COMP30024 AI Part_B Report

Group AIH1   Lanruo Su (1297549)   He Shen (1297447)

This report introduces the game agent program we designed, which uses the Minimax algorithm with Alpha-Beta pruning to solve decision problems in the Game of Testress. We have demonstrated how to improve the efficiency and effectiveness of game agents under limited resource conditions through detailed analysis of algorithm selection, performance evaluation, and other innovative technologies.

Section 1: Basis

We choose Minimax as our main algorithm due to its optimality and reliability in solving such strategy chess combat with two players. At the same time, it can effectively evaluate multiple results after several actions, and prune those actions which may result in relatively passive or disadvantageous to the player.

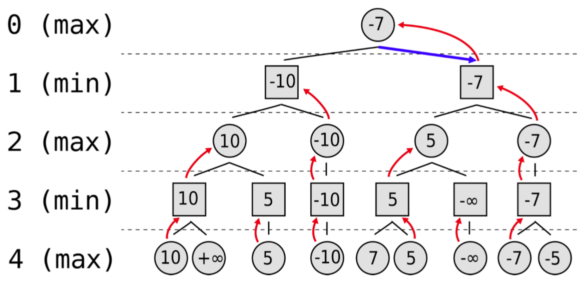


Figure 1 - minimax tree example

By recursively evaluating potential game outcomes through considering each player's possible moves, it can backtrack to determine the best action.

In terms of efficiency, we have applied additional techniques to significantly shorten the search time while ensuring optimal solutions. The first one involves dynamic depth. We will dynamically adjust the search depth based on the current game states to balance search accuracy and time. The second is to implement a transposition table, which records the previous search result with board state to system memory. Both techniques will be further discussed in **Section 3** in more detail.

The overall goal in designing our evaluation function is to ensure the heuristic purpose. Our evaluation function will use previously generated actions during the search period to calculate the potential descendants for both red and blue players. After that, it will return the difference between the number of positions that each player can place as a utility score. A positive and sufficiently large score indicates a broader range of choices compared to the opponent, which is more likely to achieve the ultimate victory. Conversely, a very small or even negative evaluation score would suggest a narrower range of choices, increasing the likelihood of loss. This evaluation score can effectively reflect the winning rate. For a detailed implementation of our heuristic, please refer to **Section 4**.

In modern days, it is always worthwhile to decide whether to use machine learning. After some discussion, we attempt to implement a Minimax pure algorithm solution. Here are several considerations that led to this decision:

1. Game complexity and information controllability:

Similar to Go, Tetress has a high level of uncertainty and a relatively complex state space. With an 11x11 cell board, even slight differences can lead to diverse place actions and ending trends. Thus, it seems that the Monte Carlo Tree Search (MCTS) algorithm is the primary consideration. However, since Tetress has much fewer cells than Go (19x19) and each piece occupies four grids, which means fewer combinations need to be considered and the overall space utilization will be larger than Go. Therefore, the Minimax algorithm can be handled and used to recursively extract the required information.

2. Resource effectiveness and limits:

MCTS randomly picks up and makes simulations to evaluate certain nodes, which may consume more resources. From the project specification, there are two limits, one is a maximum computation time limit of 180 seconds per agent per game, the other is a maximum peak memory usage of 250MB per agent per game. In this situation, adopting MCTS directly may not be a very good approach. While Minimax follows a more direct but effective searching step, it directly evaluates and compares various possible game outcomes through a deterministic approach, making it easier to control the use of resources.

3. Overall accuracy and reliability:

The results of MCTS depend on random sampling, there may be some differences in the final learning outcomes when comparing them horizontally. Considering the uncertainty of the opponent, the final result may further differentiate, leading to certain fluctuations between different runs. On the contrary, Minimax provides a deterministic solution, since the subsequent search and placement processes are determined by the values of Alpha or Beta. Due to this consistency, the result can remain unchanged when repeatedly generated in a fixed board state. We believe that this robust approach is particularly important in Tetress.

