Reversed-Reversi

Songhang Deng 11912918
Department of Computer Science and Engineering
11912918@mail.sustech.edu.cn

1. Preliminary

1.1 Problem Description

Reversed-Reversi is a famous game.Players take turns to place their chess on the board.During a play, any disk of the opponent's color that are in a straight line and bounded by the disk just placed and another discs of the current player's color are turned over to the current player's color.In this project, We should play a Reversed-Reversi game with others, but the goal of competition is to have fewer chess on the board when game finished.

1.1.1 Software

This project is written in Python with ide Pycharm

1.1.2 Algorithm

In this project, I use **Alpha Beta Pruning** to do the search.I give priori knowledges to my procedure.When the **Alpha Beta Pruning** algorithm is on the leaf point, It uses prior knowledge to calculate some consideration points and then multiplies those consideration points by preset weights to evaluate the leaf point.These weights are calculated using **genetic algorithms**.Also, some algorithons like Monte Carlotree search are used in my previous version.

1.2 Problem Applications

In this project, the method I used is very robust and can be applied in many aspects, such as other similar board games: backgammon, chess, etc. At the same time, it can also be used to adjust the parameters of many reinforcement learning algorithms, such as A3C,PPO and so on.

2. Methodology

2.1 Notation

Here is all my notations:

Notation	Meaning
b	the chessboard(a 8*8 list)
I(b)	the Location score of b
S(b)	the stabilization num of b
A(b)	the action space score of b
F(b)	the frontier num of b
N(b)	the chess num of b
w	weight
C(b)	get the score of b

all the above calculation is self substract oppenent, such as S(b), I will calculate the stable num of mine and opponent's, and return the difference as S(b).In my program, I use calculate $C(b)=I(b)+S(b)w_{S(b)}[(s-5)/20]+A(b)+F(b)w_{F(b)}[(s-5)/20]+N(b)w_{N(b)}[(s-5)/20]$, s represents the num of chess on the board

2.2 Data Structure

Here are the most important data structures in my final version.

Name	Meaning
s	the num of chess on the board
score_matrix	The location score matrix when s<=44
score_matrix1	The location score matrix when s>44
arb_matrix	The action score matrix when s<=44
arb_matrix1	The action score matrix when s>44
c2	the weight for stabilization num
c3	the weight for chess num
c4	the weight for frontier num
candidate_list	All the points you can play chess with(the last one of it is my action)
match	a map, matches board to the arrivable point list
arrivable	a list, which represents the legal moves of the player's board

2.3 Model design

My problem model is very simple, preliminary build a search tree, using iterative deepening ab pruning of the search, and the time limit within five seconds, finally improve the search order, search directly to the end of the game, try to find an action to get certainly win, if no action can make me certainly win, I will try to find an action to get draw, if no action can make me certainly win or draw, I will use two layers of ab pruning to get a move.

To improve the evaluation function, I used a genetic algorithm, with a mutation rate of 0.05 and a population size of 60, and each generation pitted all individuals against each other, keeping the higher half of the population and hybridizing to bring the population size back to 60, and keeping the ten strongest historical versions, Every individual after generation 10 will play against the historical version.

2.4 Detail of algorithms

Here are the details of my algorithms:

alpha-beta Pruning algorithm:

I use a map to store the arrivable points of board I have visited, each time I need to get arrivable points to move, I will find in the map first.Other parts in the algorithm is same as common Alpha-Beta Pruning search

---

Algorithm 1 :Alpha-Beta Pruning search

Parameter:color :the color AI hold,maxDep:the max search depth

Input: board:the chessboard

a,b:variables for Pruning

turn:the color to move

deep:the depth of now layer

lastCan:a variable to tell the function whe last player has a move or not

output: if deep==depth return best move

else if deep==0 return C(b)

else return an integer represent the best score the layer can get

```
if b in match:
  arrivable=match[b]
else
  arrivable=get_arrivable(board,turn)
  match[b]=arrivable
if turn==color:
  best=-inf
else:
  best=+inf


for move in arrivable:
   if turn == color:
      take move in board
```

```
      score=Alpha-Beta Pruning search(bd,a,b,-turn,deep-1,true)
      recover board
      if score>best:
        best=score
        bestmov=move
        alpha=max(alpha,score)
        if alpha>beta:
          return alpha
  else:
    take move in board
    score=Alpha-Beta Pruning search(bd,a,b, -turn,deep-1,true)
    recover board
     if score<best:
        best=score
        beta=min(beta,score)
        if alpha>beta:
          return beta
   if deep==maxDep:
     return move
   if arrivable is empty:
     if lastCan:
       return Alpha-Beta Pruning search(bd,a,b,-turn,deep-1,false)
     else:
       return C(b)
    if turn==color:
       return alpha
    else:
       return beta
```

