part 1

Question 1.1

> 这段话通过对 AlphaGo 的比喻，引出了计算机博弈程序设计的两种不同策略。AlphaGo作为一个使用启发式方法的深度强化学习程序，它的强大并非源于穷尽所有下法，而是通过大量的训练和不断调整策略来应对各种复杂局面。因此，它更多是通过模拟和优化自己的策略来获取优势。而在某些意义上，它并没有像一个“神”那样能够穷举所有的可能性，而更像一个“魔鬼”，因为它的力量来源于计算资源和学习能力，而不是无限的计算时间。
> 从这个角度来看，计算机博弈程序的设计有两种思路：一种是穷举法，力图遍历所有可能的局面，从而得到一个“完美”的决策树（类似于谕示机）。这种方法计算复杂度高，适用于棋盘状态空间有限的游戏。另一种是通过启发式方法和深度学习等技术，不断优化程序的策略，在博弈过程中不断从对手的反应中学习并提升自己，这种方法的优势在于它可以应对复杂的游戏和庞大的状态空间，但不能保证“完美”。
> 在设计计算机博弈程序时，应该根据问题的规模和复杂性选择适合的策略。对于较小的游戏，穷举法可能可行，而对于像 Hex 棋这样具有极大状态空间的复杂博弈，启发式搜索和强化学习可能是更合适的选择。这段话反映了计算机博弈在面对不完备信息和复杂策略时的一种常见思维模式，即不追求完美，而是通过不断的学习和优化，在合理的计算资源限制下做到“足够好”。

Question 1.2

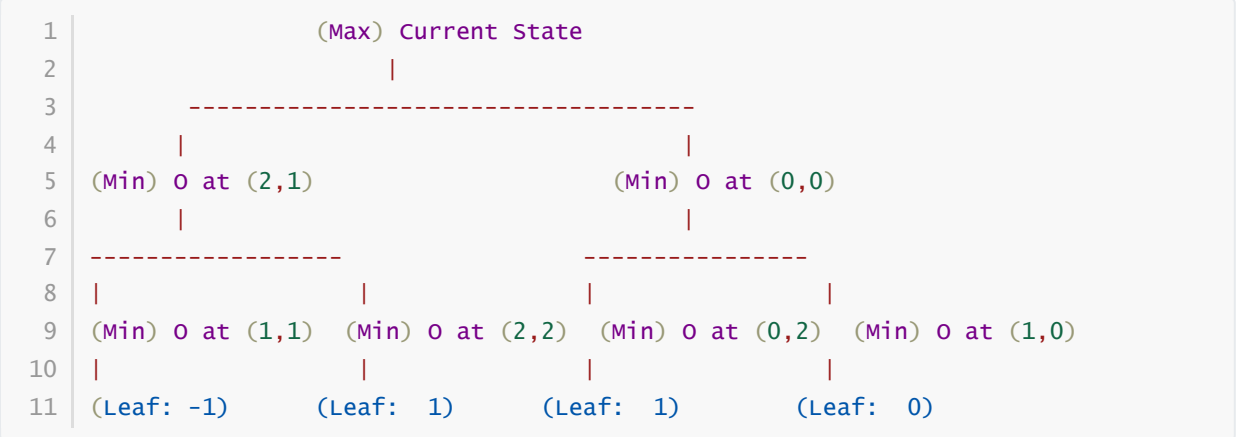
1.2.1 引言：minmax算法

Minimax 算法的基本原理

1. **游戏树**： 游戏的每个状态都可以看作树上的一个节点。树的根节点代表当前游戏状态，子节点代表所有可能的下一步状态。树的每一层交替着两个玩家的回合，一个玩家尽量做出最好的决策，另一个玩家则会尽力反制。
2. **树的深度**： 游戏树的深度通常是通过模拟游戏的步骤来确定的。当树达到最大深度时，表示游戏结束（例如，胜负已定或达到最大步数）。
3. **极大值和极小值**：
    - 在一个状态下，**Max 玩家**（例如，当前玩家）希望选择对自己最有利的动作，也就是选择使得自己得到最大收益的决策。
    - **Min 玩家**（对手）则希望选择对自己最有利的动作，也就是选择使得当前玩家损失最小的决策。
4. **递归评估**： Minimax 算法从叶子节点开始，评估每个终局状态的价值。对于 Max 玩家，叶子节点的价值就是最大的；对于 Min 玩家，叶子节点的价值是最小的。然后递归地将这些值返回到树的根节点，并通过这个过程评估整个游戏树。
5. **选择最优策略**：
    - **Max 玩家**选择能够使自己获得最大值的动作。
    - **Min 玩家**选择能够使自己避免最坏情况的动作，即选择最小的值。

算法步骤

1. **构建游戏树**：从当前游戏状态开始，生成所有可能的未来状态，直到树的叶子节点，表示游戏的终局。
2. **评估终局节点**：对于每个叶子节点，评估其值。这通常是一个简单的评分函数，返回正值、负值或零，表示游戏的胜利、失败或平局状态。
3. **递归计算每个非叶子节点的值**：
    - 如果当前节点是 **Max 玩家** 的回合，选择所有子节点中的最大值。
    - 如果当前节点是 **Min 玩家** 的回合，选择所有子节点中的最小值。
4. **返回根节点的最佳决策**：通过以上的递归评估，最终可以得到一个最优的决策，表示当前玩家应该采取的行动。

> 通俗理解
>
> > 我用井字棋游戏为例来绘制一棵简单的游戏树。
> >
> > 我们假设现在的局面是这样（轮到 Max 玩家行动）：
>
> ```
> X | O | X
> ---------
> O | X |
> ---------
>   | O | X
> ```
>
> > Max 玩家（你）可以在空格 `(2, 1)` 处下棋，得到以下新的棋盘状态：
>
> ```
> X | O | X
> ---------
> O | X |
> ---------
> O | O | X
> ```
>
> > 然后轮到 Min 玩家（对手）行动，他们会尽量让 Max 玩家失利。
>
> ## 游戏树示意图
>
> 我们从当前局面开始，构建游戏树，并逐步展开所有可能的状态。每一层的节点代表一个玩家的回合，Max 玩家和 Min 玩家交替进行。
>
> ### 游戏树
>
> 
>

解释：

1. **根节点**：我们从当前的游戏状态开始，Max 玩家要做决策。此时的局面是：

```
x | o | x
---------
o | x |
---------
  | o | x
```

