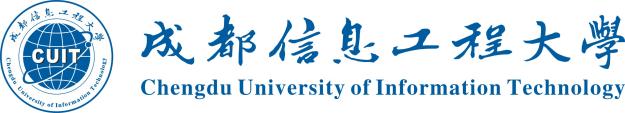
****

**实 验 报 告**

|  |  |
| --- | --- |
| **实验课程：** | **人工智能导论** |
| **实验项目：** | **Alpha-Beta剪枝** |
| **指导教师：** |  |
| **学生姓名:** |  |
| **学生学号：** |  |
| **班 级：** |  |
| **分 数：** |  |

20 年 月 日

成都信息工程大学 计算机学院

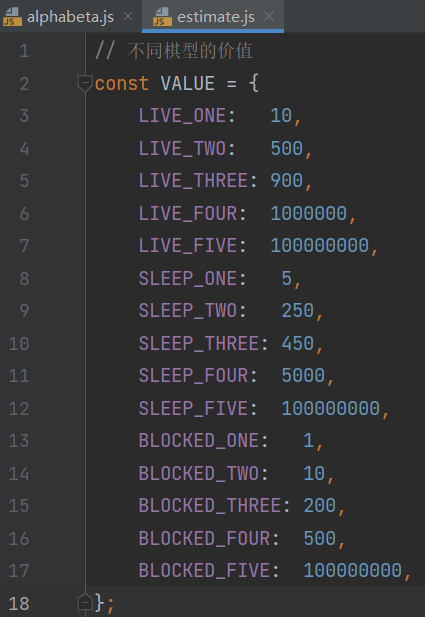
## 问题描述

采用 Alpha-Beta 剪枝搜索算法实现五子棋游戏（人机）

## 解决方案阐述

Idea打开项目，打开index.html，右上角选择edge浏览器运行，然后能看到如下画面

## 设计算法描述



Live 表示两头没堵，sleep表示堵了一头，blocked表示两头都堵，后面数字表示连子数量，冒号后面数字表示这种模式的分数。由于连子数量越多更容易胜利，堵得少更容易胜利，所以按照图里设置分数。

Alpha表示后续局面的下界，beta表示上界，由于博弈双方都是理智的，先手选择让alpha尽可能大，后手选择让beta尽可能小，对于那些不符合目标的局面不去搜索。

评价函数：对于棋盘上的己方棋子和对方棋子，分别计算每个棋子，在横竖斜4个方向的模式分数的和，的和，然后己方棋子的值减去对方棋子的值乘1.1，就是评价函数。

搜索深度为2层，对局面计算出评价函数，然后根据alpha-beta剪枝选择对当前操作者最好的局面

## 算法实现（即完整源程序，带注解）

文件1Index.html

<!DOCTYPE HTML>

<html lang="zh-CN">

<head>

<meta charset="UTF-8">

<title>五子棋</title>

<link href="./style/bootstrap.min.css" rel="stylesheet">

<link href="./style/main.css" rel="stylesheet">

</head>

<body>

<div class="container">

<div class="row">

<div class="col-md-8" style="background: url(aaa.jpeg)">

<canvas id="chess" width="480px" height="480px"></canvas>

</div>

<div class="col-md-4">

<div id="select">

<h4>请选择执棋颜色</h4>

<button class="btn btn-default" id="white">白棋</button>

<button class="btn btn-default" id="black">黑棋</button>

</div>

<button class="btn btn-default" id="reset">重新开始</button>

</div>

</div>

</div>

<script src="./script/min/jquery.min.js"></script>

<script src="./script/min/bootstrap.min.js"></script>

<script type="module" src="./script/index.js"></script>

</body>

</html>

文件2 index.js

import ChessBoard from './chess-board.js'

import {isVictory} from './estimate.js'

import nextPlace from './alphabeta.js'

// 棋盘

let chessBoard = new ChessBoard()

chessBoard.init()

chessBoard.draw()

// 棋子颜色

const BLACK = chessBoard.black

const WHITE = chessBoard.white

// 游戏是否结束

let isOver = false

// 是否为第一步

let isFirstStep = true

// 双方棋子颜色

let [my, enemy] = [null, null]

let selectDiv = document.getElementById('select')

let whiteBtn = document.getElementById('white')

let blackBtn = document.getElementById('black')

let resetBtn = document.getElementById('reset')

resetBtn.style.display='none'

whiteBtn.onclick = function() {

my = WHITE

enemy = BLACK

selectDiv.style.display='none'

resetBtn.style.display='block'

start()

