Mastering the game of Go with deep neural networks and tree search

Understanding Monte Carlo Tree Search (MCTS)

Naren Sundaravaradan

Unifie

Papers We Love, Bengaluru, 2019

What's the paper about?

- Introduces an algorithm that plays the game of Go at the highest level.

What's the paper about?

- Introduces an algorithm that plays the game of Go at the highest level.
- The algorithm introduced is easily generalizable to other games.

What's the paper about?

- Introduces an algorithm that plays the game of Go at the highest level.
- The algorithm introduced is easily generalizable to other games.
- It makes use of the existing Monte Carlo Tree Search (MCTS) algorithm and enhances it using heuristics learnt through deep learning.

What we will cover

- Describe a simpler two-player game to help illustrate the paper better

What we will cover

- Describe a simpler two-player game to help illustrate the paper better
- Derive and explore the optimal gameplay algorithm (minimax).

What we will cover

- Describe a simpler two-player game to help illustrate the paper better
- Derive and explore the optimal gameplay algorithm (minimax).
- Derive and explore vanilla MCTS

What we will cover

- Describe a simpler two-player game to help illustrate the paper better
- Derive and explore the optimal gameplay algorithm (minimax).
- Derive and explore vanilla MCTS
- Understand the Exploration-Exploitation tradeoff using Upper Confidence Trees (UCT)

What we will cover

- Describe a simpler two-player game to help illustrate the paper better
- Derive and explore the optimal gameplay algorithm (minimax).
- Derive and explore vanilla MCTS
- Understand the Exploration-Exploitation tradeoff using Upper Confidence Trees (UCT)
- Describe MCTS as used in the paper

Greedy-Coins Game

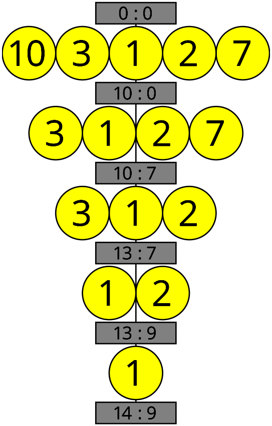
- Found in the 2012 Midsouth Programming Contest http://ccsc-ms.org/2012_Problems/index.html

Greedy-Coins Game

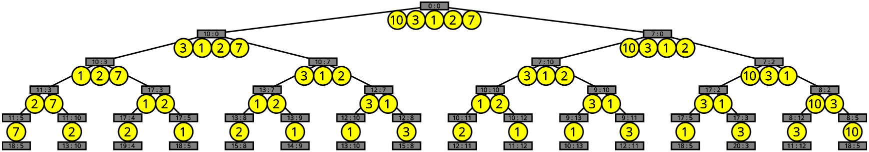
- Found in the 2012 Midsouth Programming Contest `http://ccsc-ms.org/2012_Problems/index.html`
- The aim is to collect coins worth more than that collected by the opponent

Greedy-Coins Game

- Found in the 2012 Midsouth Programming Contest `http://ccsc-ms.org/2012_Problems/index.html`
- The aim is to collect coins worth more than that collected by the opponent
- Only one coin is picked on each turn from either the left or the right



The Game Tree



Minimax

- Is an algorithm to play *optimally*. It picks the move that achieves the best score *assuming* the opponent also plays *optimally*.

Minimax

- Is an algorithm to play *optimally*. It picks the move that achieves the best score *assuming* the opponent also plays *optimally*.
- Mathematically, if s is the current state, then the best move $a \in A(s)$ is the one that maximizes

Minimax

- Is an algorithm to play *optimally*. It picks the move that achieves the best score *assuming* the opponent also plays *optimally*.
- Mathematically, if s is the current state, then the best move $a \in A(s)$ is the one that maximizes

$$\text{best score}|s = \max_{a_1 \in A}(\text{ best acheivable score } |a_1, s)$$

Minimax

- Is an algorithm to play *optimally*. It picks the move that achieves the best score *assuming* the opponent also plays *optimally*.
- Mathematically, if s is the current state, then the best move $a \in A(s)$ is the one that maximizes

$$\text{best score}|s = \max_{a_1 \in A}(\text{ best acheivable score }|a_1, s)$$

- But what is the right-hand side?