2. **第一层**（Max 玩家）：

- Max 玩家会选择一个空格来下棋。假设他选择 `(2, 1)`（最后一行，第二列），得到一个新的局面：

```
x | o | x
---------
o | x |
---------
o | o | x
```

3. **第二层**（Min 玩家）：

- 轮到 Min 玩家（对手）下棋。Min 玩家会选择一个位置，目的是让 Max 玩家尽量失去优势。例如，他选择在 `(1, 1)` 位置下棋（中间位置），得到：

```
x | o | x
---------
o | x |
---------
o | o | x
```

另一个可能是他选择在 `(2, 2)` 位置下棋，得到：

```
x | o | x
---------
o | x |
---------
o | o | x
```

这个过程会一直展开，直到游戏结束，或者达到树的最深层。

评分（叶子节点）：

- 每个**叶子节点**表示游戏的结束状态。我们评估这些状态：
- 胜利得分为 +1。
- 失败得分为 -1。
- 平局得分为 0。

在我们的例子中，最终的得分如下：

- **(Leaf: -1)**：对手赢了。
- **(Leaf: 1)**：你赢了。

> - **(Leaf: 0)**：平局。
>
> ### 回溯（递归）
>
> 1. Min 玩家会选择最小的得分，也就是选择让你得分最少的路径。
> 2. Max 玩家会选择得分最大的路径，也就是选择让自己得分最大的路径。
>
> 最终通过回溯，Max 玩家就能做出最优决策。

1.2.2 Alpha-Beta 剪枝算法

Alpha-Beta 剪枝的目的是在使用 Minimax 算法时，**减少不必要的搜索**，提高决策速度，特别是当游戏树非常大时。

关键概念：

- **Alpha (α)**：Max 玩家（你）已知的最佳得分（下界）。也就是说，Max 玩家已经看到的最好结果。
- **Beta (β)**：Min 玩家（对手）已知的最佳得分（上界）。也就是说，Min 玩家已经看到的最差结果。

剪枝的原理：

在 Minimax 算法中，每一层都会有很多子节点。如果我们在某个节点下找到一个明确的值，而且这个值已经不可能改变当前最优决策时，就可以**剪枝掉**其他不必要的子节点，避免重复计算。

剪枝的意思是**"提前停止搜索"**，跳过不可能影响最终结果的部分。通过剪枝，我们能够节省大量的计算时间。

剪枝的过程：

1. **Max 玩家**（你）尝试选择**最大**的值。
2. **Min 玩家**（对手）尝试选择**最小**的值。
3. 在这个过程中，如果我们发现某个子树中的某些值已经不可能影响最终决策（因为已经比当前的 Alpha 或 Beta 还要差），我们就**剪掉**这部分子树，跳过不再搜索它。

Alpha-Beta 剪枝的效果：

通过使用 Alpha-Beta 剪枝，Minimax 算法能跳过大量重复计算，尤其是在深度较大的游戏树中，能显著减少需要搜索的节点数量。

剪枝过程的例子：

假设我们有一个游戏树，Max 玩家和 Min 玩家轮流选择。

```
            (Max)
              |
         -------------
        |             |
      (Min)         (Min)
        |             |
     ---------     ---------
    |         |   |         |
(Leaf: 2) (Leaf: 3) (Leaf: 1) (Leaf: 5)
```

没有剪枝时：

1. Max 玩家首先选择一个子树，然后 Min 玩家对每个子树进行评估。
2. 然后，Max 玩家从评估值中选择最大的值，Min 玩家选择最小的值。

假设我们首先计算左边的子树，Min 玩家看到 `Leaf: 2` 和 `Leaf: 3`，所以 Min 玩家最终选择 `Leaf: 2`，因为这是他能得到的最小值。

然后我们计算右边的子树，Min 玩家看到 `Leaf: 1` 和 `Leaf: 5`，选择 `Leaf: 1`，因为这是最小的。

所以最终，Max 玩家会选择值为 3 的分支。

有了 Alpha-Beta 剪枝之后：

假设我们从左边的子树开始，Min 玩家会首先看到 `Leaf: 2`，然后看到 `Leaf: 3`，所以 Min 玩家选择 `2`。

现在，Max 玩家已经知道了左边的最小值是 2。

然后我们开始看右边的子树，Min 玩家看到 `Leaf: 1` 时就已经知道，右边的子树没有比 1 更小的值（因为 `Leaf: 2` 比 `Leaf: 1` 大）。**所以，Min 玩家就不需要再检查 `Leaf: 5`**，直接选择 `1`。

这样就完成了剪枝，避免了对 `Leaf: 5` 的评估，从而节省了时间。

Alpha-Beta 剪枝的步骤

1. 从根节点开始，递归地搜索树。
2. 对每个节点，更新

   ```
   Alpha
   ```

   和

   ```
   Beta
   ```

   值：

   - 如果是 Max 节点，更新 `Alpha` 为当前节点的最大值。
   - 如果是 Min 节点，更新 `Beta` 为当前节点的最小值。
3. 在搜索过程中，如果发现某个子树的值已经不可能改变当前的最优结果（`Alpha >= Beta`），则剪枝掉这部分子树，不再继续搜索。

剪枝的效果：

1. 在最好的情况下，Alpha-Beta 剪枝能把 Minimax 的搜索空间减半，甚至更多。这样一来，算法可以在同样的时间内搜索更深的游戏树。
2. 在某些情况下，Alpha-Beta 剪枝对结果的影响不大，但通常能够加速搜索过程，特别是在复杂的游戏中。

总结

- **Minimax 算法**能够找到最优决策，但在大树中会很慢。
- **Alpha-Beta 剪枝**通过"剪掉"不必要的分支来减少计算量，提高效率。
- 剪枝并不会影响最终的结果，它只是避免了重复计算，让搜索过程更加高效。

希望这个通俗的解释能帮助你理解 Alpha-Beta 剪枝的原理！如果还有疑问，随时告诉我！

1.2.3 蒙特卡洛树搜索

一、什么是蒙特卡洛树搜索（MCTS）

蒙特卡洛树搜索（Monte Carlo Tree Search, MCTS）是一种启发式搜索算法，通常用于AI游戏中，特别是在**围棋**、**象棋**、**德州扑克**等游戏中。MCTS通过模拟游戏的多个可能结果来选择最佳的游戏动作。一个典型的例子就是2016年**AlphaGo**利用MCTS击败了韩国围棋9段棋手李世石。

