RLChina Reinforcement Learning Summer School

RLChina 2022

博弈搜索算法

林 舒

中国科学院自动化研究所

2022 年 8 月 16 日

目录

目录

序列决策

序列决策（Sequential Decision Making）

通过一系列的决策来达到某种目标，且每一步的决策都会对后续决策产生影响

现实世界中许多问题都是序列决策问题：

- 路线规划
- 游戏博弈
- 机械控制
- 语言理解
- ……

序列决策问题建模

序列决策问题一般可用马尔可夫决策模型进行描述

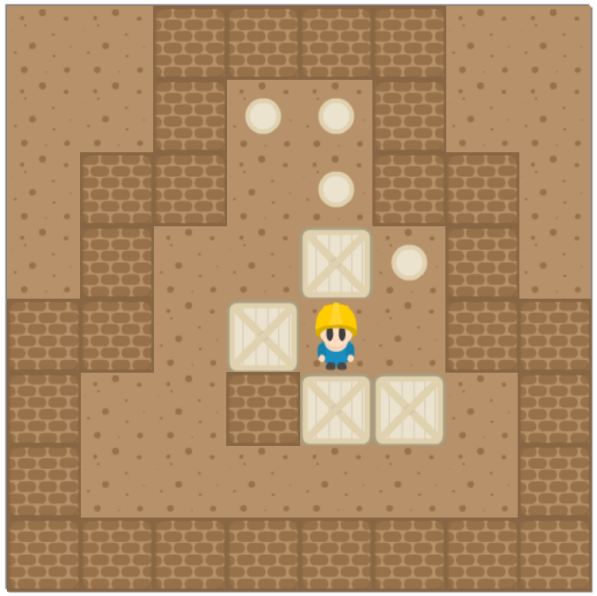
马尔可夫决策模型（Markov Decision Process, MDP）

MDP 模型用四元组 $\langle S, A(s), P(s,a), R(s,a)\rangle$ 表示，其中：

- **状态集** S：问题中所有可能出现的状态的集合。其中，$s_0 \in S$ 是初始状态，$T \subseteq S$ 是终结状态集合
- **动作集** $A(s)$：状态 $s \in S$ 下所有可能的动作集合
- **转移函数** $P(s,a) \in S$：状态 $s \in S$ 下应用动作 $a \in A(s)$ 后得到的新状态
- **奖励函数** $R(s,a)$：状态 $s \in S$ 下应用动作 $a \in A(s)$ 后得到的相应奖励

MDP 模型将序列决策问题统一为最优路径问题：
将状态视为结点，转移函数视为边，要求从起点 s_0 出发到达任意终点 $s_t \in T$，使得路径上奖励之和最大

序列决策问题示例：推箱子游戏[1]



推箱子游戏的 MDP 模型

- $S = \{$ 所有可能出现的局面集合 $\}$。s_0 如上图所示，唯一终结状态 s_t 为所有箱子都在目标位置上
- $A(s) = \{$ 上，下，左，右 $\}$
- $P(s, a)$：根据规则移动玩家和箱子位置后的新状态
- $R(s, a) \equiv -1$

[1]及第链接：`http://www.jidiai.cn/env_detail?envid=7`

通用求解算法：搜索

搜索是求解最优路径问题的经典算法：

- 搜索被称为“通用解题法”，是最重要的算法之一
- 搜索框架简单，但优化技术多样且灵活，不容易掌握
- 搜索的效率通常较低，复杂度上限不容易估计
- 搜索是人工智能领域的基础算法之一
  - 早期人工智能研究的就是如何提高搜索效率
  - 现在许多人工智能算法都与搜索有关

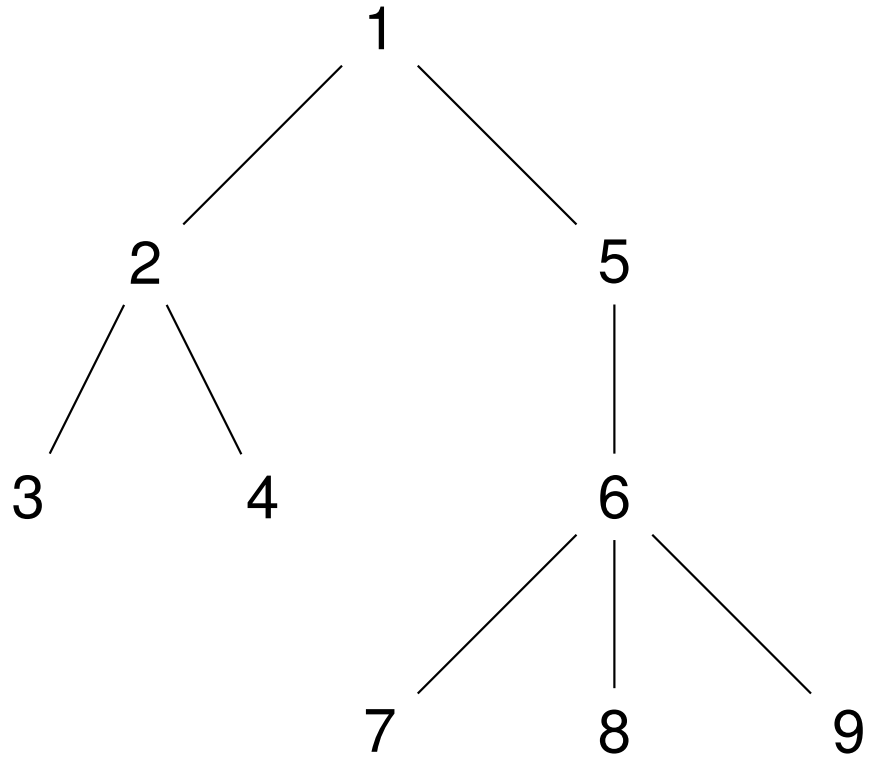
搜索分类

- 暴力搜索（枚举算法、随机游走）
- 盲目搜索（深度优先搜索、广度优先搜索）
- 启发式搜索（A*，IDA*）
- 对抗搜索（$\alpha - \beta$ 剪枝，蒙特卡洛树搜索）