Minimax

- Is an algorithm to play *optimally*. It picks the move that achieves the best score *assuming* the opponent also plays *optimally*.
- Mathematically, if s is the current state, then the best move $a \in A(s)$ is the one that maximizes

$$\text{best score}|s = \max_{a_1 \in A}(\text{ best acheivable score } |a_1, s)$$

- But what is the right-hand side?

$$\text{best score}|a_1, s = \min_{a_2 \in A}(\text{ best acheivable score } |a_2, a_1, s)$$

Minimax

- Is an algorithm to play *optimally*. It picks the move that achieves the best score *assuming* the opponent also plays *optimally*.
- Mathematically, if s is the current state, then the best move $a \in A(s)$ is the one that maximizes

$$\text{best score}|s = \max_{a_1 \in A}(\text{ best acheivable score } |a_1, s)$$

- But what is the right-hand side?

$$\text{best score}|a_1, s = \min_{a_2 \in A}(\text{ best acheivable score } |a_2, a_1, s)$$

- Keep recursing until the right-hand side has no more moves to make

Minimax

- Is an algorithm to play *optimally*. It picks the move that achieves the best score *assuming* the opponent also plays *optimally*.
- Mathematically, if s is the current state, then the best move $a \in A(s)$ is the one that maximizes

$$\text{best score}|s = \max_{a_1 \in A} (\text{ best acheivable score } |a_1, s)$$

- But what is the right-hand side?

$$\text{best score}|a_1, s = \min_{a_2 \in A} (\text{ best acheivable score } |a_2, a_1, s)$$

- Keep recursing until the right-hand side has no more moves to make

$$\text{best score}|a_n, \ldots, a_1, s = \text{final score for player}|a_n, \ldots, a_1, s$$

Minimax Demo

See **examples/minimax**

Limitations of minimax

- Game trees are large!

Limitations of minimax

- Game trees are large!
- Chess has a branching factor of around 35 and depth of around 80, that's 35^{80} games to explore.

Limitations of minimax

- Game trees are large!
- Chess has a branching factor of around 35 and depth of around 80, that's 35^{80} games to explore.
- Go has a branching factor of around 250 and depth of around 150.

Limitations of minimax

- Game trees are large!
- Chess has a branching factor of around 35 and depth of around 80, that's 35^{80} games to explore.
- Go has a branching factor of around 250 and depth of around 150.
- Even our game becomes infeasible if w start with a large number of coins.

Limitations of minimax

- Game trees are large!
- Chess has a branching factor of around 35 and depth of around 80, that's 35^{80} games to explore.
- Go has a branching factor of around 250 and depth of around 150.
- Even our game becomes infeasible if w start with a large number of coins.
- **However**, minimax requires no knowledge beyond the rules of the game!

What if we run minimax on a subset of games?

Good moves are much rarer than bad ones and are easy to miss when sampled randomly.

Monte Carlo Tree Search: What do we want?

- We want to know the probability of winning on board s given *optimal* play.

Monte Carlo Tree Search: What do we want?

- We want to know the probability of winning on board s given *optimal* play.

$$P(\text{win}|s) = \sum_{a_1,\ldots,a_N} \text{isWin}(a_1,\ldots,a_N|s)P^*(a_1,\ldots,a_N|s)$$

Monte Carlo Tree Search: What do we want?

- We want to know the probability of winning on board s given *optimal* play.

$$P(\text{win}|s) = \sum_{a_1,\ldots,a_N} \text{isWin}(a_1, \ldots, a_N|s) P^*(a_1, \ldots, a_N|s)$$

- But what is $P^*(a|s)$? It is the probability that the move a is optimal. We do not know this!

Monte Carlo Tree Search: What do we want?

