天津大学

《人工智能基础》项目实现

与/或图启发式搜索实现双跳马博弈程序

学　　院　智能与计算学部

年　　级　2018

专业班级　新工科试验班一班

学　　号　3018205377

姓　　名　李渤涟

邮　　箱　lblaoke@foxmail.com

2020 年 5 月 20 日

目录

一、实践目的

1. 知识掌握

学习与/或图启发式搜索的原理与方法；掌握双跳马博弈问题中的启发式函数设计技巧。

2. 程序实现

掌握状态、搜索图、启发式搜索、trade-off、max-min 方法和 α－β 剪枝的建模及 C++语言实现。

二、开发环境

1. 操作系统

Windows 10 家庭中文版

2. 编译环境

gcc version 8.2.0 (MinGW.org GCC-8.2.0-5)

三、项目内容

1. 双跳马博弈问题及其分析

双跳马博弈是对跳马问题的扩展，博弈双方交替移动，只需要等到双方中的一方无路可走时，就可以分出胜负。

经典的跳马问题中，胜负的关键在于是否有足够的潜在走法供选择。而在双跳马博弈中，不仅要考虑自己的潜在走法数量，还要考虑对手的潜在走法数量。在双方的棋子距离足够近的情况下，任何一方的移动都有可能对双方的潜在走法数量产生显著的影响。

令 A、B 双方在某一状态的潜在走法数量分别为 hA 和 hB，A 胜利的条件是 hA>0 且 hB=0，B 胜利的条件是 hA=0 且 hB>0。由此可以得到，A 的策略应该倾向于使 hA 更大而 hB 更小，B 则正好相反。

2. 项目整体架构

本项目包括文件“State.h”、“Search.h”、“MANvsAI.cpp”、“BLINDvsAI.cpp”、“AIvsAI.cpp”，分别是：对状态的建模、搜索图的搜索算法、玩家与 AI 进行游戏的主程序、简单程序与 AI 进行游戏的主程序以及 AI 与 AI 进行游戏的主程序。下面详细介绍每个文件中实现的功能：

“State.h”：一个状态的全部参数（棋盘、棋子位置、行棋方）、由一个状态生成其子状态的方法、计算当前节点启发值的方法、判断当前状态是否不能再展开的方法、打印当前状态对应的棋盘的方法、处理异常的方法；

“Search.h”：在一定的深度限制条件下，根据 max-min 方法搜索搜索图，并应用 α－β 剪枝合理地减小搜索量，最终得到当前节点的价值；

“MANvsAI.cpp”：在一个循环中使玩家和 AI 交替决策，并在胜负已分时退出程序；

“BLINDvsAI.cpp”：在一个循环中普通程序和 AI 交替决策，并在胜负已分时退出程序；

“AIvsAI.cpp”：在一个循环中使两个 AI 交替决策，并在胜负已分时退出程序。

所有.cpp 文件中的主程序都在功能上依赖“State.h”中的状态类和“Search.h”中的搜索函数，而且“Search.h”文件自身也依赖“State.h”中的状态类。

四、与/或图启发式搜索的一般过程

1. 与/或图规约的扩展（max-min 方法）和剪枝（α－β剪枝）

为了得到与/或图中一个节点的价值（A 为正，B 为负），需要递归展开其子状态，从相应的叶节点价值开始，逐层向上规约，最终计算得到该节点的价值。由于节点的价值可以表示为实数值，因此向上规约的过程可以使用 max-min 方法实现，对于每一个与节点（A 行动的节点），选择其子节点中的最大价值作为自己的价值，而对于每一个或节点（B 行动的节点），选择其子节点中的最小价值作为自己的价值，即：

$$value_A(S) = \max\{value_B(\{S'\})\}$$
$$value_B(S) = \min\{value_A(\{S'\})\}$$

其中{S’}表示 S 的子状态集合。

在展开子状态与规约价值的过程中，标记每个状态的价值的下界 α 和上界 β，并在展开子状态时将上下界传递给子状态，若一个状态的上界不大于下界（β ≤ α），则说明继续计算和展开当前状态不可能更新上层状态的价值，故可以跳过该节点。

2. 启发式搜索

对于每一个状态 S，利用启发式函数

$$\hat{f}(S) = \hat{g}(S) + \hat{h}(S)$$

给出其价值的估计，其中 g(S)表示从初始节点 S_0 到达 S 的最优路径的代价，h(S)表示从 S 到达目标状态（胜负已分）的最优路径的代价。上述表达式中的函数分别是对 g(S)和 h(S)的估计。

由于本项目时一个及时博弈游戏，每一步行棋都无法撤销，因此在 open 表的维护过程中，每当一步行棋完成，都要清空 open 表，并将下一步选择的行棋方式对应的状态（价值最大或最小）加入 open 表中。

3. 将与/或图和启发式搜索整合起来

为了将与/或图和启发式搜索结合起来，借鉴 trade-off 思想，先从一个节点开始将搜索图展开到一定的深度，在展开树的边界用节点的启发式函数值作为节点的价值，并通过 max-min 方法规约至原始节点，得到该节点的价值，具体算法如下所示：（以与节点的计算为例）

