Team Name: beta_come
Danielle Zhang (833445)
Ishan Sodhi (946124)

Algorithms are fun!

Artificial Intelligence Assignment 2
Report

We built an AI agent for Chexers that uses a modified MaxN algorithm with a search tree of depth 3. Reinforcement learning is applied to tune the weights of the evaluation function.

The Structure

We have created the following classes in our program:

1. Node: This class represents the structure of the nodes of the tree on which we apply the Max^n algorithm for decision making. The children for the parent nodes and the overall tree is generated by considering all the possible actions for each of the 4 players we are playing as. The node contains a representation of the game with a State object, and a vector that contains the evaluation values from each player's perspective.

2. State: This class represents the state of the game from the perspective of the given player. It includes a board representation that has the coordinates of the pieces of each player, a dictionary with counts of the number of pieces that have exited the board for each player, and the action taken by the previous player to reach to this state.

3. MaxNPlayer: This class contains the functions pertaining to our player and is being imported to play the game. The functions are described as follows:
    - init: this function is called at the beginning of the game to initialize our player.
    - Action: This function is called every time an action is requested from our agent. An action is chosen by our agent to be played.
    - Update: this function is called at the end of each player's turn and provides us a chance to update our board representation after each action.

Overall, the way our program makes intelligent decisions on how to play the game is divided into three steps:

1. Strategy determination: Check if our agent is using the maxN algorithm or the greedy strategy.

If our agent chooses to use MaxN algorithm to decide the action:

2. Decision tree generation: A tree is generated with each node representing an action made by a player, with each depth of the tree corresponding to one specific player. We generate a tree only upto depth 3 as the branching factor for the players is very high and we want to keep the number of comparisons to a balanced level. Each node of this tree has an evaluation vector associated with it. The evaluation vector contains evaluation

values from every player's perspective.

Our evaluation function uses the following features of the state of the game from the given player's perspective:

a. Number of missing pieces: This is a measure of the number of our player's pieces that have been converted into another player's pieces. We want to minimize this to 0 as we cannot win the game without our 4 pieces exiting.

b. Number of pieces left: This is a measure of the number of our agent's pieces that are left to exit the board, we want to minimize this as to make our pieces exit the board as quickly as possible.

c. Averaged exit distance: This is the average of the distances to the exit positions from all our agent's pieces. We want to minimize the distances of our pieces to the exit positions so our pieces can exit the board as quickly as possible.

d. Missing opponent pieces: The sum of the number of missing pieces for all the opponents. We want to maximise this to encourage the player to convert more opponent pieces, which decreases the probability of the opponents winning and increases the probability of our player winning since we have more pieces and more actions to consider.

We want to minimise all the above evaluation features except missing opponent pieces. Hence the initial weights are all negative except for missing opponent pieces.

3. MaxN algorithm: To manage the 3 player nature of Chexers, we used MaxN algorithm as our main strategy. MaxN algorithm is based on the principle that each player chooses a move which gives them the maximum evaluation in the evaluated vectors. This operates by recursively moving down the tree to find the maximum evaluation for each player.

If our agent chooses to use the greedy strategy:

2. Generate and evaluate all possible actions: All the possible actions of our agent are generated and their consequential states are evaluated by the evaluation function above.

3. Be greedy: Find the action with the highest evaluation value.

Machine learning

We applied Machine learning to tune the evaluation function feature weights: We applied reinforcement learning to optimize our evaluation function weights. The basic structure of our machine learning program is:

1. Play a game of Chexers with weights W. Store every action's feature values and evaluation values.

2. Based on the game result, a discounted reward is added to every evaluation value.
3. We obtain new weights W_new by the feature values and the new evaluation values using linear regression.
4. Update the weight: W += learning_rate x (W_new - W)
5. Repeat.

We assume that later the state is in the sequence of states in the game, the less it contributes to the outcome of the game's result. This assumption is made as we observe that in later states of the game, some players have a lot more pieces than the start of the game due to converting opponent's pieces with jump action, there is a smaller chance for player in disadvantage due to the lack of pieces to convert enough pieces back in order to win the game. After one round of game is played, for each player's evaluation values, a reward is decided based on the game's result. If the player wins, the reward is 1, otherwise, the reward is -1. Then for each evaluation value, if it is generated by the player's nth action, it will have a discount rate of 0.99^(n-1), which decreases as n increases. Then the reward multiplied by discount rate is added to the evaluation values.

We chose linear regression with least square as the cost function that we try to minimise for two main reasons: It is easy and straight forwards to implement, and intuitively it fits well with our evaluation function which is also linear. For every game, we regress the evaluation feature values and the new evaluation values 3 times to increase the fit.

For the games that are played for machine learning purpose, we use 3 different combinations of players: beta_come with 2 random players,2 beta_come with 1 random player, beta_come with 1 random player and 1 greedy player (greedy player uses the same evaluation feature and function as beta_come to be greedy). Since there is no randomness in beta_come or greedy player, we haven't included the combinations of 3 beta_come, and 1 beta_come with 2 greedy players as the variety of states they explore are very limited.

