CSCI1410

Final Project Capstone Writeup

Backstory:

During the process of my implementation of the bot player, most pivots happened before Milestone 2 (the first milestone that required bot performance). My initial plan was to implement using Monte Carlo Tree Search with Reinforcement Learning, because compared to the other recommended approach, adversarial search, this seems to have a much higher potential. Therefore, when completing Milestone 2, my first attempt was to write a fundamental Monte Carlo Tree Search algorithm. To make the algorithm executable was not extremely sophisticated. Still, due to its randomness, the performance was not good enough to pass the milestone, especially under the 1-second limit of the decision time.

So I tried to further implement a heuristic function in addition to the fundamental MCTS. While researching online how to implement a heuristic function in MCTS, I recognized the complexity of this approach was way beyond my expectations. Many AI bots use MCTS + RL as the foundation structure on various gameplays, such as Alpha Go, and after I viewed some of them, I realized the computational power needed for this approach is huge. Especially when we were not provided with a database of historical games, this approach couldn't partially rely on supervised learning and needed to be trained completely on reinforcement learning, which started by randomly playing against different provided bots. If my understanding was correct, this process of self-playing required a huge amount of time and computational resources. On the other hand, implementing MCTS with a heuristic function was also

stuck, as I did successfully improve its performance but it was much worse than adversarial search. Since the deadline for Milestone 2 was approaching, I decided to switch to the Minimax algorithm, at least for this milestone.

According to my attempts, the Minimax algorithm with alpha-beta pruning can run up to depth 6 within the 1-second limit. Using a very basic heuristic function only considering the numbers of PERM and TEMP blocks (of course weighted) was good enough to pass Milestone 2.

Between Milestones 2 and 3, I did more research and consulted in TA hours and on Edstem. I confirmed that the adversarial search approach also has the potential to fulfill all future requirements. Also, upon my further research on the MCTS + RL approach, I reinforced my belief that it's extremely difficult. So eventually I decided to use adversarial search until the end. In the process of implementing Milestones 3 and 4, the fundamental structure of my bot doesn't change. The differences are mainly in the optimization of the heuristic function and handling specific edge cases, which will be further described in the section below.

Description:

I will describe my current bot design in three main segments: Minimax algorithm, heuristic function, and edge case handling.

Minimax algorithm:

The Minimax algorithm is the core yet simplest part of the bot design. It's a classical implementation with alpha-beta pruning to enhance the efficiency and allow

a higher maximal depth. Layers in turn alternately choose the maximal or minimal value action, while comparing it to the alpha or beta value and pruning the branch before examining it when applicable. When the game is ended or the maximal depth limit is reached, the heuristic function is called to evaluate the resulting state.

It is worth mentioning the depth limit is set to 6 to satisfy the 1-second decision time limit. The choice of 6 is the result of careful examination and to my current understanding cannot be further enhanced. The main time cost of this Minimax algorithm is in executing the heuristic function. Since each player's move has up to 4 actions, when the depth limit is set to 6, there can be a maximum of 4^6 terminal states to use the heuristic function to evaluate. The get_safe_actions() function in the given GoTProblem class is used to eliminate unviable choices, but the number of times that the heuristic function is executed in each call of Minimax is still huge. The current implementation of the heuristic function cannot further improve its time efficiency. According to my step-by-step measurement of time cost, in each heuristic function execution, the largest time cost is on converting the list instance of the board into a numpy array, and the rest computations all have much smaller scale costs. However, numpy functions are necessary in the current design of the heuristic function, so calling numpy array is unavoidable. That's how the conclusion that the depth limit cannot be further improved is drawn.

Heuristic function:

The current design of the heuristic function employs a complicated mechanism to calculate the score of each state. First, the function checks whether the state is a

terminal state. If it is, whether the current player is winning or losing is checked, and former one receives an extremely large positive reward and latter one receives an extremely large negative penalty (±100000). If the state is not terminal, further calculation is executed.

First, the function captures the coordinates of the TEMPs of both the current player and its opponent. Then, use the current positions of the current player and the opponent to compare with the coordinates of the TEMPs and apply rewards or penalties accordingly. Only the minimum distances are recorded. If the minimum distance between the opponent's position and the current player's TEMPs is smaller or equal to 1, this means for the opponent's next movement, it can hit my TEMP and win the game, so a 100-point penalty is applied. Vice versa, if the minimum distance between the opponent's TEMPs and the current player's position is smaller or equal to 1, it means the current player may have a chance to win the game by hitting the opponent's TEMP next movement, so 100-point rewards are applied. Initially, the latter reward was not implemented so my bot won't actively attack and always plays a safe strategy. But if attacking reward is not added in the heuristic function, the bot also won't be able to defend wisely due to the nature of the Minimax algorithm.