二、蒙特卡洛树搜索与蒙特卡罗方法的区别

- **蒙特卡罗方法**（Monte Carlo method）是一类利用随机抽样来求解复杂问题的计算方法，广泛应用于物理、数学、计算机图形学等领域。它的基本思想是通过概率模拟来估计问题的解。
- **蒙特卡洛树搜索（MCTS）*是蒙特卡罗方法在*AI游戏**中的应用，专门用来在游戏中寻找最佳动作。

三、MCTS的工作原理

MCTS通过构建一个树来记录搜索过程，每次搜索过程中都会进行模拟，就像人类在下棋时会考虑对方的每一步棋。

MCTS包含4个主要步骤：

1. **选择（Selection）**：
   - 从根节点开始，选择子节点，直到选择到一个叶子节点。每次选择时，根据树中节点的统计信息决定。
   - **叶节点**表示潜在的游戏状态。
2. **扩展（Expansion）**：
   - 如果选择的节点没有子节点，就扩展它，生成新的子节点。
3. **模拟（Simulation）**：
   - 从扩展的节点开始，模拟一场完整的游戏，直到游戏结束。这个过程是完全**随机的**，目的是估计当前状态的胜负。
4. **反向传播（BackPropagation）**：

- 模拟结束后，将结果（赢或输）回传到树中的所有相关节点，更新这些节点的统计数据。

选择节点的方式——UCB1（上置信度边界1）

在选择节点时，MCTS使用了一个名为 **UCB1** 的公式：

$$UCB1 = \frac{w_i}{n_i} + C \cdot \sqrt{\frac{\ln N_i}{n_i}}$$

- w_i：当前节点的胜利次数。
- n_i：当前节点的模拟次数。
- N_i：父节点的模拟次数。
- C：常数，控制探索与利用的平衡。

UCB1 公式的意义：

- 第一部分 ($\frac{w_i}{n_i}$) 表示胜率，胜率越高的节点更容易被选中。
- 第二部分 ($\sqrt{\frac{\ln N_i}{n_i}}$) 是探索项，鼓励选择不常访问的节点，促进新的探索。

四、MCTS的实际执行过程

第一次模拟：

1. **选择**：从根节点选择一个没有子节点的节点进行扩展。
2. **扩展**：生成子节点。
3. **模拟**：选择一个子节点进行随机模拟，假设游戏以黑棋胜利结束。
4. **反向传播**：更新根节点和各个父节点的胜负统计信息。

第二次模拟：

1. 选择一个节点进行模拟，计算UCB1值，选择最大值。
2. 进行模拟，更新统计信息。
3. 继续反向传播，更新根节点和父节点的值。

第三次模拟：

重复上述步骤，直到模拟足够多次，能够提供足够的统计信息来决定最佳的游戏动作。

五、AlphaGo如何使用MCTS

AlphaGo利用MCTS与深度学习（神经网络）结合，进一步提升了计算效率和准确度。它使用了两个神经网络：

1. **策略网络（Policy Network）**：根据当前的棋盘状态，给出每个位置的评分，选择下一步最佳落点，代替传统的UCB1选择方式。
2. **值网络（Value Network）**：用来评估当前棋盘的胜负结果，避免完全依赖随机模拟。

六、MCTS的优缺点

优点：

- **探索广泛**：通过多次模拟，MCTS能够探索到很多可能的结果，从而找到较优的决策。
- **适用于大规模问题**：比如围棋这样每步棋选择数目非常大的游戏，MCTS能在有限时间内提供不错的决策。

缺点：

- **不保证找到最佳路径**：虽然MCTS能够找到较优的决策，但它并不能保证找到最优解。
- **计算量大**：MCTS的效果依赖于模拟的次数，模拟次数越多，计算量越大。

结语

蒙特卡洛树搜索（MCTS）在许多AI游戏中发挥了重要作用，特别是在围棋等复杂游戏中。通过大量的模拟，MCTS能够找到非常接近最优的策略，尽管它不能保证每次都能找到最优解，但在实际应用中已经展现出强大的能力。AlphaGo通过结合深度学习和MCTS，进一步提高了游戏策略的精确性，使得它能够战胜顶级人类选手。

> 参考链接：https://blog.csdn.net/fearlesslpp/article/details/134342648 我学习蒙特卡洛的网页
>
> 比较通俗易懂

1.2.4 蒙特卡诺树结合UCT

蒙特卡洛树搜索（MCTS）结合 **UCT（Upper Confidence bounds applied to Trees）** 算法，实际上是 **MCTS的核心算法**之一，用于在蒙特卡洛树搜索过程中平衡 **探索（exploration）** 和 **开发（exploitation）** 的需求。UCT 算法是通过上置信度界限（Upper Confidence Bound）来引导搜索过程，使得 MCTS 在决策过程中既能深入探索尚未充分评估的行动，又能更多地选择已经表现较好的行动。

1. UCT 算法概述

UCT 是对 MCTS 中 **选择（Selection）** 阶段的优化，通过一个特定的计算公式来决定从当前节点（父节点）选择哪个子节点进行模拟。这个公式的基本目标是平衡探索和开发：

- **开发（Exploitation）**：倾向于选择 **胜率较高的**路径，利用已有的信息来做出决策。
- **探索（Exploration）**：倾向于选择 **胜率未知的**路径，尝试从更多可能性中发现新的潜在好策略。

UCT的基本公式：

$$UCT = \frac{w_i}{n_i} + C \times \sqrt{\frac{\ln N_p}{n_i}}$$

其中：

- w_i：表示当前节点（子节点）赢的次数。
- n_i：表示当前节点（子节点）模拟的总次数。
- N_p：表示当前节点（父节点）模拟的总次数。
- C：是一个调节常数，用来控制探索与开发的权重。

2. UCT 算法在 MCTS 中的作用

在 MCTS 中，**选择（Selection）** 阶段是非常重要的，它决定了如何从当前节点的子节点中选取一个来进行进一步的模拟（Rollout）。UCT 算法的作用就是为每一个子节点计算一个 **UCT 值**，然后选择 **UCT 值最大的子节点**。这里的核心思想就是根据当前的胜率 $\frac{w_i}{n_i}$ 来评估每个节点的好坏，同时又考虑到选择的频率 $\frac{\ln N_p}{n_i}$，鼓励选择那些被探索较少的节点，从而引导算法进行合理的探索。

