An Implementation of Chasing Agent in Game Colosseum Survival

Xinyu Wang, Yuhe Fan

1 Introduction

In this project, we build an agent to compete in a game named colosseum survival. We use a modified chase algorithm which finds a winnable step by constantly restricting the moves of our opponent. We also add to it the mechanism of self-protection. By running games across our several version of agents we are confident that each add-in builds have its advantages and are important in determining the next move. The current algorithms, however, still have some space to improve discussed in the final section.

2 Approach

2.1 Motivation

This *colosseumsurvival* is composed by a MxM blocks chessboard with two agents take turns to move $M + 1//2$ steps and put a barrier at one side of the block. When there are two regions separated by barriers, whoever have more blocks in its regions would win.

Restrictions includes that $6 \leq M \leq 12$, two agents are randomly initialized at any position in the chessboard. The first step allows for 30s to plan the next move, however, as the games progress on, only 2s are available for planning the next move. 500Mb ram is available for a total game.

The nature of this game and its restrictions already imply some of the ways we could proceed.

First, when a chessboard is 12x12 large, a brutal force complete search is impossible, this also eliminates the use of mini-max search which also search all possibilities of moving.

Second, this game have both resemblance and difference to the game Go. Consider the resemblance, both of them have a large chessboard which provides huge possibilities of moving, but, this huge possibilities would be constrained by the presence of the opponent. That is to say, if we shift the attention from create a separate zone with as large as blocks as possible, to constrain the opponent even block it using barriers, there are less possible moves we should consider upon. This gives us a clue of deciding the moves we should take. If we only consider restricting our opponent, A* search would compute the step for us to get near at it efficiently, however, there are other considerations explained below.

Also we should recognize the difference of this game from Go. Go usually takes a long period to end due to a different criterion for ending in which it only ends when there are no free space in the chessboard, however, in our game, there is a simple and fast way to end this game by placing barriers to make two separate zones in the chessboard. Thus, with our ideal to restrict our opponent explained above, in the middle of the game, it is possible to end the game in the next move, thus a must-chose solution. This is differ from Go which only reveals its winner at the supposed end of the game. This tells us in the design, we should be greedily searching for the best move which could end the game if it exists.

MCT tree search is famous in its utility in Go, it would average across the following scenarios to chose the next move. This is exactly due to the property of Go which requires you to think the long run effect, however, as explained above, our game requires less to consider the long run effect but to greedily end the game if possible.

2.2 Explanation

To integrate our ideal of restricting our opponent, end the game with win if possible, play with safe and not to suicide. Our approach utilize two kinds of priorities, one is using the heuristic functions to capture the distance to our opponent, another using layers of condition checking to find the best move ordered from win, play with safe, until suicide.

We have an heuristic function associated with each block:

fn = Manhattan distance + num of barriers.

Since we would like to restrict the opponent, we seek to be near at it so to place the barrier, this is captured by the Manhattan distance of that block to the opponent. The number of barriers are simply a count of barriers placed at that block. The thought is that if we place our agent in a block which already almost enclosed by some barriers, the opponent may easily enclose our agent.

Thus blocks which have a smaller Manhattan distance and less number of barrier placed around would have a better value associated with it.

With each valid block now associated with a value, we turn to think how placing the barrier affect the result. Ideally the barrier should be placed at the direction of the opponent, however, sometimes placing barrier at other direction also works. Thus we have another priorities concerting the direction of barrier. We first consider ideal direction, then worse directions. Note we think that worse block with ideal direction is actually a better choice than better block with worse direction. So we order our move by the block value with ideal direction, then the block value with worse direction.

The ideal of take win if possible is implemented using condition check. Also, consider the opponent may use the same strategy as to end us with success in one shot. We also implement a self-protection technique to avoid those steps which would be ended in one shot by the opponent.

Now we turn to explain how we decide the next move in detail.

First we find all valid steps, that is steps within the range $M + 1//2$. Then we compute the heuristic function value associated with each block and sorted them based on the value.

Then in the first layer search, we transverse the sorted blocks, checking if by placing barrier at the ideal direction, we could end the game with a win. If there's no return, we again transverse the sorted blocks, checking if the non-ideal direction for placing barrier would end the game with a win.

Now if there is still have no return, we have search all possible move to end the game but failed. Thus in the second layer search, we again transverse the ordered blocks, first we check this would not end the game, for this must resulting in our own lose, this is basically a suicide. Then, this time upon the assumption we take that move, we perform the first layer search from our opponent perspective, if our opponent could end us in the next move, we assign the block with a loss value greater than 0. If we find a block with loss value being 0, we could take it. Otherwise, it basally means for all steps we could take, the opponent have a chance to win us in the next move.

In this scenario we simply choose the first possible move without suicide in our priorities, with our priorities being ordered blocks with ideal directions then ordered blocks with worse directions. Note that it may be the case that all blocks have no ideal directions.

