AU 332 Artificial Intelligence: Principles and Techniques

By: ZheHao Huang (518021910660) Yuncheng Yang(518021910727)

HW#: 1

September 27, 2020

I. GAME DESCRIPTION

A. Background

Chinese checker is a common intelligent game. The gameboard has 10 rows and 10 columns. The goal of the game is to make 10 marbles from one's starting position to the end position as quickly as possible. And assume that we are player1, we need to ensure the blue marbles move to the red marbles' positions and the yellow marbles move to the green marbles' positions. Player2 plays in the same rule.

B. Action

The marble has two types of legal moves. The first move is a simple jump over a single piece to the symmetric position about that single piece, and this move can be repeated as long as every segment of the jump is from an empty space to another empty space over a single marble regardless of our own pieces and the opponent's pieces. The second move is a simple move to an adjacent space. It should be noted that the marble can jump to one position as long as there is only one marble in the symmetric axis of the starting position and the end position.

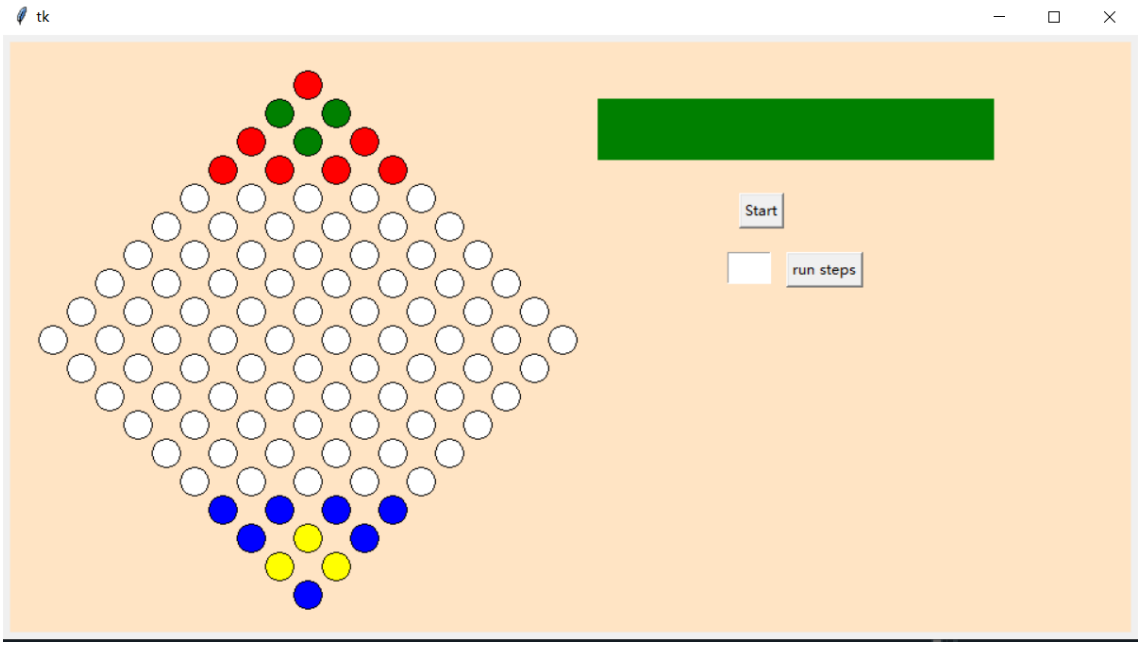
There are no forced moves in Chinese Checkers.

