COMP30024_2019_SM1

Artificial Intelligence Project B

• Introduction

In the second part of this Chexer game, it is supposed to design a game-playing algorithm as advanced as possible. In details, this project is not a simple agent like what had already done in part A, it should works on multiple-player Chexer game (it is three players this time).Therefore, in this work, there should be involved plenty of knowledge which is not used in previous as assessment, after all, the project B is much more difficult than last time.

In this report, it is going to demonstrate based on four following parts:

I. Firstly, in the next part of this report, it mainly focuses on the explanation of those basic knowledge which is utilized in this project and also has been taught from lectures.

II. For the second part (if including the introduction, it should be the third part of this report), in order to implement a more advanced and more efficient program, there must be other additional techniques which are tried to introduce in the implementation of this project. However, to some extent, they failed to be parts the final version of code. Since those learning process affected a lot on designing ideas and finally lead the group to the end of this project, it is necessary to have a discussion here.

III. Thirdly, this part is the main segment, which is used to explain the implementation of algorithms in practice, of this whole project.

IV. Finally, it shows the process of how the implementation becomes more and more optimal and effective from the beginning to end.

- Basic Knowledge from Classes

  - ***Max^n Algorithm***

All implementations are based on the Max^n algorithm, so that it is meaningful and so important to maintain the correct running of it. As the core of this project, it is extremely necessary to understand how the Max^n algorithm works.

According to the figure 2.1, it clearly shows the functions of multiple-player algorithm. In each tuple, there are three numbers representing weighted values for three players, therefore, it is a three-player game. Take the tuple (7, 3, 6) at root as an instance, in the project, these three values respectively stand as the current competitive status of red, green and blue players. If root is red player's turn, hence, the depth two represents green player's and the next is for blue. For implementation of Max^n algorithm, each player only collects the state with their highest corresponding value. Additionally, the choice of tuples is based on the principle of best actions for each player and the better action it is, the higher weighted score it will be. Therefore, blue player only needs to consider about the status from branches at depth by the third weighted value and so forth. Finally, the red player is going to choose the tuple which first value is 7, because it is more worth for red player to implement the second action rather than the 1's.

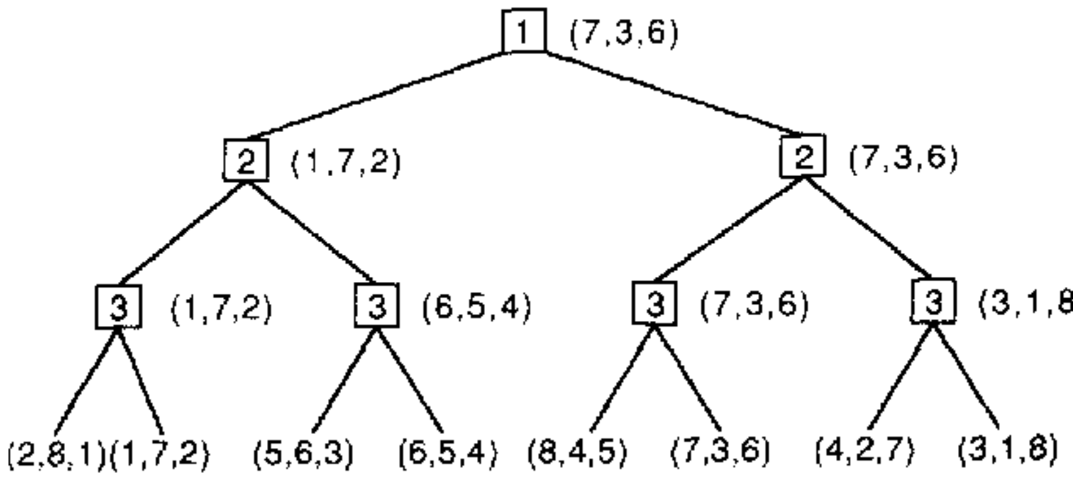

Figure 2.1 Max^n algorithm

  - ***Resource Limits***

In Resource Limits algorithm, it only works on the CPU and computer memory. In fact, if the Max^n tree can expend to the bottom, the agent can predict and compare all possible actions before making decisions, as a result, the agent can play like a master player. However, the number of Max^n tree's branches will have an exponential growth with extremely long processing time and even beyond the capability of a computer, so that the resource needs to be strictly limited by conditions. Therefore, in this project, the depth of Max^n tree should be seriously decided, in order to ensure the processing could be done during certain time and memory.

- ***Evaluation Function***

  Evaluation Function has the similar ability like Utility Function, nevertheless, they are still different. For the common point, both are used to generate values to convert practical status to visible results, which provides convenience to the further operations. For the Evaluation only, each value should be set by diverse features and features' corresponding weight, then the evaluation value is equal to the sum of all the products of each feature and its weight (Figure 2.2). In addition, Evaluation Function needs to follow three key points:

  1) Oder the terminal states in the same way as the true utility function.
  2) The computation must not take too long.
  3) The result of the evaluation value should strongly and correctly describe the actual advantages or disadvantages in that state.