目录

深度优先搜索算法思想

- 深度优先搜索算法通常采用递归或栈实现
- 算法思想：
  - 若当前状态是终结状态 $s \in T$：计算总奖励，更新解
  - 枚举每个可行动作 $a \in A(s)$：递归处理新状态 $s' = P(s, a)$
  - 返回上一步状态
- 典型示例：走迷宫



深度优先搜索基础：回溯法

回溯法框架

参数: 当前状态 s，已执行决策序列 D，当前奖励和 sum_r

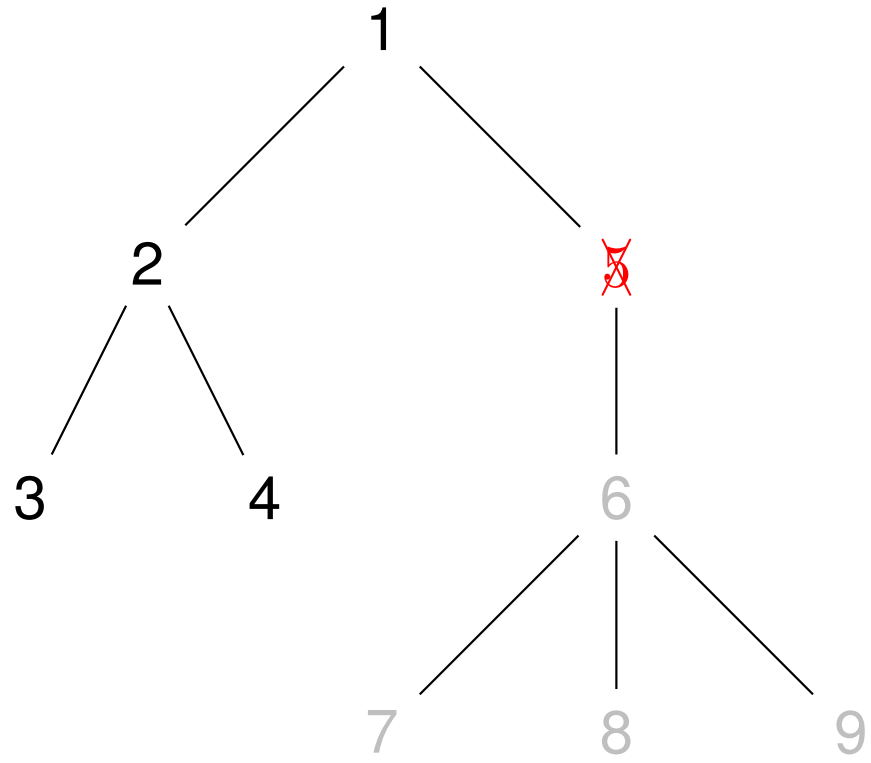
Function BACKTRACK (s, D, sum_r) **:**

```
if s ∈ T then
    if sum_r > sum_r^* then
        D^* ← D
        sum_r^* ← sum_r
foreach a ∈ A(s) do
    BACKTRACK(P(s, a), D.append(a), sum_r + R(s, a))
```

- 初始化：最优决策 $D^* \leftarrow \emptyset$，最大奖励和 $sum_r^* \leftarrow -\infty$
- 搜索入口：BACKTRACK$(s_0, [\,], 0)$
- 搜索结束后的 D^* 即为最优决策序列

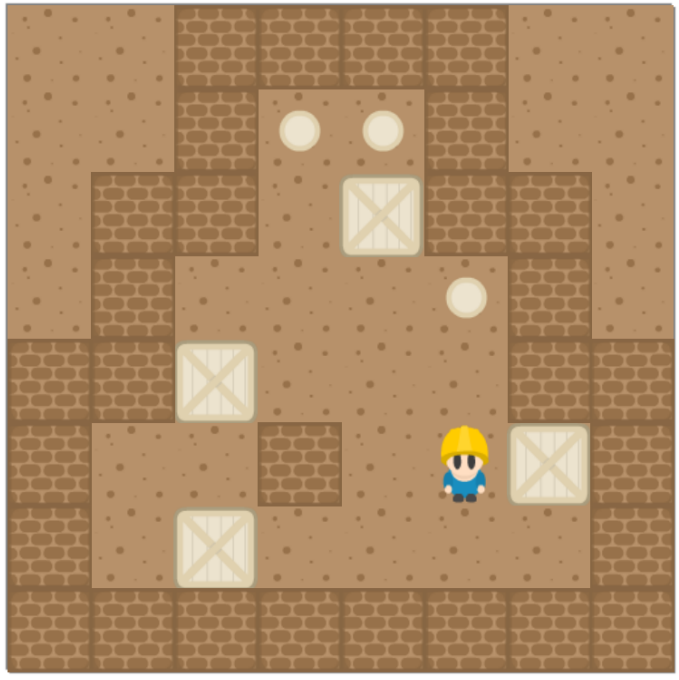
搜索的核心优化技术：剪枝

- 深度优先搜索是在回溯法的基础上加入剪枝，以减少搜索量
- 剪枝分类：
  - 可行性剪枝：若当前状态无法到达任一终结状态，直接返回
    设计思路：考虑终结状态的约束条件
  - 最优性剪枝：若未来奖励和无法大于当前最优解，直接返回
    设计思路：考虑奖励函数的性质