Due to a time limitation, instead of training the evaluation function until the weights converge, we repeated the process 500 times. The weights changed from (-1,-10,-10,1) to (-0.60735,-6.07406,-6.07713,0.61072).

Combine MaxN and greedy strategy

After a few round of games being played, there are two observations made about the nature of the game:

1. At the beginning of the game, the players don't interact with each other.
2. Towards the end of the game, getting the pieces closest to the exit positions off the board is the priority.

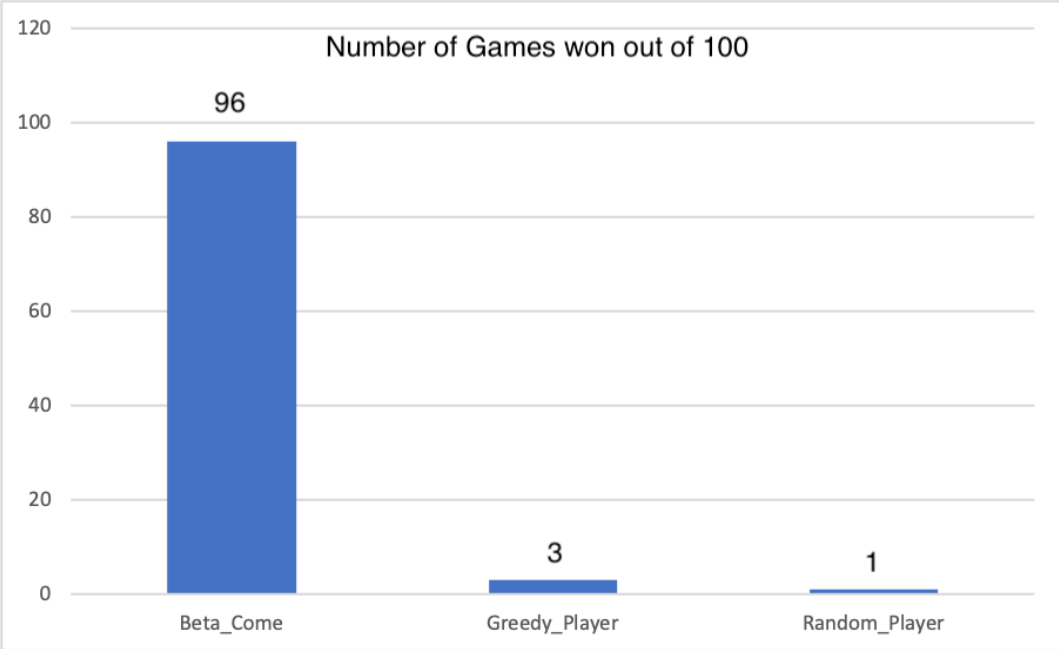
In this two scenarios, our agent doesn't necessarily need to take what other players' action into account as they have little impact on our agent's pieces. Hence a greedy strategy is also

implemented and it can be turned on and off depending on the state of the game. The agent will start with playing greedy until an opponent's piece is 2 hexes away. Later in the game, if the number of pieces on the board of our agent is higher than the number of pieces need to exit for our agent to win, greedy mode is turned back on again. This combined strategy saves the overall time taken for our agent to play as greedy strategy is a lot faster.

Reduce time complexity for maxN

MaxN is an expensive search algorithm to run. For chexer, assuming that the number of pieces each player has on the board stays as 4, and each piece has 6 neighbours to move to, MaxN has a branching factor of 24. If the max^n search tree has a depth of n, the time complexity isO(24^n). Since later in the game, some players can have more than 4 pieces as they perform jump actions, the branching factor is higher than 24. Hence to reduce the time complexity for MaxN, we have to make some sacrifice. We chose the search tree depth to be 3 as it can see the possible actions from every player for one round. When the algorithm expands the tree, if the player of that tree depth has at least 4 pieces on the board, the algorithm only look at the possible actions of 3 pieces from the player that are the closest to the exit positions. This result in our agent usually having one piece left behind in the starting position while other pieces move forward. However, with the evaluation function encouraging jump action to convert other player's pieces, this sacrifice is more or less compensated.

Overall effectiveness of our program:



As can be observed from the graph, our agent beta_come won 96 out of a total of 100 games played against a Greedy Player and Random player. The greedy player chose its action at each stage based on the basis of immediate reward instead of long term strategic rewards. The random player generated moves at random at every stage. Overall, this lets us know that looking 3 steps ahead and utilizing the evaluation function enables our agent to perform far better than other forms of players.

Other strategy

We considered using MiniMax algorithm to do game search. However, in 3-player game, since each min-player is just a combination of actions, if we choose to expand the tree for n players taking action and b is the average branching factor, the number of states explore is still b^n, which result in a time complexity of O(b^n). The only difference it has with MaxN is the depth of the tree. Since it has the same time complexity as MaxN and makes the paranoid assumption, we decided that MiniMax does not represent the nature of the game well.

We tried implementing our previous A* algorithm to decide which actions to take or to effectively decrease the number of possible actions generated for the MaxN algorithm to work on. Our previous algorithm would devise a strategy as to how many least number of actions are required when our pieces worked together to win the game, but it actually ended up slowing down the game because of A* being run at every actions. Therefore, we didn't implement that in our current code.