid[grid.length - j - 1][i];
        }
    }

    return out;
}

public static int[][] horizontalFlip(int[][] grid) {
    int[][] out = new int[grid.length][grid.length];
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid.length; j++) {
            out[i][grid.length - j - 1] = grid[i][j];
        }
    }
    return out;
}

public static int[][] verticalFlip(int[][] grid) {
    int[][] out = new int[grid.length][grid.length];
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid.length; j++) {
            out[grid.length - j - 1][j] = grid[i][j];
        }
    }
    return out;
}
```

RAVE

Stats are stored as the amount of simulations visiting the node and the amount of reward given to the node. The individual rewards are recorded in order to inform UCB-Tuned.

```
/* update the stats for this node */
private void updateStats(int type, double value) {

  /* for the uct value or amaf value, dependent on the type input */
    if(type == 0) {
      simulationRewards.add(value);
    } else {
      amafRewards.add(value);
    }
  if(type == 0) {
    nVisits[1]++;
  }
  nVisits[type]++;
    totValue[type] += value;
}
```

The method to update amaf stats uses type 1. It works through the subtree recursively, checking if the move being played matches the one on the amafMap recorded during the most recent simulation. If it does, then the stats are updated, and if the node that was or was not updated has children, then those children are updated too.

```
/* share the updated values across the subtree of the move too */
private void updateStatsAmaf(Color[] amafMap, List<TreeNode> children, double
simulationResult) {

  /* for every child of this node */
  for (TreeNode c : children) {
      Color inverseColor = c.currentGame.getNextToPlay().inverse();
      c.currentGame.getNextToPlay().inverse();
      /* if the move being played for this node matches the move being played
in the simulation */
      if(inverseColor == amafMap[c.currentGame.getMove(0)]) {

        /* update the total value of that node with the simulation result */
        c.updateStats(1, simulationResult);
      }

