Theory of Computer Games Homework 2

Written by 朱紹瑜 (Student ID: R11922047)

How to Execute My Program?

1. Compile the Agent, the Game, and Baseline Programs

```
 make
cd game
make
cd baseline
make
cd ..
```

2. Run the program

```
./game/game -p0 ./main.out -p1 <baseline-program> -r <number-of-rounds>
```

For example, to compete against the `greedy` baseline for 10 rounds, run `./game/game -p0 main.out -p1 ./baseline/greedy -r 10`.

Algorithms, Heuristics, and Other Techniques

1. Monte-Carlo Tree Search (MCTS) with Upper Confidence Bound (UCB)

The core of this program is a Monte-Carlo tree search.

Set the given position as the root and expand it by one level, where each leaf represents the result of applying a valid ply to the initial position. Run random simulations 100 times per leaf node.

Then repeat the following steps multiple times until the time is nearly up.

1. **Selection**

   Select the principal variation path (PV path). Follow the top-down direction, if the current node is a MAX node, pick the child with the largest score; otherwise, pick the one with the smallest score. That is to say, for each level, select the child node following the formula below.

   $$ \begin{cases} \argmax_i \frac{W_i - L_i}{N_i} + c\sqrt{\frac{\log N}{N_i}} \quad \text{if the node is a MAX node} \\ \argmax_i -\frac{W_i - L_i}{N_i} + c\sqrt{\frac{\log N}{N_i}} \quad \text{if the node is a MIN node} \end{cases} $$

2. **Expansion**

   Expand the leaf node of the PV path selected from the previous step by 1 level. Here we apply an “all ends” expansion strategy. All valid plys are expanded into a new node.

3. **Simulation**

   For each newly created node, perform 100 playouts. Throughout the playouts, both players play randomly.

4. **Back Propagation**

   Update the scores for the ancestor nodes.

Note that there are several slight modification done to our MCTS algorithm.

1. **Early termination**

   Given the rule of playing “pass” as a move, the standard MCTS expands nodes of terminal positions with one node further. However, since that the result can be directly judged from the terminal position, we directly add the simulation results and do not expand nodes of terminal positions.

2. **Different exploration constant for opening, middle game, and end game**

   We set the exploration constant (the coefficient c in the UCB formulation) as different values for the opening, the middle game, and the end game. The constant starts with a larger one and decreases as the game proceeds. We determine the game phases by the total manhattan distance of the current position of the cubes to their target corner.

2. Rapid Action Value Estimate (RAVE)

For each node, we maintain both the actual score and the all-move-as-first (AMAF) score. Given the simulation results of a leaf node, we search for its ancestors, if any of its ancestor has a child node whose corresponding ply matches the one of the leaf node, update the results to the child node’s AMAF simulation results.

During the computation of UCB scores, blend the actual scores and the AMAF scores with RAVE. Let $v_1(P)$ be the score of a position P with purely actual simulations. Let $v_2(P)$ be that of P with both actual and virtual simulations. The revised score is $v_3(P) = \alpha \times v_i(P) + (1-\alpha) \times v_2(P)$, where $\alpha \in [0, 1]$ is a coefficient that decreases as the number of actual simulations increases.

3. Parallelization

The number of simulations is critical for Monte-Carlo based algorithms. To increase the number of simulations done in the given time limit, in the simulation phase, we distribute the computation to 4 threads to run in parallel. Through this technique, the required time for a single batch of simulation decreases and results in a larger number of simulation done.

Implementation Detail

1. Node Data Structure

The following is the data structure of a tree node.

```
class Node {
    static ldbl explorationConst;
    Board board;
    Ply ply;
    Node* parent;
    vector<Node*> children;
    int depth;
    long long simCnt;
    long long winCnt;
    long long loseCnt;
    ldbl cSqrtLogSimCnt;
    ldbl sqrtSimCnt;
    ldbl avgScore;
    long long amafSimCnt;
    long long amafWinCnt;
    long long amafLoseCnt;
    ldbl amafAvgScore;
    mutex simResLock;
};
```

`board` stores the position of all existing cubes, while `ply` records the ply that leads the parent node to the current node. The simulation results are recorded in `simCnt` (total number of simulation), `winCnt` (the number of winning), and `loseCnt` (the number of losing). Note that the the number of draws can be calculated by `simCnt - winCnt - loseCnt`. Each node also maintain its `cSqrtLogSimCnt` ($c\sqrt{\log N}$), `sqrtSimCnt` ($\sqrt N$), and `avgScore` ($\frac{W - L}{N}$) for UCB score computation.

2. Random Number Generator

To ensure the randomness of our random simulations, we apply the C++ random number generator provided by PCG (https://www.pcg-random.org/).

Experiment and Discussion

We compare the performance of different algorithms by running them against the 3 baselines (random/greedy/heuristic) for 10 rounds. The results are listed below. Note that the winning result is shown in the format of #Win : #Lose : #Draw. As for the number of simulation, average leaf depth, and max leaf depth, we record the statistics of the first round.

	Pure Monte-Carlo	MCTS	MCTS + RAVE	MCTS + RAVE + Parallelization
vs. Random	10 : 0 : 0	10 : 0 : 0	10 : 0 : 0	10 : 0 : 0
vs. Greedy	4 : 6 : 0	10 : 0 : 0	9 : 1 : 0	8 : 2 : 0
vs. Heuristic	0 : 10 : 0	2 : 8 : 0	3 : 7 : 0	5 : 5 : 0
#simulation	179,300~2,625,900	144,100~3,334,100	150,400~3,045,500 (AMAF: 247,400~11,674,000)	454,500~294,759,564 (AMAF: 962,400~307,540,564)
average leaf depth	1	3.33~8.52	3.23~7.93	3.95~11.35
max leaf depth	1	4~10	4~10	5~19

By comparing MCTS-based methods to pure Monte-Carlo, we can see that the mini-max concept does improve the performance of the agent.

With RAVE, the number of AMAF simulations is a lot more than the actual simulations. However, it does not make much difference on the agent’s performance.

By applying multi-thread execution, the number of simulation (both actual and AMAF) increased by about 3~4 times. The tree is searched slightly deeper. And the performance is also slightly better than the one without parallelization.