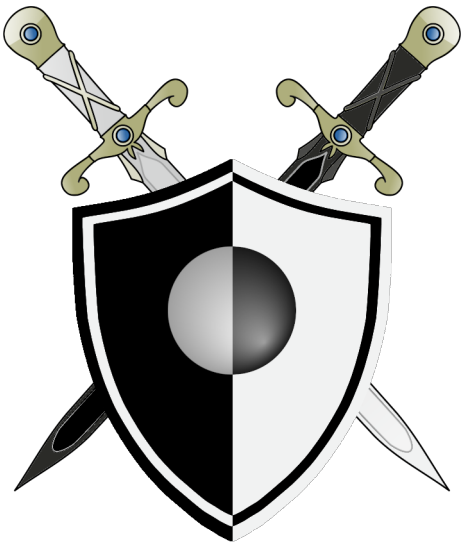
Final Report

by

GoSlayer Team

January 23, 2020



Team members:

Role	Name	Section	B.N.
Leader	Ahmed Khaled	1	3
Co-leader	Mai Mahmoud	2	25
Research Team	Ibrahim Mahmoud	1	1
Design and Tmplementation Team	Ahmed Ibrahim	1	2
	Ahmed Essam	1	6
	Kareem Emad	2	10
	Mohamed AbdelKereem	2	20
	Yasmeen Ahmed	2	32
Integration Team	Aisha Mousa	1	32
	Abdelrahman Mohamed	2	1
GUI - Interface Team	Ahmed Salah	1	5
	Hady Maher	2	29
Communication and Testing Team	Reem Ashraf	1	24
	Maryam AboulFetouh	2	24
Documentation team	Sara Maher	1	28
	Sayed Kotb	1	30

Abstract

Solving Go with an `AI Agent` was thought to be a very hard problem, until a few years ago. Here, we present our own agent `GoSlayer`, that uses Monte Carlo tree search (MCTS) and Last Good Reply With Forgetting (LGRF-1), which offers an improvement in performance over vanilla MCTS[4].

Acknowledgments

We would like to thank Prof. Nevin Darwish and Eng. Yahia Zakaria for providing invaluable guidance, advice and a good competitive environment.

Contents

Chapter 1

Introduction

Go is a two-player, Chinese board game, that was invented more than $2,500$ years ago. Each player places pieces, called stones, on the intersections on a board grid aiming at surrounding more territory than the opponent. One player uses black stones and the other uses white ones (The player with the white stones plays 2^{nd} and for that they take 6.5 extra points). The player can also capture the opponent's stones by surrounding them. The game ends when both player skip their turn to play, or when one of the players resignates. The winner is determined by counting the stones each player has captured and their surrounded territory. Although the rules are simple, Go is a very complex game, due to the number of alternatives to consider per move. The lower bound on the number of legal board positions in Go has been estimated to be 2×10^{170}.

Our project deals with a 19×19 Go board. We developed an `AI Agent` to play the game, as well as the `GUI` and server-client communication. The game agent supports 2 modes of operations, either `AI Agent` vs. `AI agent` or `AI Agent` vs. `a human`

`player`. We apply a modified version of the Chinese Go rules, where captured stones are not removed. Because Go is a zero-sum, sequential, partially-observable and deterministic strategy game, with a branching factor that starts at 361, so using regular `alpha-beta` search is incredibly inefficient. The following chapters go through: a review of the related literature, a walk through the algorithms and heuristics implemented, the different experiments tried and future work.

Chapter 2

Literature Review

2.1 Temporal-difference search in computer GO [3]

2.1.1 Markov decision process (MDP)

A Markov decision process typically consists of:

- A set of states **S**
- A set of actions **A**, that govern the transition between each state **s** to another state **s'**
- A reward function

$$R^a_{ss'} = E[r_{t+1}|s_t = s, a_t = a, s_{t+1} = s']$$

that specifies the expected reward for a given state transition.

- A return function

$$R_t = \sum_{k=t}^{T} r_k$$

  which is the total reward accumulated in an episode from time **t** until reaching the terminal state at time **T**

- A policy

$$\prod(s, a) = P(a_t = a | s_t = s)$$

  that maps each state **s** to a probability distribution over all the possible actions.

