**Game of The Generals AI**

The implementation of the AI in Game of The Generals (GOG) uses a version of Monte Carlo Tree Search (MCTS) that was modified to fit the board game. The algorithm covered first starting will be spawning, then the MCTS algorithm for finding good moves. The challenges encountered and techniques used will be covered last.

**Spawning**

There are two stages that the AI does to generate a starting spawn for all its pieces. The second stage is where the AI chooses from 2 types of spawning.

First Stage

The first stage is when it spawns the flag in either the back row or the middle row. According to guides, it is recommended to spawn the flag on the back row, but to introduce some randomness such that the player does not know that the flag will always be at the back, the flag has a 75% chance to spawn at the back row and a 25% chance to spawn on the middle row. After it spawns the flag, it is also recommended to spawn a few pieces to “guard” the flag on its sides or on its front. For this implementation, there are 3 pieces that guards the flag: 1 spy, 1 private, and 1 3/4/5-star general, where each n-star general is equally as likely to be chosen to be spawned near the flag.

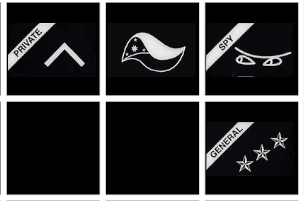
****

Fig. 1. Flag was spawned with guard pieces

Second Stage

After the first stage, the AI then moves on to the second stage where it decides to either do 2 approaches: “Conservative” or “Blitz”, both having a 50% chance to be chosen to keep the player on his toes. For both “Conservative” and “Blitz” spawning, each piece’s rank has a value assigned to them. For example, the spy has a value of 7.50 and the Colonel has a value of 4.05. The ranks’ values are derived from the GOG manual.

Conservative Spawning

If the AI chooses this method, the board is now divided into 3 “flanks”. It then tallies each piece that is spawned on each flank. For example, if a spy is spawned on the left flank, the left flank’s total value is increased by 7.50. This also considers the pieces spawned on the first stage of spawning. While there are pieces that are still not spawned on the board, it chooses the least valued flank and spawns a piece that has not yet been spawned. This repeats until there are no more pieces.



Fig. 2. Spawns pieces such that each flank is close in sum value

Blitz Spawning

If the AI chooses this method, the board chooses 1 flank and concentrates most of the most valuable pieces in that chosen flank. In detail, it sorts the remaining pieces that have not been spawned yet in descending order and keeps spawning on the flank until it is full. It then moves on to randomly spawning the other pieces on the other flanks.



Fig. 3. Spawns most of the strong pieces in one flank

**Monte Carlo Tree Search**

Since in this game the other player’s pieces are hidden, this makes it an imperfect information game. The coded AI must consider simulations of alternate outcomes and must also keep track of information discovered from past board states.

Keeping track of the other player’s hidden pieces

The AI keeps track of each of the hidden piece’s possible ranks and a pool of pieces. Each time there is a “battle” between 2 pieces, the AI will be able to acquire some information about the other player’s hidden piece. It could either be higher or lower in rank, a spy, or a private depending on if the AI’s piece wins or loses the battle. It eliminates all the possibilities of what the hidden piece could be. A piece is “discovered” when there is only 1 possibility left. After a piece is discovered, that rank is removed from the pool of pieces. This pool exists so that when a rank is removed from the pool, no other piece can have the same rank. For example, when the AI discovers the player’s 5-star general, the rank is removed from the pool of pieces. This then implies that no other piece can *possibly* be that rank and the 5-star general rank is removed from the list of possibilities of all other pieces.

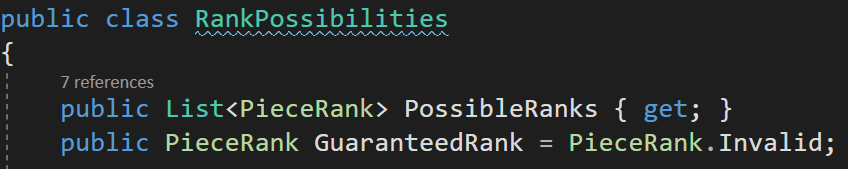


Fig. 4. Each hidden piece will have a list of possible ranks

Searching and search space

The AI must look at each of its valid moves and evaluate them using a heuristic which will be discussed on the next section. For each of its valid moves, the AI will explore all counter moves, which will then repeat recursively until a fixed depth. The maximum depth set in this game is 3 which only takes the AI a few seconds to find a good move. A maximum depth of 4 takes the AI about 25 seconds which is good for harder difficulties.

