In this project, alpha-beta algorithm is adopted to determine the next move for player along with the use of machine learning to set evaluation function. The depth of our search limit is up to 5 in submitted version due to the storage and time restriction of the assignment.

choice of search algorithm and evaluation function

This chess game is a deterministic perfect-information game, and alpha-beta pruning is actually perfect for this type of game. Moreover, since there are time and storage restriction, alpha-beta pruning algorithm is favored because it can cut extra nodes off to reduce time complexity and set depth limit compared with other similar algorithms like minimax and MinimaxCutoff. Therefore, this game is categorified into a search problem using α–β algorithm.

Firstly, we implement a basic α–β algorithm that can search in depth of 3 in limit times.

The algorithm will return the most valuable leaf at max cut-off depth, where the value of leaf is measured by evaluation function at that state.For evaluation function, there are 6 features.

Features 1-5: number of single stack consists of (1-5) tokens. Stack consists of more than 5 tokens will be considered as 5-token stack.

Feature 6: number of own-side stack on board.

Our idea of evaluation function design is that: the evaluation function should reflect real situation of how good the state is. In other words, the evaluation function that only evaluate current state should be: (number of own side token - number of opponent). However, due to the limitation of steps that algorithm can look forward, using evaluation function described above may lead to a sequence of state with same evaluation score followed by low-score state (state after explosion). To solve this, we must add some subjective features to predict how likely is the current state to win in the future. It is easy to find states with some features are likely to win:

1. sparse distributed stacks
2. stacks consist of multiple tokens.

To motivate algorithm to form stacks, average weight per token of 2-token-stack and 3-token-stack is set to be higher than single token. As for 4-token-stack and 5-token-stack, because there are too many tokens in one stack, average weight per token is set to be similar to single token.

To motivate distributed sparsely, we considered several options like “calculate the minimum spanning tree of tokens” or “how many opponent tokens is needed to explode all own-side tokens”. The method above is very time-consuming, a little increase of time complexity in evaluation will increase the time complexity of the whole algorithm exponentially, because evaluation function is called at the bottom of the search tree.

At last, we choose (number of opponent stacks – number of own stacks) as feature to evaluate the density of tokens, because less stacks means there is less probability that two stacks are close. Although it might not be appropriate, it is a trade-off between quality of feature and time complexity.

Different versions, modification and effectiveness of program

We have created 2 versions of game-playing program, the first one is the basic α–β we described before. As for the second one, we made some modification based on the first one:

we introduced a filter_move function to reduce search space of moves and achieves depth of 5.

We choose the second one for submission because it has better performance.

The structure of two versions are the same, there are two modules except player: “utility” and “board”. “utility ” contains the body of the algorithm including α–β, filter_move(in second version) and evaluation function, while “board” is responsible for recording state of board, updating the board and finding the move.

After implementation of basic α–β(first version), we found that α–β with depth of 3 have weak ability to plan ahead, for example, it will not make reaction until there is an opponent token nearby, which is too late for both defense and attraction to opponent token. In order to have a better performance, the algorithm must look several steps ahead.

In second version(submitted version), to increase depth of the algorithm, we introduce function “filter_move()” to reduce number of possible moves in search space. This function uses an intuitive way to evaluate if a token after one or two moves can explode near opponent tokens. If it is recognized that there is no opponent token nearby after several moves, then the first move will be excluded from search space. One problem we face is that: at the beginning of the game, after using this filter_move() function, there is no valid search space. In this case, we use a α–β with depth 1 without using filter_move() function to find next move.

This intuitive method successfully increases the search depth to 5, however, sometimes it will ignore some good actions. This is because we use human experience to help algorithm filter search space. Due to the defect of our game perception, we may consider some good move as bad move. Another drawback of the program is that the variance of runtime in different state is large. This is because the number of moves excluded by filter_move() function varies among different states.

As for time complexity of α–β, version one has $O(b1^{(3/2)})$ in perfect ordering, worst case is $O(b1^3)$. For version two, best case is $O(b2^{(5/2)})$, worst case is $O(b2^5)$. Note that $b2 < b1$ because we discard some moves in search space in version two.

Moreover, there are some extremely sensitive functions which are used heavily to find possible moves.Time complexity for one function named Single_token_move is $O(n+m)$ and this function is also called in a loop in another function named find_move. As a result, these two functions combined inevitably and significantly influence the performance of our program. The program can perform well with depth limit set=5. However, in some cases, when there are lots of separate independent tokens and depth limit is large, it would consume far more time due to the function find_move and the data structure we used to store tokens.

Attempt on machine learning

We tried to use machine learning to modify the weight of features. We choose td(lambda) rather than tdleaf(lambda), because tdleaf(lambda) is required to find features of best leaf found at max cut-off depth, which is difficult for α–β algorithm, where td(lambda) only focus on evaluation of current state.

To obtain training instance, we create a environment for self-play and record the moves of each game. After each game, we update weight by

$w_j \leftarrow w_j + \eta \ \Sigma$ (from i = 1 to N-1)$(\partial r(s_i , w)/\partial w_j)$ [Σ (from m=i to N-1) m=i λ m−i dm]

Where $d_i = r(s_{i+1}, w) - r(s_i , w)$, $r(s_i , w) = \tanh(eval(s_i , w)$

We set learning rate η to be 0.1 and λ to be 0.6.The weight w converges only after several games and we got a weird weight, it seems that evaluation function overfit the training case.

The performance of algorithm using new w is even worse.