- We want to know the probability of winning on board s given *optimal* play.

$$P(\text{win}|s) = \sum_{a_1,\ldots,a_N} \text{isWin}(a_1,\ldots,a_N|s)P^*(a_1,\ldots,a_N|s)$$

- But what is $P^*(a|s)$? It is the probability that the move a is optimal. We do not know this!
- We, however, do know that

$$P^*(a|s) = \frac{P(\text{win}|a,s)}{\sum_{a\in A(s)} P(\text{win}|a,s)}$$

Monte Carlo Tree Search: What do we want?

- We want to know the probability of winning on board s given *optimal* play.

$$P(\text{win}|s) = \sum_{a_1,\ldots,a_N} \text{isWin}(a_1,\ldots,a_N|s)P^*(a_1,\ldots,a_N|s)$$

- But what is $P^*(a|s)$? It is the probability that the move a is optimal. We do not know this!
- We, however, do know that

$$P^*(a|s) = \frac{P(\text{win}|a,s)}{\sum_{a\in A(s)} P(\text{win}|a,s)}$$

- We have now defined the RHS in terms of a smaller LHS. We can solve for $P^*(a|s)$.

Monte Carlo Tree Search

- Below is a possible method for solving for $P^*(a|s)$.

Monte Carlo Tree Search

- Below is a possible method for solving for $P^*(a|s)$.
- Starting at board s, play one game, G_a for each possible first move $a \in A(s)$ followed by random moves till the end.

Monte Carlo Tree Search

- Below is a possible method for solving for $P^*(a|s)$.
- Starting at board s, play one game, G_a for each possible first move $a \in A(s)$ followed by random moves till the end.
- Suppose one of the games has the moves $a_1, \ldots, a_N$. This gives us an estimate, $P_e(\text{win}|a, s)$ for $P(\text{win}|a, s)$ since

Monte Carlo Tree Search

- Below is a possible method for solving for $P^*(a|s)$.
- Starting at board s, play one game, G_a for each possible first move $a \in A(s)$ followed by random moves till the end.
- Suppose one of the games has the moves $a_1, \ldots, a_N$. This gives us an estimate, $P_e(\text{win}|a, s)$ for $P(\text{win}|a, s)$ since

$$P_e(\text{win}|a, s) = \frac{\text{isWin}(G_a)}{\sum_{a \in A(s)} \text{isWin}(G_a)}$$

- Now that we have $P(\text{win}|a, s)$ for $a \in A(s)$, we can estimate $P(a|s)$ using the formula for $P^*(a|s)$.

Monte Carlo Tree Search

- Below is a possible method for solving for $P^*(a|s)$.
- Starting at board s, play one game, G_a for each possible first move $a \in A(s)$ followed by random moves till the end.
- Suppose one of the games has the moves $a_1, \dots, a_N$. This gives us an estimate, $P_e(\text{win}|a,s)$ for $P(\text{win}|a,s)$ since

$$P_e(\text{win}|a,s) = \frac{\text{isWin}(G_a)}{\sum_{a \in A(s)} \text{isWin}(G_a)}$$

- Now that we have $P(\text{win}|a,s)$ for $a \in A(s)$, we can estimate $P(a|s)$ using the formula for $P^*(a|s)$.
- Now, the next time we play a few more games don't choose the first move randomly but choose it according to $P_e(\text{win}|a,s)$.

Monte Carlo Tree Search

- Below is a possible method for solving for $P^*(a|s)$.
- Starting at board s, play one game, G_a for each possible first move $a \in A(s)$ followed by random moves till the end.
- Suppose one of the games has the moves $a_1, \ldots, a_N$. This gives us an estimate, $P_e(\text{win}|a, s)$ for $P(\text{win}|a, s)$ since

$$P_e(\text{win}|a, s) = \frac{\text{isWin}(G_a)}{\sum_{a \in A(s)} \text{isWin}(G_a)}$$

- Now that we have $P(\text{win}|a, s)$ for $a \in A(s)$, we can estimate $P(a|s)$ using the formula for $P^*(a|s)$.
- Now, the next time we play a few more games don't choose the first move randomly but choose it according to $P_e(\text{win}|a, s)$.
- Monte Carlo tells us that this process (repeated) guarantees that $P(a|s) \longrightarrow P^*(a|s)$.