- A value function

$$V^{\pi}(s) = E_{\pi}[R_t | s_t = s]$$

  which is the expected return from state **s**, following the policy.

- An action value function

$$Q^{\pi}(s, a) = E_{\pi}[R_t | s_t = s, a_t = a] = V^{\pi}(s'),$$

  which represents the expected return after selecting an action **a**

2.1.2 Temporal-difference learning with local shape features

Temporal-difference learning (TD learning) is a model-free method for policy evaluation that bootstraps the value function from subsequent estimates of the value function. We first need to find a suitable value function in order to be able to bootstrap.

Value function approximation

In the lookup table, the value function has a distinct value $\mathbf{V(s)}$ for each state $\mathbf{s}$. However, that is not practical in large environments,like in the game of `Go`, so the value function must be approximated.
A common and successful approximation approach is to use a feature vector $\boldsymbol{\phi}(\mathbf{s})$ to represent a state $\mathbf{s}$, and approximate the value function $\mathbf{V(s)}$ by a linear combination of the features $\boldsymbol{\phi}(\mathbf{s})$ and weights $\boldsymbol{\theta}$. In this case,

$$V(s) = \phi(s) \cdot \theta$$

. Now we need to find some good features to fill in $\boldsymbol{\phi}(\mathbf{s})$.

Local shape features

Local shape features are a simple representation of shape knowledge in the game of `Go`. It is a way of representing the set of local board configuration within all square regions up to size $m \times m$.

The state in our game of `Go`, $s \in \{\ , \circ, \bullet\}^{19\times 19}$ [1], consists of a state variable in each intersection of size 19×19 board. However, we can't extract good knowledge from the raw state. That's why we define a local shape $\mathbf{l}$ to be a specific configuration of state variables within some size $k \times k$ of the board, where $l \in \{\ , \circ, \bullet\}^{k\times k}$.

We exhaustively enumerate all possible shapes that can exist within all square regions up to size $k \leq m$.

[1] meaning whether s contains no stone, a white stone, or a black stone respectively

The local shape feature $\boldsymbol{\phi_i}(\mathbf{s})$ has value 1 in state $\mathbf{s}$, if the board matches exactly the $\mathbf{i^{th}}$ local shape $\mathbf{l_i}$ and value 0 otherwise. The local shape features are then combined into one large feature vector $\boldsymbol{\phi}(\mathbf{s})$. This feature vector is very sparse, so calculating its linear combination is not computationally expensive.

There's also the concept of weight sharing that we use to exploit symmetries of the `Go` board. The idea behind weight sharing is that if some arbitrary local shape is good and guarantees a high success probability for me, it would make sense for that same local shape to be bad if my opponent has it. We do this by defining all local shapes that are rotationally or reflectionally symmetric (even if it is in another location on the board) to be in the same class and hence, have the same weight. However, each local shape that belongs to that class but is inverted such that all the stones are flipped to the opposite color is given the same weight but in the negative direction (negative weight sharing).

2.1.3 Learning algorithm

Our objective is to win the game of `Go`. This can be represented through a binary reward function which gives a reward of $\mathbf{r = 1}$ if we win and $\mathbf{r = 0}$ if our opponent wins, with no intermediate rewards. We approximate the value function by a logistic-linear combination of local shape features.

$$V(s) = \sigma(\phi(s) \cdot \theta), \ where\ \sigma(x) = \frac{1}{1 + e^{-x}}$$

We measure the TD-error between the current state $\mathbf{V(s_t)}$ and the value after both player and opponent have made a move $\mathbf{V(s_t + 2)}$. In order to update the weights

in a way that reflects the learning nature of the algorithm, we use the following equation,

$$\Delta\theta = \alpha \frac{\phi(s_t)}{\left\|\phi(s_t)\right\|^2}(V(s_{t+2}) - V(s_t))$$

where $\boldsymbol{\alpha}$ is the learning rate.

Now that we have defined a value function and a way to improve it, we need to define the policy by which we are going to choose each action $\boldsymbol{\alpha}$ from each state **s**. The policy we are going to be using is $\boldsymbol{\epsilon}$ **− greedy algorithm** which is really a simple policy. It chooses a random action from all possible actions from the current state with a probability of $\boldsymbol{\epsilon}$, and it chooses the action **a** that maximizes the value function $\mathbf{V(s)}$ with a probability of $\mathbf{1} - \boldsymbol{\epsilon}$. This policy combines both the exploration and exploitation parts of the algorithm, and each can be controlled by changing the value of $\boldsymbol{\epsilon}$.

