亚马逊棋实验报告

刘振宇 元培学院 1700017853

概述

本次实验题目为“亚马逊棋”[1]，主要是要求写一个可以进行亚马逊棋人工对弈的 C 程序，要求程序具有菜单选择、字符界面、存盘复盘等简单能力，同时要求 AI 具有一定的思考能力。

本次实验题目由我自己一个人完成，部分代码（如常量定义、变量声明、穷举所有情况）来自老师提供的 botzone 上的随机策略样例程序[1]。其他所有内容，包括界面设计，走棋合法性判断、存盘复盘、AI 策略等等内容全部由自己独立完成。

游戏提供了多种不同的模式，也提供了多种不同难度的 AI 可供用户选择，并且可以实现存档读档，基本完全达到了实验题目的基本要求。

在 AI 和核心算法部分，本程序最大的亮点在于实现了一个过程复杂但行之有效的打分方式，对当前局势下各种走棋方式的优劣程度进行打分。利用这种打分方式，搭建起了整个 AI 的行棋策略。

项目结构

1. 头文件：score.h

这个头文件主要包含了用于对当前局势进行打分的一系列函数以及其辅助函数。本程序采用了一个较为复杂的打分算法：基于当前局势计算 5 个特征值：评价占地情况的 territory 特征值 t1, t2，评价位置特征、地理优势的 position 特征值 c1, c2，和评价棋子灵活度的 mobility 特征值 m，对于每种局势下的打分为上述五个特征值的加权平均，每个特征值的具体含义与计算方法在之后的打分算法部分有详细解释。

头文件 score.h 中包含了 14 个函数和一系列的常量定义，变量声明。利用这些函数的组合可以对当前局势分别计算 5 个特征值，并计算加权平均。具体每个函数的作用，函数之间的相互关系会在之后的打分算法部分详细叙述。

2. 主程序：主程序.cpp

主程序中首先定义了几个辅助函数：

- bool Check_End(int color, bool print_result)：检查游戏是否结束，结束返回 true；
- int Cntlocked(int color)：计算某种颜色棋子中被锁死不能移动的个数，方便之后行棋策略的确定
- void PrintMap()：在屏幕上打印棋盘信息，棋子用"○"和"●"表示，障碍用"Δ"表示。
- void PrintMatrix(int a[8][8]：以棋盘的模式打印某个矩阵的信息，仅仅在调试阶段使用。

在这几个辅助函数的支撑下，进一步定义了几个下棋策略的函数，分别是：

- void RandStep()：采取随机的策略，先列举所有的策略，然后随机选择其一。在老师提供的范例基础上仅仅增加输出了一些提示信息。
- void ScoreOnly(int color)：先穷举所有可能的情况，然后对每一种情况进行打分，在打分时，5 个特征值采取各自固定的权重，选择打分最高的一步棋。
- void Lock_Score(int color, int ID)：先检查有没有致胜棋，如果有的话直接行棋。之后检查能不能通过行棋锁死对方的棋子，如果可以的话就行棋。再之后检查能不能防止自己的棋子被锁死，可以则行棋。上述都不满足的话就对所有情况打分，而且

随着棋局的进行自动改变每个特征值在打分中所占的权重，选择打分最高的一步行棋。具体的行棋策略在之后详述。

主函数为游戏的主体，进入主函数之后首先打印菜单界面，并根据用户的输入确定游戏模式并初始化棋盘，这里用一个 switch 语句来分支到各个模式：

Case 1：人机对弈模式，模式中首先允许用户选择 AI 的难度，以及自己所执子的颜色，黑色先行棋。确认好这些游戏基本信息后开始进行游戏。通过一个永真循环来进行用户、计算机的分别行棋，并在每个人行棋之前检查游戏是否结束，结束则跳出循环并输出胜方信息。

在用户输入指令时，可以通过输入‘s’来进行存盘，实际上就是把当前的棋盘数组信息、用户执子颜色以及所选 AI 难度以此写入到一个文件中，之后可以进行读盘。存盘完成后，询问用户是否继续游戏，不继续则退出。

Case 2: 人人对弈模式，这个模式的代码实现较为简单，只需要两方依次输入，并在每一方行棋之前检查游戏是否结束即可。

Case 3: 两个 AI 之间对弈，对用户没有使用价值，主要是用于调试阶段。通过计算机之间的快速对弈检查能否正确判断游戏结束，并且可以用来对不同 AI 算法的优劣进行比较，从而不断优化算法和参数（避免 botzone 上对局要排队，而且还要修改代码输入输出的的问题）。