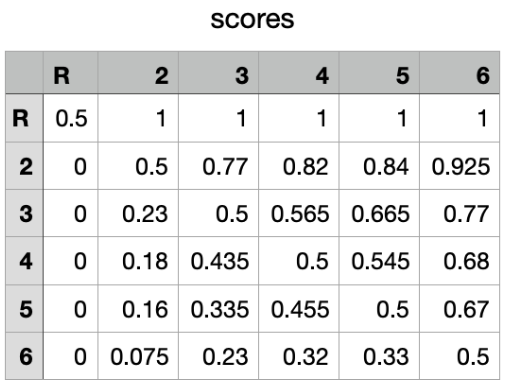
If all above steps have no return, it leave us the only option of suicide.

The main ideal could be captured in the pseudocode attached in the appendix.

Thus to summarize our approach, we assign a priority to our possible moves based on the ideal of restricting our opponent and protecting our agent being restricted, then following this priority greedily check if we could end the game with a win. If winning at this stage is not possible, we follow the priorities but to choose a safe step. Thus we're always looking forward one step. Note that sometimes even a safe step is not possible, then we again try to avoid suicide. The implementation is that to commit suicide if that is the only option.

3 Performance

We present our agent scores when fight 100 games. R is denoting the random agent, 2 is denoting agent 2 which only have heuristic function for Manhattan distance, 3 is denoting agent 3 with full heuristic function including Manhattan



scores

	R	2	3	4	5	6
R	0.5	1	1	1	1	1
2	0	0.5	0.77	0.82	0.84	0.925
3	0	0.23	0.5	0.565	0.665	0.77
4	0	0.18	0.435	0.5	0.545	0.68
5	0	0.16	0.335	0.455	0.5	0.67
6	0	0.075	0.23	0.32	0.33	0.5

distance and number of barriers, 4 is denoting agent 4 builds upon agent 3 with first layer search checking if this step is winnable. 5 denotes agent 5 which adds a third layer search which prevent itself from suicide. Agent 6 is our complete implementation which also have second layer search for safe move.

Each wins grants a score of 0.01, even games gives 0.005 to both side.

From the score table we could observe each improved version of agent indeed have advantages over the previous, with agent 2 could win random player with certainty, agent 6 could win agent 2 with almost possibility of 1.

3.1 Pros

The most significant advantages from the design of this agent is that it is greedy to take the win if possible, at the same time it is cautious to take the safe move. To choose a move involves several considerations, and one is superior than others depending on the situation. By using a heuristic function to chase the opponent, we deliberately checks several conditions in the order from wining to safe play. This approach actually allows us to always reaches the best move we could take as possible according to current situation. While we check our conditions using a sorted order of moving steps, we expect to determine the step early in the sorted queue thus being efficient enough.

Our fully implemented agent is both efficient in time and space. If we place two of our agent to fight in a 12x12 chessboard in 100 runs, the total time is approximately 3 minutes, and it only uses 50 mb ram. While if plays on 6x6 chessboard, it would only use 0.1s. This efficiency again is due to our design of looking forward a step, instead of searching deeply and average across all effect.

3.2 Cons

The disadvantages actually also comes from the fact we are checking the best moves only depends on the current situation. The performance is estimated to be best at 10x10 chessboard, however, when the chessboard is smaller, in 6x6, the opponent would have time to search for deeper moves hence have advantages over our agent.

4 Future Improvements

We are only looking forward one steps currently due to the restriction of time given only 2s for computation, however, if examined carefully, as the game progress, the possible valid steps to check is actually shrinking, that gives us an opportunity to search deeper by adding more layers of search. Also consider in the first run, we have the most possible valid steps for checking, but we also have 30s to make plan. Our current approach uses 50 Mb for 100 runs, with one more layers it would requires 2500Mb, which still fits the requirement. Thus it is possible to build further layers to looking forward into 2 moves.

Learning during playing is not allowed this time, but if possible, the heuristic function could have adjustable assigned weights which get trained during playing, thus we could find the most appropriate weights for Manhattan distance and the number of barriers in our heuristic functions. Now both of them have weights equal to one.

5 Appendix

```
pos = Sort reachable blocks by heuristic function

first layer search - search for end with success
for pos:
    for ideal direction:
        if placing barrier end the game:
            if win:
                return this move

for pos:
    for worse direction:
        if placing barrier end the game:
            if win:
                return this move

second layer search - search for safe move
for pos:
    for ideal direction:
 if placing barrier not end the game:
        fist layer + second search from opponent perspective
         if opponent win:
              loss value > 0
         else:
              loss value  = 0
          if loss value = 0:
              return this move

for pos:
    for worse direction:
 if placing barrier not end the game:
        fist layer + second search from opponent perspective
         if opponent win:
              loss value > 0
         else:
              loss value  = 0
         if loss value = 0:
              return this move

third layer search - to not suicide
for pos:
 for ideal direction:
  If placing barrier not end the game:
  return

for pos:
 for worse direction:
  If placing barrier not end the game:
  return

return first move - suicide
```