剪枝示例：推箱子游戏

- 可行性剪枝条件：有箱子无法到达目标位置（如下图所示）



- 最优性剪枝条件：
  已进行步数 + 估计未来步数下限 $\geq$ 当前最优步数

深度优先搜索

深度优先搜索框架

参数: 当前状态 s，已执行决策序列 D，当前奖励和 sum_r

Function `DFS`(s, D, sum_r)**:**

- **if** s 满足可行性剪枝或最优性剪枝条件 **then**
  - **return**
- **if** $s \in T$ **then**
  - $D^* \leftarrow D$
  - $sum_r^* \leftarrow sum_r$
- **foreach** $a \in A(s)$ **do**
  - `DFS`$(P(s, a), D.append(a), sum_r + R(s, a))$

深度优先搜索其他基本优化

去除重复状态

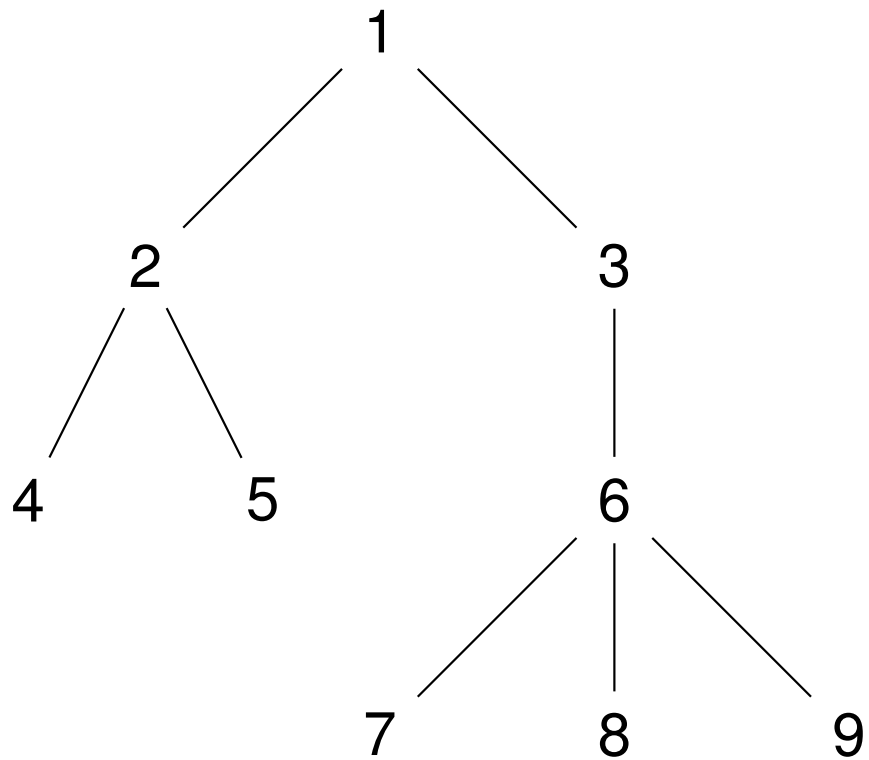
- 记录所有已访问状态集合，遇到新状态时，若集合中已存在相同或等价状态，则直接返回
- 优点：减少重复计算，避免死循环
- 缺点：需要额外空间存储集合，以及额外时间判断等价性

迭代加深

- 限制递归深度上限，并逐步放宽限制
- 优点：提升最优性剪枝效果，避免陷入很深的死胡同
- 缺点：需要额外时间重复计算浅层状态，但一般可忽略

广度优先搜索算法思想

- 广度优先搜索算法通常采用队列实现
- 算法思想：
  - 一开始队列中只有初始状态相关信息 $Q = [s_0]$
  - 若队列不为空：
    - 从队列中取出一个状态 $s = Q.pop()$
    - 若当前状态是终结状态 $s \in T$：计算总奖励，更新解
    - 枚举每个可行动作 $a \in A(s)$：将新状态加入队列 $Q.push(P(s, a))$
- 典型示例：洪水泛滥



广度优先搜索基础：泛洪填充法

泛洪填充法框架

Function FLOODFILL():

$D^* \leftarrow \emptyset$

$sum_r^* \leftarrow -\infty$

$Q \leftarrow [\langle s_0, [\,], 0 \rangle]$

while $Q \neq \emptyset$ **do**

$\langle s, D, sum_r \rangle \leftarrow Q.pop()$

if $s \in T$ **then**

if $sum_r > sum_r^*$ **then**

$D^* \leftarrow D$

$sum_r^* \leftarrow sum_r$

foreach $a \in A(s)$ **do**

$Q.push(\langle P(s,a), D.append(a), sum_r + R(s,a) \rangle)$

广度优先搜索：泛洪填充法 + 去重 & 剪枝

广度优先搜索框架

Function `BFS()`**:**

$D^* \leftarrow \emptyset$, $sum_r^* \leftarrow -\infty$, $Q \leftarrow [\langle s_0, [\,], 0 \rangle]$

$visited \leftarrow \{s_0\}$

while $Q \neq \emptyset$ **do**

$\langle s, D, sum_r \rangle \leftarrow Q.pop()$

if $s \in T$ **then**

$D^* \leftarrow D$, $sum_r^* \leftarrow sum_r$

foreach $a \in A(s)$ **do**

$s' \leftarrow P(s, a)$

if *not* s' 重复出现或满足剪枝条件 **then**

$visited \leftarrow visited \cup \{s'\}$

$Q.push(\langle s', D.append(a), sum_r + R(s, a) \rangle)$

return D^*

广度优先搜索其他基本优化

双向搜索

- 同时从起始状态和终结状态出发搜索，直至相遇
- 优点：减少展开状态数，提高搜索效率
- 缺点：要求终结状态数较少

调整搜索顺序

- 优先从队列中选取更好的状态进行扩展
- 优点：更快找到较优或最优解
- 缺点：需要精心分析问题性质

小结：深度优先搜索 VS 广度优先搜索

- 深度优先搜索基于栈，广度优先搜索基于队列
- 广度优先搜索不需要递归实现，额外时间开销小
- 深度优先搜索不需要存储经过的状态，空间复杂度低
- 根据问题特性选择算法：
  - 状态数量多：深度优先搜索
  - 重复状态：深度优先搜索 + 状态去重，广度优先搜索
  - 深度不定：深度优先搜索 + 迭代加深，广度优先搜索
  - 重复状态 + 深度不定：广度优先搜索