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Figure 2.2 Evaluation Function formula

- ***Lazy Evaluation***

  Lazy Evaluation is a kind of pruning algorithm to improve the efficiency. For its functionality, it delays the evaluation value until that value is needed. In other words, when evaluating a state, the component of the vector will not be computed until it is used to compare with others. In Figure 2.3, while it comes to the depth * (blue player's turn), only the third evaluation value in each tuple is used to do the comparison. Hence, the remaining evaluation cannot affect anything to results, and then, they will be ignored during the comparison until current vector is chosen to the upper depth. Eventually, the time is saved due to the lack of those wasting reading values.

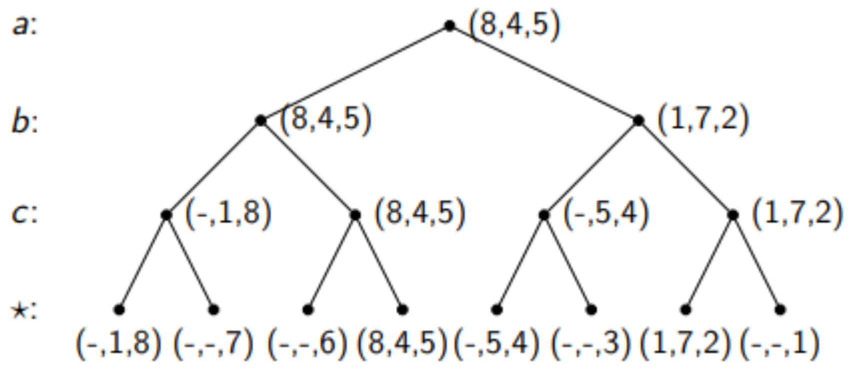

Figure 2.3 Lazy Evaluaiton

● Alternative Algorithms Attempt

➢ ***Minimax Algorithms***

Initially, Minimax algorithm is the first attempt before the choice of Max^n algorithm. Minimax algorithm computes the minimax decision from the current state and only works on two-player competitive games. Since this project is a three-player, it is necessary to introduce the Paranoid Reduction and Alliance Reduction.

Paranoid:

In order to reduce the number of players to simplify the game, it is supposed to regard all opponent players as a single and beat it. Take the red player as an example, in code, when initializing all players, players who are not red are considered as movable blocks and the aim of this game is trying to jump over blocks and exiting four pieces successfully.

Alliance:

In order to beat the strongest player, making a deal with the other enemy to cooperate each other, avoid battles between alliances and then eliminate the player who is closest to win. For alliance cooperation, regard alliance's pieces as safe and sub pieces which have the same colour with self's but is not moveable. Since in practice, alliances, most time, try not to hurt each other, in order to balance their own competitive ability and avoid to lose by the strongest player.

Drawbacks of Paranoid:

i. There would be a large number of branches and most of them might be wasteful, so that it has a hard requirement for the CPU, memory and time.
ii. Since the competitive players are both regarded as blocks, as a result, self cannot have a very clear view that who is going to win the game, who is currently battling with self and so one so forth. Therefore, it would be harder to figure out a strategy to conquer the current embarrassed situation.
iii. From overview, if the player is fortunate, this player may find out a path to exit. However, this is not a finding-shortest-path game and also can lose pieces, even lose the game, thus, the player can only wins game with hardly opportunity.

Drawbacks of Alliance:

i. Low cost to be alliances without any contract, thus, it would be extremely hard to distinguish which player is trustable.
ii. After battling on the public channel, it was found that most groups may not involve teamwork attempts, which causes failing to make a deal with others and even lost pieces as a gift. Totally speaking, it is really beyond the ability to make a team with others, after all, these AI communication does not seem like social communication so easy.

- ***Reinforcement Learning (TD leaf)***

  Reinforcement learning train machine to achieve high marks in their training environment. This process requires lots of attempts and learn to make future actions better. Machine is not provided with suggestions from the training environment, it only receives the award or penalties from the environment.

Q-learning, for instance, give penalty value for losing pieces, give award value for get closer to the destination. Machine have a chart of memory record the awards for actions of states. The new action will update the old by calculate the difference of the actual award and predicted award.

At first, Max^n was the not the first choice in this project. Since this is a kind of competitive game, if the agent can learn by itself and can decide rational actions, it will be the initial expectation for this game and the best result. Therefore, Temporal- Difference Lean algorithm was introduced in this project.

TD Leaf is a kind of machine learning and the most popular approaches in reinforcement learning. For TD leaf, there are still some features needed to give an evaluation value to its current state and the number of other players' exiting pieces was the feature to weight the evaluation value. Firstly, evaluation values have to be set those evaluation value manually, since there is not any absolutely correct features' weight which can be referenced. Secondly, before becoming a successful agent, it has to study from competitions, in order to adjust those manually setting values so that to make sure that evaluation values are close to the correct amount which should be in fact.

However, this creative and advanced algorithm did not work as good as what it was supposed to by many reasons. Before battling with others, the initial value of features was set as 1 and it played as a random player, since there was not any objective adjustment to decide which next action would be better. After hundreds of battles, the evaluation value raised up to roughly 11, and then, increased slowly with extremely small value of decimals. Until so far, this learning process helped the agent could successfully win random players but not other groups and even greedy players. Eventually, in order to keep competitive strength with other groups and Matt' s tests, this approach was abandoned and try other algorithms.