3. UCT 算法的工作流程

MCTS 和 UCT 的结合通常遵循以下四个步骤：

1. **选择（Selection）**：
   从根节点开始，选择一个子节点。选择的依据是 UCT 值，公式为：

   $$UCT = \frac{w_i}{n_i} + C \times \sqrt{\frac{\ln N_p}{n_i}}$$

   在每一步中，节点会根据 UCT 值进行选择，既关注每个节点的 **胜率**，又关注它被 **探索的次数**。
2. **扩展（Expansion）**：
   如果当前节点没有被完全扩展，则会扩展一个新的子节点，加入到搜索树中。
3. **模拟（Simulation）**：
   从扩展的节点开始，进行 **随机模拟**，直到游戏结束，得到模拟的结果（胜负）。模拟过程可以使用随机策略，也可以使用一些启发式方法来提高模拟的效率。
4. **反向传播（Backpropagation）**：
   将模拟的结果反向传播到搜索树中。每个父节点根据其子节点的模拟结果更新其统计数据（如胜负次数和总模拟次数）。

4. 探索与开发的平衡

通过 UCT 算法，MCTS 在每一轮选择子节点时，可以在 **探索** 和 **开发** 之间取得良好的平衡：

- **开发（Exploitation）**：如果某个子节点的胜率高且模拟次数多，则该节点的 $\frac{w_i}{n_i}$ 会比较大，从而使得该节点在选择时更容易被选中，体现出 **开发** 的思想。
- **探索（Exploration）**：如果某个子节点的模拟次数很少（即 n_i 很小），则 $\sqrt{\frac{\ln N_p}{n_i}}$ 项会变大，从而鼓励对这些尚未充分探索的节点进行 **探索**。

通过调整常数 C，我们可以控制 **探索** 和 **开发** 的权重：

- 如果 C 较大，算法倾向于 **探索更多的节点**，即使它们的胜率较低。
- 如果 C 较小，算法倾向于 **开发** 已知的最优节点，即选择那些胜率较高的路径。

5. UCT 在实际应用中的优势

1. **自动平衡探索与开发**：UCT 算法通过公式自然地平衡了探索和开发，在搜索树中既不会停留在当前胜率较高的路径，也能保证探索那些可能隐藏更优解的路径。
2. **无须完备的评估函数**：与传统的算法（如 Minimax）依赖于明确的评估函数不同，UCT 基于 **随机模拟** 来评估节点的好坏，因此可以在 **没有完整评估函数** 的情况下，也能有效地进行决策。
3. **适应性强**：UCT 适应了大部分不确定性较高的环境，尤其适用于博弈类问题，如 **围棋**、**国际象棋**、**德州扑克** 等，尤其是当搜索空间巨大且无法完全评估时，UCT 能够有效探索。

4. **效果逐渐收敛**：随着模拟次数的增加，UCT 会逐渐趋向最优解，因此，MCTS + UCT 可以在足够多的模拟下给出较为准确的决策。

6. UCT 算法的挑战和局限性

1. **计算成本高**：尽管 UCT 是一个相对高效的启发式算法，但它依然需要大量的模拟才能获得准确的结果，对于一些复杂的博弈，模拟的次数可能非常庞大，因此计算资源的消耗不可忽视。
2. **没有绝对的最优保证**：尽管 UCT 在多次模拟下能够收敛到一个接近最优解的解，但它并不一定能保证 **全局最优解**。特别是在某些游戏状态空间较大时，仍然可能会出现局部最优。

总结

UCT 是 **蒙特卡洛树搜索（MCTS）** 中的一个关键部分，它通过动态平衡 **探索** 和 **开发** 的策略，有效地在 **决策树** 中进行选择。通过计算每个节点的 **UCT 值**，可以保证在模拟过程中既能探索未曾尝试的动作，也能利用已有的成功经验来作出决策。UCT 算法使得 MCTS 在 **不确定性** 和 **复杂性** 较高的问题中表现出了强大的能力，特别是在游戏 AI 和决策支持系统中取得了广泛应用。

1.2.5 两种算法的异同

相同点

1. **应用场景**：
   两者都是 **用于决策树**（游戏树）搜索的算法，广泛应用于 **博弈类问题**（如围棋、国际象棋、德州扑克等）。它们的目标都是通过搜索找到最佳的行动路径。
2. **搜索树的构建**：
   它们都通过构建 **树状结构** 来表示游戏的状态转移，并在树中选择最优的行动路径。搜索树的根节点通常表示当前的游戏状态，子节点表示接下来的状态。
3. **优化目标**：
   两者的最终目标都是通过优化搜索过程来提高决策效率，减少不必要的计算。剪枝通过提前剪掉不可能导致最优解的路径，而MCTS通过模拟和回溯来逐步发现较优的决策路径。
4. **启发式**：
   都带有一定的启发式搜索，虽然方式不同。剪枝算法的启发式是通过评估函数来估算每个节点的最优值；MCTS则是通过模拟来估算每个决策的胜率。

不同点

1. 算法原理

- **剪枝算法（如Alpha-Beta剪枝）**：
  - 是 **确定性** 搜索算法，常用于 **Minimax** 类型的搜索树中，主要用于 **游戏树** 的搜索。
  - 剪枝的目的是减少不必要的搜索，避免遍历不可能导致最佳解的分支，从而提高搜索效率。
  - 它通过对树中的某些分支进行“剪枝”来减少计算量。具体地，如果一个节点的值已经不可能影响父节点的选择，则会停止进一步的搜索。

    - **Alpha-Beta剪枝** 是Minimax算法的一种优化版本，它维护两个变量 **α** 和 **β** 来跟踪当前搜索路径的最优值，如果某条路径的值已经不可能超过（或低于）当前的α或β值，就可以剪枝，避免继续搜索该路径。
- **蒙特卡洛树搜索（MCTS）**：
    - 是 **概率性** 搜索算法，基于 **模拟** 来选择动作，不需要完整遍历所有的节点。
    - MCTS的核心是通过随机模拟游戏的多个可能路径来估计各个决策点的胜率，然后根据这些结果逐步更新搜索树。
    - 它使用 **选择**、**扩展**、**模拟** 和 **反向传播** 四个步骤来搜索最佳决策。
    - 由于MCTS依赖大量的随机模拟（rollouts），它能够处理一些极为复杂的决策问题，如围棋这类非常大搜索空间的问题。

