Comp 424
Project Report

Spiros Mavroidakos
260689391

April 8, 2019

1 Program Specifications and Motivations

The way in which the program works is relatively simple yet effective. The program begins in the $chooseMove(...)$ method of the given $StudentPlayer$ class template. This method will instantiate an instance of the $ZeroSystem$ class and run the $EngageZeroSystem()$ method from the newly instantiated object to get the best move. $ZeroSystem$ is the given codename of the implemented AI agent and the AI agent will be refered to as the zero system throughout the report.
The $ZeroSystem$ class takes in a $PentagoBoardState$ as a constructor argument. This board state will be the current state of the board (i.e. the same one given as input to the $chooseMove(..)$ method of the $StudentPlayer$ class). The $EngageZeroSystem()$ object method will begin by initializing a variable that will be used to store the best win/loss ratio found and another variable to store a candidate move corresponding to that best ratio. It will then run a Monte-Carlo style tree search in order to determine the next move.
This method is where the descent phase of the Monte-Carlo search begins. The tree policy is a simple one, a move is deemed promising and therefore expanded on if applying the move will not lead to the opponent winning on their turn. The motivation behind this is that we wish to expand the most amount of nodes from the current state but we do not want to waste time gathering statistics for a move that will automatically lead to a loss if the opponent is playing an optimal game. Prior to checking if our opponent can win, the program checks to see if the applied move has made us win the game and if it did, it simply returns the move as there is no point in continuing as we have won. The descent phase is implemented in the program by getting all the legal moves available from the current state of the board (that given in the constructor) and iterating through them. The move iteration corresponds to doing the checks outlined above and if neede, we move on to the rollout phase.
The rollout phase corresponds to the $EngageZeroSytem()$ method calling the $PerformMonteCarloSimulation(...)$ method of the $ZeroSystem$ class. The purpose of this method is to run a certain amount of trials (limited by the constant $TrialLimit$ in the $ZeroSystem$ class which is set to 300). A trial is over when engame is reached. Endgame is either a win, loss or a draw. Once the trial limit has been reached, the simulation will end and the the result of the simulation will be returned to the $EngageZeroSytem$ method for processing. A trial limit has been set in order to avoid going through every possible move which will inevitably lead to a timeout and therefore an automatic loss. The $PerformMonteCarloSimulation(...)$ method takes in a board state, a move and a $SimulationResult$ object which is a custom object that will keep track of the wins, losses and draws for a simulation.

This method will apply the move and then check if a winner was determined. If a win, loss or draw occured, then the *SimulationResult* object will be updated accordingly and this result update is one part of the update phase. If engame has not been reached, then the simulation goes on. At this point, the program checks if the opponent can make a move that leads directly to a victory (i.e. opponent wins instantly by applying a move after the one we just did). If the opponent wins, then the porgram gives a heavy loss penalty (i.e. 15 losses in this case) as checking all possible opponent moves is an expensive operation so this type of penalty is needed to avoid a timeout. If this is not the case, then a random opponent move is chosen and applied. The program then checks for a winner and updates the results if needed (another instance of the update phase). If no winner has been determined, then we get all legal moves and iterate through them. The iteration corresponds to recursively calling the $PerformMonteCarloSimulation(...)$ method. This recursion will continuing untill all moves have been exhausted or the trial limit is reached.
The update phase occures when engame is reached and the *SimulationResult* object is updated. Updating the result object is enough to backpropagate the reward of a trial as the object will be manipulated by all the trials for a particular move (The move in question is one of the candidate moves, i.e. a legal move from the board inputed to the $ZeroSystemconstructor$).
The growth phase of Monte-Carlo search in this case is skipped. In this implementation of Monte-Carlo search, the rollout phase will be continuous. It will be continuous in the sense that once an endgame is reached, the program will return to the level above it in the tree and begin a new trial from there until the trial limit has been reached or endgame is achieved and so on. The motivation for this has to do with the fact that timeouts may occur if we return to add the first state of the rollout to the tree as per the growth phase. Furthermore, the use of the *SimulationResult* object allows for a continuous update of the statistics and therefore the statistics do not need to be initialized by the growth phase for each rollout as it is done on the fly.
Once the maximum amount of trials limit has been reached, the program returns the results of the trials to $EngageZeroSystem(...)$. The results are then processed by comparing the win/loss ratio to that of the best ratio currently found. If the ratio for the newest run is better than the current best, then we save this new ratio and make this move the candidate move. The candidate move will be returned to the *chooseMove* method of the *StudentPlayer* class as the move to be applied.
A Monte-Carlo style search was chosen in this case due to the fact that it is an interesting approach to game playing. Due to the timeout constraits, it also seemed like a safe bet as it can easily be tuned to support time limits by limiting the amount

of trials being done.

