TCG2023 HW2

b09902004 資工四 郭懷元

How to Compile

Use this command to compile `agent`:

```
make
```

Algorithms & Heuristics

Rich game result representation

I give each game result a score using these rules:

- If we win by eliminating all the opponent's pieces, the score is 1.0
- If we win by reaching the goal square, the score is 0.9
- If the opponent win by reaching the goal square, the score is 0.1
- If the opponent win by eliminating all our pieces, the score is 0.0

Note that the score falls in the range $[0.0, 1.0]$. This should make UCB variable tuning easier, as we can start with values that would work with Bernoulli variables.

UCB

I use the fine-tuned version of UCB:

$$\mathrm{UCB}_i = \mu_i + c\sqrt{\frac{\ln N}{N_i}\min\left\{\sigma_i^2 + c_1\sqrt{\frac{\ln N}{N_i}},\ c_2\right\}}$$

Due to the significant time cost of game simulation, I decided to use the value of c_1 and c_2 directly from the paper, and only search for optimal c myself. These are the values in the final version:

$$\begin{cases} c &= 0.6 \\ c_1 &= 1.414 \\ c_2 &= 0.25 \end{cases}$$

However, I did not find significant improvement over the UCB without variance. I suspect the reason might be that my result representation is still too simple to differentiate the quality of games.

MCTS

I implemented vanilla MCTS with some shortcuts for my final agent. For better efficiency, I simulate a game tree node 25 times when it was first added or selected as PV leaf.

Shortcut

To make each simulation quicker and more "realistic", instead of randomly sample moves or use MCTS to evaluate each move, if a candidate move wins immediately, we will just do it.

Simplified pseudo code for my shortcut logic:

```
def MCTS(current_game_state):
    # Shortcut
    if len(all_possible_moves) == 1:
        return all_possible_moves[0]
    for move in all_possible_moves:
        if move wins the game immediately:
            return move
    # Shortcut end

    return typical_MCTS_procedure()

def simulate():
    while not state.over():
        # Shortcut
        if len(all_possible_moves) == 1:
            state.do(all_possible_moves[0])
            continue_while_loop()
        for move in all_possible_moves:
            if move wins the game immediately:
                state.do(move)
                continue_while_loop()
        # Shortcut end
        state.do(random_select_move())
```

Expansion policy

I use a simple `sim_count_threshold` policy, where a game tree node is expanded only after the total simulation count on that node crosses a universally same threshold. The threshold I settled with is 200.

Sampling temperature

I tried adding constant temperature on both PV finding and move choosing. But to my surprise, this technique seems to only hurt the performance, therefore I didn't keep this in my final version. The stats are included in Experiment Results

You can still compile a version with temperature during move choosing with `make temp`. The compiled binary is `agent_temperature`.

Programming

To avoid memory allocation taking up too much time during game simulation in MCTS, I used a pre-allocated array of game tree nodes, then manage them myself.

Due to the effectiveness of using a board-less state object in hw1, I also tried using a board-less state object this time. But my preliminary results shows no change to the total simulation count. Therefore I didn't include this change in my final version.

Experiment Results

Different algorithms

1000 rounds, same hyper-parameters for all settings

opponent	MCS	MCTS with normal UCB	+ rich game representation	+ tuned UCB
`easy`	991W - 9L	986W - 14L	991W - 9L	990W - 10L
`normal`	783W - 217L	962W - 38L	952W - 48L	957W - 43L
`hard`	477W - 523L	793W - 207L	825W - 175L	808W - 192L

Different c

1000 rounds

opponent	$c = 1.4$	$c = 1.0$	$c = 0.6$	$c = 0.2$
`easy`	-	990W - 10L	992W - 8L	-
`normal`	-	957W - 43L	953W - 47L	-
`hard`	833W - 167L	808W - 192L	832W - 168L	796W - 204L

It seems that fine-tuning c doesn't bring much difference. I used $c = 0.6$ in my submission.

Temperature @ PV finding

Note: $K = \frac{1}{T}$, against `hard`, 200 rounds

$K = 0$	$K = 2$	$K = 5$	$K = 10$	$K = 20$	$K = 40$
93 W - 107L	115W - 85L	136W - 64L	143W - 57L	158W - 42L	161W - 39L

The result shows that less randomness (larger K) leads to better performance, so I didn't include this in my submission.

Temperature @ move choosing

Note: $K = \frac{1}{T}$, against `hard`, 500 rounds

$K = 0$	$K = 2$	$K = 5$	$K = 10$	$K = 20$	$K = 40$
18 W-482L	86W - 414L	175W - 325L	225W - 275L	273W - 227L	351W - 149L

The result here also shows that less randomness (larger K) leads to better performance, so I didn't include this in my submission.

References

- People I discussed with: 陳可邦
- Finite-time Analysis of the Multiarmed Bandit Problem
- Node-Expansion Operators for the UCT Algorithm