4. Real-time response capability：

Using search algorithms that do not require machine learning naturally saves a portion of machine learning time. The Minimax algorithm can already provide a satisfactory result while making good and quick decisions if we do not pursue a comprehensive optimal solution and only consider the current agent's ultimate victory as the goal of implementation. This approach already enables the agent to respond to real-time combat chess games effectively.

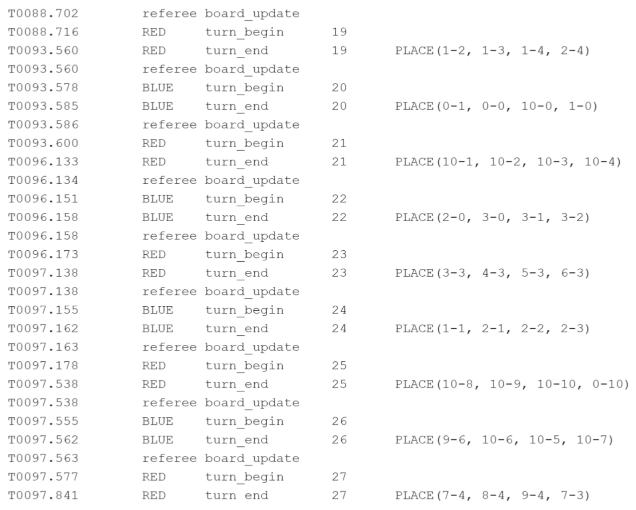
Section 2: Performance

Time complexity and space complexity provide the basic performance-related parameters, offering initial insights and facilitating subsequent testing and optimization processes.

The time complexity of the basic Minimax algorithm is usually O(b ^ d), where b is the number of possible actions for the player in each turn, and d is the number of possible actions for the opponent and oneself to continue searching after assuming the current action is taken. Its time complexity will increase exponentially. Thus, the duration of search turn can be very long, possibly far exceeding the limit of 180 seconds. Therefore, we decided to use Alpha-Beta pruning to prune unnecessary search branches which may lead to unfavorable results to control and optimize the duration. The principle is to use the Alpha and Beta values at both ends to control the maximum or minimum values that the Min and Max layers can accept. In the most ideal scenario, the time complexity can be reduced to O(b ^ (d / 2)).

In addition to time complexity, space complexity also needs to be considered. Under default conditions, its space complexity is mainly determined by the stack of recursive calls, and the number of stacks corresponds to the depth of the search, d. So the spatial complexity is O(d). Unlike time complexity, even with pruning, it still does not change the maximum possible depth of recursive calls, since it is determined by Tetress' own rules, with the actual actions of the agent under the dynamic depth strategy in our program. Hence, the space complexity of using and not using pruning can be considered as the same.

However, as we have applied additional techniques such as dynamic depth and transposition tables (mentioned before), they would also affect time and space complexity to a certain extent. In the actual performance of the program, which includes various chess game states such as the opening, middle, and ending. The CPU time for each turn is approximately 5-10 seconds. In specific situations, such as when the player represented by the agent has a much greater winning trend than the opponent, the place action time will be reduced to about 1 second. The above actual performance tests are all based on the Gradescope platform. We believe that such performance is basically in line with our expectations and should also be at a normal level.