代码实践：推箱子游戏的广度优先搜索算法

- https://github.com/jidiai/SummerCourse2022/course2/examples/bfs-sokuban/submission.py
- https://gitee.com/rlchina/summercourse2022/course2/examples/bfs-sokuban/submission.py

目录

从盲目搜索到启发式搜索

- 深度优先搜索和广度优先搜索都是盲目搜索
- 盲目搜索主要靠剪枝来减少搜索量，从而提高效率
- 盲目搜索一般用预定顺序访问各状态，不考虑问题特性
- 启发式搜索加入知识改变搜索顺序，指导更快找到最优解

A* 算法基础：最优优先搜索

最优优先搜索框架

Function $\mathrm{A}^*()$ **:**

- $Q \leftarrow [\langle s_0, [\,], 0\rangle]$
- $visited \leftarrow \{s_0\}$
- **while** $Q \neq \emptyset$ **do**
  - $\langle s, D, sum_r\rangle \leftarrow Q.pop(\arg\max f(s \mid s \in Q))$
  - **if** $s \in T$ **then**
    - **return** D
  - **foreach** $a \in A(s)$ **do**
    - $s' \leftarrow P(s, a)$
    - **if** *not* s' 重复出现或满足剪枝条件 **then**
      - $visited \leftarrow visited \cup \{s'\}$
      - $Q.push(\langle s', D.append(a), sum_r + R(s, a)\rangle)$

A* 算法

- 在广度优先搜索的基础上引入估价函数改变搜索顺序
- 估价函数 $f(s)$ 由两个部分组成 $f(s) = g(s) + h(s)$，其中：
  - $g(s)$ 是起点到 s 的估计奖励，一般定义为实际奖励 sum_r
  - $h(s)$ 是 s 到任意终点的估计奖励
- 对 $h(s)$ 函数的要求：
  - $h(s)$ 与真实值 $h^*(s)$ 越接近效果越好
  - $h(s)$ 的计算不能太复杂，否则影响搜索效率
  - 可采纳性：不低估未来奖励，确保找到的第一个解是最优解

$$h(s) \geq h^*(s)$$

  - 一致性：$f(s)$ 随转移单调不增，保证状态不会被重复处理

$$h(s) \geq R(s,a) + h(P(s,a))$$

估价函数设计示例

大部分问题中，仅需考虑对 $h(s)$ 的设计：

- 最短路径（平面）问题，用当前点 s 与终点 t 距离的相反数作为未来奖励

  欧拉距离：$h(s) = -\sqrt{(s.x - t.x)^2 + (s.y - t.y)^2}$

- 迷宫（网格）问题，用当前点 s 与终点 t 距离的相反数作为未来奖励

  曼哈顿距离：$h(s) = -\left(|s.x - t.x| + |s.y - t.y|\right)$

- 推箱子，
  - 用箱子的最小移动距离的相反数作为未来奖励
  - 用不考虑障碍时人的最小移动距离的相反数作为未来奖励

A* 算法的局限性

- A* 算法基于广度优先搜索
- A* 算法空间复杂度较高，状态数多甚至无限时无法存储
- 如何避免存储状态?

- 回忆在盲目搜索中，深度优先搜索是不需要存储状态的
- 能否将估价函数与深度优先搜索结合?

IDA* 算法

- IDA* 算法：使用深度优先搜索 + 迭代加深来实现 A* 算法
- 实现方式：
  - 限定估计奖励 $f(s)$ 的下限（深度）为 f_{min}
  - 在搜索过程中，若 $f(s) < f_{min}$ 则进行最优性剪枝
  - 若完成一轮搜索后没有找到解，适当减少 f_{min} 再次搜索