    /* if there is more subtree to explore */
    if(c.children.size() > 0) {

      /* recursively iterate through the whole subtree */
      updateStatsAmaf(amafMap, c.getChildren(), simulationResult);
    }
  }
}
```

Experiments

These experiments were conducted using the gogui-twogtp feature, which allows the user to play two AI's against each other. All of these experiments are conducted using self-play on the 9x9 board. The win% for black, variance of that result, games, iterations and the type of tree development are listed in the table. To begin with, we analysed black VS white.

Testing	Black win%	Variance	Games	Iterations	Time/Iterations
Black VS White	48%	7.10%	100	1000	Iterations

UCB1, SimpleUCB, UCB-Tuned,

Testing	Black win%	Variance	Games	Iterations	Time/Iterations
ucbVSucbTuned	14%	4.90%	50	1000	Iterations
UCB1vsUCBTunedWeighted1Ex	20%	12.60%	10	1000	Iterations
simpleUCBvsUCB1	0%	0%	10	1000	Iterations

UCT, AMAF, RAVE, MC-RAVE, RAVE Skip

Testing	Black win%	Variance	Games	Iterations	Time/Iterations
UCTvsRAVEcheck	33%	7.30%	42	1000	Iterations
UCTvsWeightedRave	40%	15.50%	10	1600	Time
UCTvsRAVEcheck	33%	7.30%	42	1000	Iterations
1000ravevs1500rave	55%	5%	100	1000	Iterations
noRaveSkipVSraveSkip	44%	7%	50	1000	Iterations

Binary Scoring, Even Scoring

Testing	Black win%	Variance	Games	Iterations	Time/Iterations
binaryVSnonBinary	100%	0%	20	1600	Iterations
noEvenScoringVSevenScoring	30.80%	9.10%	26	1000	Iterations
5EvenScoringVS15EvenScoring	60%	6.90%	50	1000	Iterations
15EvenScoringVS10EvenScoring	44%	7%	50	1000	Iterations

Opening Book

Testing	Black win%	Variance	Games	Iterations	Time/Iterations
OpeningBookVSNoOpeningBook	61%	4.90%	100	1000	Iterations

Exploration Weight

Testing	Black win%	Variance	Games	Iterations	Time/Iterations
noExploreWeightVSExploreWeightUCT	70%	8.40%	30	600	Iterations

exploreWeightVSnoExploreWeight	88%	4.60%	50	1000	Iterations
exploreWeightVSnoExploreWeight	86%	4.90%	50	1000	Iterations
noExploreWeightVSExploreWeight	11.80%	7.80%	17	2300	Iterations

Bonus First Play Urgency, First Play Urgency

Testing	Black win%	Variance	Games	Iterations	Time/Iterations
0.1FPUvs1FPU	28.60%	9.90%	21	1600	Iterations
noFPUVSFPU	0%	0%	35	1600	Iterations
noBonusFPUVSbonusFPU	72%	9%	25	600	Iterations
0.9FPUVS10FPU	54.80%	7.70%	42	1000	Iterations
0.9FPuvs0.8FPU	55%	5%	100	1000	Iterations

Bonus Patterns, Bonus Avoid Eyes

Testing	Black win%	Variance	Games	Iterations	Time/Iterations
5000bonusPatternVS50BP	38.50%	13.50%	13	1600	Iterations
MaxRobust-500VSMaxRobust5000	67.10%	6%	70	1000	Iterations
50PatternVS500Pattern	60%	6.90%	50	1000	Iterations
0VS137Pattern	50%	7.10%	50	1000	Iterations
0PatternVS50Pattern	50%	6.90%	50	1000	Iterations
137PatternVS200pattern	48%	7.10%	50	1000	Iterations
17patternvs50pattern	46%	7%	50	1000	Iterations

Simulate Avoid Eyes, Simulate Patterns, Simulate Mercy Rule

Testing	Black win%	Variance	Games	Iterations	Time/Iterations
simEyesVSnoSimEyes	67.10%	6%	70	1000	Iterations
simPatternsVsnoSimPatterns	60%	6.90%	50	1000	Iterations
simMercyRuleVsnoSimMercyRule	50%	7.10%	50	1000	Iterations

Vary Sim Eyes, Vary Sim Patterns

Testing	Black win%	Variance	Games	Iterations	Time/Iterations
varyEyes0.01VSvaryEyes0.1	34%	6.70%	50	1000	Iterations
0.3VaryVS0.5Vary	58%	4.90%	100	1000	Iterations
0.3VaryVS0.2Vary	42.90%	18.70%	7	1000	Iterations
0.01varyEyesVS0.05varyEyes	44%	5%	100	1000	Iterations

pickRobust, pickMax, pickSecure, pickMaxRobust,

Testing	Black win%	Variance	Games	Iterations	Time/Iterations
MaxMeanVSmostVisited	16.70%	15.20%	6	N/A	Iterations
MaxChildVSMaxRobust	26.70%	11.40%	15	1000	Iterations

PolicyVSMaxRobust	13.30%	8.80%	15	1000	Iterations
MostSimulatedVSMaxRobustBoth5k	76.90%	5.80%	52	1000	Iterations

clearMemory, pruneNodes, developPruning,

Testing	**Black win%**	**Variance**	**Games**	**Iterations**	**Time/Iterations**
0DynamicTreeVS2DynamicTree	48%	7.50%	37	1000	Iterations
2dynamicVS4dynamic	44%	9.90%	25	1000	Iterations

Future Work

I know that this implementation can compete competitively with just a few more modifications. In the future I am planning to implement features found in Fuego [Enz+10] in order to compete in Computer Go tournaments, as well as attempting to integrate some personally observed domain knowledge. With the release of this work open-source, anyone could develop this project into a full-fledged commercial software.

Multi-Threading