2 Theoretical basis of the approach

As stated in the prior section, a variant of Monte-Carlo tree search(MCST) was used. MCST uses a tree policy in order to choose a promising leaf node from the root state. Once a promising node is discovered, the default policy is applied for both players which corresponds to running a simulation until the endgame The value of a move is then obtained by analysing the results of the simulation and the move with the best statistics is chosen. MCTS usually has 4 phases: The first being the descent phase which corresponds to applying the tree policy. The second being the rollout phase which is running the default policy from the promising nodes found in the descent phase. Phase 3 is the update phase where all the statistics for each node is updated. Finally, we have the growth phase which corresponds to adding the first state in the rollout to the tree and initializing its statistics.
The above corresponds to the regular implementation of MCST. The one applied here disregards the growth phase and makes some other minor changes to suit the situation. The descent phase is used to choose promising nodes to explore with the only criteria being that if we apply our move, then our opponent does not win on there move. The descent phase will therefore only reach nodes that are exactly one move ahead of our current state. The rollout phase will then run the default policy described in the prior section. Finally, the update phase is done at the end of every trial which occurs when a win, loss or draw is determined. The statistics are automatically updated to the top of the tree by the use of the *SimulationResult* object. The growth phase is ignored due to the timeout constraints given. This implementation makes it so that the growth phase does not need to happen as the program only uses the win/loss ratio of the promising nodes (those added by the descent phase) to determine the next move. With that in mind, it would be a waste to return to the level directly below the descent phase nodes. If the program kept returning to the promising nodes, then it would increase the probability of timing out. The other difference if that once a endgame has been reached, a simulation does not begin at the promising node, but at the parent of the engame node to avoid timing out.

3 Advantages, Disadvantages, Expected Failure Modes and Weaknesses

The advantages of this approach is that it is simple, quick, uses minimal memory and uses statistics to make a decision. An advantage is that nodes that lead to losses in 1 move are pruned. However, there are many disadvantages as well. The main Disadvantage is that when exploring nodes in the rollout phase, the opponent's moves are random and therefore the validity of the statistics become questionable as a random move is not an optimal move. Furthermore, there is no type of pruning being done based on the "goodness" or score of a state. The only nodes that are pruned are thoses that lead directly to a loss. The weakness of this pruning method is that this pruning is only effective if we can guard against the win. It is possible that the win is guaranteed for the opponent (i.e. more than 1 way to win) and thus this pruning does not stop the opponent from reaching a state where they set themselves up to be in the unguardable position. A weakness of the approach is the fact that the descent phase only evaluates the simplest criteria for leaf node expansion. If the descent phase had more criteria like a state score to represent the "goodness" of the state, then the gathered statistics would be more meaningfull. Also, the next move is chosen by a win/loss ratio which is not the most effective heuristic. Also, the fact that the win/loss ratio is only counted at the level below the root node is also problematic as we lose information as to the moves that led to each win, loss or draw. Some moves might have worse ratios but can overall be better moves in the long run.

4 Other approaches

The first iteration of the Zero System was a pure implementation of MCTS which used all 4 phases as they were outlined in section 2. However, this lead to too many timeouts and the only way to avoid them was to reduce the amount of trials to a very small number. The problem was the growth phase and going back up the tree after each rollout. The growth phase requires you to go all the way back up the tree to add the first state in the rollout to the tree. However, this takes time and always resulted in timeouts. To avoid this, the number of simulations was limited to a very small amount. This lead to the implementation of the current approach wich completely ignores the growth phase so the tree does not grow. When the two agents faced off, the current approach defeated the original every time due to the lack of information in the original. The original simply did not have enough statistics to

make good guesses on the best moves. Many refinements were attempted to try and make the original feasible but the amount of simulations being run was not enough to compete with the current approach.

5 Improving the Player

One of the biggest improvements that could be done would be to limit the amount of nodes being explored by the descent phase. To make this happen, a heuristic function could be written in order to score each of the states that are reachable from the current one and then we can chose a subset of them to explore. Another improvement would be to change the way in which how the opponents moves are chosen. Currently, the opponents moves are random which is not representative of an actual player which can lead to faulty statistics. The opponents moves could be chosen using a heuristic function that evaluates the "goodness" of a state by analysing the placement of the pieces for example.