IDA* 算法

IDA* 算法框架

参数: 当前状态 s，已执行决策序列 D，当前奖励和 sum_r

Function $\texttt{IDA}^*(s, D, sum_r)$**:**

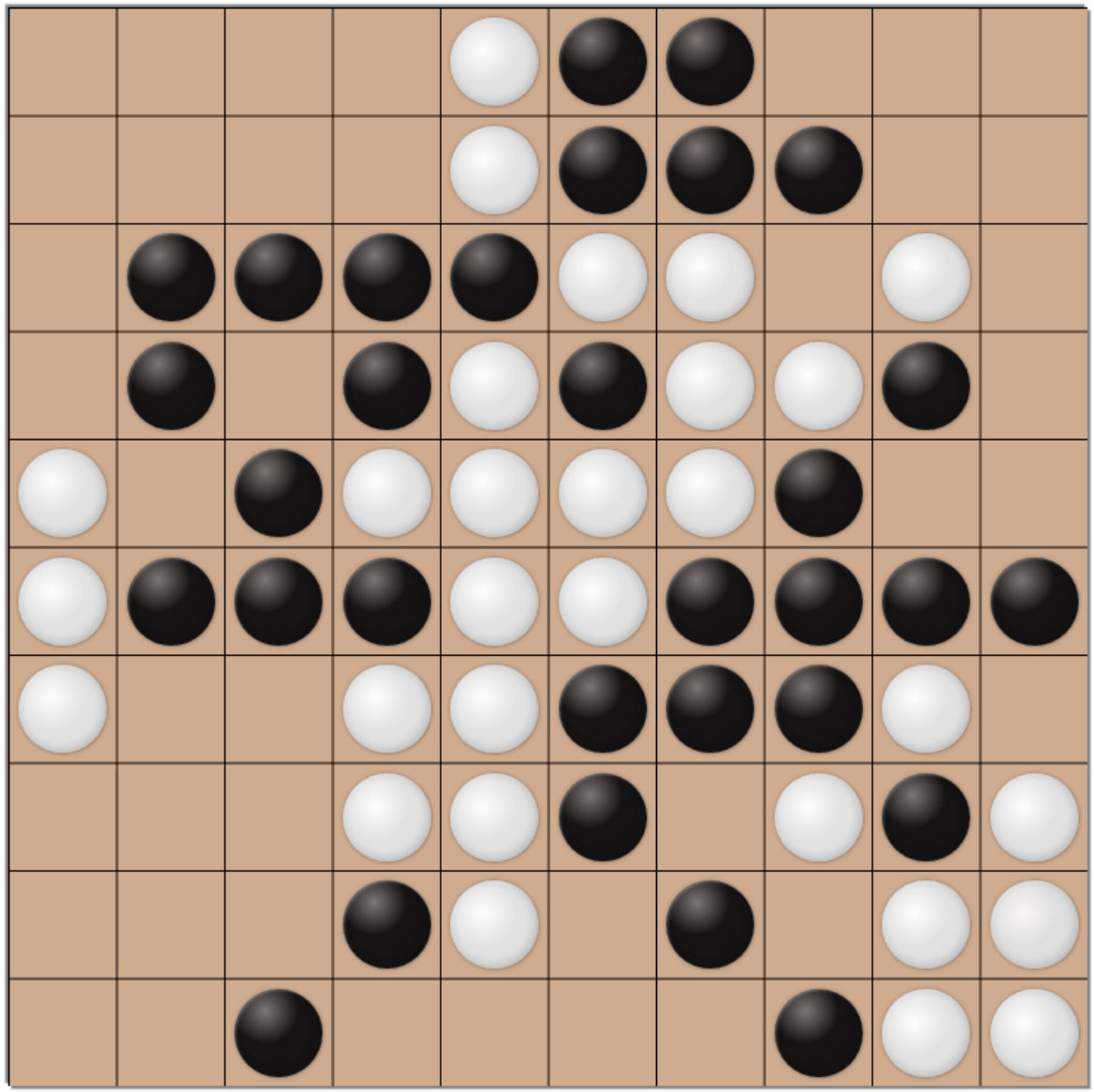
- **if** $f(s) < f_{min}$ 或 s 满足其他剪枝条件 **then**
  - **return**
- **if** $s \in T$ **then**
  - $D^* \leftarrow D$
  - $sum_r^* \leftarrow sum_r$
- **foreach** $a \in A(s)$ **do**
  - $\texttt{IDA}^*(P(s,a), D.append(a), sum_r + R(s,a))$

目录

对抗博弈环境

- 上面介绍的盲目搜索和启发式搜索都是针对单决策者环境
- 当环境中出现两个或更多决策者时，需要使用对抗搜索
- 本讲只考虑最简单、却是最重要的一类博弈：双人零和博弈
  - 两个决策者
  - 双方最终的总收益值为 0
  - 完全信息，即双方均充分知晓游戏规则及当前状态
  - 状态转移具有确定性
  - 双方轮流决策
  - 一定在有限步内到达终结状态
  - 双方均采用最优策略

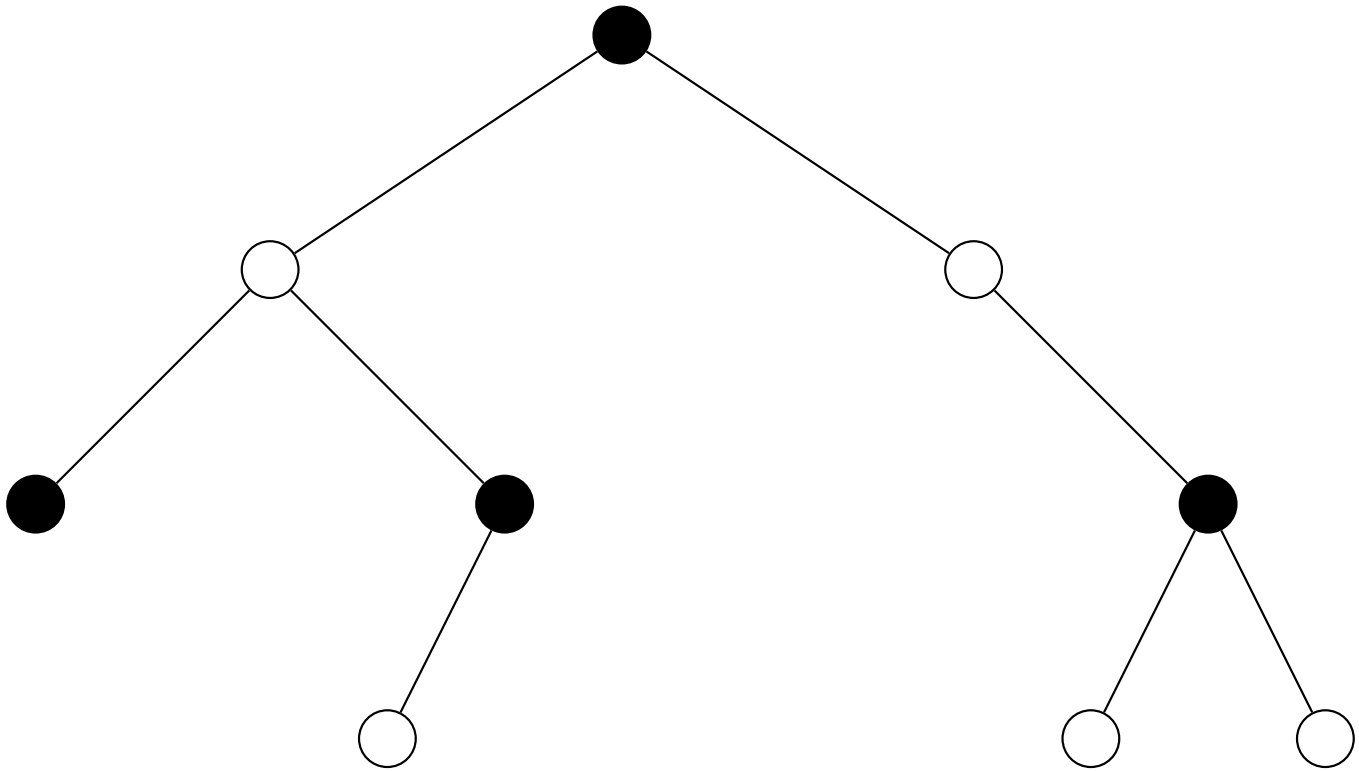
双人零和博弈示例：翻转棋[2]