Case 4：读盘。即从上次存盘的文件中读取当前的棋盘数组信息、用户执子颜色以及所选 AI 难度，并在当前局势下继续游戏。读盘继续进行的游戏可以正常存盘。

Case 5：退出程序

主程序之后有一段被注释掉的内容，是用于调试阶段遍历棋盘，验证两次深度优先搜索的结果以及打分函数正确性所用的，对程序正常运行无价值。

3. 存档文件：Archive.txt

以文本文档的形式储存当前的棋盘信息（gridInfo），用户所执子颜色和所选的 AI 难度，数据之间用空格分开。

亮点：打分算法

前文中提到，程序最大的亮点在于采用了一个较为复杂庞大但是行之有效的打分算法。

利用 AI 实现对弈，最简单的办法当然是穷举所用的走法，选其中最好的一个。但是如何评价一个走法是好还是坏，这是一个很难量化的问题。我能想到的当然是：首先，能致胜的走法当然最好，没有致胜走法就优先走能锁住别人棋子的方法；如果还没有，就想办法把自己可能被锁住的棋子移开。但是这些情况毕竟是少数，大多数情况下棋局中都没有满足这三种情况的走法，这时候量化打分每一个走法的作用就体现出来了。

1. 打分原理：

通过查阅已有文献，我找到了 5 个可以量化评价当前局势的特征值[2]，评价占地情况的 territory 特征值 t1, t2，评价位置特征、地理优势的 position 特征值 c1, c2，和评价棋子灵活度的 mobility 特征值 m。这 5 个特征值从 3 个方面量化了当前局势对两方的优劣程度，具体每个特征值的意义，计算方法展示如下[2]：

- territory 特征值：t1, t2

territory 特征值反应的是两方对棋盘格的占领情况。对于一个空的棋盘格，分别计算黑，白两种棋子到达这个格点所需要的最小步数，认为需要步数少的一方占领了这个空格。通过对全局所用空格的判断，可以确定哪一方占领的空格数目更多，从而评价局势优劣。

这里涉及到的走棋方式有两种：Queenmove 和 Kingmove，前者是指向 8 个方向上只要没有障碍就可以随意移动，而后者则是指只能一次向八个方向之一移动一步。

在这两种行棋方式下，可以分别计算出每个某一方到达某个空格 a 的最小步数$D_i^j(a)$，其中 i 表示走法，i=1 表示采用 Queenmove，i=2 表示采用 Kingmove。j 表示棋子颜色，j=1 表示白色，j=2 表示黑色。为了帮助理解，图 1 中展示了一种情况下的$D_i^j(a)$，左侧为 Queenmove，右侧为 Kingmove，每个空格中左侧的点为白色棋子到达需要的最小步数，右下角为黑色棋子最少步数。

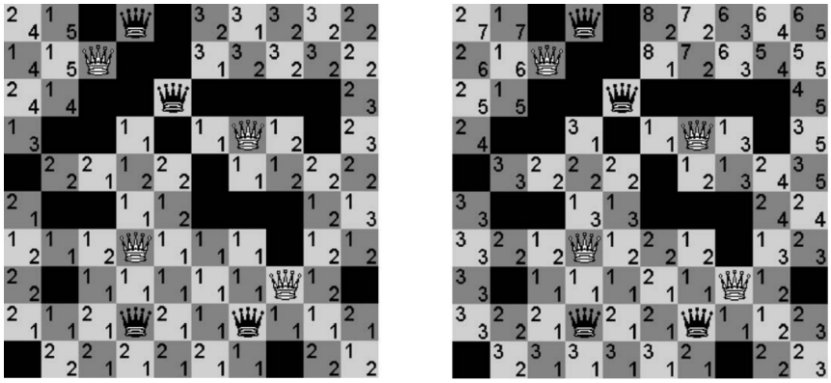

图 1 两种走法下两种颜色到每个空格的最小步数（左为 Queenmove，右为 Kingmove）

跟根据计算出来的每个空格的$D_i^j(a)$值，可以计算在 Queenmove 和 Kingmove 下的 territory 特征值，其计算公式为

$$t_i = \sum_{\text{empty squares } a} \Delta(D_i^1(a), D_i^2(a)),$$

其中：

$$\Delta(n,m) = \begin{cases} 0 & \text{if } n = m = \infty, \\ \kappa & \text{if } n = m < \infty, \\ 1 & \text{if } n < m, \\ -1 & \text{if } n > m, \end{cases}$$

i 取 1, 2 分别计算，分别得到在 Queenmove 和 Kingmove 下的两个 territory 特征值 t1，t2。可以看出，对于一个空格，如果白色棋子需要的步数少（即白色控制了这个空格），则对应 t 加 1，反之则减 1。于是，t 越大说明白色控制的范围越大，黑色控制的范围越小。