Genetic algorithm:

In this algorithm, I added some individuals to the initialization, and they would just be greedy to pick the move that would make the least amount of their chesses, also I store the strongest ten version, each turns all individuals should play to them

---

Algorithm 2 :Genetic algorithm

Input:

waiterNum:The total number of individuals

playNum:The game num of each individual color should play

random init a list of ai, named waiter

waiter[0:4]'s c2,c3,c4 is all zero, score_matrix is an 8*8 list which has same value at anywhere.

total=0

loop infinity:

for j from 0 to waiterNum-1:

for i from 1 to playnum:

waiter[j] using black play to waiter[i+j]

if waiter[j] wins:

waiter[j].blackwin++

if tie:

waiter[j].tie++

else:

waiter[i].whitewin++

for j from 0 to waiterNum-1:

for i from 1 to playnum:

waiter[j] using white play to waiter[i+j]

if waiter[j] wins:

waiter[j].whitewin++

if tie:

waiter[j].tie++

else:

waiter[i].blackwin++

if total>10:

for j from 0 to waiterNum-1:

waiter[j] using black and white play to ten history winners

modify waiter[j]'s win num like above

sort waiter by(min(blackwin,whitewin),tie) in decending order

push waiter[0] into history winners

pop the last half of waiter

print waiter

play in history winners, pop the worst one

print history winners

for i from 0 to waiternum/2:

random choose one from waiter to product with waiter[i], all the gene is represent by signed binary number, the gene is inherited from father or mother by the same probility, the mutation is 0 to 1 or 1 to 0, the probility of mutation is 0.05.

get son of above

append son to waiter

getarriable algorithm:

I use a simple algorithm to get arriable points of the board, just think about all empty locations, and judge whether the location is legal move or not

---

Algorithm 3 :getarriable

---

input:

board,turn

direction=[[0,1],[0,-1],[1,0],[-1,0],[1,1],[1,-1],[-1,1],[-1,-1]]

result=[]

for i from 0 to 7:

for j from 0 to 7:

if board[i] [j] is empty:

for k from 0 to 7:

if [i,j] can turned over opponent's chess in direction[k]

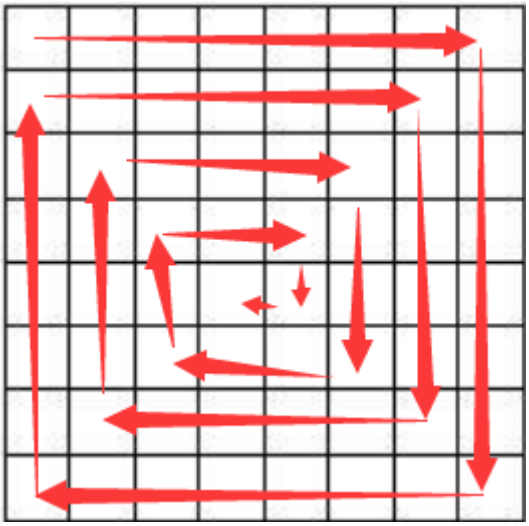
result.append([i,j])

break

return result

winner search:

When num of chess is no smaller than 54, I will search to the end of the game, if I have a move to consure my win, I will take the move, else, if I have a move to draw, I will take the move, else, I will just using Alpha-Beta Pruning search of deep 2.To improve the search efficiency, I make an search order, which is from the boundary to the middle, which is like this:



Near the end of the game, the legal placement is highly likely to be distributed on the boundary, and searching in this order is easy to have large pruning, greatly reducing the search time.

Algorithm 4 :winner search

Parameter:color :the color AI hold,maxDep:the deep of layer which will output a point

Input: board:the chessboard

turn:the color to move

deep:the depth of now layer

lastCan:a variable to tell the function whe last player has a move or not

output: if deep==depth return best move

else if winner is self return inf

else if draw return 0

else return -inf

```
if turn == color:
  best=-inf
else
  best=inf
bestMov=-1
s=0
for move in order1:
 if move is valid:
  s=1
  if turn == color:
    take move in board
    score=winner search(board,-turn,deep-1,true)
```

```
      recover board
      if best < score:
       best=score
       bestMov=move
       if deep==maxDep and best==inf:
        break
       else if best==inf:
        return best
  else:
   take move in board
   score=winner search(board,-turn,deep-1,true)
   recover board
    if score<best:
      best=score
      if best ==-inf
       return -inf
if s==0:
  if lastCan:
    return winner search(board,a,b,-turn,deep-1,false)
  else:
    if winner is self return inf
    else if draw return 0
    else return  -inf
if deep==depth return bestMov
else return best
```

3.Empirical Verification

3.1 dataset

In the feasibility test, I used simple logic, that is, traversing eight directions to determine whether each point is legal or not, and passed 9 of the test samples. Later, after checking my logic, I found that I ignored the situation that there were no feasible points and passed the test after considering them.I have written a battle program locally. When I find that the performance of my program on the online battle platform does not meet my expectations, I will download the log and put it into the local battle program.Then I will use debug mode to observe why the performance of the program does not meet my expectations

3.2 Performance measure

I used the local play platform mentioned above to test my performance. Here is my environment.