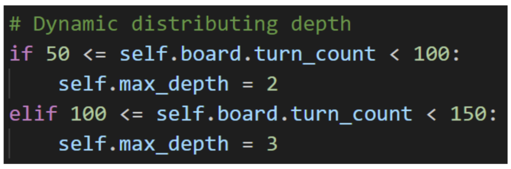
```
T0088.702       referee board_update
T0088.716       RED     turn_begin      19
T0093.560       RED     turn_end        19      PLACE(1-2, 1-3, 1-4, 2-4)
T0093.560       referee board_update
T0093.578       BLUE    turn_begin      20
T0093.585       BLUE    turn_end        20      PLACE(0-1, 0-0, 10-0, 1-0)
T0093.586       referee board_update
T0093.600       RED     turn_begin      21
T0096.133       RED     turn_end        21      PLACE(10-1, 10-2, 10-3, 10-4)
T0096.134       referee board_update
T0096.151       BLUE    turn_begin      22
T0096.158       BLUE    turn_end        22      PLACE(2-0, 3-0, 3-1, 3-2)
T0096.158       referee board_update
T0096.173       RED     turn_begin      23
T0097.138       RED     turn_end        23      PLACE(3-3, 4-3, 5-3, 6-3)
T0097.138       referee board_update
T0097.155       BLUE    turn_begin      24
T0097.162       BLUE    turn_end        24      PLACE(1-1, 2-1, 2-2, 2-3)
T0097.163       referee board_update
T0097.178       RED     turn_begin      25
T0097.538       RED     turn_end        25      PLACE(10-8, 10-9, 10-10, 0-10)
T0097.538       referee board_update
T0097.555       BLUE    turn_begin      26
T0097.562       BLUE    turn_end        26      PLACE(9-6, 10-6, 10-5, 10-7)
T0097.563       referee board_update
T0097.577       RED     turn_begin      27
T0097.841       RED     turn end        27      PLACE(7-4, 8-4, 9-4, 7-3)
```

Figure 2 - Gradescope submission log

Section 3: Technical Creatives

We have designed and implemented some additional technologies to ensure that our agent can work better, both are designed under a performance optimization perspective.

1. Dynamic Depth: In the initial stage of the game, due to the large number of choices that can be placed on the board and less effect on the final game result, we will reduce the search depth and instead use evaluation functions to score each possible action. The advantage of doing this is that during the search, only one layer needs to be expanded, and the decision-making power for selecting the best solution is indirectly handed over to the utility score provided by the evaluation function, rather than searching through nodes. It is obvious that searching only one layer can significantly reduce search time, and what is worth considering and discussing is the principle and performance of heuristic search schemes in the evaluation function, as it largely determines the direction and accuracy in this early game period. When the game enters the middle and later stages, we will gradually increase the search depth to better find the combination combo that can truly defeat the opponent. Overall, dynamic depth provides more flexible space for our program's Minimax algorithm, allowing for different search depths at different game stages and thus balancing accuracy with search duration.



```
# Dynamic distributing depth
if 50 <= self.board.turn_count < 100:
    self.max_depth = 2
elif 100 <= self.board.turn_count < 150:
    self.max_depth = 3
```

Figure 3 - dynamic depth setting

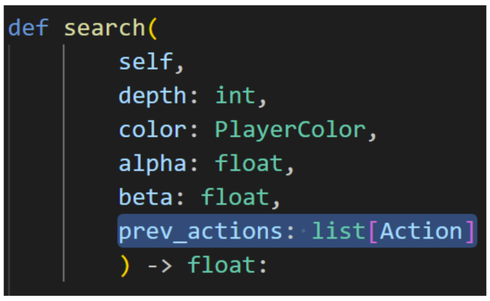
2. Transposition Table: Till this point, the algorithm as a whole can work as expected and ensure efficiency. However, there may still be some time issues encountered in some extremely rare situations, especially in residual games. Under these circumstances, the search duration per turn may still take quite a long time to complete due to the fact of more messy distributed branches and deeper node expansions. We have decided to add a transposition table to store the board state at a certain depth with its corresponding utility score. Considering the limitation of 250MB memory, we have decided to only store the state plus its utility score when the depth is greater or equal to 2, while avoiding storing all node expansions, as well as detailed Alpha and Beta values, to avoid unnecessary space waste. When storing the state, the function assigns a unique hash value to represent the board, so that only when a completely consistent board is accessed in the future can its corresponding utility score be provided. Hence, there is no need to perform subsequent node expansions. Instead, the utility score is directly obtained. By using a trade-off with memory, a lot of search time is saved.