Furthermore, an additional 20-point penalty is added if the current player's position is not adjacent to any PERM grid or is completely surrounded by PERM grids. This was not implemented until the Milestone 4. The former is used to avoid situations when the opponent or a white walker is nearby and the current player can't get to a safe place in one movement. The latter is used to avoid situations when the

current player goes deeply into its whole PERM grids and cannot find a way out due to the depth of the Minimax algorithm being limited (will be further explained in the later section).

Lastly, the final evaluation score is majorly calculated based on the number of PERM and TEMP grids of the current player. The score is designed to be 3 times the number of PERM grids plus 1 times the number of TEMP grids. This weighting allocation is the result of intensive experiments, and it is so far the best strategy to guarantee that the bot will expand in a reasonably rapid way and will not be so risky that the depth of 6 cannot handle. The final evaluation of each game state is this score plus the rewards or penalties mentioned above.

Edge case handling:

Using the implementation of the Minimax algorithm with a heuristic function described above, the bot's game strategy pattern is similar to TA Bot 1. However, there are limitations to using the Minimax algorithm (which will be further discussed in the later section), so some edge case handling strategies are added to further improve the performance.

Firstly, under certain circumstances, the depth limit of the Minimax algorithm will be reduced to 1. This is because the Minimax algorithm assumes the opponent is playing under the same strategy but in reality, that’s not the case. That’s why purely relying on the Minimax algorithm will let the bot make unwise choices, especially in some dangerous edge cases. These circumstances include: 1. The opponent’s position is adjacent (not including diagonally) to one of my TEMP grids. 2. The opponent’s

position is near (2 grids close) to my current position and I'm the first mover. 3. The white walker's position is adjacent (including diagonally) to one of my TEMP grids or my current position. It is easy to tell that these circumstances are defined to directly avoid sudden death. Theoretically speaking, a completely correctly designed Minimax algorithm should be able to avoid these situations directly, but there might be some unknown flaws in my current implementation that stop it from being so (which will be further discussed in a later section). These edge cases are especially effective when facing bots with aggressive attacking strategies, including Attack Bot, TA Bot 1 and 2.

Another very similar edge case is also included. When my position is adjacent to the opponent's position and it's now my turn to move, directly hit the opponent to win the game. This edge case is defined because in the provided get_safe_actions() function, the opponent's position is considered as an unsafe movement, but in the actual game execution, hitting the opponent will win the game. The effect of this implementation can also be replaced by changing the get_safe_actions() function, but I chose the approach above because it also provides value for enhancing efficiency by avoiding calling the Minimax algorithm.

The last edge case included is the beginning of each game. This is specifically designed against TA Bot 2. Under the implementation of the Minimax algorithm, the heuristic function, and the edge case handling above, the current bot design is good at attacking or guarding against the opponent or white walker. However, compared to all the other bots, TA Bot 2 has a much wiser and more aggressive expansion strategy, which leads to it always winning over my bot by claiming more grids. This is

especially obvious at the beginning of each game, where TA Bot 2 always chooses to go in one direction with many steps and then return, claiming a linear shape territory, which is very efficient. However, this pattern is not reproducible under the current Minimax design. Because the depth is limited, although the linear movement at the beginning is the most efficient approach, the Minimax algorithm cannot foresee the state when we return and claim the land as PERMs, therefore it's likely to later randomly move and lead to death. In addition, the performance of my bot is heavily dependent on the territory we already claimed, because it has a relatively conservative strategy compared to TA Bot 2, the larger the claimed territory is, the faster it can be to claim more grids. Therefore, under these considerations, I decided to manually implement the beginning strategies in a way that is similar to TA Bot 2 to claim a linear territory, based on whether my bot is the first mover or the second, and whether there is a white walker in the map. So in my current bot design, the first several steps are hard-coded based on the game situation, and then the Minimax algorithm is used for further decisions. Under the current implementation described above in this section, the bot can pass the required winning rate thresholds in Milestone 4.

Shortcomings:

The shortcomings of my current bot design will be discussed in three topics: unknown bugs and glitches, heuristic function's scope, and the limitation of the current approach.

Unknown bugs and glitches:

As mentioned in the section above, many edge case handling strategies that are added are theoretically not needed in addition to the Minimax algorithm. However, the real experiments show that they actually significantly enhance the performance of the bot, which means there might be some issues within the current implementation of the Minimax algorithm. Due to the deadline approaching, I don't have enough time to accurately locate them, but they might exist in the process of switching max and min selection and the calculation of the heuristic function. The negative consequence of these flaws is mainly that the terminal states are not handled correctly, which creates the need for edge case handling strategies.