}

blackBtn.onclick = function() {

my = BLACK

enemy = WHITE

selectDiv.style.display='none'

resetBtn.style.display='block'

start()

}

function start() {

// 如果我方是白棋，让黑棋走第一步

if (my === WHITE) {

chessBoard.oneStep(7, 7, BLACK)

chessBoard.initBorder(7, 7)

isFirstStep = false

}

chessBoard.chess.onclick = function(e) {

if (isOver) {

return

}

// 因为有内边距 padding，所以 x - padding 到 x + padding 范围内

// 都会落在 x 上，y 坐标同理

let i = Math.floor(e.offsetX / chessBoard.gridSize)

let j = Math.floor(e.offsetY / chessBoard.gridSize)

if (chessBoard.isEmpty(i, j)) {

if (isFirstStep) {

chessBoard.initBorder(i, j)

isFirstStep = false

}

chessBoard.oneStep(i, j, my)

if (isVictory(chessBoard, [i, j], my)) {

isOver = true

// temp solve delay problem

setTimeout("alert('my win')", 0)

} else {

[i, j] = nextPlace(chessBoard, enemy)

chessBoard.oneStep(i, j, enemy)

if (isVictory(chessBoard, [i, j], enemy)) {

isOver = true

// temp solve delay problem

setTimeout("alert('enemy win')", 0)

}

}

}

}

}

// 重新开始

resetBtn.onclick = function() {

isOver = false

isFirstStep = true

;[my, enemy] = [null, null]

chessBoard.chess.onclick = null

selectDiv.style.display='block'

resetBtn.style.display='none'

chessBoard.init()

chessBoard.setBorder(0, 15, 0, 15)

chessBoard.reDraw()

}

文件3 estimate.js

// 不同棋型的价值

const VALUE = {

LIVE\_ONE: 10,

LIVE\_TWO: 500,

LIVE\_THREE: 900,

LIVE\_FOUR: 1000000,

LIVE\_FIVE: 100000000,

SLEEP\_ONE: 5,

SLEEP\_TWO: 250,

SLEEP\_THREE: 450,

SLEEP\_FOUR: 5000,

SLEEP\_FIVE: 100000000,

BLOCKED\_ONE: 1,

BLOCKED\_TWO: 10,

BLOCKED\_THREE: 200,

BLOCKED\_FOUR: 500,

BLOCKED\_FIVE: 100000000,

};

// 保存四个方向每一行的棋子情况

let row = [];

let col = [];

let leftSlash = [];

let rightSlash = [];

// 根据连子数(ctn)和封堵数(blk)

// 给出一个评价值

function getValue(cnt, blk) {

if (blk === 0) { // 活棋

switch (cnt) {

case 1: return VALUE.LIVE\_ONE;

case 2: return VALUE.LIVE\_TWO;

case 3: return VALUE.LIVE\_THREE;

case 4: return VALUE.LIVE\_FOUR;

default: return VALUE.LIVE\_FIVE;

}

} else if (blk === 1) { // 眠棋

switch (cnt) {

case 1: return VALUE.SLEEP\_ONE;

case 2: return VALUE.SLEEP\_TWO;

case 3: return VALUE.SLEEP\_THREE;

case 4: return VALUE.SLEEP\_FOUR;

default: return VALUE.SLEEP\_FIVE;

}

} else { // 死棋

switch (cnt) {

case 1: return VALUE.BLOCKED\_ONE;

case 2: return VALUE.BLOCKED\_TWO;

case 3: return VALUE.BLOCKED\_THREE;

case 4: return VALUE.BLOCKED\_FOUR;

default: return VALUE.BLOCKED\_FIVE;

}

}

}

// 根据一行棋的情况

// 给出color棋在这一行的评值

function evaluateLine(line, color) {

let value = 0; // 评估值

let cnt = 0; // 连子数

let blk = 0; // 封闭数

const MY = color; // 己方

const OT = -color; // 对方

// 从左向右扫描

let lineLength = line.length;

for (let i = 0; i < lineLength; i++) {

if (line[i] === color) { // 找到第一个己方的棋子

// 还原计数

cnt = 1;