- position 特征值:c1, c2

可以看出，territory 特征值只能反映两种走法下，两种颜色对空格的控制权，但是并不能反映出控制权的差值以及其反应的位置特征。因为在 territory 特征值中，只关心哪一个步数更小，但是不关心小的幅度有多大。因此需要引入 position 特征值来描述这一位置特征。文献中给出的 position 特征值计算公式如下：

$$c_1 = 2 \sum_{\text{empty squares } a} 2^{-D_1^1(a)} - 2^{-D_1^2(a)},$$

$$c_2 = \sum_{\text{empty squares } a} \min(1, \max(-1, (D_2^2(a) - D_2^1(a))/6)).$$

文中表示这个公式是一个经验公式，并没有严格的推导过程。从公式中我们可以看出，$D_1^1(a)$越小，$D_1^2(a)$越大，c1 特征值越大，说明白色棋子离空格的步数越少，对应的打分越高。当白，黑到一个空格的距离分别为 1, 2 时，c1=0.5，分别为 2, 3 时，c1=0.25，分别为 1, 3 时，c1=0.75，从而可以反映出距离差值大小，进而表示出位置信息。

- mobility 特征值：m

亚马逊棋的最终目的是把对方的棋子全部锁死，因此两方被锁住的棋子也是局势的很重要的一部分。但是从前面的讨论中可以看出，territory 和 position 特征值都只关心对空格的

控制情况，但是没有考虑某一方的棋子被限制的情况，因此需要引入第五个特征值：灵活度特征值来反应某一方棋子移动的灵活度。

灵活度特征值的计算方法如下：

步骤 1：计算棋盘中每一个空格的灵活度（即每个空格周围的 8 个邻居中空格的个数）

步骤 2：记录棋子 a 采用 Queenmove 走法一步之内所能到达的所有空格

步骤 3：对于 2 中的每一个空格，将其空格的灵活度除以当前空格到棋子 a 的距离，把得到的结果相加，即得到当前棋子 a 的灵活度。

例如，对图 2 中左上角的白棋，它的灵活度为 F(a) = 7 / 1 + 6 / 1 + 5 / 1 + 3 / 1 + 3 / 1 + 5 / 2 + 4 / 2 + 7 / 2 + 4 / 2 + 5 / 3 = 35.25。

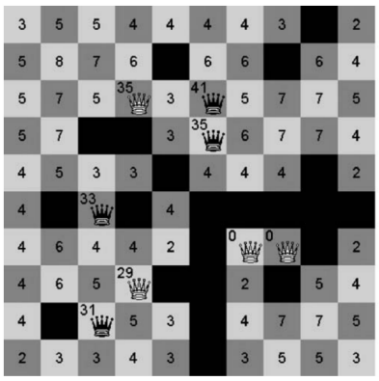

图 2 灵活度的计算

而 m 特征值，即为白色棋子灵活度之和减去黑色棋子灵活度之和：

$$m = \sum_{w} F(w) - \sum_{b} F(b)$$

可以看出，m 特征值越大，对应白色棋子灵活度越大，黑色棋子灵活度越小。

有了上面的 5 个特征值，我们可以把他们加权平均得到最终对局势的总打分值：

$$score = t1 \times a + t2 \times b + c1 \times c + c2 \times d + m \times e$$

公式中的 abcde 为 5 个特征值的权重，具体的权重留作之后讨论。

2. 代码实现：

打分算法的所有函数都被封装在头文件 score.h 中，下面简单介绍上述打分方法的代码实现。

首先可以看出，territory 特征值和 position 特征值的计算都依赖于计算出一种颜色的棋子到达某个空格的最小步数，因此首先需要计算出两种棋子在两种行棋方式下，到达每一个空格的最小步数，储存在四个二维数组

```
int KingMove_w[GRIDSIZE][GRIDSIZE];
int KingMove_b[GRIDSIZE][GRIDSIZE];
int QueenMove_w[GRIDSIZE][GRIDSIZE];
int QueenMove_b[GRIDSIZE][GRIDSIZE];
```

中。

对于 kingmove 的计算较为简单，只需要从每一个棋子出发，向四个方向进行深度优先收缩(BFS)，搜索到每一个空格时和当前储存的这个空格的最小步数比较，取较小者储存即可。score.h 中的函数 void King_BFS(int x, int y, int cnt)即为从每个棋子开始进行深度优先搜索，void CalKingMove(int color)函数用于找到每个棋子，并储存当前颜色到达每个空格需要的最小步数。如果不能到达，则把这一点的值记为 999。