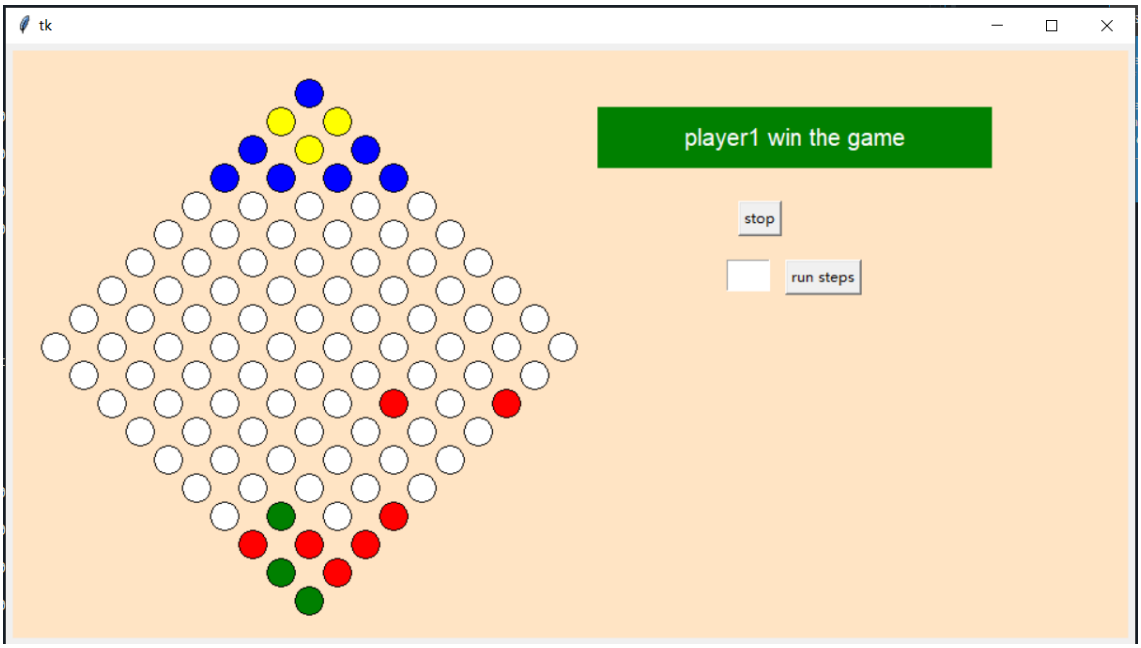
C. Special Rules

This game will give one second for each player to generate a move *timeout(agent.getAction, state)*. By the end of the one second, if the program does not assign a valid move to the *agent* action, it will lose the game immediately. Besides, after 100 turns, if any marble of one player are still in its own triangle, the player loses its game immediately.

II. OUR BIEJUANLE(TEAMNAME)

A. Preparation for the game





Before we start to implement our algorithm we first design a test tool for our game. As we can see the test tool enables us to take a single step or any steps as we want therefore we can make a more clear judgement about how the marbles take action and find some bugs in our code, which really helps to improve our algorithm efficiently.

B. Heuristics

We design our own Heuristics to help evaluate whether an action is worthwhile to take. We not only take into account the efficiency to fill in the target positions, but also use some constraints to ensure we can pad the target positions with the right marbles and not get stuck. In the following demonstration we will take player1 as an example to show how we score an candidate.

1. If the action is that a blue piece goes to $(1, 1)$, it will get the highest score and skip the following other evaluation steps.

2. If the action is moving a marble that's already at the target position, it will get the lowest score and skip the following other evaluation steps. And we define the target positions of blue pieces are $\{(1, 1), (3, 1), (3, 3), (4, 1), (4, 2), (4, 3), (4, 4)\}$ and of yellow pieces are $\{(2, 1), (2, 2)\}$, while we find that if a yellow piece stops at $(3, 2)$ it may hinder other yellow pieces from going to the right places, so it's not in the target positions of yellow pieces.

3. We consider only one blue pieces can exist in $\{(1, 1), (2, 1), (2, 2)\}$. So if the action is that a blue piece goes into that top triangle from outside when a blue one has already been there, it will get the lowest score and skip the following other evaluation steps.

4. We consider if two yellow pieces are already at $(2, 1)$ and $(2, 2)$, we will keep the third yellow piece out of $(1, 1)$ and give this action the lowest score. Skipping the following other evaluation steps.

5. We come up with a formula to evaluate the other actions:

$$Eval(a) = f_1(a) + f_2(a) + f_3(a) - f_4(a) \tag{1}$$

where $a = (pos_{now}, pos_{new}), pos = (row, col)$.

Before we explain our various functions, first and foremost, since the board is different from the gameboard with square grids and it's actually with hexagonal grids. I'd like to introduce some formulas to convert the coordinate system and two methods to compute the distance between two points.