算法 4.3.1：与/或图规约方法
输入：当前状态 S,搜索深度限制 D，下界 α，上界 β 输出：$value_A(S)$

begin
　if S 的深度超过 D OR S 无子状态 then
　　return $\hat{f}(S)$
　end
　value←-∞
　for n∈S 的子状态 do
　　value←max{value,$value_B$(n,D,α,β)}
　　α←value
　　if α>=β then
　　　跳出循环
　　end

```
    end
    return value
end
```

在上述过程的基础上，确定行棋策略的方法是：根据上述过程计算子节点的价值，并找出最大或最小价值的节点作为下一步行棋的策略。详细步骤如下所示：（以或节点的策略为例）

算法 4.3.2：启发式搜索方法
输入：当前状态 S，搜索深度限制 D 输出：下一步对应的状态 S’

begin
S’ ← ε
minValue←+∞
for n∈S 的子状态 do
if minValue>$value_B$(n,D,-∞,+∞) then
S’ ←n
minValue←$value_B$(n,D,-∞,+∞)
end
end
return S’
end

五、实现细节

1. 状态与搜索图的建模及实现

在本项目的实现过程中，“状态”是整个程序逻辑的基石。使用 C++面向对象方法实现 class State，再在此基础上用状态连接成的数构成搜索图。如下图所示为“状态”的成员参数：

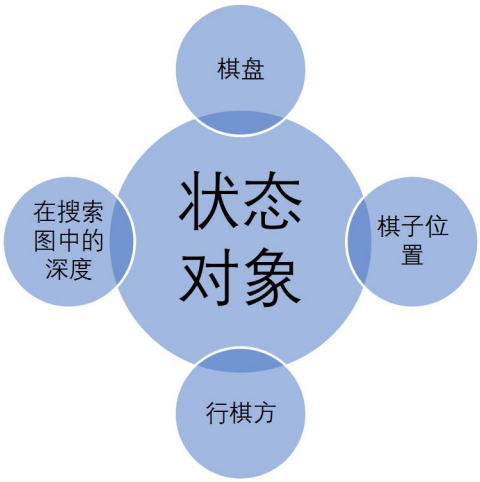


图 5.1.1 State 成员参数

此外，每个状态还应该包括至多 8 个指向其子状态的指针（因为每一步行棋都有至多 8 种选择）。由这些指针将状态对象连接成树状结构，就得到了搜索图。

在状态的构造函数中，要完成如下步骤：

①判断状态的合法性，必要时拒绝创建对象并退出；

②从父状态那里继承参数；

③根据预先指定的行棋方式更新状态参数；

④计算状态的价值估计，并暂存起来。

状态的生成和连接由搜索程序控制，搜索程序会决定生成哪些些状态，不生成哪些状态，以节省实现与内存。

2. 启发式函数设计

从初始状态到达当前状态的路径代价对于 A、B 双方而言都是相同的，因为双方交替行棋。为了体现“策略惯性”（执行过好策略的一方当前的策略可能更好），将该项代价的估计定义如下：

$$\hat{g}(S)=\begin{cases}0 & ,\text{S是初始状态}\\ 0.25\cdot\hat{f}(S_f) & ,\text{S不是初始状态}\end{cases}$$

其中 S 和 S_f 分别表示当前状态以及其父状态，系数 0.25 保证了历史信息在价值估计中所占的权重不会太大。

而对于从当先状态到目标状态的代价估计，使用当前棋子位置的局部信息完成。设 hA 和 hB 分别为 A、B 棋子周围的空交叉点数量，Size 为棋盘大小，则

$$\hat{h}(S)=(hA-hB)\sqrt{2\cdot Size-m(A,B)}$$

其中 m(A,B)表示 A、B 棋子之间的 Manhattan 距离，根号下的负距离项使得两棋子相距越近，两者的可行走法数越重要，体现了局面的生死攸关程度。

最后，将上述函数整合成完整的价值估计函数，并将已知胜负的状态的价值设为无穷，有

$$\hat{f}(S)=\begin{cases}+\infty & ,\text{A必胜}\\ -\infty & ,\text{B必胜}\\ \hat{g}(S)+\hat{h}(S) & ,\text{其他}\end{cases}$$