// 检查左侧是否封闭

if (i === 0 || line[i-1] === OT) {

// 如果棋子在棋盘的边界，或者上一个棋子为他方棋子

blk = 1;

} else {

blk = 0;

}

// 计算连子数

for (i = i+1; i < lineLength && line[i] == MY; i++) {

cnt++;

}

// 看右侧是否封闭

if (line[i] === OT || i === lineLength) {

blk++;

}

// 计算评估值

value += getValue(cnt, blk);

}

}

return value;

}

// 根据棋盘chessBoard状况

// 给出棋子颜色为color在chessBoard的评值

function evaluateState(chessBoard, color) {

const BOARD\_SIZE = chessBoard.boardSize

// 初始化(重置)行数组

for (let i=0; i<BOARD\_SIZE; i++) {

row[i] = [];

col[i] = [];

}

for (let i=0; i<BOARD\_SIZE\*2-1; i++) {

leftSlash[i] = [];

rightSlash[i] = [];

}

// 将chessBoard中的棋子分四个方向存储为单行值

for (let i = 0; i < BOARD\_SIZE; ++i){

for (let j = 0; j < BOARD\_SIZE; ++j){

row[j].push(chessBoard[i][j]);

col[i].push(chessBoard[i][j]);

leftSlash[j-i+BOARD\_SIZE-1].push(chessBoard[i][j]);

rightSlash[i+j].push(chessBoard[i][j]);

}

}

// 评值

let colorValue = 0;

let notColorValue = 0;

// 累加行状态评估值

for (let i=0; i<BOARD\_SIZE; i++) {

colorValue += evaluateLine(row[i], color);

notColorValue += evaluateLine(row[i], -color);

colorValue += evaluateLine(col[i], color);

notColorValue += evaluateLine(col[i], -color);

}

for (let i=0; i<BOARD\_SIZE\*2-1; i++) {

colorValue += evaluateLine(leftSlash[i], color);

notColorValue += evaluateLine(leftSlash[i], -color);

colorValue += evaluateLine(rightSlash[i], color);

notColorValue += evaluateLine(rightSlash[i], -color);

}

return colorValue-1.1\*notColorValue;

}

// 在 place 落 color 颜色棋子后，该棋子是否在[cx, cy]方向的这一行取得胜利

function victoryInLine(chessBoard, place, color, cx, cy) {

let cnt = 1;

let [i, j] = place;

let [x, y] = [i + cx, j + cy];

for (; chessBoard.is(x, y, color); x += cx, y += cy) {

cnt++;

}

[x, y] = [i - cx, j - cy];

for (; chessBoard.is(x, y, color); x -= cx, y -= cy) {

cnt++;

}

// 超过 5 颗棋子取得胜利

return cnt >= 5

}

// 判断 chessBoard 中 place 位置的 color 颜色棋子是否取得胜利

function isVictory(chessBoard, place, color) {

// 依次从四个方向判断是否取得胜利

return (

victoryInLine(chessBoard, place, color, 1, 0) // row

|| victoryInLine(chessBoard, place, color, 0, 1) // col

|| victoryInLine(chessBoard, place, color, 1, 1) // left, x+1,y+1 or x-1,y-1

|| victoryInLine(chessBoard, place, color, -1, 1) // right, x+1,y-1 or x-1,y+1

)

}

export {evaluateState, isVictory};

文件4 chess-board.js

class ChessBoard extends Array {

constructor(...args) {

super(...args)

// 棋盘大小为 16\*16 行列

this.boardSize = 16

// 棋格大小为 30\*30 的方格

this.gridSize = 30

// 棋盘内边距（棋盘必须有内边距，

// 不光是美观，同时可以为落子点判断提供左右范围）

this.padding = this.gridSize / 2

// 棋子半径

this.pieceRadius = 13

// 棋子颜色

this.empty = 0

this.black = 1

this.white = -1

// 搜索位置边界值

this.i\_min = 0

this.j\_min = 0

this.i\_max = this.boardSize

this.j\_max = this.boardSize

// 搜索边界每次扩充大小

this.range = 2

this.chess = document.querySelector('#chess')

this.context = this.chess.getContext('2d')

}

// 初始棋盘状态

init() {

for (let i = 0; i < this.boardSize; i++) {

this[i] = new Array(this.boardSize)

for (let j = 0; j < this.boardSize; j++) {

this[i][j] = this.empty

}

}

}

show() {

console.table(this)