2. 搜索方式

- **剪枝算法（Alpha-Beta剪枝）**：
    - 采用 **深度优先搜索**（DFS）的方式，按层次搜索游戏树。
    - 它使用递归的方式深度遍历树的分支，逐步"剪掉"不必要的分支，从而减少搜索空间。
    - 剪枝仅仅是在对已有路径进行优化，不需要随机模拟。
- **蒙特卡洛树搜索（MCTS）**：
    - 采用 **广度优先搜索**（BFS）的方式，构建一个树状结构并在其中进行多次随机模拟。
    - 每次模拟的结果会影响父节点的决策，从而帮助算法选择更有可能带来胜利的动作。
    - 它不需要穷尽所有可能的路径，而是通过 **随机模拟** 快速评估各个节点。

3. 时间复杂度

- **剪枝算法（Alpha-Beta剪枝）**：
    - 在最好的情况下，Alpha-Beta剪枝可以将搜索复杂度从 $O(b^d)$ 降低到 $O(b^{d/2})$，其中 b 是每个节点的子节点数，d 是树的深度。
    - 通过剪枝，Alpha-Beta算法能够显著减少遍历的节点数，但其依然是 **确定性** 的，依赖于搜索树的深度。
- **蒙特卡洛树搜索（MCTS）**：
    - MCTS的时间复杂度是由模拟次数决定的，复杂度为 $O(n \cdot t)$，其中 n 是模拟的次数，t 是每次模拟的时间。
    - 随着模拟次数的增加，MCTS的结果会更加准确，但这会导致计算成本增加。
    - MCTS的表现会随着搜索空间的增大而提高，但仍然需要大量的模拟。

4. 处理复杂度

- 剪枝算法（Alpha-Beta剪枝）
    - 更适合 **有限搜索空间** 的问题，能够在不需要大量模拟的情况下，快速确定最优解。
    - 它的性能受限于游戏树的深度，如果游戏树的深度很大，它的效果会逐渐下降。
- 蒙特卡洛树搜索（MCTS）
    - 更适合处理 **大规模的搜索空间**，特别是那些每一步都有非常多选择的复杂游戏，如围棋、德州扑克等。
    - MCTS通过 **随机模拟** 可以处理非常复杂的游戏情况，而不需要穷尽所有可能的搜索路径。

5. 准确性与最优性

- **剪枝算法（Alpha-Beta剪枝）**：
  - 能保证 **找到最优解**，前提是搜索树完整，且能够在有限时间内遍历到最优的决策节点。
  - 如果剪枝策略有效，Alpha-Beta可以保证找到最佳决策。
- **蒙特卡洛树搜索（MCTS）**：
  - MCTS不一定能保证找到最优解，因为它依赖于 **随机模拟**，尽管随着模拟次数的增加，结果会越来越精确，但由于模拟是随机的，始终无法保证找到最优解。
  - 通过 **足够多的模拟**，MCTS的结果通常可以接近最优解，但不会100%保证。

6. 适用场景

- **剪枝算法（Alpha-Beta剪枝）**：
  - 适用于 **确定性** 游戏，尤其是 **棋类游戏**（如象棋、国际象棋等），其中每一步的状态都可以明确计算并评估。
  - 游戏树的规模相对适中，且每一步决策比较明确。
- **蒙特卡洛树搜索（MCTS）**：
  - 适用于 **复杂、不确定性** 较高的游戏，尤其是 **大规模游戏**（如围棋、德州扑克等）。
  - 游戏的状态变化多样，难以通过常规算法进行完全搜索，MCTS通过模拟来弥补这一点。

总结对比

特性	剪枝算法（Alpha-Beta剪枝）	蒙特卡洛树搜索（MCTS）
搜索方式	深度优先搜索（DFS）	广度优先搜索（BFS）
适用类型	确定性问题，棋类游戏	不确定性问题，复杂的游戏
时间复杂度	$O(b^{d/2})$	$O(n \cdot t)$，依赖模拟次数与每次模拟时间
最优性	保证最优解	随着模拟次数增加，结果更准确，但无法保证最优解
计算成本	较低，但树的深度增加时计算量会增长	需要大量模拟，计算成本随模拟次数增加
准确性	高，但依赖树的深度和剪枝效果	高，随着模拟次数增加准确度提高，但模拟是随机的
应用场景	游戏树较小，明确评估每一步的情况	游戏树庞大，涉及大量随机因素的游戏（如围棋）

总结

- **剪枝算法**（如Alpha-Beta剪枝）在计算资源有限且需要寻找最优解时表现更好，特别是在搜索树较小且每一步决策明确的情况下。
- **蒙特卡洛树搜索**适合用于复杂的、不确定性的游戏，其优势在于能够通过大量的模拟探索庞大的搜索空间，但由于依赖于随机模拟，它无法保证找到最优解。

Part 2：Hex 博弈程序实现

1. 整体设计与架构

主要包含以下模块：

- **HexAI 类**：封装棋盘状态、评估函数、搜索算法（Minimax、MCTS）和判断胜负的方法。
- **蒙特卡洛模拟函数**：利用随机模拟来评估候选落子的胜率。
- **辅助函数**：例如 `swap_move`、`format_move`、`parse_move` 等，辅助实现交换规则、格式转换和用户输入解析。
- **主程序流程（main 函数）**：负责处理用户输入、选择先手、交换规则的决策以及轮流落子直至结束。

2. 蒙特卡洛模拟与随机游戏

2.1 随机游戏模拟

函数 `simulate_random_game_static` 用于从当前棋盘状态开始，随机模拟后续的落子过程，直到棋盘填满，并判断最终谁获胜。

```
def simulate_random_game_static(board, player):
    board_copy = board.copy()
    size = board_copy.shape[0]
    moves = [(i, j) for i in range(size) for j in range(size) if board_copy[i, j] == 0]
    random.shuffle(moves)
    cur_player = player
    for move in moves:
        board_copy[move] = cur_player
        cur_player = -cur_player
    temp = HexAI(size)
    temp.board = board_copy
    if temp.is_winner(1):
        return 1
    elif temp.is_winner(-1):
        return -1
    else:
        return 0
```

2.2 针对单一步骤的模拟

函数 `simulate_for_move` 为每个候选落子进行多次模拟，并统计该落子下获胜的次数。

```python
def simulate_for_move(args):
    move, board, player, simulations = args
    win_count = 0
    for _ in range(simulations):
        board_copy = board.copy()
        board_copy[move] = player
        result = simulate_random_game_static(board_copy, -player)
        if result == player:
            win_count += 1
    return move, win_count
```