3. max-min 方法搜索搜索图

按照深度优先的顺序，从当前节点展开搜索图至深度限制，应用如下规则得到当前节点的价值：

①对于每一个达到搜索边界的状态，用 $\hat{f}(S)$ 作为自身的价值；

②对于每一个与状态，选择子状态价值中的最大值作为自身的价值；

③对于每一个或状态，选择子状态价值中的最小值作为自身的价值。

4. α－β 剪枝

在搜索过程中应用 α－β 剪枝的具体方法如算法 4.3.1 所示。

5. 交替行棋的流程

游戏主程序的逻辑如下流程图所示：

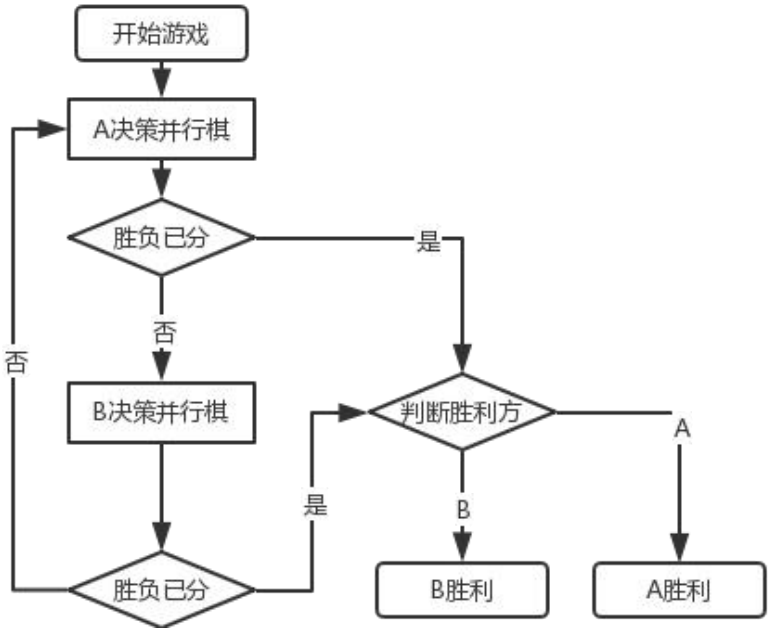


图 5.5.1 游戏主程序逻辑

六、程序效果

1. 人类玩家 vs 本程序

下表记录了人类玩家与本程序游戏时的胜负情况，具体的棋局情况详见附录。

搜索深度\AI 胜负\棋盘大小	6	8
2	负	胜
4	负	胜
8	胜	胜

表 6.1.1 MANvsAI

可以发现人类玩家在棋盘较小时有较高的胜率，因为小棋盘对应的搜索图本身比较简单，在小棋盘上体现不出程序的优势。

2. 瞎子爬山法程序 vs 本程序

下表记录了普通程序与本程序游戏时的胜负情况，具体的棋局情况详见附录。

搜索深度\AI 胜率\棋盘大小	6	8	10
4	0.5	0.6	0.8
8	0.7	0.7	0.8
12	0.9	0.8	*

表 6.2.1 BLINDvsAI

其中*表示搜索过程中内存溢出，游戏无法进行下去。

可以看出随着棋盘大小和搜索深度的增加，应用瞎子爬山法的程序越来越无法战胜本程序，因为瞎子爬山法过分依赖启发式函数，常常看不清发展数步后的局势。

3. 本程序 vs 本程序

下表记录了本程序自对局时的胜负情况，具体的棋局情况详见附录。

搜索深度\先手胜率\棋盘大小	6	8	10
4	0.4	0.5	0.4

8	0.4	0.4	0.7
12	0.3	*	*

表 6.3.1 AIvsAI

其中*表示搜索过程中内存溢出，游戏无法进行下去。

可以看出，先手胜率基本维持于 0.4~0.5，这说明双跳马博弈游戏很可能对后手更有利一些。

七、实践总结

1. 心得与收获

本项目的推进过程中，我对启发式搜索和与/或图有了更深刻的认识，充分感受到了专业的人工智能方法的简洁与强大。借由此项目，我掌握了设计博弈游戏 AI 的一般方法，理解了将本项目的方法推广到其他游戏的途径。最重要的是，我深刻理解到了人工智能搜索的特质，即动态性和局部性，这为我进一步学习人工智能打下了基础。

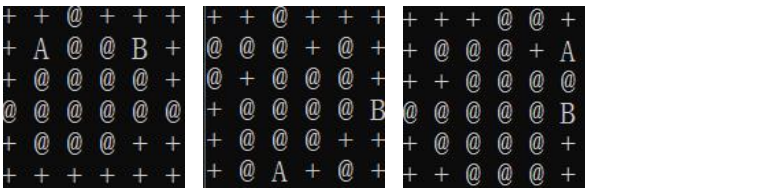
2. 开发过程中遇到的困难

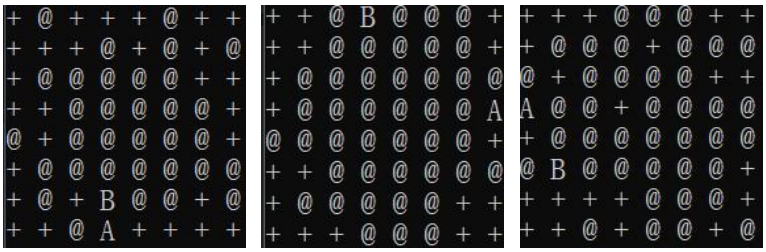
程序的开发过程中我遇到了很多困难，例如：不知道该如何用指针将状态连接成网络、不知道怎么将启发式搜索应用于与/或图。这些困难的解决往往伴随着进度的加速推进和理解程度的加深，正是由于我解决了这些困难，我才能完成项目并对知识有深入的理解。