[2]及第链接http://www.jidiai.cn/env_detail?envid=2

双人零和博弈基本思路

- 前提：双方均采用最优策略
- 搜索中状态分为黑点（我方决策点）和白点（敌方决策点）
- 在任意一条路径中黑点和白点交替出现
- 黑点目标使我方奖励最大，白点目标使我方奖励最小
- 我方称为 MAX 方，敌方称为 MIN 方



极大极小搜索

极大极小搜索框架

参数: 当前状态 s，决策者 p

Function MiniMax(s, p) **:**

- **if** $s \in T$ **then**
  - **return** $reward(s, p)$
- **foreach** $a \in A(s)$ **do**
  - $s' \leftarrow P(s, a)$
  - $f(s') \leftarrow -\text{MiniMax}(s', opponent(p))$
- $a^*(s) \leftarrow \arg\max_{a \in A(s)} \{f(P(s, a)\}$
- **return** $f(p(s, a^*(s)))$

α-β 剪枝

- 极大极小搜索提供了双人零和博弈的基本框架，但效率较低
- 可以加入最优性剪枝——α-β 剪枝
  - α 为已找到的我方奖励下界，β 为已找到的我方奖励上界
  - 对于敌方来说，已找到的奖励下界和上界分别为 $-\beta$ 和 $-\alpha$
  - 剪枝条件：$\exists a \in A(s), f(P(s,a)) \geq \beta$

α-β 剪枝

加入 α-β 剪枝的搜索框架

参数: 当前状态 s，决策者 p，奖励下界 α，奖励上界 β

Function AlphaBeta(s, p, α, β) **:**

- **if** $s \in T$ **then**
  - **return** $reward(s, p)$
- **foreach** $a \in A(s)$ **do**
  - $s' \leftarrow P(s, a)$
  - $f(s') \leftarrow -\text{AlphaBeta}(s', opponent(p), -\beta, -\alpha)$
  - **if** $f(s') \geq \beta$ **then**
    - **return** $f(s')$
- $a^*(s) \leftarrow \arg\max_{a \in A(s)} \{f(P(s, a)\}$
- $\alpha \leftarrow \max\{\alpha, f(p(s, a^*(s)))\}$
- **return** $f(p(s, a^*(s)))$

翻转棋 AI 实现思路

- 双人零和博弈，直接套用极大极小算法 +α-β 剪枝框架
- 规模较大，需要限制深度，设计状态价值估计函数
  - 方案 1：状态价值 = 我方棋子数 − 敌方棋子数
  - 方案 2：状态价值 = 我方棋子加权和 − 敌方棋子加权和
    以 8×8 为例：

20	-3	11	8	8	11	-3	20
-3	-7	-4	1	1	-4	-7	-3
11	-4	2	2	2	2	-4	11
8	1	2	-3	-3	2	1	8
8	1	2	-3	-3	2	1	8
11	-4	2	2	2	2	-4	11
-3	-7	-4	1	1	-4	-7	-3
20	-3	11	8	8	11	-3	20

