CS303 AI Project1 Report: Reversed-Reversi

HuangZitong

12012710

November 1, 2022

1. Introduction

1.1. Zero-Sum Game

Zero-sum game is a situation in game theory, which reference one side's gain will lead to another side's loss. That is to say, there is no possibility for two side to cooperation. It can be represent by:

$$\sum_{k=1}^{n} S(k) = 0$$

- $S(x)$: the income player x can get in zero-sum game
- n: the number of players
- k: the k-th player

1.2. Application

In daily life, zero-sum game is used in lot fields:[1]

> …Other examples of zero-sum games in daily life include games like poker, chess, and bridge where one person gains and another person loses, which results in a zero-net benefit for every player[2].In the markets and financial instruments, futures contracts and options are zero-sum games as well.[3]

In this project, zero-sum theory is main used by describe reversi(a kind of chess game). A further explanation of reversi will be shown later.

1.3. Purpose

This report is mainly write to explain the first project of CS303 course. And, this report will give a better explanation of some algorithm used, theory, fomuler and method used in project code.

2. Preliminary Problem Description

2.1. Reversi

Reversi is a famous chess game origin from UK, and carried by Japaness at 19th centry, with a serious of simple rule:

- There is a chessboard with 8×8 size, and two players represent black and white
- When two self chess "clamp" some enemy chess —crosswise, verticle or oblique, it will change these chesses' color to self color.
- Every step should change at list one enemy chess to self chess

Note 1. This project is about Reversed-Reversi, which means AI will try to loss a reversi game.

2.2. Formulation

Some variable should be defined first:

- A: Agent side
- E: Agent's opposite side
- x : a specific player, where $A \in x$ and $E \in x$
- $n(x)$: the number of x's chess at a given time
- $N(x)$: when the game is over, the value of $n(x)$

  Example. $N(A)$reference to the chess number of A when game is over.

- $S(x_1,x_2)$: when the game is over, the difference of x_1's chess and x_2's chess. $S(x_1,x_2) = N(x_1) - N(x_2)$

  Example. $S(A,E) = N(A) - N(E)$

- Chessboard state: the state of chessboard when game is at a certain point
- Evaluation function: A function to evaluate the score of a given chessboard state
- self chess: chess in agent's color
- enemy chess: chess not in agent's color
- F_n: the n-th evaluation function's value at a given chessboard state
- w_n: the n-th evaluation function's weight
- F: final evaluate function, $F = \Sigma w_n F_n$
- grid score: score that some evluate function give to a chessboard grid
- chessboard score:sum of grid score earned by a player
- action point: the number of position a player can play
- timeout step: agent need more than 5s to decide this step

1. Reference from https://en.wikipedia.org/wiki/Zero-sum_game

2. Von Neumann, John; Oskar Morgenstern (2007). *Theory of games and economic behavior* (60th anniversary ed.). Princeton: Princeton University Press. ISBN 978-1-4008-2946-0. OCLC 830323721.

3. Kenton, Will. "Zero-Sum Game". *Investopedia*. Retrieved 2021-04-25.

The problem can be defined as a optimized problem—Target function is $S(A, E)$, the objective of an agent is to minimized $S(A, E)$

3. Procedure

3.1. Introduction

Solve this problem need such procedure: usable test, evaluate function design and step decide(decide which step to go).

When dealing with step decide, a minimax algorithm optimized by alpha-beta pruming and heuristic cutoff is used.

3.2. Usable Test

Before all algorithms, we have to get the positions where we can place our chess.

The procedure of this step is:

1. find all self chesses
2. for every self chess, determine the position can be played because of this self chess.
3. Remove duplicate position.