八、附录：实验数据终局棋盘状态

下面列出了实验中得到的终局棋盘状态，其中“+”表示没有走过的交叉点，“@”表示已经走过的交叉点。

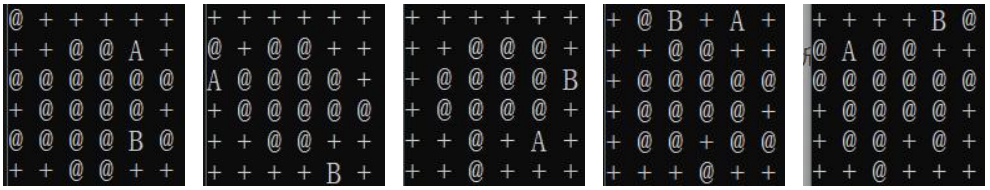
1. 人类玩家 vs 本程序

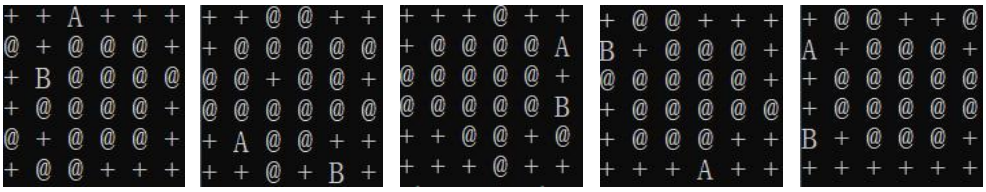




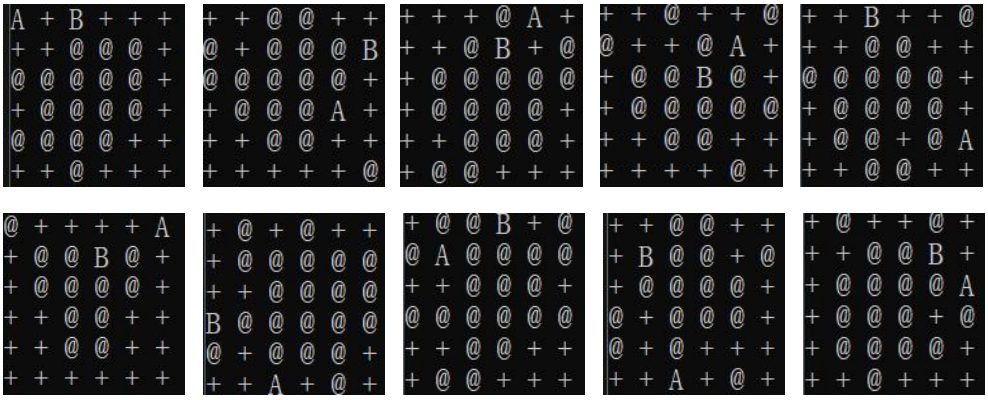
2. 瞎子爬山法程序 vs 本程序

2.1. 6×6 棋盘，深度 4