- ## Actual Implementations
  - ***Max^n Algorithm***

    In the actual implementation of Max^n tree in this project, its node is slightly different from the one which is demonstrated before. Currently, in each node, it is not only restoring the evaluation value of each different colour player, but also keeping the action for that current state. Take the Figure 2.3 as an example, if the node with evaluation vector (8, 4, 5) is chosen to the root from level b, the successfully corresponding red player's action will be passed to root as well. Therefore, it will be easier to detect possible actions for the next turn.

    In fact, there are five depths in the actual Maxn algorithm and they go deeper in hierarchy. For first three depth, they plays the same operation like what is mentioned before. When there is a tie meet in Maxn, additional evaluation is applied. Danger value shows the number of pieces in danger of a player, this calculation involved prediction up to two actions. The calculation for Cases is high in cost, and the weight for then is not so high, so we designed to calculate them when needed
  - ***Evaluation Function***

    Since it is a complex agent this time, there are plenty of features can impact on results. However, like what is discussed before, these features should be hierarchical, which means there must be some vital features or some can affect the prediction but not that much. For the most essential feature is the number of pieces, since, the insufficient number of pieces will never lead to a champion. In addition, there are other details can control the game tendency. According to the conclusion from the last segment, the danger level, killed capacity and safety environment are considered before the next-action output and all scores are showed in Figure 4.1.

    The features in evaluation:

    Num_pieces: the total number of pieces of a player, getting higher when the player eat pieces, and getting lower when losing pieces.

    Score: score is 0 when no exit action made and it is 1 when an exit action is made.

    Distance_to_exit: -1 * the sum value of all pieces of a player to exit hexes.

    The features in additional evaluation:

Danger: -1 * number of pieces is exposed to enemy without the protection of friend piece
Kill: kill is 0 when no kill action is made and it is 1 when an kill action made
Safety: sum of the distance between all pieces of a player.

The overall weight given to the features:
Num_pieces >=(<) Score > Danger >=(<) Distance_to_exit > Safety > Kill
In the process of Maxn, action with larger value of evaluation result is chosen to be the best action for each player. Getting more pieces brings more advantages than anything else, so I gave it the highest weight weight. The weight for score should be higher than num_pieces when the player is safe or close to win, but the score should be low when there are no enough pieces (pieces on board plus the score of game) to win. Danger is given higher weight than Distance_to_exit when the player is not in the dominant situation, It is harder to get pieces back when it has only 3 pieces. But the Distance_to_exit is given higher weight when the player have much more pieces than others, at this situation, move and approach exit is the priority. Safety makes pieces come closer when there are similar states(the sum of other features gives same values) available. Kill, reduces opponents is another case of improve self.

#pieces > 4: Agent will try to avoid all possible battle fields, because battles will cost a lot and it is not necessary to kill others in order to gain sufficient pieces to win.

#pieces < 4: Go back to battle fields and try to grab non-immediately convertible pieces from others.

#pieces = 4: Keep moving and be ready to any dangers (be killed) happen.

Others: From the first level of hierarchal evaluation value (Danger Level), the higher level will not be allowed to access until there are the same evaluation value of vectors comparing at the same level. Therefore, in this design, it tries to avoid dangerous situations first. If there has to be a collision with others, the agent will try to kill the competitor first. Eventually, if both conditions are evoked and also states values come to tie, the next layer, Safety Level, is going to detect how safe it will be after eating that pieces (if will be ate back immediately or ate by others).

Pieces > 4	+12000
Pieces = 4	+10100
Pieces < 4	-100
Others	+100

Figure 4.1 Evaluation Vaules Table

● Improvements

In fact, this report has illustrated a lot the growth of this project potentially so far, such as Minimax algorithm to Max^n algorithm, hierarchal evaluation value and so forth. Besides all which are involved before, there is another improvement on the correctness and efficiency.

In the first version of coding, there were four depths in Max^n algorithm and the fourth is used to detect if own pieces are safe after one round. At the beginning of design, the four-depth Max^n sounded reasonable and worked well during initial test. Nevertheless, after large number of tests, it was clear that if the agent considered the fourth level as the detection of dangers, it means the agent will consider whether it is safe after two self turns. Therefore, sometimes, the agent could not correctly judge if it is safe, according to two moves vs one move's condition. In order to avoid to many depths to complete the game in given time, the fourth depth was cancelled, then, more specific and hierarchal evaluation value was deign in this project.

Time complexity reduced by lazy evaluation, better choice of data type (lots of set and tuple), avoid use of 'copy.deepcopy()' and replace this with '.copy' make code run ten times faster, avoid the unneeded evaluation calculation.

Actually, complex evaluation function will not better than a simple one, It is hard to defeat the very early version of the implementation of Max^n. Lots of attempts can be found in 'moreversion' folder in 'oldversion' folder.

'simplemaxn' in the folder 'oldversion' is good in performance though its evaluation function is very simple, with only two major features. This 'simplemaxn' is the first complete version and one of the best versions I believe.