In the code given, each point is represented by a offset of rows and columns as (row, col). But we can't directly get the distance between two points from that denotation. So we convert the offset coordinates into cube coordinates like (x, y, z).

Algorithm 1 offset_to_cube

Input: row, col
Output: x, y, z
1: $x = row$
2: $y = col - \frac{row - (row \mod 2)}{2}$
3: $z = -x - y$
4: **return** x, y, z

One of the advantages of why we convert the offset coordinate system into cube coordinate one is that in the cube coordinate system, we can directly compute the corresponding Manhattan distance and Euclidean distance as follows.

Algorithm 2 cube_Manhattan_distance

Input: $a = (a.row, a.col), b = (b.row, b.col)$
Output: d

$a.x = a.row$
$a.y = a.col - \frac{a.row - (a.row \mod 2)}{2}$
$a.z = -a.x - a.y$
$b.x = b.row$
$b.y = b.col - \frac{b.row - (b.row \mod 2)}{2}$
$b.z = -b.x - b.y$
$d = max(abs(a.x - b.x), abs(a.y - b.y), abs(a.z - b.z))$
return d

Algorithm 3 cube_Euclidean_distance

Input: $a = (a.row, a.col), b = (b.row, b.col)$
Output: d

$a.x = a.row$
$a.y = a.col - \frac{a.row - (a.row \mod 2)}{2}$
$a.z = -a.x - a.y$
$b.x = b.row$
$b.y = b.col - \frac{b.row - (b.row \mod 2)}{2}$
$b.z = -b.x - b.y$
$d = \sqrt{(a.x - b.x)^2 + (a.y - b.y)^2 + (a.z - b.z)^2}$
return d

More details about hexagonal grids and how to convert into cube coordinate system can be found on https://www.redblobgames.com/grids/hexagons/.

Besides, I'd like to introduce a dictionary of *targets* by what we guide our pieces to the right places. Its keys are the types of pieces that haven't filled in the corresponding positions and its values are the coordinates of these pieces at the order of first padding $\{1 : (1,1), (3,1), (3,3)\ 3 : (3,2)\}$ and then padding $\{1 : (4,1), (4,2), (4,3), (4,4)\ 3 : (2,1), (2,2)\}$.

After that, we can start to go though our functional Heuristics.

- f_1 is like a magnet repulsing the pieces away from the lower half plane. That means the pieces on the lower part of the board are more likely to go forward than the pieces that are quite closer to the target position.

$$f_1(a) = \begin{cases} pos_{now}[row] - 10 & pos_{now}[row] > 10 \\ 0 & pos_{now}[row] \leq 10 \end{cases} \tag{2}$$

- f_2 takes vertical displacement into account and rewards those big advances. We want our pieces arrive the target positions as quickly as possible. And this effect will be remarkable in the early on time of the game.

$$f_2(a) = \begin{cases} 2 \times \alpha \times (pos_{now}[row] - pos_{new}[row]) & pos_{now}[row] - pos_{new}[row] > 3 \\ \alpha \times (pos_{now}[row] - pos_{new}[row]) & pos_{now}[row] - pos_{new}[row] \leq 3 \end{cases} \tag{3}$$

$$\alpha = \begin{cases} 1 & if\ there\ is\ no\ pieces\ in\ the\ lower\ half\ plane\ of\ the\ board \\ 1.5 & otherwise \end{cases} \tag{4}$$

- f_3 assigns different target positions with different scores. We consider it is more important to make yellow marbles to the right places than to pad blue ones.The examples are as follows:
  If the color of pos_{now} is blue:

$$f_3(a) = \begin{cases} 1 & targets[1][row] = 4 \\ 5 & targets[1][row] = 3 \\ 2 & targets[1][row] = 1 \end{cases} \tag{5}$$

  If the color of pos_{now} is yellow:

$$f_3(a) = \begin{cases} 6 & targets[3][row] = 3 \\ 4 & targets[3][row] = 2 \end{cases} \tag{6}$$