---

3. HexAI 类详解

3.1 初始化

构造函数初始化棋盘状态、棋盘大小以及相邻六个方向的定义，用于后续连通性判断和路径搜索。

```python
class HexAI:
    def __init__(self, size=11):
        self.size = size
        self.board = np.zeros((size, size), dtype=int)
        self.directions = [(1, 0), (0, 1), (1, -1), (-1, 1), (-1, 0), (0, -1)]
```

3.2 评估函数及其子模块

综合评估函数

函数 `evaluate` 结合了连通性评估、最短路径、桥接模式和对手威胁分析来计算当前局面的评分。

```python
def evaluate(self, player):
    connectivity_score = self.evaluate_connectivity(player)
    path_score = self.enhanced_shortest_path(player)
    opponent = -player
    opponent_threat = self.opponent_threat_analysis(opponent)
    bridge_score = self.evaluate_bridging_patterns(player)
    score = (
            connectivity_score * 15 +
            path_score * 100 +
            bridge_score * 20 -
            opponent_threat * 80
    )
    return score
```

连通性评估

`evaluate_connectivity` 函数利用广度优先搜索（BFS）寻找所有己方棋子连通区域，并根据区域规模、边界连接、边缘奖励及空邻居数量给予分数。

```python
def evaluate_connectivity(self, player):
```

```
    size = self.size
    score = 0
    visited = np.zeros((size, size), dtype=bool)
    connected_regions = []

    for i in range(size):
        for j in range(size):
            if self.board[i, j] == player and not visited[i, j]:
                region = []
                queue = deque([(i, j)])
                visited[i, j] = True
                while queue:
                    x, y = queue.popleft()
                    region.append((x, y))
                    for dx, dy in self.directions:
                        nx, ny = x + dx, y + dy
                        if 0 <= nx < size and 0 <= ny < size and self.board[nx, ny] ==
player and not visited[nx, ny]:
                            queue.append((nx, ny))
                            visited[nx, ny] = True
                connected_regions.append(region)

    for region in connected_regions:
        region_size = len(region)
        edge_connections = 0
        if player == 1:
            top_connected = any(x == 0 for x, y in region)
            bottom_connected = any(x == size - 1 for x, y in region)
            edge_connections = 10 * (top_connected + bottom_connected)
        else:
            left_connected = any(y == 0 for x, y in region)
            right_connected = any(y == size - 1 for x, y in region)
            edge_connections = 10 * (left_connected + right_connected)

        edge_bonus = 0
        if player == 1:
            for x, y in region:
                if x == 0:
                    edge_bonus += 2
                if x == size - 1:
                    edge_bonus += 2
        else:
            for x, y in region:
                if y == 0:
                    edge_bonus += 2
                if y == size - 1:
                    edge_bonus += 2

        empty_neighbors = 0
        for x, y in region:
            for dx, dy in self.directions:
                nx, ny = x + dx, y + dy
                if 0 <= nx < size and 0 <= ny < size and self.board[nx, ny] == 0:
```

```
                    empty_neighbors += 1

        region_score = region_size * region_size + edge_connections + edge_bonus + empty_neighbors * 0.5
        score += region_score

        if (player == 1 and top_connected and bottom_connected) or \
                (player == -1 and left_connected and right_connected):
            score += 1000
    return score
```

最短路径评估

`enhanced_shortest_path` 使用类似 Dijkstra 算法的思想为己方寻找一条从起始边到目标边的最短“成本”路径，并以成本反比的方式转换为评分。

```
def enhanced_shortest_path(self, player):
    opponent = -player
    if player == 1:
        start_edge = [(0, j) for j in range(self.size)]
        end_edge = [(self.size - 1, j) for j in range(self.size)]
    else:
        start_edge = [(i, 0) for i in range(self.size)]
        end_edge = [(i, self.size - 1) for i in range(self.size)]

    dist = np.full((self.size, self.size), float('inf'))
    pq = []

    for x, y in start_edge:
        cost = 0
        if self.board[x, y] == 0:
            cost = 1
        elif self.board[x, y] == opponent:
            cost = 5
        elif self.board[x, y] == player:
            cost = 0

        if cost < float('inf'):
            heapq.heappush(pq, (cost, x, y))
            dist[x, y] = cost

    while pq:
        d, x, y = heapq.heappop(pq)
        if (player == 1 and x == self.size - 1) or (player == -1 and y == self.size - 1):
            return 100 / (1 + d)
        if d > dist[x, y]:
            continue
        for dx, dy in self.directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < self.size and 0 <= ny < self.size:
                new_cost = d
```

```
                if self.board[nx, ny] == 0:
                    new_cost += 1
                elif self.board[nx, ny] == opponent:
                    new_cost += 5
                elif self.board[nx, ny] == player:
                    new_cost += 0
                if new_cost < dist[nx, ny]:
                    dist[nx, ny] = new_cost
                    heapq.heappush(pq, (new_cost, nx, ny))
    return 0
```

对手威胁分析

该函数评估对手从其起始边到目标边的路径，并根据其评分乘以不同的倍率反映对手的潜在威胁。

```
def opponent_threat_analysis(self, opponent):
    opponent_path_score = self.enhanced_shortest_path(opponent)
    if opponent_path_score > 50:
        return opponent_path_score * 1.5
    elif opponent_path_score > 25:
        return opponent_path_score * 1.2
    else:
        return opponent_path_score * 0.8
```

桥接模式评估

`evaluate_bridging_patterns` 以及辅助函数 `_check_bridges` 检测棋子间是否存在有利的桥接布局，并给予额外分数。

```
def evaluate_bridging_patterns(self, player):
    score = 0
    size = self.size
    for i in range(size):
        for j in range(size):
            if self.board[i, j] == player:
                score += self._check_bridges(i, j, player)
    return score

def _check_bridges(self, x, y, player):
    bridge_score = 0
    size = self.size
    bridge_directions = [
        [(1, 0), (1, -1)],
        [(1, 0), (0, 1)],
        [(0, 1), (-1, 1)],
        [(0, 1), (1, -1)],
        [(-1, 0), (-1, 1)],
        [(-1, 0), (0, -1)]
    ]

    for dir1, dir2 in bridge_directions:
        dx1, dy1 = dir1
```

```
        dx2, dy2 = dir2
        x1, y1 = x + dx1, y + dy1
        if not (0 <= x1 < size and 0 <= y1 < size):
            continue
        x2, y2 = x + dx2, y + dy2
        if not (0 <= x2 < size and 0 <= y2 < size):
            continue
        x3, y3 = x + dx1 + dx2, y + dy1 + dy2
        if not (0 <= x3 < size and 0 <= y3 < size):
            continue

        if (self.board[x3, y3] == player and
                self.board[x1, y1] == 0 and
                self.board[x2, y2] == 0):
            bridge_score += 3
        elif (self.board[x3, y3] == 0 and
              (self.board[x1, y1] == 0 or self.board[x2, y2] == 0)):
            if (self.board[x1, y1] == player or self.board[x2, y2] == player):
                bridge_score += 2
            else:
                bridge_score += 1

        x4, y4 = x + 2 * dx1, y + 2 * dy1
        x5, y5 = x + 2 * dx2, y + 2 * dy2
        if (0 <= x4 < size and 0 <= y4 < size and
                0 <= x5 < size and 0 <= y5 < size):
            if (self.board[x1, y1] == 0 and self.board[x4, y4] == player and
                    self.board[x2, y2] == 0 and self.board[x5, y5] == player):
                bridge_score += 4
    return bridge_score
```