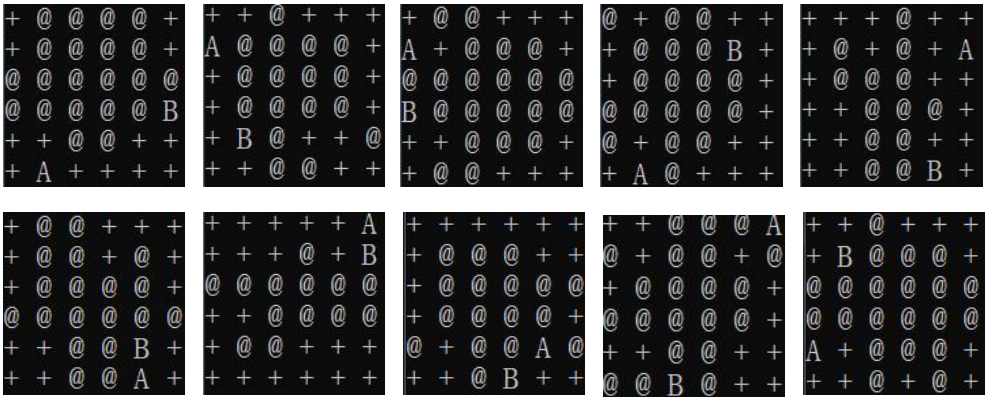




2.2. 6×6 棋盘，深度 8



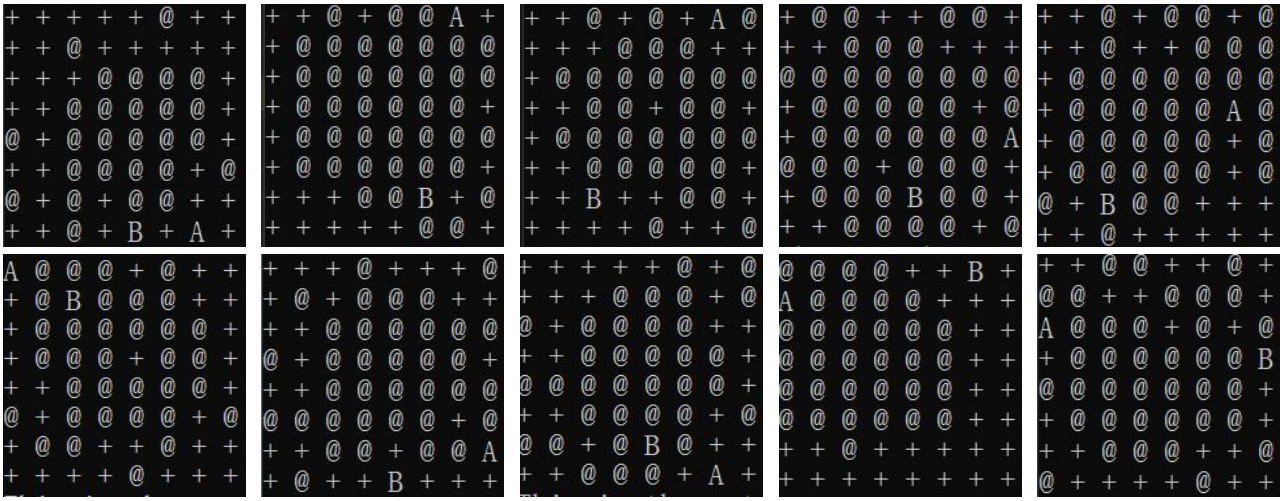
2.3. 6×6 棋盘，深度 12



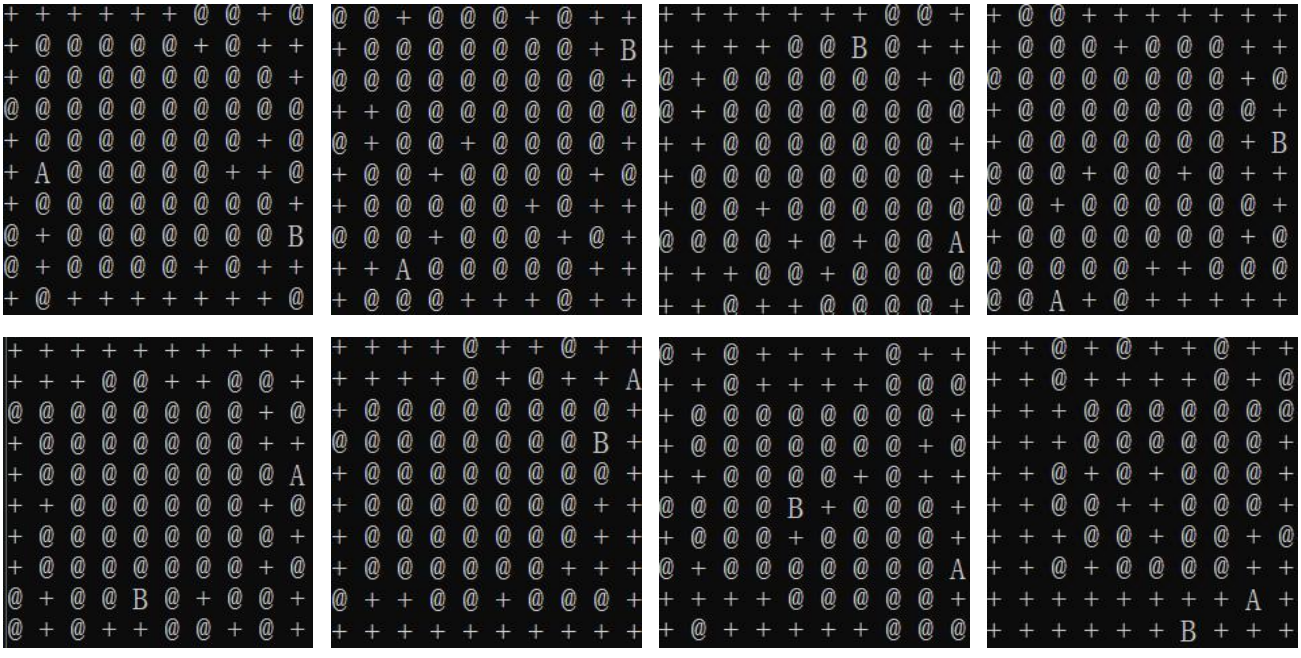
2.4. 8×8 棋盘，深度 4

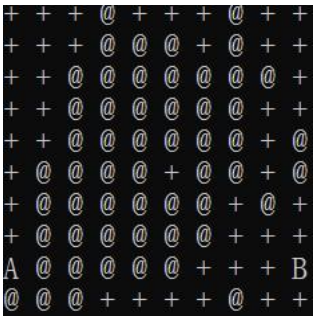
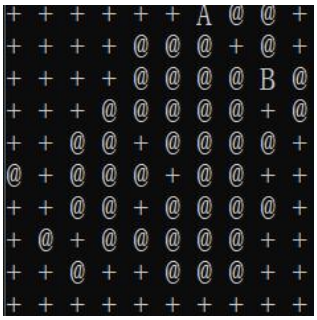
2.5. 8×8 棋盘，深度 8