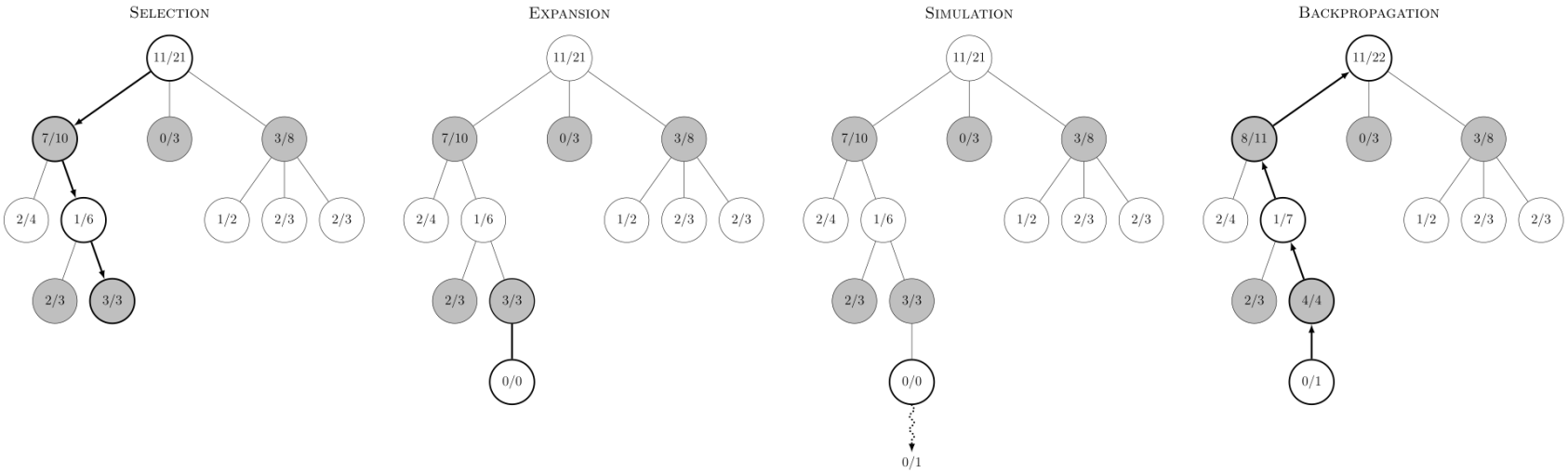
代码实践：翻转棋游戏的 α-β 剪枝实现

- https://github.com/jidiai/SummerCourse2022/course2/examples/alphabeta-reversi/submission.py
- https://gitee.com/rlchina/summercourse2022/course2/examples/alphabeta-reversi/submission.py

α-β 剪枝的不足

- 以找到最优解为目标，而非尽快找到一个较优解
- 不评估各分支权重，无法将算力集中在重要分支上
- 过分依赖人类提供的评估函数

蒙特卡洛树搜索 MCTS



蒙特卡洛树搜索重复进行以下四个步骤[3]：

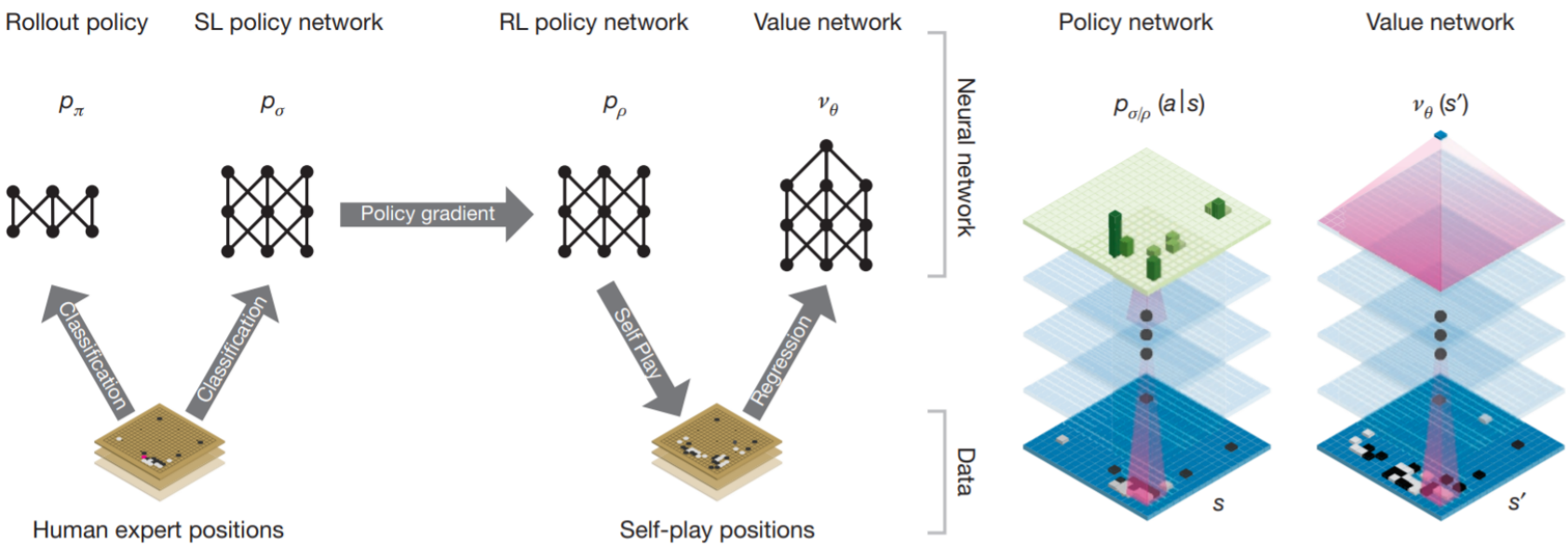
1. 选择：根据节点数据选择动作（ϵ-greedy 或 UCB 等）
2. 扩展：到达未访问节点，创建新节点并加入树中
3. 模拟：从新节点出发随机行动到达终结节点，重复多次
4. 回溯：用模拟结果更新沿途各节点数据