其他辅助评估函数

`bridge_potential`、`virtual_connection` 和 `blocking_value` 等函数，用于评估某一位置的桥接潜力、直接连接和对对手的阻挡作用。

3.3 搜索算法

HexAI 类实现了两种搜索方法以供 AI 决策：

Minimax 算法（带 Alpha-Beta 剪枝）

```
def minimax(self, depth, player, alpha, beta):
    if depth == 0 or self.is_winner(1) or self.is_winner(-1):
        return self.evaluate(player), None

    best_move = None
    empty_cells = [(i, j) for i in range(self.size) for j in range(self.size) if self.board[i, j] == 0]
    empty_cells.sort(key=lambda move: self.evaluate_move(move, player), reverse=True)

```

```
    if player == 1:
        max_eval = -float('inf')
        for i, j in empty_cells:
            self.board[i, j] = player
            eval, _ = self.minimax(depth - 1, -player, alpha, beta)
            self.board[i, j] = 0
            if eval > max_eval:
                max_eval = eval
                best_move = (i, j)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval, best_move
    else:
        min_eval = float('inf')
        for i, j in empty_cells:
            self.board[i, j] = player
            eval, _ = self.minimax(depth - 1, -player, alpha, beta)
            self.board[i, j] = 0
            if eval < min_eval:
                min_eval = eval
                best_move = (i, j)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval, best_move

def evaluate_move(self, move, player):
    i, j = move
    self.board[i, j] = player
    score = self.evaluate(player)
    self.board[i, j] = 0
    return score
```

蒙特卡洛树搜索（MCTS）

利用多进程加速对所有空位候选落子的模拟，统计各个落子的胜率，最后选择胜率最高的作为最佳落子。

```python
def mcts(self, player, simulations=400000, num_processes=mp.cpu_count()):
    empty_cells = [(i, j) for i in range(self.size) for j in range(self.size) if self.board[i, j] == 0]
    if not empty_cells:
        return None
    total_moves = len(empty_cells)
    simulations_per_move = simulations // total_moves
    args_list = [(move, self.board.copy(), player, simulations_per_move) for move in empty_cells]

    with mp.Pool(processes=num_processes) as pool:
        results = pool.map(simulate_for_move, args_list)

    move_scores = {move: wins / simulations_per_move for move, wins in results}
    return max(move_scores, key=move_scores.get)
```

获取最佳落子

根据参数选择使用 MCTS 或 Minimax 算法来决定最佳落子：

```python
def get_best_move(self, player, depth=17, use_mcts=True):
    if use_mcts:
        return self.mcts(player)
    else:
        _, best_move = self.minimax(depth, player, -float('inf'), float('inf'))
        return best_move
```

3.4 胜负判断

函数 `is_winner` 利用深度优先搜索检查是否存在从起始边到目标边的连通路径，从而判断某个玩家是否获胜。

```python
def is_winner(self, player):
    visited = set()

    def dfs(x, y):
        if (x, y) in visited:
            return False
        if player == 1 and x == self.size - 1:
            return True
        if player == -1 and y == self.size - 1:
            return True
        visited.add((x, y))
        for dx, dy in [(1, 0), (0, 1), (1, -1), (-1, 1), (-1, 0), (0, -1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < self.size and 0 <= ny < self.size and self.board[nx, ny] == player:
                if dfs(nx, ny):
                    return True
        return False

```

```
    for i in range(self.size):
        if player == 1 and self.board[0, i] == 1 and dfs(0, i):
            return True
        if player == -1 and self.board[i, 0] == -1 and dfs(i, 0):
            return True
    return False
```

4. 辅助函数

交换规则与格式转换

- **交换函数** `swap_move`：将棋步坐标沿主对角线反射，用于交换棋子位置。

```
def swap_move(move):
    """Swap a move by reflecting it across the diagonal of the board"""
    row, col = move
    return (col, row)
```

- **格式化函数** `format_move`：将棋步转换格式（如 "a1"）。

```
def format_move(move):
    """Convert a move tuple (row, col) to a standard format like 'a1'"""
    row, col = move
    return f"{chr(col + ord('a'))}{row + 1}"
```

- **解析输入** `parse_move`：将用户输入的字符串解析成棋盘坐标，进行基本校验。

```
def parse_move(move_str, size):
    move_str = move_str.strip().lower()
    if len(move_str) < 2:
        return None
    col_char = move_str[0]
    row_str = move_str[1:]
    if not col_char.isalpha() or not row_str.isdigit():
        return None
    col = ord(col_char) - ord('a')
    row = int(row_str) - 1
    if 0 <= row < size and 0 <= col < size:
        return (row, col)
    return None
```