Monte Carlo Tree Search: Example

See **examples/vanilla**

Monte Carlo Tree Search: Limitations

It can take a long time for $P(a|s) \longrightarrow P^*(a|s)$

Speeding up MCTS: Upper Confidence Trees

- Consider ten slot machine each having a jackpot probability given by $P^*(\text{win}|M_i)$ which we don't know.

Speeding up MCTS: Upper Confidence Trees

- Consider ten slot machine each having a jackpot probability given by $P^*(\text{win}|M_i)$ which we don't know.
- What is the best strategy for you to figure out $P^*(\text{win}|M_i)$ while *minimizing* your losses?

Speeding up MCTS: Upper Confidence Trees

- Consider ten slot machine each having a jackpot probability given by $P^*(\text{win}|M_i)$ which we don't know.
- What is the best strategy for you to figure out $P^*(\text{win}|M_i)$ while *minimizing* your losses?
- The MCTS we described is not the best strategy.

Speeding up MCTS: Upper Confidence Trees

- Consider ten slot machine each having a jackpot probability given by $P^*(\text{win}|M_i)$ which we don't know.
- What is the best strategy for you to figure out $P^*(\text{win}|M_i)$ while *minimizing* your losses?
- The MCTS we described is not the best strategy.
- A strategy called Upper Confidence Trees picks the next machine to play by seeing which machine has the highest following value

Speeding up MCTS: Upper Confidence Trees

- Consider ten slot machine each having a jackpot probability given by $P^*(\text{win}|M_i)$ which we don't know.
- What is the best strategy for you to figure out $P^*(\text{win}|M_i)$ while *minimizing* your losses?
- The MCTS we described is not the best strategy.
- A strategy called Upper Confidence Trees picks the next machine to play by seeing which machine has the highest following value

$$\frac{1}{N_i}\sum_{j=1}^{N_i} x_{ij} + \sqrt{\frac{2\ln\sum_i N_i}{N_i}}$$

N_i = number of times machine i is played

x_{ij} = win or loss when playing machine i the jth time

Speeding up MCTS: Example

See **examples/uct**

Speeding up MCTS: AlphaGo

- Even MCTS with UCT has a few limitations.

Speeding up MCTS: AlphaGo

- Even MCTS with UCT has a few limitations.
- First, when we got to a leaf node, we end up playing random games again.

Speeding up MCTS: AlphaGo

- Even MCTS with UCT has a few limitations.
- First, when we got to a leaf node, we end up playing random games again.
- The solution is to use a heuristic that can give *quick* indications of which moves should be preferred. AlphaGo achieves this using a deep learning model trained against human-expert moves from 30 million games.

Speeding up MCTS: AlphaGo

- Even MCTS with UCT has a few limitations.
- First, when we got to a leaf node, we end up playing random games again.
- The solution is to use a heuristic that can give *quick* indications of which moves should be preferred. AlphaGo achieves this using a deep learning model trained against human-expert moves from 30 million games.
- Second, when a move is selected, UCT uses no prior knowledge of gameplay to make a better decision.

Speeding up MCTS: AlphaGo

- Even MCTS with UCT has a few limitations.
- First, when we got to a leaf node, we end up playing random games again.
- The solution is to use a heuristic that can give *quick* indications of which moves should be preferred. AlphaGo achieves this using a deep learning model trained against human-expert moves from 30 million games.
- Second, when a move is selected, UCT uses no prior knowledge of gameplay to make a better decision.
- AlphaGo uses a prior distribution of moves learnt by a deep learning model using reinforcement learning and human-expert moves.

Speeding up MCTS: AlphaGo Example

See **examples/alphago**

Thank You

https://github.com/nanonaren/PWL_AlphaGo