Reversed Reversi Project
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY
September 28 – October 28, 2022, Shenzhen, Guangdong, China

CS303 - ARTIFICIAL INTELLIGENCE

MCTS FOR REVERSED REVERSI

Liu Leqi (12011327)[1]

[1]Southern University of Science and Technology, Shenzhen, China

ABSTRACT

Reversi (or known as Othello) is a popular board game originated from England. The object of reversed Reversi is in contrast of Reversi. In this project, I implemented an AI based on the Monte Carlo tree search (MCTS) algorithm. Besides, the upper confidence bound (UCB1) optimization is also utilized on the algorithm.

Keywords: Reversed Reversi, AI, MCTS

1. INTRODUCTION

Reversi (or known as Othello) is a popular board game originated from England. Players take turns placing disks on the board with their assigned color facing up. The object of the game is to have the more discs turned to display the player's color when the game is over. However, reversed Reversi is in contrast. That is, players need to obtain less discs turned in their color when the game is over. In this project, we should implement an AI to play reversed Reversi.

In 2016, AlphaGo beat Lee Sedol, the world champion of Go, which astonished the whole world. The key of AlphaGo is the Monte Carlo tree search and deep strategy value network. The core idea of the Monte Carlo tree search (MCTS) is putting the resorece on the more valuable branches. Thus, it is effective for gambling in large scale search space. In real world, MCTS also applies in fields of statistics such as planning, optimization and control systems.

Therefore, the search algorithm used in this project is MCTS algorithm and its upper confidence bound (UCB1) optimization.

2. PRELIMINARY

NOMENCLATURE

v	node of the Monte Carlo tree
s	state of the game
$a(s)$	action taken on the state
Δ	reward of the simulation
$Q(v)$	value of the node
$N(v)$	number of visited times of the node

Documentation for `asmeconf.cls`: Version 1.30, October 31, 2022.

In the game of reversed Reversi, the state of the game is the current discs distribution on the board and the player who is to place a disc. In simplification, we can use a class called *Node* to store the state of the game. Then every time when player places disc, the state will change and so do the state. Therefore, the problem can be formulated as:

- Initial state: $\text{Node}(B_0, C_1)$
- Transition Function: $f(\text{Node}(B_i, C_1), a) \rightsquigarrow \text{Node}(B_{i+1}, C_2)$
- Terminal state: $\text{Node}(B_t, C_j)$ $(j = 1, 2)$

where $B = \{B_0, B_1, ..., B_t\}$ is the discs distribution on the board for start B_0 and end B_t, $C = \{C_1, C_2\}$ is the two players, and a is the action taken in turn.

3. METHODOLOGY

3.1 General Workflow

There are 4 steps taken in MCTS: selection, expansion, simulation, backpropagation. We will always iterate to execute them until reach the computational budget.

1. Selection: Start from root node and select the best child node recursively until reaching the leaf node.
2. Expansion: Add one or more child nodes to the selected leaf node.
3. Simulation: Rollout from the expanded node until the terminal state. And then get the reward.
4. Backpropagation: Back up the reward from the expanded node to the root node.

The following two policies are taken in MCTS.

1. Tree policy: UCB1 function

$$\frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$$

where v' is the child of v, c usually takes 1.414.

2. Default policy: Based on an evaluation function, make a proper choice in each action.

MCTS with UCB1 optimization is also called UCT algorithm.

3.2 Detailed Algorithm

3.3 Analysis

The main idea of MCTS is Monte Carlo method, which is a probabilistic approach. As mention above, the decision of MCTS is based on huge amount of simulation. And then it will take the move that has the highest win rate.
It is obvious that the UCB1 function is a significant deciding factor:

$$\frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$$

where v' is the child of v, c usually takes 1.414.
The factor (f_1) on the left of plus mark denotes the win rate of one node, and the factor (f_2) on the right of plus mark helps to find the node that is less searched. Therefore, we can tuning the hyper-parameter c to balance f_1 (depth trend) and f_2 (breadth trend). Usually, c takes 1.414 as for approximate $\sqrt{2}$.

