Gomoku AI Report

Weijie Deng, Yifei Chen

1. General Structure:

Basically, our gomoku AI is a modified combination of Monte Carlo Tree Search (MCTS) and Heuristic Search, with the class we define as StateNode. The specific function of it and its contribution to MCTS will be expanded as follow.

2. StateNode Class

2.1. Properties

StateNode class serves as the state of certain board. It receives the board as a 2D list and the turn right now (in True or False pattern in terms of convenience in turn transition) as compulsory inputs, and optional inputs includes the number of simulations it has (default 0), the number of win game(default 0, not the perceptual win and we will discuss it later), its parents (default None, an StateNode instance, the parent state of the contemporary state), and reach_move (default None, we called it so but actually it is the move its parent, if it has, takes to reach the contemporary state)

Another properties the StateNode has include the board's features (calculated based on the board), terminal state (calculated based on the features), the evaluation of the board (calculated based on the features), its children (default empty list, a list of StateNode instances, the child states contemporary state will have), the hint (a little smaller range of what the next move could be, calculated based on the board). To simplify the sampling process, we define the modified score (a list of modified score its children have, if it has, the modify rules will be discussed later).

2.2. Feature

To extract the features a board has, firstly we extract the vertical, horizontal and diagonal lines from the board one by one, and the total features of the board is the summary of each single lines. We set that a line only has one most prior feature at the same time, and the priority levels of different patterns are manually set as follow (if we let 1 as the player's move who is evaluating the board, x as the rival's move or the boundary of board):

L5: '11111' L4: '011110' S4: 'x11110', '11101', '11011'
L3: '01110', '010110' S3: 'x11100', 'x01110x', 'x11010', '0110010'
L2: '0110', '01010', '010010' S2: 'x1100', 'x10100', 'x10010' L1: '010' S1: 'x10'
'None': not mentioned
Priority level: L5 > L4 > L3> S4 > L2 > S3 > S2 > L1 > S1 > None
We judge the form with regular expression, and detect the feature in the current state and the rival perspective respectively.

2.3. Evaluation

To have better performance, the scoring rules we set is not zero-sum, so different sides have different scoring even based on the same board, and alpha-beta pruning is not available in this case. For example, we are player A at board M and it's A's turn right now, and we are going to decide the move to take, so we calculate the scores of child(A) from A's perspective and choose the best to act. Then, we calculate the score based on the number of features a board has, combined with some manual knowledge for example some winning patterns. The scoring rules are listed according to priority order.
me 'L5'>0: 100000; rival 'L5'>0: -100000; rival 'S4'>0: -40000; me 'L4'>0: 40000;
(me 'S4'>0 and 'L3'>0) or (me 'S4'>1): 20000; rival 'L3'>1: -20000; rival 'L3'>0: -10000
me 'L3'>1: 20000
If all of the cases above don't cover the current board, let 'L3': 500, 'S4': 300, 'L2': 100, 'S3': 50, 'S2': 5, 'L1': 2, 'S1': 1, 'None': 0, and the evaluation is the sum of the number of each feature in current state's view product the score of this feature, and minus the number of each feature in rival's view product the score of this feature.

2.4. Get Children

To narrow the searching spaces we explore, we only consider the vacant positions which is located in the 5*5 grid centered by a certain chessman already on the board. A state will only consider these new boards to generate its children. Such possible positions are stored in a list self.hint.

Every time we apply the method self.get_child(), we create its children whose board inherit the board from the parent but take a new move suggested by self.hint. Children of a parent will have large similarity in features compared with the parent. So every time we create new child, we plug in the optional input 'parent', and let the child inherit the feature from its parent, and just recheck the features in the lines containing the newest move. This method would save the time by almost one order of magnitude.