For each board state down the search tree, it evaluates the board to a heuristic that is unique to the AI and to the player. With this evaluated score, it is subtracted by the average evaluated score of all the other player’s board states. This recursively repeats until the maximum depth. Finding averages only applies to the levels of the tree below the first level because on the first level, the AI must choose one move so it can not average the results out of all the moves.

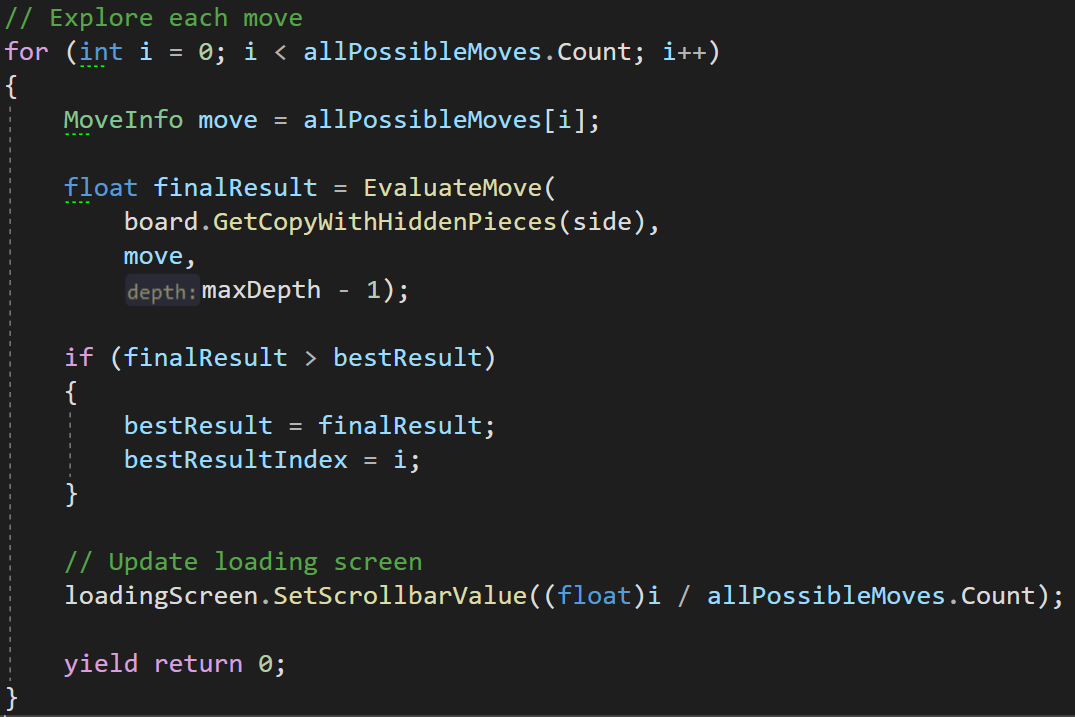


Fig. 5. The AI explores every move and chooses the one with the highest evaluation

Simulating unknown outcomes

In this game, when a piece battles another piece, it is uncertain whether your piece will win or lose the battle. This is a problem which is solved from keeping track of each hidden piece’s rank possibilities. It is simple enough to move a piece without invoking a “battle” between 2 pieces, but if the move requires a battle, then the AI must simulate what happens for both winning and losing the battle. Using the list of possible ranks, the AI calculates its chance of its piece winning against a piece that is hidden based on the other piece’s possibilities. It then multiplies the chance of it winning and losing with the heuristics of the outcome of winning and losing respectively and adding both of those together. For example, when the AI’s piece has a 75% chance of winning against the human’s piece, the evaluated score of the outcome where the piece won the battle is multiplied by 0.75 while the evaluated score for outcome of losing is multiplied by 0.25.