2.6. 8×8 棋盘，深度 12

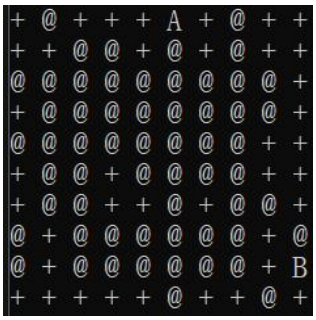
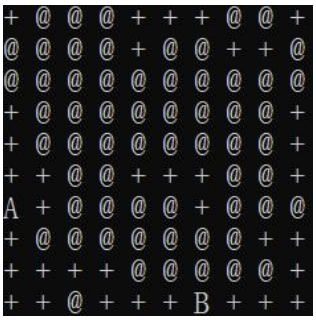
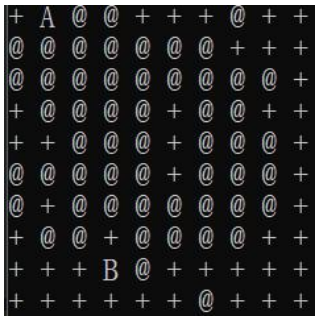
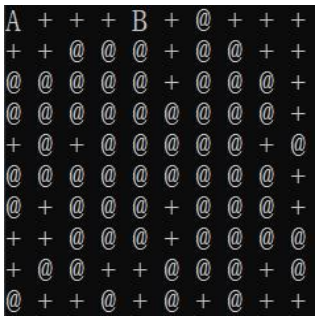
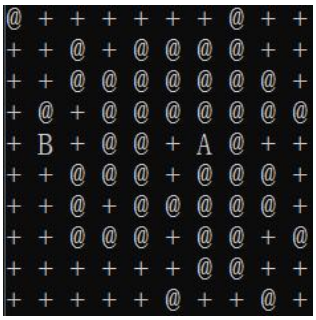
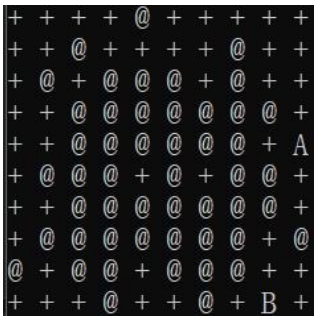
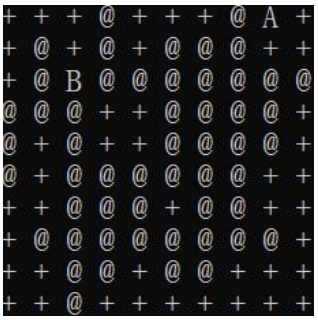
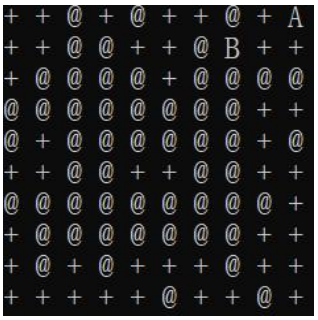
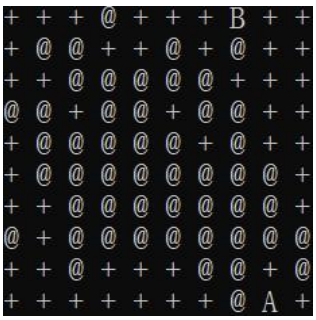
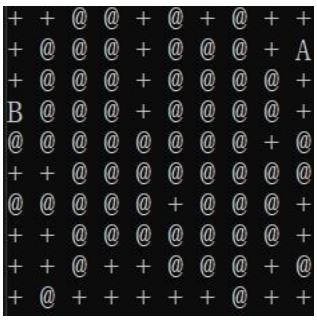


2.7. 10×10 棋盘，深度 4



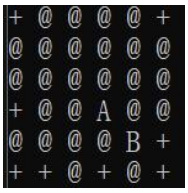
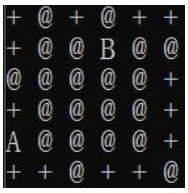
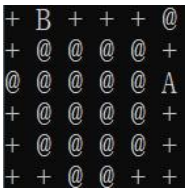
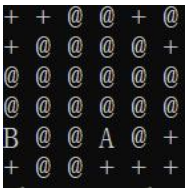



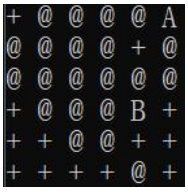
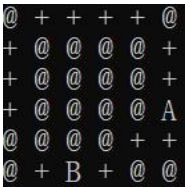
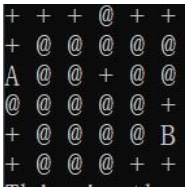
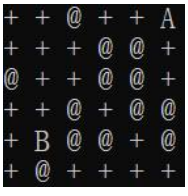
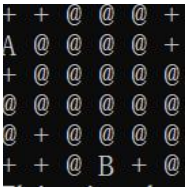
2.8. 10×10 棋盘，深度 8