}

// 绘制棋盘

draw() {

this.context.strokeStyle = "BFBFBF"

for (let i = 0; i < this.boardSize; i++) {

// 棋盘竖线

this.context.moveTo( // 起点

this.padding + i \* this.gridSize,

this.padding

)

this.context.lineTo( // 终点

this.padding + i \* this.gridSize,

this.padding + (this.boardSize - 1) \* this.gridSize

)

this.context.stroke()

// 棋盘横线

this.context.moveTo(

this.padding,

this.padding + i \* this.gridSize

)

this.context.lineTo(

this.padding + (this.boardSize - 1) \* this.gridSize,

this.padding + i \* this.gridSize

)

this.context.stroke()

}

for (let i = 3; i < this.boardSize; i += 4) {

for (let j = 3; j < this.boardSize; j += 4) {

this.context.beginPath()

this.context.arc(

this.padding + i \* this.gridSize, // x

this.padding + j \* this.gridSize, // y

this.gridSize / 6, // radius

0, // startAngle

2 \* Math.PI // endAngle

)

this.context.closePath()

this.context.fill()

}

}

}

// 重新绘制棋盘

reDraw() {

this.chess.setAttribute('height', `${this.boardSize \* this.gridSize}px`)

this.draw()

}

// 在i, j位置落color棋子

oneStep(i, j, color) {

this.resetBorder(i, j)

this.context.beginPath()

this.context.arc(

this.padding + i \* this.gridSize, // x 坐标

this.padding + j \* this.gridSize, // y 坐标

this.pieceRadius, // 半径

0, // startAngle

2 \* Math.PI // endAngle

)

this.context.closePath()

let gradient = this.context.createRadialGradient(

this.padding + i \* this.gridSize + 2, // x1

this.padding + j \* this.gridSize - 2, // y1

this.pieceRadius, // r1

this.padding + i \* this.gridSize + 2, // x2

this.padding + j \* this.gridSize - 2, // y2

0 // r2

)

if (color == this.black) {

gradient.addColorStop(0, "#0A0A0A")

gradient.addColorStop(1, "#636766")

this[i][j] = this.black

} else if (color == this.white) {

gradient.addColorStop(0, "#D1D1D1")

gradient.addColorStop(1, "#F9F9F9")

this[i][j] = this.white

}

this.context.fillStyle = gradient

this.context.fill()

}

// 根据第一次落子位置x, y初始搜索边界

initBorder(x, y) {

if (x - this.range >= 0) {

this.i\_min = x - this.range

}

if (x + this.range <= this.boardSize) {

this.i\_max = x + this.range

}

if (y - this.range >= 0) {

this.j\_min = y - this.range

}

if (y + this.range <= this.boardSize) {

this.j\_max = y + this.range

}

}

// 根据非第一次落子位置x, y重置边界

resetBorder(x, y) {

const [i\_min, i\_max, j\_min, j\_max] = this.getBorder()

if (

x - this.range >= 0

&& i\_min > x - this.range

) {

this.i\_min = x - this.range

}

if (

x + this.range <= this.boardSize

&& i\_max < x + this.range

) {

this.i\_max = x + this.range

}

if (

y - this.range >= 0

&& j\_min > y - this.range

) {

this.j\_min = y - this.range

}

if (

y + this.range <= this.boardSize

&& j\_max < y + this.range

) {

this.j\_max = y + this.range

}

}

setBorder(i\_min, i\_max, j\_min, j\_max) {

this.i\_min = i\_min

this.i\_max = i\_max

this.j\_min = j\_min

this.j\_max = j\_max

}

getBorder() {

return [this.i\_min, this.i\_max, this.j\_min, this.j\_max]

}

is(i, j, color) {

return this[i] && this[i][j] === color

}

isEmpty(i, j) {

return this.is(i, j, this.empty)

}

isBlack(i, j) {

return this.is(i, j, this.black)

}

isWhite(i, j) {

return this.is(i, j, this.white)

}

// 根据棋盘情况

// 返回可以落子的位置

possiblePlaces() {

let places = []

let [i\_min, i\_max, j\_min, j\_max] = this.getBorder()

for (let i = i\_min; i < i\_max; i++) {

for (let j = j\_min; j < j\_max; j++) {

if (this.isEmpty(i, j)) {

places.push([i, j])

}

}

}

return places

}

}

export default ChessBoard;