- f_4 calculates the distance between the new position of that action and the nearest position of its target. And through some experiments, it turns out that Manhattan distance works better than Euclidean distance. We want our pieces gradually get closer to the target positions, so we need to minus this part in the total evaluation function.

$$f_4(a) = \begin{cases} min(cube\_Manhattan\_distance(pos_{new}, targets[1][j])) & pos_{now}\ is\ blue \\ min(cube\_Manhattan\_distance(pos_{new}, targets[3][j])) & pos_{now}\ is\ yellow \end{cases} \tag{7}$$

What's more, there are also some conditions such as when one marble may be left behind and we can not find other marbles to help the left one move forward quickly. Therefore, we try to prevent this situation from happening and let the marbles move in groups to get greater effect.

Last but not least, we will judge the number of our marbles left in the lower part of the board. And if the number is less than 4, we add the action score of those marbles to prevent them from being isolated.

C. Search method

We first tried the minimax algorithm and alpha-beta pruning to make the adversarial search. But it turned out that even we only took into account two steps of our actions and one step of the opponent, it still took more than 1 second to search. So we come up with a new algorithm to not only do the adversarial search, but also effectively control the number of the expanded nodes of the decision tree and the number of the depth of that tree. We first make the forward direction search to select the node with the top-k scores, and then we sum the highest scores of each path to the leaf node of the search tree to finally determine which action will be taken. The following figures aim to give a clear demonstration of our search method.

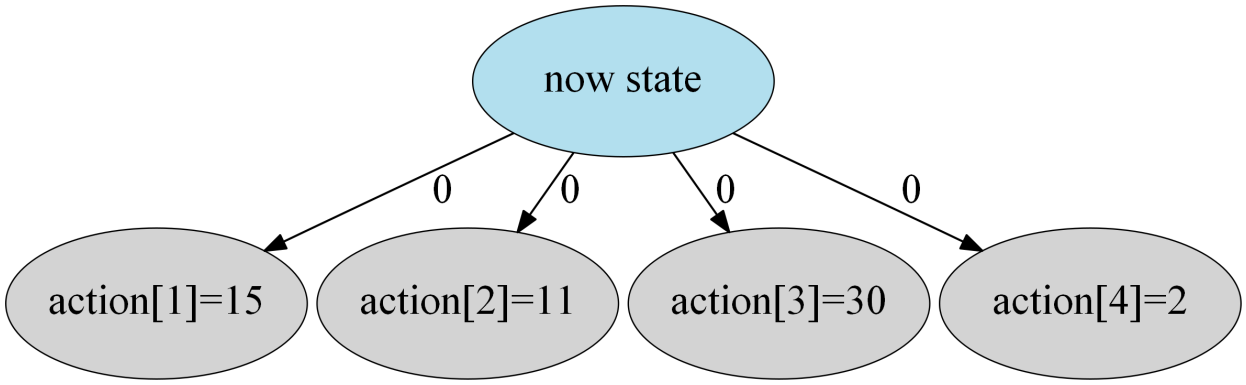


Figure1: Assuming in the current state we have four legal actions to select. And we can score these actions in the forward direction. The accumulations of scores on the initial paths are all 0.