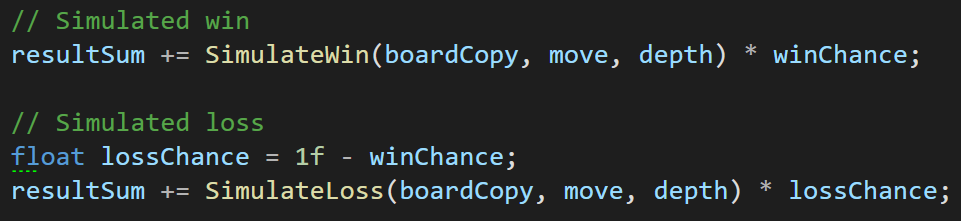


Fig. 6. Both outcomes are simulated

Heuristics

When a board state needs to be evaluated, there are multiple variables that must be considered. One is that the evaluation for the AI is different from the evaluation of the other player which is the human. This is because of the inherent limitations of the game. The AI does not know the pieces of the other player so it must evaluate the other player’s board state slightly differently.

Both of evaluating the AI and the other player’s board state include “Openness” and “Aggressiveness”. Openness measures how many valid moves the player can do. Generally, the more options a player has, the better state that player is. Aggressiveness is a measure of how much a piece is willing to attack another enemy piece. Because the AI keeps track of enemy pieces, it can calculate its chance of winning. With a high win chance, the AI is incentivized to attack. Inversely, with a low win chance, the AI is then disincentivized to attack.

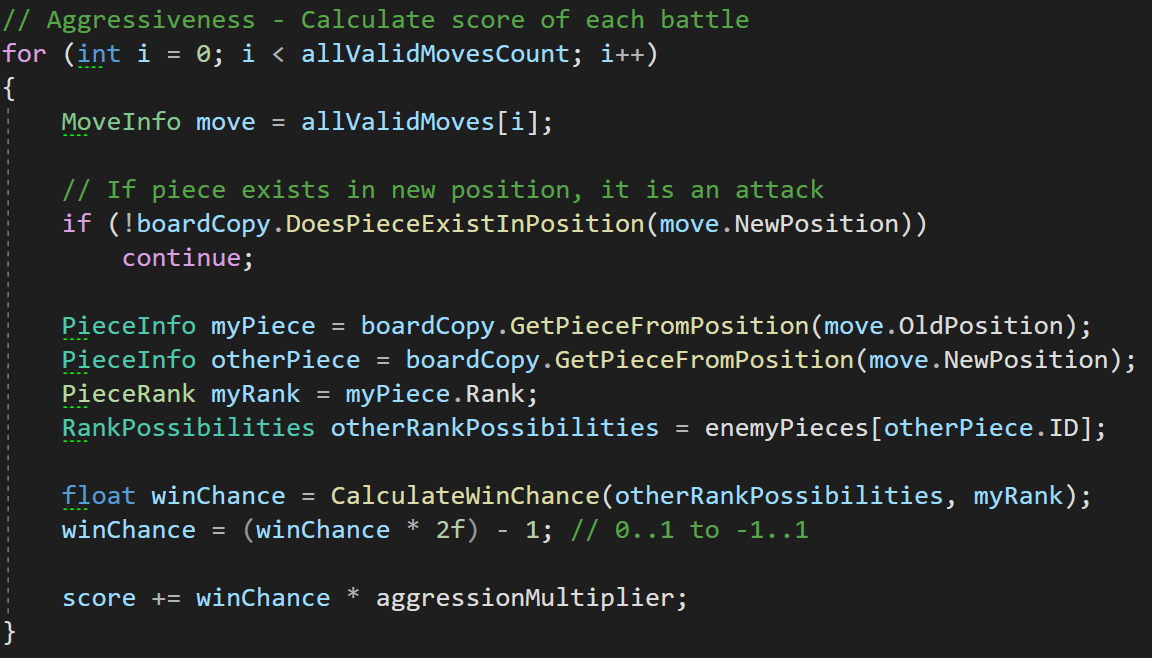


Fig. 7. Aggressiveness as “Offensiveness” and “Defensiveness” combined

Another way of viewing it is that it is a combination of “Offensiveness” and “Defensiveness” where it attacks when it is incentivized enough to do so and finds alternative moves when the chance of winning the battle is low or until more information is found. Values above 50%-win chance yields a positive number, which means that it incentivizes an attack. Values below 50% yields a negative number, which disincentivizes attacking.