With multi-threading, Fuego is capable of running 12,000 simulations a second. [Enz+10]. Without multi-threading, my software is capable of running 1000. Once the implementation has been adjusted so that it fits the multi-threaded architecture, I believe it will become tenfold stronger.

Simulation Policy

The weakest part of the current software is its simulation policy. Because it does not feature heuristics for detecting Atari, the simulations do not contain knowledge of how to put opponents into Atari, or avoid it. This is a massive tactical flaw. Even though this is fairly trivial to implement compared to the complexity of MoGo patterns, I believe that it will result in a monumental increase in playing strength. The software should no longer make game-changing tactical errors based on capturing or self-Atari once this change has been developed.

Pondering

Currently, when the opponent is thinking, the tree rests still. In Fuego [Enz+10] the tree is further developed while the opponent waits to take their move. I believe that this would greatly increase my programs playing strength, and I am excited to see the possible results.

Conclusion

The implementation was developed in Java, and tested using a 9x9 board. Even though it is completely capable of playing at any board size at low-amateur level, without multi-threading time optimisation or complex domain specific problem solving, the software detailed in this report is not at the level of commercial software. Through extensive testing and coverage of key modern MCTS features, this implementation gives researchers the opportunity to use the groundwork here to develop a competitive application. Before this, a statistical analysis across every feature of an implementation had not been released to the public. This information should be invaluable for researchers looking for information on the effectiveness of different MCTS features, or looking to fine-tune the values for their implementation. Covering the main developments in the leading Go implementations, this report and implementation lays the groundwork for further developments in the Java Go world.

References

[Bru93] Bernd Bruegmann. “Gobble: Monte Carlo Go”. 1993. url: http://www.cgl.ucsf.edu/go/Programs/Gobble.html.

[Bou07] Bouzy, Bruno. "Old-fashioned Computer Go vs Monte-Carlo Go". IEEE Symposium on Computational Intelligence and Games, April 1–5, 2007, Hilton Hawaiian Village, Honolulu, Hawaii. url: http://ewh.ieee.org/cmte/cis/mtsc/ieeecis/tutorial2007/Bruno_Bouzy_2007.pdf.

[BH03] Bruno Bouzy and Bernard Helmstetter. “Monte Carlo Go Developments”. In: Advances in Computer Games conference (ACG-10), Graz 2003. Ed. by Ernst. Kluwer, 2003, pp. 159/174. url: http://www.math-info.univparis5.fr/~bouzy/publications/bouzy-helmstetter.pdf.

[Bau11] Peter Baudis. “MCTS with Information Sharing”. 2011. url: http://pasky.or.cz/go/prace.pdf.

[Iee12] Cameron Browne, Member, IEEE, Edward Powley, Member, IEEE, Daniel Whitehouse, Member, IEEE, Simon Lucas, Senior Member, IEEE, Peter I. Cowling, Member, IEEE, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton. “A Survey of Monte Carlo Tree Search Methods”. IEEE Transactions on Computational Intelligence and AI in Games, Vol 4, No.1, March 2012. url: http://ccg.doc.gold.ac.uk/reports/browne_tciaig12_1.pdf

[LR85] * T. L. Lai and H. Robbins, “Asymptotically Efficient Adaptive Allocation Rules,” Adv. Appl. Math., vol. 6, pp. 4–22, 1985

[A95] R. Agrawal, “Sample mean based index policies with zero (log n) regret for the multi-armed bandit problem,” Adv. Appl. Prob., vol. 27, no. 4, pp. 1054–1078, 1995.

[ABF02] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-time Analysis of the Multiarmed Bandit Problem,” Mach. Learn., vol. 47, no. 2, pp. 235–256, 2002.

[KS06] L. Kocsis and Csaba Szepesvari. Bandit based monte-carlo planning. In Proceedings of ECML, pages 282–293, 2006.

[Rob52] Herbert Robbins. “Some aspects of the sequential design of experiments”. In: Bulletin of the American Mathematics Society. Vol. 58.1952, pp. 527/535.

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

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

[Boo90] Mark Boon. “A Pattern Matcher for Goliath”. In: Computer Go 13 (1990), pp. 12/23.

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

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

[KS06] Levente Kocsis and Csaba Szepesvári. “Bandit Based Monte-Carlo Planning”. In: Machine Learning: ECML 2006. Ed. by Johannes Fürn-kranz, Tobias Sche”er, and Myra Spiliopoulou. Vol.

4212. Lecture Notes in Computer Science. 10.1007/11871842.29. Springer Berlin / Heidelberg, 2006, pp. 282/293.