文件5 alphabeta.js

import {evaluateState} from './estimate.js'

// 博弈树探索深度

let LIMIT\_DEPTH = 2

// 1. 记录搜索博弈树落子前旧的边界

// 2. 在 place 位置落子，并得到落子后的棋局估值

// 3. 恢复落子前的边界值与棋盘状态

function getWeight(chessBoard, alpha, beta, color, searchDepth, place, isMax) {

// 保存旧边界

let [old\_i\_min, old\_i\_max, old\_j\_min, old\_j\_max] = chessBoard.getBorder();

// 落子并更新搜索边界

let [i, j] = place;

if (isMax) {

// max 层，color 一方落子，取 color 能得到最大估值的情况

chessBoard[i][j] = color;

} else {

// min 层，color 对方落子，取 color 能得到最小估值的情况

chessBoard[i][j] = -color;

}

chessBoard.resetBorder(i, j);

let weight = alphabeta(chessBoard, alpha, beta, color, searchDepth+1);

// 恢复棋盘上一个状态与边界值

chessBoard[i][j] = chessBoard.empty;

chessBoard.setBorder(old\_i\_min, old\_i\_max, old\_j\_min, old\_j\_max);

return weight;

}

// 从博弈树的 0 层进入，递归搜索可能的棋盘状态

// 其他层返回颜色为 color 的棋子在此层的估值( min 值或 max 值)

// 递归结束后回到 0 层，返回 color 棋子最优落子位置

function alphabeta(chessBoard, alpha, beta, color, searchDepth) {

// 此层是取极大值还是极小值

let isMin = searchDepth % 2 === 1

let isMax = !isMin

// 此层是否为 top 层（top 层也是 max 层）

let isTop = searchDepth === 0

// 到探索树叶子节点，直接返回 color 一方在这种情况下的估值

if (searchDepth >= LIMIT\_DEPTH) {

// console.log(searchDepth, place)

return evaluateState(chessBoard, color);

}

// 否则继续向下探索，估值由下层节点确定

// top 层，color 一方落子，取 color 能得到最大估值的情况

// 并返回 color 取得最大估值时的位置

if (isTop) {

let max = -Infinity;

let maxPlace = null;

for (let place of chessBoard.possiblePlaces()) {

let weight = getWeight(chessBoard, alpha, beta, color, searchDepth, place, isMax);

// top/max 层取最大值

if (max < weight) {

max = weight;

alpha = max;

maxPlace = place;

}

// if (alpha < max) {

// alpha = max;

// maxPlace = place;

// }

// top 层不存在上一层，所以这里不用剪枝

// beta cut-off: 上一层的beta大于这层的alpha，则不用再继续搜索

// if (beta <= alpha) {

// break;

// }

}

return maxPlace;

}

// max 层，color 一方落子，取 color 能得到最大估值的情况

if (isMax) {

let max = -Infinity;

for (let place of chessBoard.possiblePlaces()) {

let weight = getWeight(chessBoard, alpha, beta, color, searchDepth, place, isMax);

// max 层取最大值

if (max < weight) {

max = weight;

}

if (alpha < max) {

alpha = max;

}

// beta cut-off: 上一层的 beta 大于这层的 alpha，则不用在继续搜索

if (beta <= alpha) {

break;

}

}

return max;

} else {

// min 层，color 对方落子，取 color 能得到最小估值的情况

let min = +Infinity;

for (let place of chessBoard.possiblePlaces()) {

let weight = getWeight(chessBoard, alpha, beta, color, searchDepth, place, isMax);

// min 层取最小值

if (min > weight) {

min = weight;

}

if (beta > min) {

beta = min;

}

// alpha cut-off: 上一层的 alpha 已经大于这层的 beta，则不用继续搜索

if (beta <= alpha) {

break;

}

}

return min;

}

}

// 根据棋盘情况

// 返回 color 棋子下一个应该落子的位置

function nextPlace(chessBoard, color) {

let alpha = -Infinity;

let beta = +Infinity;

return alphabeta(chessBoard, alpha, beta, color, 0);

}

export default nextPlace;

## 实验结果测试与分析

图片略

AI和人类互有输赢。设置搜索深度为2，速度很快。设置搜索深度为3，AI反应速度略下降。深度为4，AI反应速度非常慢。

## 思考及学习心得

进一步理解了minmax算法，以及alpha-Beta剪枝是如何对其优化的