[3] 图片作者：Rmoss92

代码实践：翻转棋游戏的蒙特卡洛树搜索实现

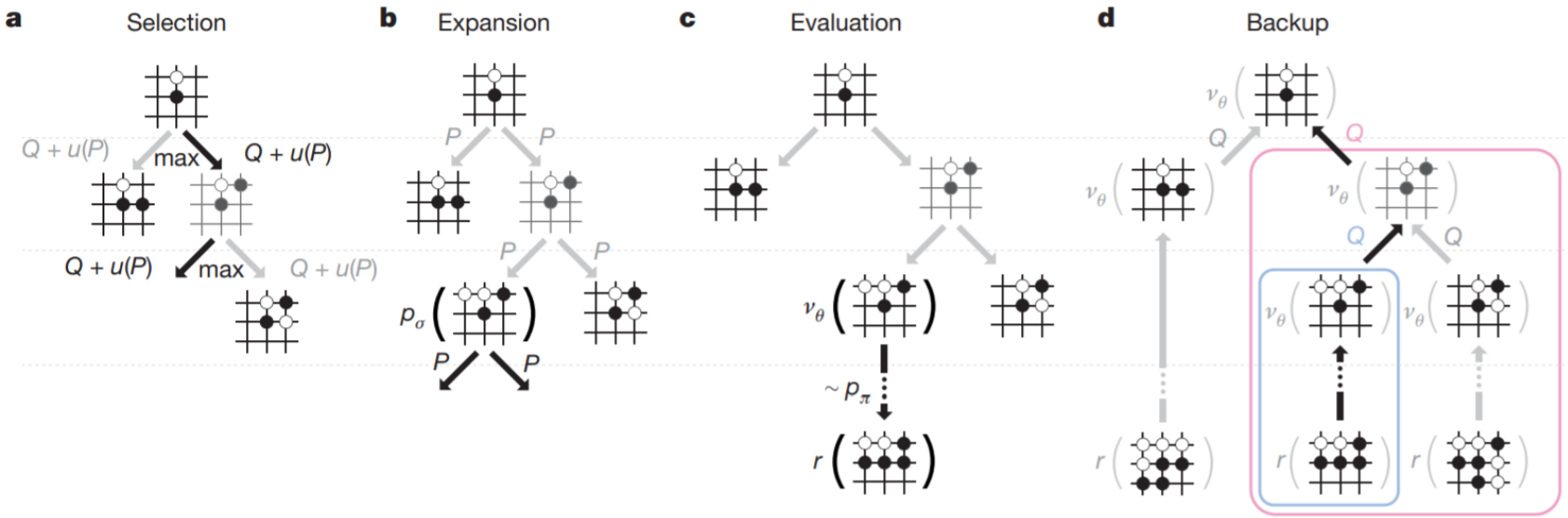
- https://github.com/jidiai/SummerCourse2022/course2/examples/mcts-reversi/submission.py
- https://gitee.com/rlchina/summercourse2022/course2/examples/mcts-reversi/submission.py

AlphaGo 介绍[4]



[4]Mastering the game of Go with Deep Neural Networks & Tree Search

AlphaGo 中的蒙特卡洛树搜索[5]



1. 选择：根据 $Q + u(P)$ 选择节点
2. 扩展：创建新节点，用 P_σ 初始化概率
3. 模拟：结合估值网络 v_θ 和快速走子策略 P_π 进行采样
4. 回溯：更新 Q

[5] Mastering the game of Go with Deep Neural Networks & Tree Search

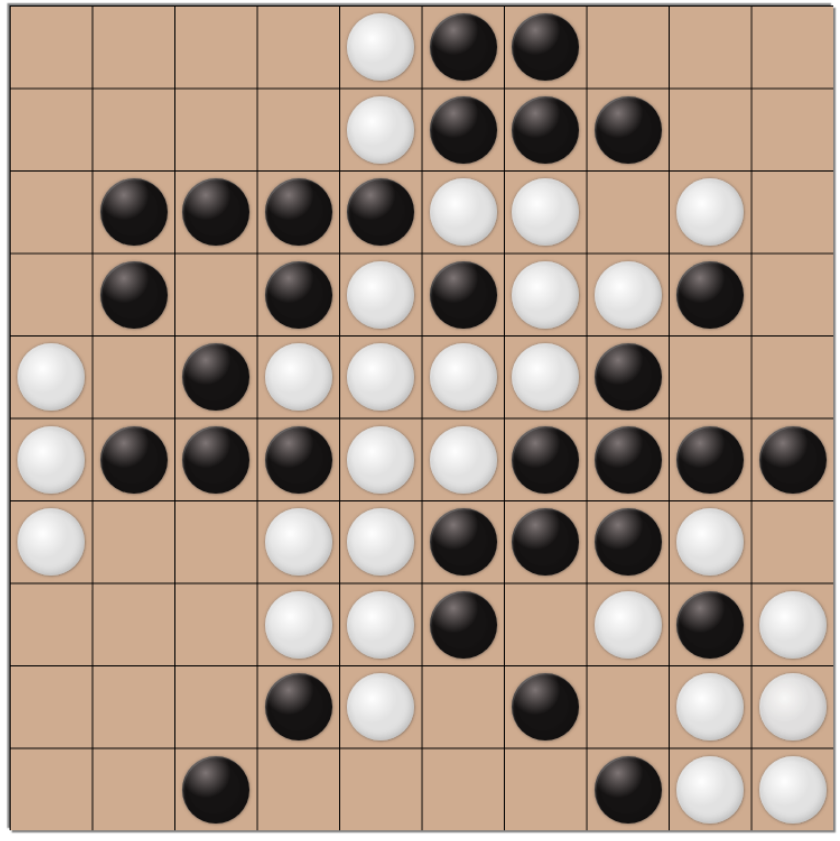
目录

总结

博弈搜索算法

- 盲目搜索
  - 深度优先搜索：基于递归或栈
  - 广度优先搜索：基于队列
- 启发式搜索
  - A* 算法：广度优先搜索 + 估价函数
  - IDA* 算法：深度优先搜索 + 估价函数 + 迭代加深
- 对抗搜索
  - α-β 剪枝：极大极小搜索 + 最优性剪枝
  - 蒙特卡洛树搜索：选择 $\rightarrow$ 扩展 $\rightarrow$ 模拟 $\rightarrow$ 回溯

作业：翻转棋



- 作业说明：使用 α-β 剪枝或蒙特卡洛树搜索实现翻转棋 AI
- 及第入口：`http://www.jidiai.cn/env_detail?envid=2`
- 作业要求：在及第上提交，排名高于随机（Jidi_random）