Heuristic function's scope:

The current heuristic function already includes the relationships between the positions of the current player, the opponent, and the TEMP and PERM grids. Due to the open-ended nature of the heuristic function, there must exist better designs that are not yet used here. To further revise it, a better understanding of the game and more experiments are needed. A better heuristic function can further improve the performance a lot. Also, although mentioned earlier that the time efficiency seems to be maximized, there might be ways to further improve it (maybe avoid using numpy), which can lead the depth limit to increase and let the bot choose more wisely.

Limitation of the current approach:

Since the beginning of my work on this final project, I have realized that adversarial search has less potential compared to MCTS + RL, and I still hold this belief today. No matter what optimization is implemented, the search depth of the

Minimax algorithm is always fixed and limited. In such complicated gameplay, where there are almost uncountable possible possibilities (similar to Go), the ability of adversarial search is certainly restricted. To achieve better performance, a new approach is needed, such as using reinforcement learning.

Future work:

To further improve the performance of my bot, my future work closely corresponds to the shortcomings covered above. The most urgent need is to find and fix the unknown bugs so that the edge cases don't need to be handled manually. Hard-coded strategies are always not comprehensive enough compared to a proper algorithm. In addition, with more experiments and a better understanding of the game, a wiser heuristic function can be designed to further improve the bot.

Most importantly, since several bots are now implemented, such as my own bot based on adversarial search, TA Bots, and more, letting them play against each other can create a valuable game history database, which can be used for supervised learning. On the base of that, the original plan of implementing MCTS + RL might become realizable. Like Alpha Go and many other AI bots for gameplaying, this can achieve much higher potential compared to adversarial search and should be the ultimate solution to this project.

Appendix (READMEs for the previous milestones):

Milestone 1:

For the development of my AI bot, here's my current plan: I will initially use Adversarial Search because I'm more familiar with it. The Minimax algorithm with alpha-beta pruning should be enough to meet the criteria of Part 2. However, because the game's rules are complex, for the later milestones with higher winning rate requirements, I think I will use the Monte Carlo Tree Search algorithm instead to reach a higher potential. When using MCTS, the performance is significantly influenced by the heuristic function, and designing a good h function can be challenging or even unfeasible in this case because of the complexity of the game, so I think I will incorporate reinforcement learning with the MCTS to enhance its capabilities further. However, I haven't figured out a way to train the model since we don't have any given gaming data.

Milestone 2:

During the Part 2 implementation, I initially used MCTS with a heuristic function, but the performance was really bad. I realized in order to achieve better performance, RL is required if I want to use MCTS. However, after careful reflection on it, I think the required computational power is too large. So I eventually switched to using adversarial search. The current implementation uses Minimax with alpha-beta pruning. The maximum depth is set to 6, and the current heuristic function mainly considers the number of PERM and TEMP slots (a PERM worth 2 times more than a TEMP), the distance between my TEMP and the opponent, and the distance between me and opponent's temp. As the plan for the future, I'll try to optimize the

current algorithm to increase the maximum depth while keeping within 1 second time limit. Also, let the heuristic function be more accurate. One current limitation to solve is that the minimax algorithm assumes the opponent is using the same logic, which makes it not stable when dealing with opponents with different strategies and styles. Right now this is handled as a special case, when the enemy is close to my TEMP or me, the maximum depth will be reduced to 1 for a more urgent reaction. One possible better solution is to based on the opponent's current pattern guess its strategy. Also, I want to factor in that player 1 starts with top-left and player 2 starts with bottom-right.

Milestone 3:

During the Part 3 implementation, frankly, there's no major change in the design, because the implementation during Part 2 appears to be good enough to satisfy most of the criteria for this milestone (on RandBot, AttackBot, and SafeBot). But there are indeed some minor optimizations and debugging made in order to better perform against TABot1. Firstly, the h function has been revised in a way that better handles attacking: when the opponent's temps are close, my bot now will get close and try to attack (realized by a positive reward of getting close to the opponent's temps). Therefore, now the bot can both attack more smartly and also better protect against attacks since we are using the Minimax algorithm. Additionally, outside of the Minimax algorithm, in the decide() function, I added a function to handle some edge cases, because the performance of my bot against TABot1 was particularly bad in small maps. The edge case I'm currently handling is when it's our turn and the

opponent is nearby, we should directly eat it instead of continuing to expand in order to win immediately. This edge case handling is unexpectedly effective, which leads me to think of adding more similar handings in future implementations. Also, I run my bot against TABot2, although there's no requirement for that in this milestone. The edge case attacking also works well here but some more edge cases should be considered to protect better against attacks. Also, especially in large maps, TABot2's strategy of expanding is very aggressive and smart. The current bot's expansion might be too conservative, so the logic might need to be revised. Also, we need to consider hyenas in our next milestone (which is involved now but for sure needs to be revised). These compose the TODO list for now.