对于 Queenmove 的计算思路与 Kingmove 类似，但是由于每一步可以向 8 个方向走任意步，对应的情况远多于后者，而且很容易出现不同递归层数之间搜索区域的重合，因此需要进行适当的剪枝。比如当当前步数已经多余对应节点储存的步数时，直接放弃后续对这个节点的搜索等。Queenmove 步数的计算同样通过 void Queen_BFS(int x, int y, int cnt)和 void CalQueenMove(int color)两个函数实现。如果无法到达同样记为 999。

为了验证上述两个深度优先搜索函数的程序正确性，我对初始棋盘上每一个空格相对于白棋的 Queenmove 和 Kingmove 步数进行了计算（图 3），结果显示两个函数工作正常。

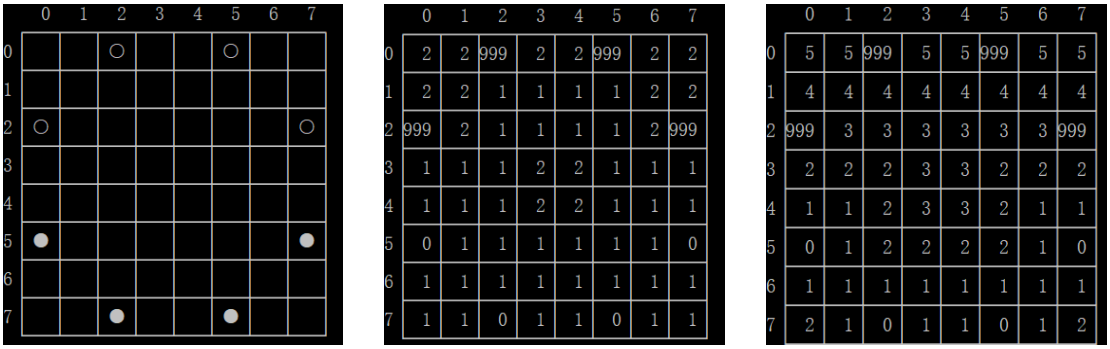

图 3 对 Queenmove 和 Kingmove 计算正确性的验证

(左：棋盘信息，中：Queenmove 不熟，右：Kingmove 步数)

有了每一个空格相对于某种颜色的最少移动次数，就可以据此计算出两个 territory 特征值和两个 position 特征值，函数

int territory(int w[GRIDSIZE][GRIDSIZE], int b[GRIDSIZE][GRIDSIZE]);

double position1(int w[GRIDSIZE][GRIDSIZE], int b[GRIDSIZE][GRIDSIZE]);

double position2(int w[GRIDSIZE][GRIDSIZE], int b[GRIDSIZE][GRIDSIZE])

即是用来计算这 4 个特征值。

之后就是来计算 mobility 特征值，根据步骤，首先计算每一个空格的灵活度，函数 void CalBlankMobility()函数便是实现此功能。之后，根据空白的灵活度计算每一个棋子的灵活度，函数 double CalPieceMobility()实现这个功能。最后则是计算每种颜色的灵活度，使用函数 double CalColorMobility()。

有了上面的各个函数，便可以计算出所有 5 个特征值，并把它们加权平均得到对当前局势的总打分。函数 double CalScore(double a = 0.2, double b = 0.2, double c = 0.2, double d = 0.2, double e = 0.2)便是最终的打分函数。它可以依次调用上面提到的各个函数，依次计算出 5 个特征值，并把它们加权平均，返回最终的得分值。函数的 5 个参数即为对应的权重，如果不认为规定，默认值为平均权重，均为 0.2。

3. 权重讨论：

对于计算得到的 5 个特征值，其权重分配也是决定打分算法优劣的一个重要部分。每个特征值均针对的是当前局势的一个方面，而在比赛的不同阶段，对于不同因素的考虑权重有较大的差异，因此需要根据比赛的进行适当调整 5 个特征值的权重[3]。

首先考虑 t1 特征值，它在比赛中的作用应该是越来越大。因为在比赛的初期，大家采取，障碍较少，大家采取 Queenmove 控制的空格数相仿，圈地必要性不大。但是当进行到了中、后期阶段，大家的棋子往往被分隔开，棋盘被划分成小的区域，这时候尽量占领最多的空格意味着更大的圈地面积。所以 t1 的权重应该逐渐增大。