Chapter 3

Applied Methodology

In this chapter, we go through the methodology and implemented algorithms for the Go game agent.

3.1 The Game Agent

We implemented a Monte Carlo Tree Search (`MCTS`) based `AI agent`. `MCTS` is divided into these four phases:

1. **Selection**: start from root (current game state) and successively select child nodes until a leaf node (L) is reached.

2. **Expansion**: if L is not the end of the game, create one (or more) child nodes and choose node (C) from one of them. Child nodes are any valid moves from the game position defined by (L).

3. **Playout**: may be as simple as choosing uniform random moves until the game is decided (for example, in chess: the game is won, lost or drawn).

4. **Backpropagation**: use the result of the playout to update information in the nodes on the path from (C) to root.

We first implemented `vanilla MCTS` with Upper Confidence Bound 1 applied to trees (`UCT`) [1] selection policy and random playout policy. This didn't provide a good performance, so we implemented the following heuristics for the playout policy:

- Defend groups in `atari` in the neighborhood of last play.
- Attack enemy groups in atari in the neighborhood of last play.
- Decrease the liberties of a nearby enemy group while increasing the liberty count of a friend group.
- Attack enemy groups in atari if any exists in the whole board.
- Play randomly.

These are subsets of the heuristics that `Fuego`[2], a go playing agent that uses `MCTS`, uses.

The performance still wasn't great, so we implemented the Last Good Reply-1 (`LGR-1`), Last Good Reply-2 (`LGR-2`) and Last Good Reply With Forgetting (`LGRF-1`)[4] heuristics for the playout phase, and the performance of the agent improved significantly after using (`LGRF-1`), so we integrated it.

[1]
$$UCT = \frac{\omega_i}{n_i} + c \times \sqrt{\frac{\ln N_i}{n_i}}$$

ω_i: stands for the number of wins for the node considered after the i^{th} move.
n_i: stands for the number of simulations for the node considered after the i^{th} move.
N_i: stands for the total number of simulations after the i^{th} move run by the parent node of the one considered.
c: is the exploration parameter (theoretically equals $\sqrt{2}$, in practice usually chosen empirically).

[2] `http://fuego.sourceforge.net/`

We also implemented Rapid Action Value Estimation (`RAVE`)[2][3] policy for the selection phase, replacing the normal `UCT` policy, which again improved the performance. The agent was extremely slow at this stage, so it couldn't do the required simulations in a reasonable time, so we:

- Changed the engine to cache more data.
- Applied some optimizations on the heuristics, which resulted in 4× improvement on the number of simulations done in the same time.
- Reused the relevant nodes by neatly organizing their data, instead of creating the tree search from scratch every move.
- Implemented a few `Matching Heuristics` which enable the agent to easily recognize some patterns and directly apply a pre-computed solution.

3.2 The Graphical User Interface (GUI)

We implemented the visual module of the game engine as follows:

- Implemented the interface in 3 separate pages, namely, start, game and winner screens.
- Implemented the interface for 4 different game modes, namely:
  1. `Agent` vs `Agent`.
  2. `Human` vs `Agent`.
  3. `Human` vs `Human`.

[3]See Appendix A

4. Agent vs Remote.

- Added sound effects for game moves.
- Added players' names, timers and game statistics.
- Created a promo video for the game engine using Adobe After Effects [4].

3.3 Communication between the Engine and the GUI

We implemented Go text protocol (GTP) interface between GUI and Engine as defined by GTP version 2 draft[1]. The implementation is as follows:

- The GUI part is a JavaScript implementation of the protocol that contains a JavaScript WebSocket interface to send messages between the engine and the GUI using the native JavaScript WebSockets library.
- Engine part is divided into 2 parts:
  - **Server**: a Websocketpp [5] server that can send GTP requests, receive game configuration from GUI and send final game winner/score.
  - **Human Player**: an Agent that can request moves from GUI. Hmuman player had a problem with busy waiting till it gets the server response so we made it block and sleep then the server would awaken it when it receives the response through a handle that is set during the agent's construction.