The algorithm incentivizes having the flag be out-of-reach from the other player. The difference with evaluations for the AI and the other player is that the AI does not know where the flag of the other player is. The AI can only evaluate for the flag’s safety of its own pieces only. This is a severe hindrance to the quality of the AI, but it is an inherent limitation to the game. Of course, the AI could also “discover” the flag for instances such as there is only one remaining piece on the board left but for most of the game, the AI will not know where the flag is. This was the reason why evaluations for the other player’s board state does not include the flag’s safety.

**Challenges Encountered**

Deep Copy or Undo

To perform searches, I need to look at multiple future board states. This means that the algorithm must have a data structure that represents the board. I had two options: copying the board for each state down the search or copying the board only once and just undoing the action when backpropagating. A pragmatic approach to this would be creating an implementation for both and profiling the performance but this is tedious and would take much time. I decided to just copy the board for each move because an “undo” system would introduce much unneeded complexity.

Forwardness heuristic

After some playtesting, it can be observed that when there are only one or two pieces left that is out of reach from the AI, it does not know what to do and plays randomly. This can simply be alleviated by increasing the maximum depth of the tree search so that it is able to evaluate the winning moves that will finish the game, but this will severely impact the time it takes to run the algorithm. The simple solution that was implemented was that there is a slight incentive to move forward. This is added to the evaluated score for moves that move a piece forward. This makes it such that when there are no moves can finish the game, the AI will just push forward towards the player.

**Techniques**

Guaranteed loss

Since the AI tracks the other player’s hidden pieces, it can often figure out what a particular piece can be. With this knowledge, we can offer an optimization to the search algorithm where if it can tell that an attacking move is *guaranteed* to lose, then it does not expand and simulate that move. The code uses “less than 0.01” and not “equal to 0” to guard against floating-point precision errors.

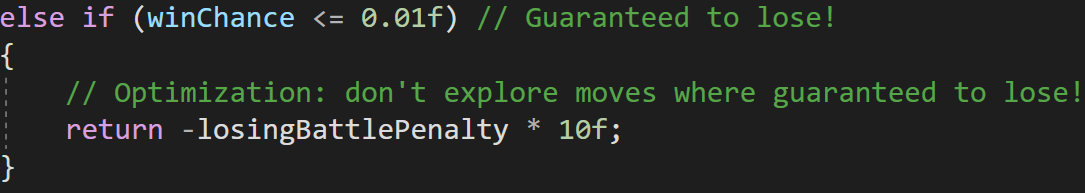


Fig. 8. Situations of a piece is guaranteed to lose are skipped

Guarding the AI against cheating

To make sure that the AI does not cheat, when the board is copied to test and evaluate future board states, the board “invalidates” the human player’s pieces. This is just to be sure that it does not cheat. It is also good for debugging because the game does not know how to handle pieces that have “Invalid” ranks.



Fig. 9. Deep copies the board but hides the pieces of the other player



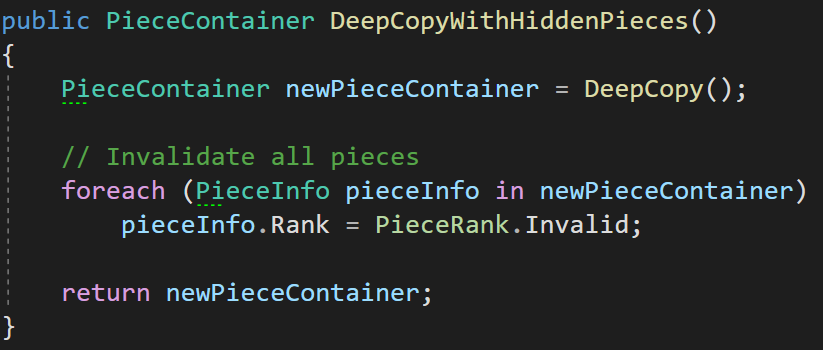


Fig. 10. Sets the rank of all pieces to “Invalid”