When dealing with 2-th step, the pseudo-code is

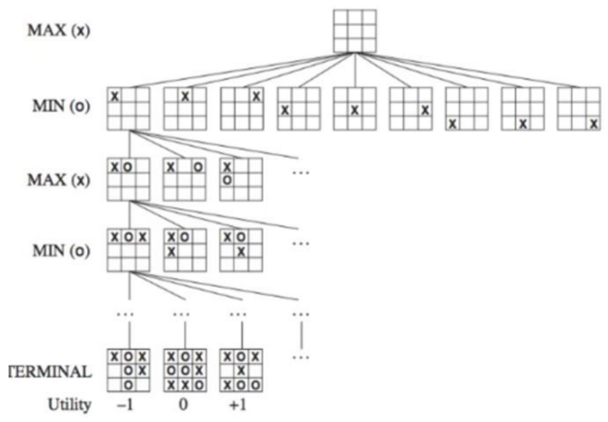
```
*/
find the positions can be played because of the chess located in position
*/
function find(chessboard, position)
begin
    possiblePosition = []
    acts <- [(1,1),(1,0),(0,1),(-1,-1),(-1,0),(0,-1),(1,-1),(-1,1)]
    for act from acts[0] to acts[7]
    begin
        cnt <- 0
        x, y <- position.x, position.y
        while x,y in size(chessboard)
        begin
            x, y += act
            cnt++
        end
        if cnt > 0 then add (x,y) to possiblePosition
    end
    return possiblePosition
end
```

3.3. Step Decided

3.3.1. Minimax

Procedure

To solve such zero-sum game, a common solution (or, algorism) is Minimax algorism. An example graph is:



At the bottom of the search tree, evaluation function will score the chessboard. Then, Max layer will choose the highest score in next layer, that is, Min layer; And then, the Min layer will choose the lowest score in next layer, that is, the next Max layer.

Result

By using this algorithm, agent will choose the highest minimal score(global highest score is decided by how fool opponent player is, but minimal higest score is decided by agent).

Deficiency

Obviously, we will found the time complexy of minimax algorithm is

$$O(M^N)$$

where N is max step possible, M is conditions each step may have

It's easy to see the time complexy is terrible—for reversi, N can easily reach 64, and even M = 2 the time complexy will get 2e19.

Thus, we need an optimizing way

3.3.2. Alpha-Beta Pruming

For each node, we can discover that if we know a subnode is worse than another subnode, than we don't need know how bad it is. Thus, we can define an α and a β

α: the minimal score a node should have, defined by

$$\alpha = \begin{cases} \max(\alpha, \text{score of all subnode}) & (\text{Max layer}) \\ \alpha \text{ from its parent node} & (\text{Min layer}) \end{cases}$$

β: the minimal score a node should have, defined by

$$\beta = \begin{cases} \beta \text{ from its parent node} & (\text{Max layer}) \\ \min(\beta, \text{score of all subnode}) & (\text{Min layer}) \end{cases}$$

if a node's $\alpha \geqslant \beta$, we can stop search immediately.

After pruming, the time complecy can get[4]

$$O\left(\sqrt{M^N}\right)$$

3.3.3. Heuristic Cutoff

Obciously, even by using alpha-beta pruming, the time complexy is still very high. To deal with this stituation heuristic cutoff can be used.

4. reference by https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

Before calling the minimax function, algorism add a parameter d means depth. while minimax reach the d-th layer, algorism will cutoff search, and return a score depend on evaluate function. Then, minimax search will turn back with the score, and choose the action.

To avoid time exceed, depth in this project is 3

Note. Since when game is about over, M in time complexy will be small enough to search, so when empty grid on chessboard is less than 5, depth will be ∞

3.3.4. Pseudo-code

The pseudo-code of minimax in this project is following.

```
function miniMax(depth, chessboard)
begin

    function max_value(depth, chessboard, alpha, beta)
    begin
        if GameOver or depth is 0
        then return Score(chessboard)
        $v$ <– -$\infty$
        for every action can be down
        begin
            if alpha > beta then break
            $v_0$ <– min_value(depth - 1,
                                ChessboardAfterAction,
                                alpha, beta)
            alpha <–max($v_0$, alpha)
            $v$ <– max($v$, $v_0$)
        end
    end

    function min_value
    begin
        if GameOver or depth is 0
        then return Score(chessboard)
        $v$ <– $\infty$
        for every action can be down
        begin
            if alpha > beta then break
            $v_0$ <– max_value(depth - 1,
                                ChessboardAfterAction,
                                alpha, beta)
            beta <– min($v_0$, beta)
            $v$ <– min($v$, $v_0$)
        end
    end

    return max_value(depth, chessboard, -$\infty$, $\infty$)
end
```

3.4. Evaluate Function

In this project, three evaluate function is set to evaluate the chessboard state[5].

$$F_1 = \text{chessboard score}(\text{self chess})$$

$$F_2 = \text{action point}(\text{self chess}) - \text{action point}(\text{enemy chess})$$

$$F_3 = n(\text{self chess}) - n(\text{enemy chess})$$

5. Reference to 黑白棋AI：局面评估+AlphaBeta剪枝预搜索 - 知乎 (zhihu.com)

Note. score give to each grid is following:

10000	-200	250	250	250	250	200	10000
-200	-45	-1	-1	-1	-1	-45	-200
250	-1	3	2	2	3	-1	250
250	-1	2	1	1	2	-1	250
250	-1	2	1	1	2	-1	250
250	-1	3	2	2	3	-1	250
-200	-45	-1	-1	-1	-1	-45	-200
10000	-200	250	250	250	250	-200	10000

It is worth to mention that since we are designing a reversed-reversi AI, F_1 and F_3 should be reversed. That is to say, the acture evaluate functions are:

$$F_1 = 0 - \text{chessboard score}(\text{self chess})$$

$$F_2 = \text{action point}(\text{self chess}) - \text{action point}(\text{enemy chess})$$

$$F_3 = 0 - (\, n(\text{self chess}) - n(\text{enemy chess})\,)$$

To distinguish the importance of each function, weights are needed. The final total evaluate function is:

$$F = w_1 \times F_1 + w_2 \times F_2 + w_3 \times F_3$$

where w_1, w_2, w_3 are F_1, F_2, F_3's weight. In this project, they are 10, 5 and 1

3.5. Working flow

1. Start the algorithm
2. Usable test find all position can be played
3. Start Minimax algorithem
4. While Minimax search reach the cutoff layer, use evaluate function to score the chessboard state
5. Depend to the score evaluate function return, decide each node's alpha and beta, continue acting Minimax.
6. Minimax return a point, agent do action.

4. Experiments

4.1. Environment

4.1.1. Hardware Environment

CPU: Intel(R) Core(TM) i7-10875H CPU @ 2.30GHz
GPU: NVIDIA GeForce RTX 2060

4.1.2. Software Environment

Operation System: Windows 11
Python: Python 3.9.12
IDE: VSCode with python extension
Numpy: 1.16.1
Other package used in python : math, time, random

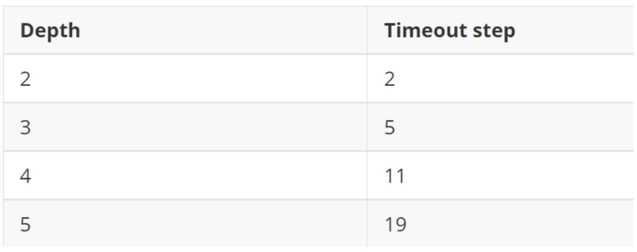
4.2. Dataset

Most of test chessboard is build by myself, mainly to test whether the agent can decide and do action correctly. Test chessboard is mainly consitute by some extreme boundary conditions, such as

-1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0

4.3. Time analysed

With the increase of depth, timeout step will increase. The general case is:



Depth	Timeout step
2	2
3	5
4	11
5	19

Since timeout step will be cutoff immediately, in this project depth = 3 is a balanced choose.

5. Conclusion

5.1. Evaluate algorism

5.1.1. Advantage

- Simple: Minimax search is easy to build for beginner
- Long-term version: Evaluate function is mainly contribute by the action point and chessboard score, not chess change or chess number difference.
- Sprint: While using sprint method, agent can use the best evaluate function — win or loss — when game is about over.

5.1.2. Defect

- Grid score on chessboard should change while game going
- Stable chess(chess cannot be change)is an important concept in reversi game, however it is not used in evaluate function.
- Time complexy is still too high, there are a lot of space to optimized.
- Depth should change while game going, such as larger at the start and smaller when middle.

5.2. Space to improve

To improve the algorithm, such method can be take:

- Change the grid score while game is going: such as, while a corner is taken by enemy, grids in side around the corner should have a higher score
- Time check can be added, if the action points of enemy/agent is too large, depth can be decreased properly.

Bibliography

1. Von Neumann, John; Oskar Morgenstern (2007). *Theory of games and economic behavior* (60th anniversary ed.). Princeton: Princeton University Press. ISBN 978-1-4008-2946-0. OCLC 830323721.
2. Kenton, Will. "Zero-Sum Game". *Investopedia*. Retrieved 2021-04-25.
3. Orion Nebula.黑白棋AI：局面评估+AlphaBeta剪枝预搜索 [EB/OL].(2018-05-09)[2022-10-31]. https://zhuanlan.zhihu.com/p/35121997