之后考虑 t2 特征值，t2 特征值主要是基于 Kingmove 走法，实际是争抢自己棋子周围紧挨着的空格。但是当游戏进行到了后期，棋子周围往往有较多的障碍，对紧挨着的空格争抢意义不大。而且往往棋盘被分割成许多个小的部分，一块圈地一旦被一方占领就不存在对方的进攻问题，此时 t2 作用几乎为 0。

对于 c1，c2，主要是反应棋子在棋盘中的位置分布。在前期，可以根据位置分布调整走棋位置，但是中后期，由于较多障碍存在，棋子的分布受到限制，位置分布的意义不大。而且一旦圈地形成之后，在棋子圈地中的位置对局势几乎没有影响，因此其权重逐渐递减。

最后，对于 m 特征值其对局势的影响也是呈下降趋势，因为到了中后期，灵活度普遍受到较大的控制，这时候只有圈地大小对局势的影响最大。

基于上述讨论，我决定认为把棋局划分为前、中、后三个阶段，每个阶段采用不同的权重打分，划分的依据为已经进行的走棋步数。为了评判不同参数条件下打分的效果，我采了的战胜随机策略 AI 所需要的平均步数作为评价标准（两个有逻辑的 AI 之间对弈结果固定，而且往往出现交换先后手前后一胜一负的情况，难以评估），通过多次尝试，确定了各个阶段的较为合理的权重（表 1），这样的权重下战胜随机数 AI 平均需要 30 步左右。

表 1 评估因子的权重

	评估特征	territory		position		mobility
	特征值	t1	t2	c1	c2	m
	权重	a	b	c	d	e
阶	前期	0.22	0,47	0.13	0.13	0.05
段	中期	0.30	0.28	0.20	0.20	0.02
	后期	0.80	0.10	0.05	0.05	0

收获

通过完成亚马逊棋的实验题，我感觉自己的收获很多，体现在一下几个方面：

1. 学习并了解了 AI 下棋的简单工作原理和实现过程。
2. 提高了自己的编程能力，练习使用课堂上学习到的编程技巧来解决实际遇到的问题。
3. 通过对 5 个特征值权重参数的目测胜率+手动调整，第一次真切感受到了调参对于一个算法的重要作用。
4. 为了更好地调整上述的 5 个特征值的权重参数，得到最合理的参数，我自己通过搜集资料，查阅文献等方式，学习了通过机器学习的算法进行调参的基本原理以及简单的实现方法。但是由于对应算法较为复杂，无法完全理解，随作罢。
5. 为了得到能力更强的 AI，需要进行更深层级的搜索。我通过查阅资料，学习了极大值极小值的算法，以及 alpha-beta 剪枝算法的原理。但是由于 AI 算法部分提交时间较早，未能完全实现。

未来改进

1. 对于错误输入的容错率不够。

这次的亚马逊棋程序对用户的输入格式有很高的要求，必须严格按照格式输入才能进行。虽然我在读入数据阶段做了很多优化，基本可以保证只要输入的是数字，即使数据有明显问题也能正确报错，并提醒用户重新输入。但是如果用户输入的不适数字，就会导致 cin 的错误，进而导致整个程序无法正常进行。

后续的优化过程可以把用户输入全部读入到字符串中，在进行处理，这样就不会出现 cin 读入时数据类型出现错误。

2. 无法存在多个存档

由于在程序中存盘，读盘的文件名都是固定的，因此目录下只能存在一个存档文件，无法存在多个存档，后续可以继续改进。

3. 参数优化问题

目前使用的参数是我自己通过 AI 对弈结果，手动调整得到的一组参数，当然不是最好的参数。将来希望通过简单的机器学习方法，通过计算机实现 AI 的自动对弈和自动调参，从而得到更合理的权重参数。

4. 搜索层数不够

这也许是当前程序最大的硬伤：只进行了一层搜索。虽然打分函数很复杂，但是层数不够导致对可用算力的利用率很低。之后可以通过极大值极小值算法进行多层搜索打分，同时利用 alpha-beta 剪枝算法减小搜索分支数，得到搜索更深的 AI，也许能够使 AI 的对弈水平得到极大的提升。

参考文献

[1] Amazons: Botzone wiki. https://wiki.botzone.org.cn/index.php?title=Amazons. 2018.11.14

[2] Lieberum J . An evaluation function for the game of amazons[J]. Theoretical Computer Science, 2004, 349(2):230-244.

[3] 郭琴琴, 李淑琴, 包华. 亚马逊棋机器博弈系统中评估函数的研究[J]. 计算机工程与应用, 2012, 48(34):50-54.