[4] https://www.adobe.com/mena_en/products/aftereffects.html

[5] WebSocket++ is an open source header only C++ library that implements RFC6455 (The WebSocket Protocol). It allows integrating WebSocket client and server functionality into C++ programs. It uses interchangable network transport modules including one based on C++ iostreams and one based on Boost Asio.https://github.com/zaphoyd/websocketpp

We also implemented a GTP Command Line Interface (CLI) for the engine to provide a GTP game interface and to be able to play with other engines using a Go board editor like Sabaki[6].

3.4 Communication between the Engine and the Game server

We implemented the module responsible for communication between the engine and the game server as follows:

- Implemented a Game Manager that is responsible for creating/destroying a Game Runner instance when needed in a separate thread as well as running the client-side module.
- Implemented a client-side module responsible for connecting to, reconnecting to and communicating with the Game Server through event handlers using Websocketpp[7].
- Implemented JSON encoder and decoder module to parse received messages from JSON format and encode outgoing messages into JSON format using RapidJSON[8].
- Implemented a Game Runner module responsible for creating and initializing a game instance and assigning both a Local and Remote agent to it.
- Implemented a Remote Agent that when asked to generate a move blocks until the received move from the Game Server is passed to it in an atomic

[6]https://sabaki.yichuanshen.de/
[7]https://github.com/zaphoyd/websocketpp
[8]https://rapidjson.org/

`ActionMessage` struct.

- Implemented a `Local Agent` that generates a move using the wrapped `Monte Carlo Tree Search Agent` then proceeds to send it to the `Game Server`.

3.5 Testing the Game Engine

We designed and executed tests to validate the work done by other teams and to make sure that it conforms to the specifications set by the acceptance criteria. Testing activities included:

- Setting the acceptance criteria for each part of the system.
- Designing a test suite using a testing framework called `Catch 2`[9].
- Executing the test suite and reporting results.
- Retesting any fixes for reported defects by performing regression testing.

[9]https://github.com/catchorg/Catch2

Chapter 4

Experiments and Future work

In this section we highlight the methods we experimented with and the results we obtained. While building the game agent, we tried the following:

- We tried using `vanilla MCTS` and as a result the game agent managed to non-randomly play against human players but the performance was not satisfactory.
- We then tried combining `RAVE` with `vanilla MCTS` and it immediately showed better performance and dominated the `non-RAVE` agent in 5 consecutive games.
- Afterwards, we tried using `LGR` with the previous configiration and ran a couple of experiments at which the `LGR` agent won and lost equally.
- We used `GNU Go` [1] to test how good is our agent, but we lost every time.

We considered implementing other techniques to increase the agent performance such as:

[1] GNU Go is a free software program by the Free Software Foundation that plays Go.

- We considered implementing `Zobrist hashing` [2] to encode previously seen states so that we can use its statistics when encountered instead of starting from scratch, but the idea was discarded due to its development and space cost.

- We considered implementing `MCTS Initialization Heuristics` that biases the `MCTS`.

- We considered running `MCTS` in parallel.

- We considered creating a time manager to allocate time for each move dynamically based on the expected number of moves.

[2] `Zobrist hashing` is a hash function construction used in computer programs that play abstract board games, such as chess and Go, to implement transposition tables

Chapter 5

Conclusion

Based on our experiments, we decided to go with implementing `MCTS` with `LGRF-1` as it gave the best results on average. Still, there are many promising techniques and optimizations that we wanted to try, but unfortunately, we didn't have the time. We added them in the future work section and we hope to be able to implement these techniques and see how they will affect the performance in the near future.

Appendix A

Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go

The paper [2] adds 2 extensions to the `vanilla MCTS`:

1. The Rapid Action Value Estimation (`RAVE`) algorithm, shares the value of actions across each sub-tree of the search tree. RAVE forms a very fast and rough estimate of the action value; whereas normal Monte-Carlo is slower but more accurate. It uses the all-moves-as-first heuristic from each node of the search tree, to estimate the value of each action. RAVE provides a simple way to share knowledge between related nodes in the search tree, resulting in a rapid, but biased estimate of the action values.

2. The second extension, heuristic Monte-Carlo tree search, uses a heuristic function to initialise the values of new positions in the search tree. We demonstrate that an effective heuristic function can be learnt by temporal-difference learn-

ing and self-play; however, in general any heuristic can be provided to the algorithm.