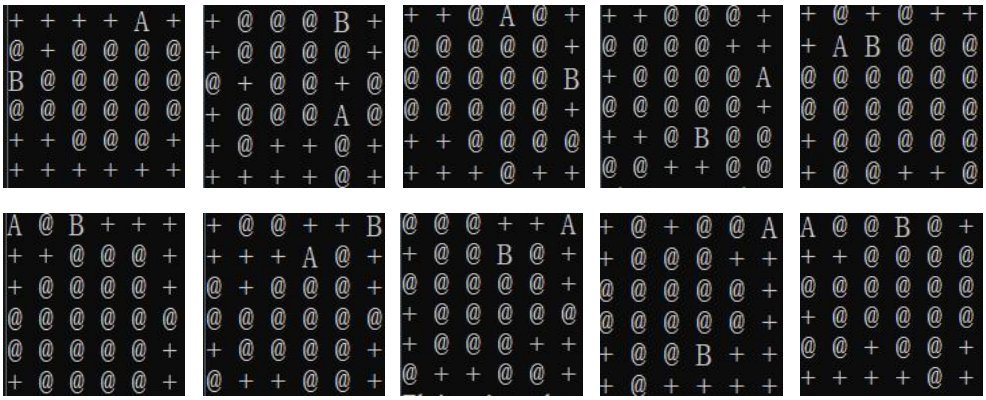
3. 本程序 vs 本程序

3.1. 6×6 棋盘，深度 4

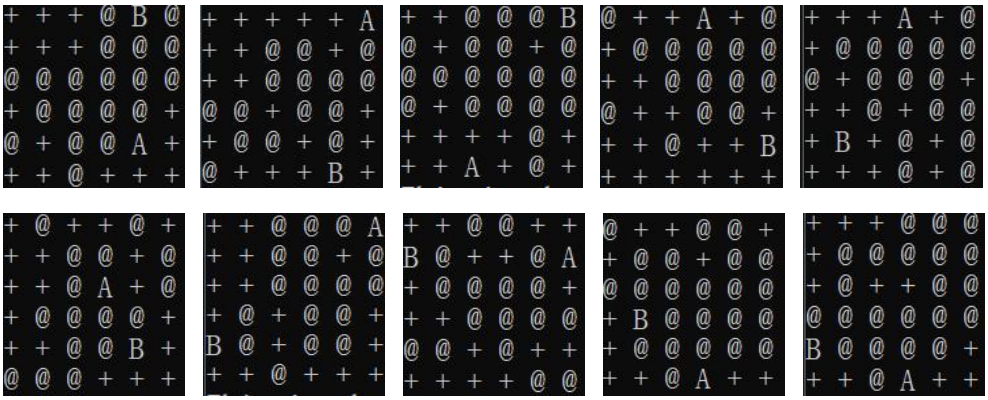





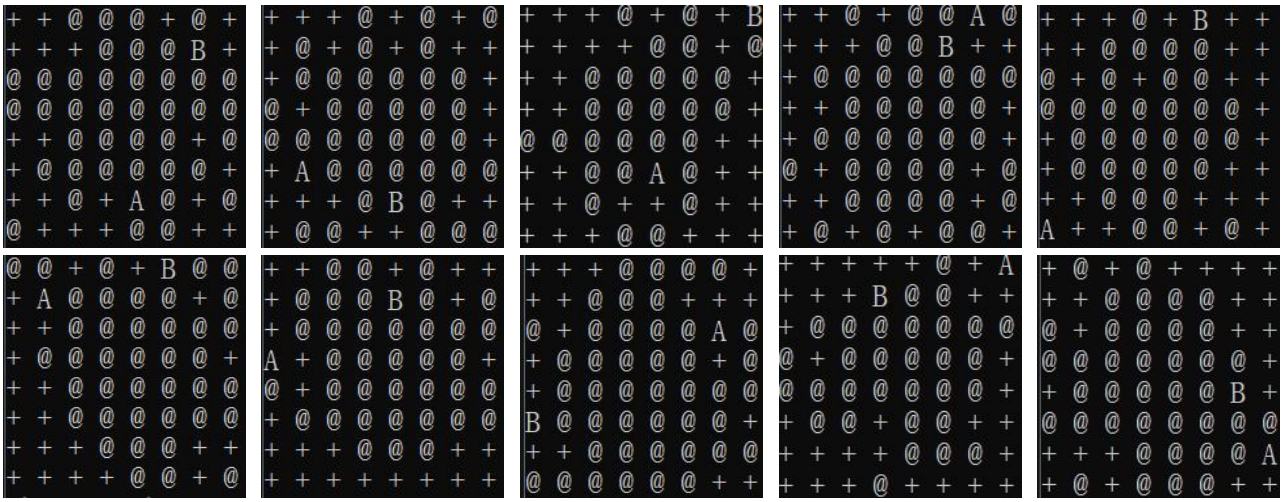
3.2. 6×6 棋盘，深度 8



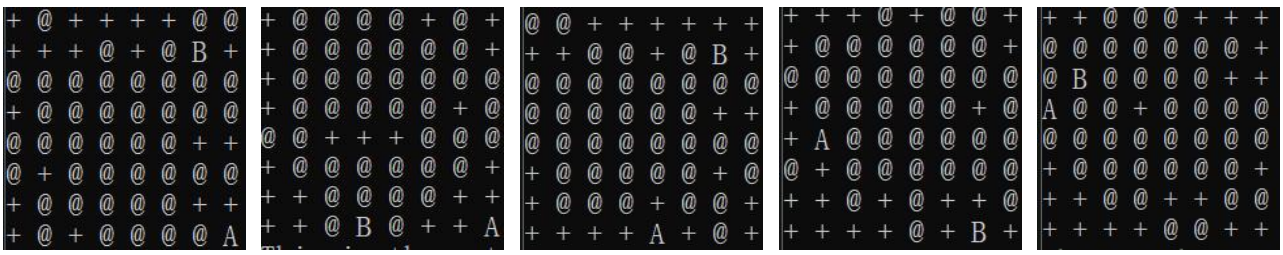
3.3. 6×6 棋盘，深度 12