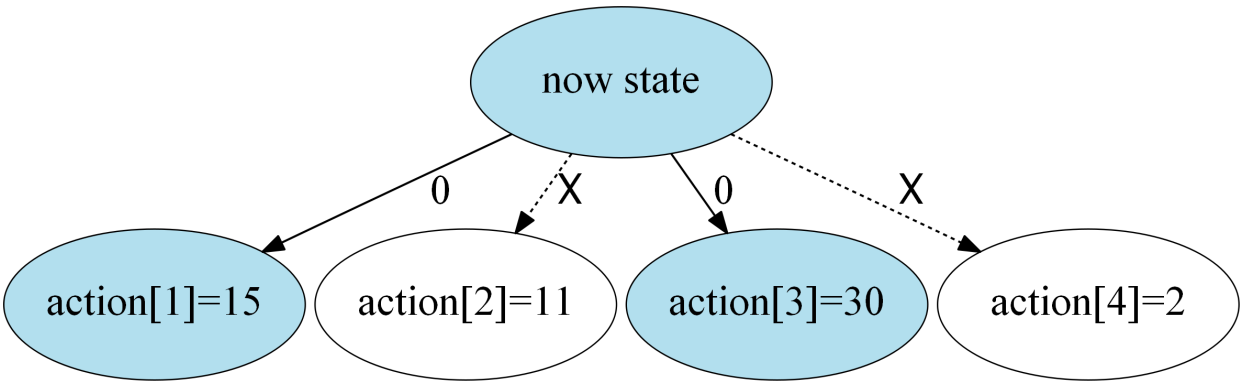


Figure2: If we pick top-2 actions, *action*[1] and *action*[3] will continue to be expanded. And *action*[2] and *action*[4] will be pruned.

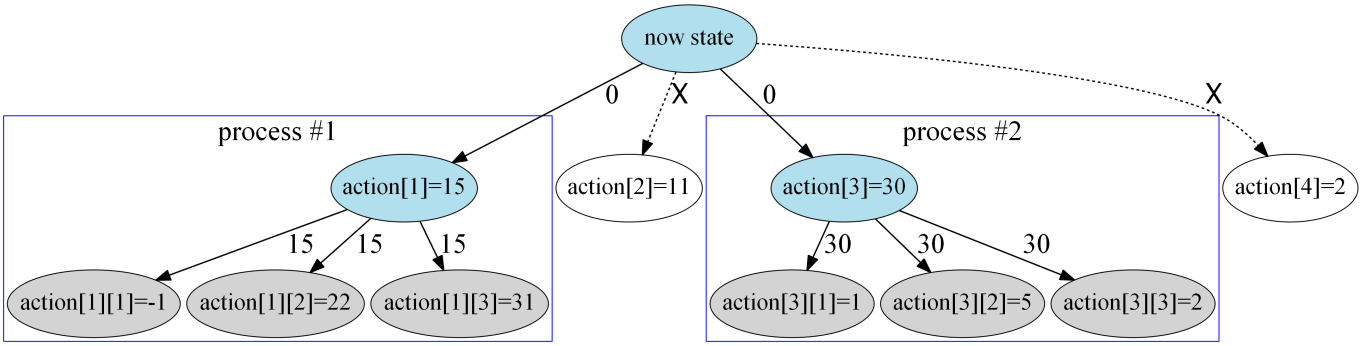


Figure3: When we expand the node of *action*[1], we will do process 1 which firstly change the state of the board according to *action*[1], secondly generate the optimal one-step-lookahead action of the opponent and thirdly play this action on the board. Process 2 is similar to process 1. Then we get the legal actions of the state after process 1 and 2 respectively. We also need to accumulate the previous scores on each paths.

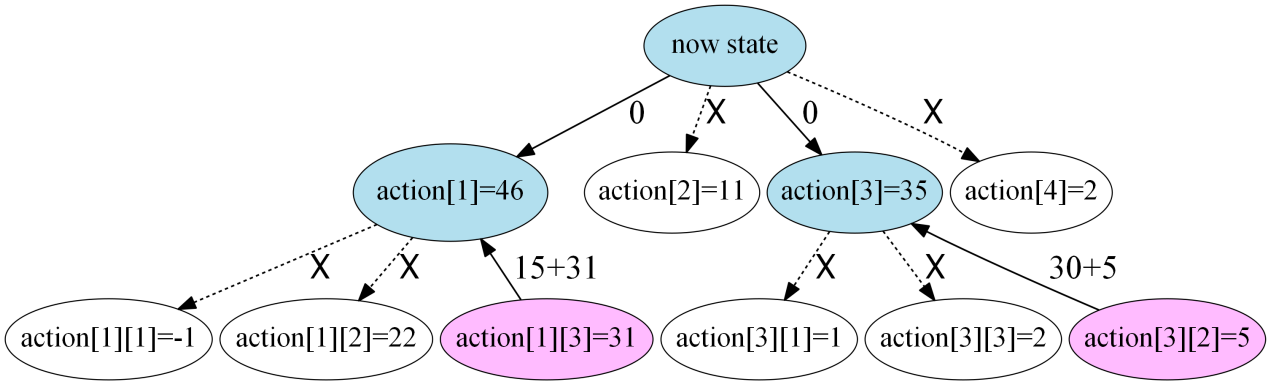


Figure4: Assuming the depth of the search tree is 2. So in the last layer we just return the highest scores accumulated on the paths backward to the top-2 actions in the first layer because one of these actions is the actual step we will take in the current turn.