> Intel(R) Core(TM) i7-10700K CPU @ 3.80GHz 3.79 GHz

I use decorator to limit my program to 4.8 seconds, then print out the number of levels searched and find that my program can search more than four levels in any case, which is enough for a more forward-looking evaluation function. In addition, I compared the results of the three, four, and five levels of search in many case and found that 90% of the time they gave the same results, indicating that my evaluation function was a bit farsighted.In the last days, I rarely lost in the platform, mostly two wins or one loss. In the final round robin, I get rank 2 with a win percentage of 95.2 percents and a loss percentage of 3.85 percent.

3.3 Hyperparameters

In my program, the hyperparameters are the weights of each position and the weights of the actions of each position, as well as the weights of the stabilizer, the frontier chess.

At first, I set these hyperparameters empirically, and later, I adjusted them using genetic algorithms, and I used multiple processes, playing round robin matches between 60 individuals in each generation.Each time, I compare the individuals using (min(blackwins,whitewins),tie nums).The min(blackwins,whitewins) can ensure the robustness.I pop the last half of these individuals each time.I also recorded the survival time of each individual and pitted them against the ten most powerful versions in history to prevent population degradation.I pass all usability tests, and win 95.2 percents in the robin round.

3.4 Experimental results

My program turned out to be pretty good, I pass all the ten usability tests, win 95.2 percents in the robin round and take rank 2^{nd} at last.In my genetic algorithm, the optimal individual winning rate in each generation of individuals is basically over 60% within the population, and the ten strongest individuals in history are basically generated in the last 50 generations, which indicates that my population has not undergone significant degradation.

3.5 Conclusion

During this project, I learn many knowledge.I found some ways to improve alpha-beta search, and I learn gene algorithm which I totally know nothing about it before.Also I found the importance of robustness and stability, at first, I let each individual just play ten terms with others, and the result is very unstable, some individuals even has no lose!So at last, I make each individual play with all others.Also, I use min(blackwin,whitewin) to ensure the robustness, avoid some individuals just win when using black or white.

In the final version of my algorithm, there are many disadvantages as well as many advantages. The disadvantage of my approach is obvious. In the genetic process, all opponents that each program encounters will only consider a few perspectives that I have given them, and my program may struggle when I encounter opponents who consider more than these perspectives in a round-robin match. In other words, the prior knowledge of the program needs to be given by me. In addition, My division of stages is only based on the number of chess pieces, without comprehensive consideration of the situation, which is largely limited by the number of

parameters. I once wanted to weight some situations at corner and eder, but there are 27 situations with only 3 pieces in corner, and the number of parameters increases exponentially. In a real match, the black and white players are considered differently, but splitting them would have doubled the number of arguments. I didn't splitting them in the end, so my programm used the same way of thinking regardless of it is black turn or white turn.Also, because of my select way, the worst individual in my population has only a win percentage less than half.To make improve, I think during genetic algorithm, we can reserve less individuals.In my experience, no individual has lived for 100 generations during over 3 thousands generations, that means the algorithm is far away to convergence, even maybe small degeneration.So we can store more good history versions, and give more prior knowledge to the program. Also we need to control the range of the parameters.In my program, the total searching space is 2^{346}, it's a terrible space!

In this project, my program also has many advantages. My calculation method of individual fitness makes it have strong robustness and will not perform badly no matter what kind of opponent it faces.When the indivual can not learn directcly, the robustness is important. Secondly, my evaluation function is farsighted and considers the situation as far as possible under the circumstance of limited computing power. The characteristics of genetic algorithm make my program still have a relatively good performance in a large search space. In the case of not finding a way for individuals to learn directly, genetic algorithm plays a very good role.

4.references

[1]知乎专栏. 2021. 黑白棋*AI*：局面评估+*AlphaBeta*剪枝预搜索. [online] Available at: https://zhuanlan.zhihu.com/p/35121997 [Accessed 4 November 2021].

[2]简书. 2021. 【算法】超详细的遗传算法*(Genetic Algorithm)*解析. [online] Available at: https://www.jianshu.com/p/ae5157c26af9 [Accessed 4 November 2021].