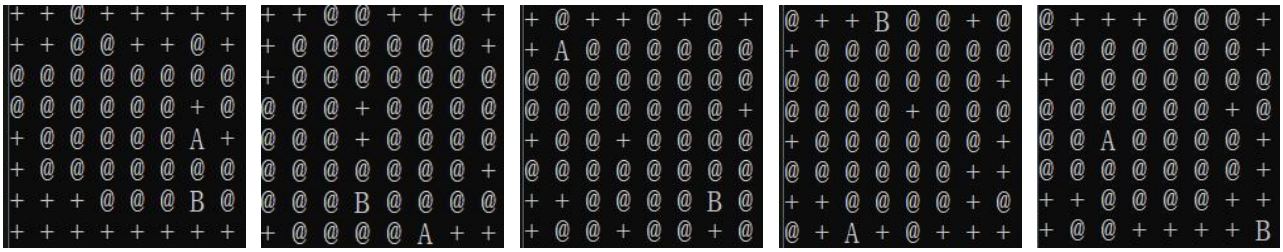


3.4. 8×8 棋盘，深度 4

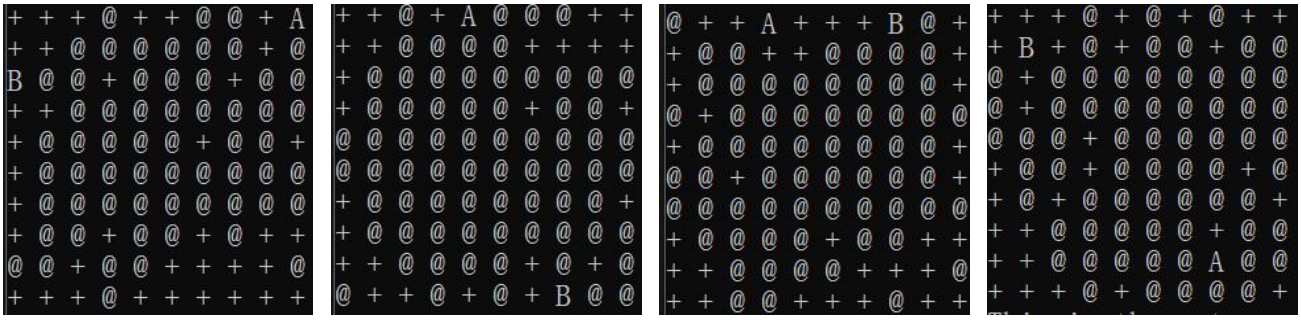


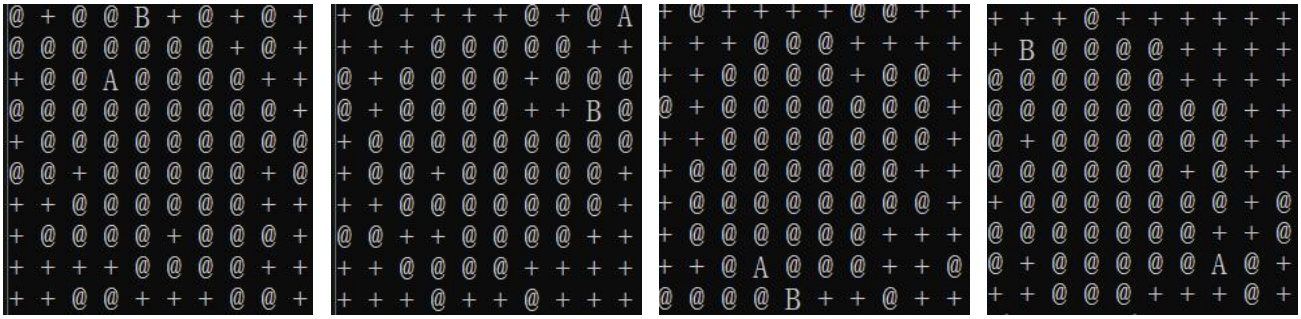
3.5. 8×8 棋盘，深度 8

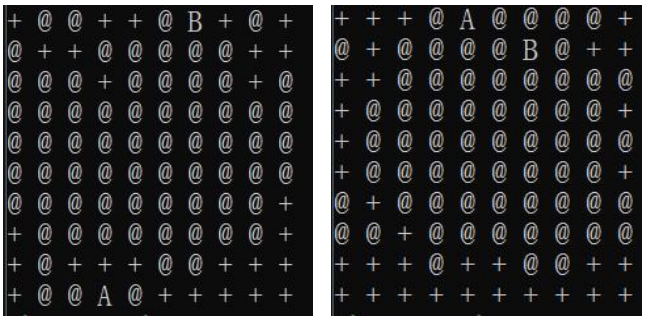




3.6. 10×10 棋盘，深度 4







3.7. 10×10 棋盘，深度 8