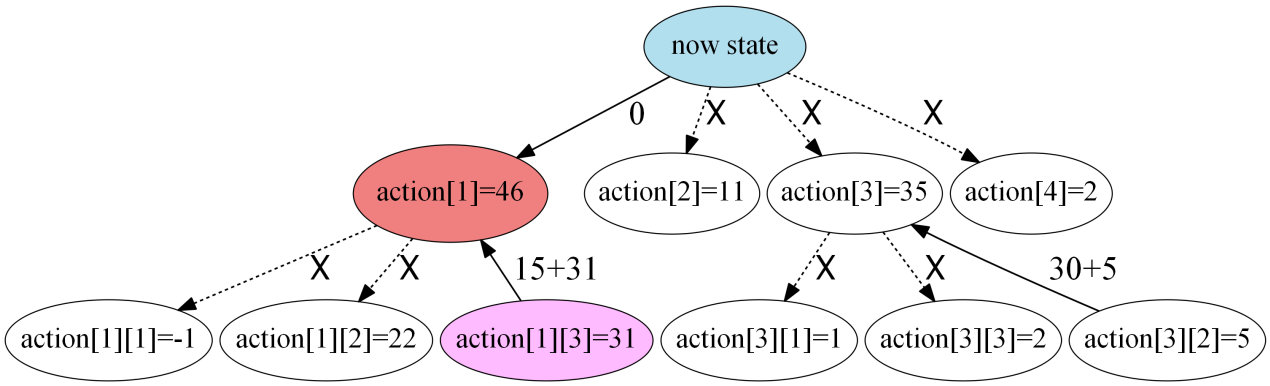


Figure5: We traverse the updated top-k actions in first layer, and find the action with the highest score in the exmaple is $action[1] = 46$. Although the score of $action[1]$ in the forward direction search is lower than the one of $action[3]$, after backward update the value of $action[1]$ in the next few turns is higher than $action[3]$ so we finally select it.

Using our search method, though we may lose some excellent actions which will show their values in the long-term playing, we still can get our ideal actions with controllable execution time which can be estimated by the max depth and the top-k nodes we set. We can ensure our action at least works better than the greedy search with the same heuristics because if we set the max depth equal to 0, that will work equivalently to the greedy search, which really helps us improve stability of our search method.

Actually, our agent may encounter the following situation: the agent predicts the most valuable step in the next few steps, but the opponent does not follow its own ideas, so this move is almost difficult to achieve. Therefore, in our consideration, we believe that the current return is more important, and the deeper the search level, the less important the return will be (because it is likely to be impossible to achieve).

We consider the decay factor γ, the reward of the action a is

$$R = Eval(a) * \gamma^{depth} \tag{8}$$

III. DISCUSSION & CONCLUSION

To conclude, we think that the algorithm is the key to make the agent win the game. And how to make the agent think like human is a very interesting topic. Given the fact that machine can calculate more quickly and effectively than human but it make some stupid mistakes from our sight, our core strategy is to build the proper evaluation function and combine the function with the moving strategy to get the ideal action. And to search for the best moving strategy we need to search for more than one steps to find the best step in the long run. However, there are some defects in our code. For example, we do not consider the difference between the player1 and player2. Given the fact that the first player have some advantages and the last player may have some disadvantages.Therefore the strategy of them may be different.

The combination of game and assignment is really triggering our great interest to dig into the development of our own designed algorithm. It not only helps us apply the algorithms learned in the class to solve the real problem, but also develops our programming skill. We learned how to gradually improve our algorithm and implement our abstract idea on the application through experiments. What's more, teamwork contributes a lot to our final algorithm. Close cooperation and frequent discussion also help us make progress efficiently. This homework gives us a good opportunity to finish a project in both theory and practice.