Since the evaluation of children are from the parent's view, and the parent has to decide the child with high scores. In normal case we select the children with the up to top 3 highest scores as the children the parent has. So children of a certain state (at most 3) are relatively worth more consideration in our feature rules. If there is some necessary move to take, such as alarming patterns, we distinguish each case into several alarming groups. Grouping rules as following:
A. If I take that move and get L5, return this child immediately.
B. If rival takes that move and get L5. If group A is empty, return the best child in this group.
C. If I take that move and get more than 2 'S4' or 1 'S4' and 1 'L3'. If group A, B are empty, return the best child in this group.
D. If rival take that move and get more than 2 'S4' or 1 'S4' and 1 'L3'. If group A, B, C are empty, return the best child in this group.
E. If me take that move and get more than 1 'L3'. If group A, B, C, D are empty, return the best child in this group.
F. If rival take that move and get more than 1 'L3'. If group A, B, C, D, E are empty, return the best child in this group.
In all of these cases, or if there is only one move to take, the number of the child would be 1, and

we would directly use that child without MCTS or other procedures.

3. MCTS

Generally we apply the similar principles as normal MCTS: firstly every time we let the current state be the root of the tree, and obey certain principles to select the leaves (children) of the root, and when meeting an unexpanded leaf, we create its children and randomly simulate the game in each children for several times, finally update the simulation records (times of simulation and winning) from the leaf to the root. After certain times of loop the tree search would converge, and the root would use the same principles to select the best child in the first layer children as the output of MCTS. What differ us are listed in several aspects:

3.1. Random simulation

As the get_child method mentioned above, a state would have only up to 3 children of high reference at the same time, which would narrow the search range and enhance the quality of random play, and in return decrease the demanded times of simulations to reach convergence. Meanwhile, due to the scoring rules, a child might have negative score but it may be still worth exploration. To modify the sample rules, firstly we modify the scores of three children, which sum up the absolute value of all the negative scores, and let every scores plus the sum to generate the modified scores. For instance, -300, -100 and 200, the modified scores of these three are 100, 300 and 600. The formula is:

$$add = \sum_{score(child)<0} -score(child)$$

$$MS(child) = score(child) + add$$

Then,we apply score_sample method: with the modified scores, we sample the children list based on the score each children has. Specifically, we have the formula to calculate the probability of picking up child A:

$$P(A) = \frac{MS(A)}{\sum_{child} MS(child)}$$

Based on the definition above, we create random_play as a method of a StateNode, which return the result of score_sample to serve as random simulation.

3.2. Definition of win index

Often it is a chance that the current board is so unclear that it would need too much times of simulation to reach any terminal states, for example in the openings. Instead of the method of recording 'win' until reach that, we just random_play for up to 2 moves (if one of the players wins the simulation stops for sure), and consider the evaluation of board at that time. Generally, we create a linear map from the range of that evaluation [-10000, +10000] to the new index [0, 1], and see the new index as the alternative of win index. If the evaluation is higher than 10000 or lower than -10000, we simply view them as 10000 or -10000 for convenience. In fact from the scoring rules, it's obvious that the winner of the board is doomed if the evaluation is that high or that low. So every random_play will end in up to 2 steps, and return a win index in the range of [0, 1]. We use this win index to update all the parents of current state.

3.3. Child selection principle

We choose the selected child from a certain expanded state in terms of win index and modified scores of the boards of its children. The judge function is defined as:

$$Judge(A) = \frac{win(A)}{sim(A)} + \frac{MS(A)}{\sum_{child} MS(child)}$$

The function will balance the significance of the board features and the performance in random simulations.

The task is time-limited, so it's possible that a certain child has not been simulated when the process is killed, then the win ratio part of function Judge would be 0.5. Or sum of MS(child) is 0, then the scoring part of function Judge would be 0.

4. Performance

Generally we limit time for per step in 2 seconds, and it's tested that the initial state has been through about tens of simulations, which is acceptable because there has been at least 4 times of comparison between the children in the first layer, and the tree search has considered the situation at least 4 steps ahead (maybe the same as a novice human).

By the way, MCTS seldom converges in 2 seconds, but we have to do so since the time limit of this task is so wired that the time for per match has the same order of magnitude as the time for per step. It's tested by us that if given a normal time limit (15 seconds per step and no total time limit), the Elo rating of our AI would increase from about 1300 to about 1600. Such a pity!