4. EXPERIMENTS

4.1 Setup

4.1.1 Environment.

- Windows 10 & Ubuntu 20.04
- Python 3.8.8
- NumPy 1.20.1
- timeout decorator 0.5.0

4.1.2 Value Map. The value map of table 2 is originally from MCTS+Roxanne of table 1. And based on the performance in the round robin, I modified it. Notice that we need to take the negative of it for reversed Reversi. Therefore, the corners are valueless and the X and C positions are worthy.

TABLE 1: ROXANNE VALUE MAP

1	5	3	3	3	3	5	1
5	5	4	4	4	4	5	5
3	4	2	2	2	2	4	3
3	4	2	0	0	2	4	3
3	4	2	0	0	2	4	3
3	4	2	2	2	2	4	3
5	5	4	4	4	4	5	5
1	5	3	3	3	3	5	1

Algorithm 1 MCTS WITH UCB1: UCT

function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TreePolicy(v_0)
    for each child v' of v_1 **do**
      $\Delta \leftarrow$ DefaultPolicy($s(v')$)
      BackUp(v', Δ)
    end for
  end while
  return a(BestChild($v_0, 0$))
end function

function TREEPOLICY(v)
  while v is not leaf **do**
    $v \leftarrow$ BestChild(v, C_p)
  end while
  for each action a in $A(s(v))$ **do**
    add a new child v' to v with
    $s(v') \leftarrow f(s(v), a)$
    $a(v') \leftarrow a$
  end for
  return v
end function

function BESTCHILD(v, c)
  return $\arg\max_{v' \in children\ of\ v} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$
end function

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ based on empirical evaluation
    $s \leftarrow f(s, a)$
  end while
  return reward for state s
end function

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta$
    $\Delta \leftarrow -\Delta$
    $v \leftarrow$ parent of v
  end while
end function

TABLE 2: FINAL VALUE MAP

500	-25	10	5	5	10	-25	500
-25	-45	1	1	1	1	-45	-25
10	1	3	2	2	3	1	10
5	1	2	1	1	2	1	5
5	1	2	1	1	2	1	5
10	1	3	2	2	3	1	10
-25	-45	1	1	1	1	-45	-25
500	-25	10	5	5	10	-25	500

4.2 Results

- Within 1 second for 500 computational budget in my local computer. (in round robin I set the iteration never halting until $4.8s$, then cut it down and get the result from the searched list.)
- Using neither greater nor smaller hyper-parameter c has better performance then $c = 1.41$.
- Three default policy in simulation. The random choice has the best performance among random, value map, and Roxanne.

4.3 Analysis

My MCTS passed the 10 usability tests given by student assistants. But the stability of MCTS is bad. It would not always win the same opponent, which was out of my expectation.

5. CONCLUSION

MCTS is an excellent algorithm for gambling in large scale search space and no more needs on domain-specific knowledge. It is only interested in the simulation win rate rather than exploring much more states to find a best next move like minimax algorithm. However, due to the feature of probability, MCTS is not stable on the final result, which is hardly to improve. As mentioned in introduction, we can use RCNN (residual convolutional neural network) to estimate the state of game and policy and value network for probability like AlphaGo.

REFERENCES

[1] B. Rose, Othello. Macmillan Education, 2005.

[2] Canosa, R. "Analysis of Monte Carlo Techniques in Othello".

[3] G. Chaslot, S. Bakkes, I. Szita, en P. Spronck, “Monte-Carlo Tree Search: A New Framework for Game AI”, AIIDE, vol 8, bll 216–217, 2008.

[4] C. B. Browne et al., "A Survey of Monte Carlo Tree Search Methods," in IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, pp. 1-43, March 2012, doi: 10.1109/TCIAIG.2012.2186810.