Section 4: Auxiliary Tools

We reuse some helper functions from Project Part A, such as `adjust_coord`, which is a function to adjust the coordinates if it overflows, and also `get_all_tetrominoes`, a function that defines every shape and their corresponding coordinates. These functions are separately placed in the utils.py file.

However, some other functions have been adjusted accordingly for the changes made in Part B, with the biggest being the evaluation function, named as `heuristic`. We have redesigned the whole evaluation function for it to consider from the perspective of the game. The goal here is to get the probability of the current agent winning based on the board state, in order to return the utility score. In the meantime, we tried many implementation cases, such as directly obtaining the difference in the number of placed pieces between the red and blue on the board, or determining based on the proportion of adjacent blanks between the red and blue. However, these solutions either have poor heuristic results or have relatively long calculation times. Finally, we came up with a solution that accepts both sides' place actions, and then determines the number of their sets for both sides. By subtracting their numbers, the final utility score can be used.

Although this approach requires significantly less computational time compared to the others, after actual testing, the time performance in some board states is still not ideal. To generate a sequence of place actions for a given player on a chessboard, it is necessary to traverse the entire 11x11 chessboard, with a time complexity of up to O(11 x 11). We need to consider how to avoid such duplicate generation and computation. The first idea we came up with is to use the transposition table we have implemented, which may require some modifications to further store the place actions of both sides. However, due to the frequent retrieval of evaluation functions throughout the entire program, it can quickly consume the memory. Then, we suddenly realized that both actions had already been generated once when expanding nodes, but they did not correspond to the same board state. The former's actions corresponded to the previous board, while the latter corresponded to the board after the former player had deployed its tetromino. However, we discovered these two actions are on the same search branch and can somehow still reflect the performance. After some testing, the time required for the evaluation function has been reduced by nearly half, and its accuracy is almost identical to the original solution.



```python
def search(
        self,
        depth: int,
        color: PlayerColor,
        alpha: float,
        beta: float,
        prev_actions: list[Action]
        ) -> float:
```

Figure 4 - accessibility of previous sections

In addition to the evaluation function, we have also added a `generate_random_action` on top of the `generate_successor_actions` to make their first random move at the beginning of the game. In order to avoid certain race conditions, such as when both sides happen to be close to each other and placed in the same column or row, resulting in a higher winning rate for the first player, we have added additional judgment to generate more fair placement schemes multiple times.

Finally, we also adjusted the `is_valid_place_action` function accordingly, now it can accept a color attribute to check whether the corresponding coordinate neighbor to be placed is the same color. At the same time, we have also optimized the overall evaluation of whether it is valid. For those unreasonable actions, they will be directly rejected.

Conclusion

In this project, our team developed a game agent based on the Minimax search algorithm for Tetress. In order to improve the overall search efficiency, we used Alpha-Beta pruning and dynamic search depth strategy, to effectively improve computational efficiency, ensuring that the algorithm has excellent performance under limited resource conditions. In addition, we have also introduced transposition tables to further improve the search process in the later stages of the game, reduce duplicate expansions, and make the program perform better.

Overall, our evaluation shows that the agent is able to make quick and accurate decisions throughout the game. Although we have chosen a pure algorithmic solution that does not use machine learning, this choice is based on the complexity and information controllability of the game, and has been proven to be effective in dealing with various situations. Moreover, for us personally, this project not only enhances our understanding of strategy game AI design, but also lays a solid foundation for us to develop efficient AI agents in a wider range of application fields in the future.

Reference

Figure 1, A minimax tree example, Nuno Nogueira (Nmnogueira),
https://en.wikipedia.org/wiki/Minimax#/media/File:Minimax.svg

Figure 2, Gradescope Autograder Log,
https://www.gradescope.com/courses/744332/assignments/4356771/submissions/253010705

Figure 3 & 4, COMP30024-Project-AIH1 / agent / program.py,
https://github.com/cxlanyagege/COMP30024-Project-AIH1/blob/main/agent/program.py