We will only talk about `RAVE`.

A.1 All-Moves-As-First

We define the `AMAF` value function $\tilde{Q}^{\pi}(s,a)$ to be the expected outcome z from state s, when following joint policy π for both players, given that action a was selected at some subsequent turn,

$$\tilde{Q}^{\pi}(s,a) = \mathbb{E}_{\pi}[z|s_t = s, \exists u \geq t \text{ s.t. } a_u = a]$$

The level of bias, $\tilde{B}(s,a)$, depends on the particular state s and action a,

$$\tilde{Q}^{\pi}(s,a) = Q^{\pi}(s,a) + \tilde{B}(s,a)$$

The all-moves-as first value $\tilde{Q}^{(}s,a)$ is the mean outcome of all simulations in which action a is selected at any turn after s is encountered,

$$\tilde{Q}(s,a) = \frac{1}{\tilde{N}(s,a)} \sum_{i=1}^{N(s)} \tilde{\mathbb{I}}_i(s,a) z_i$$

, where $\tilde{\mathbb{I}}_i(s,a)z_i$ is an indicator function returning 1 if state s was encountered at any step t of the i^{th} simulation, and action a was selected at any step $u \geq t$, or 0 otherwise; and $\tilde{N}(s,a) = \sum_{i=1}^{N(s)} \tilde{\mathbb{I}}_i(s,a)$, which counts the total number of simulations used to estimate the `AMAF` value. In order to select the best move with reasonable accuracy, Monte-Carlo simulation requires many simulations from every candidate

move. The AMAF heuristic provides more information: every move will typically have been tried on several occasions, after just a handful of simulations. If the value of a move is unaffected, this can result in a much faster rough estimate of the value.

A.2 RAVE

The RAVE algorithm combines MCTS with the AMAF heuristic. RAVE generalises over sub-trees, with the assumption that the value of action a in state s will be similar from all states within sub-tree $\tau(s)$. Thus, the value of a is estimated from all simulations starting from s, regardless of exactly when a is played, and the action with the maximum AMAF value in sub-tree $\tau(s_t)$ is selected, $a_t = \underset{b}{\operatorname{argmin}}\, \tilde{Q}(s, a)$.

Appendix B

Monte-Carlo Tree Search Enhancements for Havannah

The Last-Good-Reply[1] (LGR) policy is an enhancement used during the play-out step of the MCTS algorithm. Rather than applying the default simulation strategy, moves are chosen according to the last good replies to previous moves, based on the results from previous play-outs.
There are several variants of LGR:

1. The first one is LGR-1, where each of the winner's moves made during the play-out step is stored as the last good reply to the previous move. During the play-out step of future iterations of the MCTS algorithm, whenever possible, the last good reply to the previous move is always played instead of a random move.

2. The second variant is LGR-2, which stores the last good reply to the previous

[1]The paper contains many optimization policies, we will only be talking about the LGR policy, which we implemented.

two moves, so the last good replies are based on more relevant samples. During the playout, the last good reply to the previous two moves is always played whenever possible. If there is no last good reply known for the previous two moves, `LGR-1` is tried instead. Therefore, LGR-2 also stores tables for `LGR-1` replies.

3. A third variant is `LGR-1` with forgetting, or simply `LGRF-1`. It works exactly like `LGR-1`, but now the loser's last good replies are deleted if they were played during the play-out.

4. The fourth and last variant is `LGR-2` with forgetting, or simply `LGRF-2`. Thus, the last good reply to the previous two moves is stored and after each play-out, the last good replies of the losing player are deleted if they have been played.

Bibliography

[1] G Farnebäck. Specification of the go text protocol, version 2, draft 2, october 2002. *URL http://www. lysator. liu. se/ gunnar/gtp. Ultimo acceso: setiembre de*, 2005.

[2] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[3] David Silver, Richard S. Sutton, and Martin Müller. Temporal-difference search in computer go. *Machine Learning*, 87(2):183–219, May 2012.

[4] Jan A Stankiewicz, Mark HM Winands, and Jos WHM Uiterwijk. Monte-carlo tree search enhancements for havannah. In *Advances in Computer Games*, pages 60–71. Springer, 2011.