5. 主程序流程

主函数 `main()` 负责处理游戏的整体流程，包括玩家选择先手、处理交换规则以及轮流落子直到游戏结束。

```
def main():
    size = 11
    game = HexAI(size)
```

```
    human_player = 1  # 默认人类连接顶到底
    ai_player = -1    # 默认 AI 连接左到右
    use_mcts = True
    first_move_made = False
    ai_first_move = None

    # 根据第一行输入判断先手
    first_input = input()

    if first_input.lower() == "first":
        # AI 先手，调整双方身份
        human_player = -1
        ai_player = 1
        ai_first_move = game.get_best_move(ai_player, depth=17, use_mcts=use_mcts)
        game.board[ai_first_move] = ai_player
        print(format_move(ai_first_move))
        first_move_made = True
        current_player = human_player

    elif first_input.lower() == "finish":
        print("游戏结束")
        return

    else:
        # 人类先手，输入的第一步为人类落子
        first_move = parse_move(first_input, size)
        if first_move is None:
            print("无效输入，请重新输入。")
            return
        if game.board[first_move] != 0:
            print("该位置已被占用，请重新选择。")
            return

        # 人类第一步落子
        game.board[first_move] = human_player
        first_move_made = True

        # AI 判断是否使用交换规则
        normal_eval = game.evaluate(ai_player)

        game.board[first_move] = 0
        game.board[first_move] = ai_player
        swap_eval = game.evaluate(ai_player)

        game.board[first_move] = 0

        if swap_eval > normal_eval * 1.2:
            swapped_move = swap_move(first_move)
            if (0 <= swapped_move[0] < size and 0 <= swapped_move[1] < size and
game.board[swapped_move] == 0):
                game.board[first_move] = 0
                game.board[swapped_move] = ai_player
                print("change")
```

```
                current_player = human_player
            else:
                game.board[first_move] = human_player
                move = game.get_best_move(ai_player, depth=17, use_mcts=use_mcts)
                game.board[move] = ai_player
                print(format_move(move))
                current_player = human_player
        else:
            game.board[first_move] = human_player
            move = game.get_best_move(ai_player, depth=17, use_mcts=use_mcts)
            game.board[move] = ai_player
            print(format_move(move))
            current_player = human_player

    # 主游戏循环
    while True:
        if game.is_winner(human_player):
            print("human win")
            break
        if game.is_winner(ai_player):
            print("AI win")
            break

        if current_player == human_player:
            move_input = input()
            if move_input.lower() == "finish":
                print("游戏结束")
                break

            if first_move_made and ai_player == 1 and move_input.lower() == "change"
and ai_first_move is not None:
                game.board[ai_first_move] = 0
                swapped_move = swap_move(ai_first_move)
                game.board[swapped_move] = human_player
                current_player = ai_player
                continue

            move = parse_move(move_input, size)
            if move is None:
                print("无效输入，请重新输入。")
                continue
            if game.board[move] != 0:
                print("该位置已被占用，请重新选择。")
                continue

            game.board[move] = human_player
            current_player = ai_player

        else:  # AI 的回合
            move = game.get_best_move(ai_player, depth=17, use_mcts=use_mcts)
            if move is None:
                print("平局! ")
                break
```

```
            game.board[move] = ai_player
            print(format_move(move))
            current_player = human_player

if __name__ == "__main__":
    main()
```

6. 多进程与性能考虑

在蒙特卡洛搜索（MCTS）中，代码利用 `multiprocessing.Pool` 将模拟任务分配给多个 CPU 核心，以加速大量模拟（默认 400000 次），从而提高搜索效率。

7. 特别说明

> 在这段代码中，MCTS 部分使用了下面这行代码来确定使用多少个进程：
>
> ```
> num_processes = mp.cpu_count()
> ```
>
> 这行代码会返回系统中的 CPU 核心数，也就是说，默认情况下程序会尝试启动与 CPU 核心数相同的进程数。
>
> ## 解释 CPU 数量级对性能的影响
>
> 1. **多核心系统（例如 4 核、8 核或更多）**
>   - 如果你的机器拥有 4 核或 8 核 CPU，程序就能并行运行 4 个或 8 个模拟任务。这可以大幅提高蒙特卡洛搜索的效率，因为多个进程可以同时运行并完成各自的模拟任务，从而加快总的计算速度。
> 2. **低核心数或老旧 CPU（例如 1 核或 2 核）**
>   - 如果 CPU 核心数较少，启动的并行进程也就少，整个 MCTS 部分的并行速度会受限。例如，如果系统只有 1 个核心，所有模拟任务都只能串行执行，这会显著降低搜索效率。
>   - 此外，部分低性能的 CPU 在面对大量进程调度和频繁的进程间通信时，可能会出现资源争用或调度开销较高的问题，从而导致总体性能下降。
>
> > 运行这部分代码需要电脑具备一定的性能（cpu数量），否则可能会超时。代码在本机上测试运行时间正常。
> >
> > 附：
> >
> > ```
> > PS C:\Users\34544> cd "C:\Users\34544\Desktop\【2025年工高招生】实践面题目附件\附件\A-attachment\code"
> > PS C:\Users\34544\Desktop\【2025年工高招生】实践面题目附件\附件\A-attachment\code> python main.py
> > Referee load
> > 玩家1对阵玩家2...玩家1先手,执红棋...
> > 在get_response时，先手的进程意外终止
> > 先手 共消耗32.1125秒
> > 后手 共消耗0.1027秒
> > 比赛结果GameResult.p1_win
> >
> > 玩家2对阵玩家1...玩家2先手,执红棋...
> > 先手落子重复，判负
> > 先手 共消耗0.001秒
> > 后手 共消耗6.7213秒
> > 比赛结果GameResult.p2_win
> >
> > 玩家1，胜场积分为2.0
> > 玩家2，胜场积分为0.0
> > ```

8. 总结

1. 程序启动与初始化

- **初始化棋盘和玩家**
  - 创建一个 `HexAI` 对象，初始化一个大小为 11×11 的棋盘（全 0 矩阵）。
  - 根据用户输入确定谁先手：
    - 如果输入 `"first"`，则 AI 先走并将身份设为玩家 1（连接顶到底），人类为玩家 -1。
    - 否则，默认人类先走（玩家 1）并根据输入的棋步进行落子。

---

2. 判断局面和决策过程

- **局面评估**
  - **评估函数** `evaluate(player)`：
    - **连通性判断**：利用 BFS 找出己方棋子连通区域，判断棋子群是否连接了目标边（例如顶到底或左到右），以及区域大小、边界连接情况等。
    - **最短路径算法**：通过类似 Dijkstra 算法计算从起始边到目标边的最短代价路径，代价越低（路径越短）则说明局面越有利，评分为 `100 / (1 + d)`。
    - **桥接模式检测**：检查棋子之间是否存在桥接结构，给出额外分数。
    - **对手威胁分析**：使用对手的最短路径得分来衡量对手的威胁，并在总分中扣分。
- **搜索决策**
  - **Minimax 搜索（带 Alpha-Beta 剪枝）**：
    - 递归搜索未来若干步局面（深度控制在 17 层），在叶节点通过 `evaluate` 进行局面评分，反复比较所有候选落子的评估值，选择得分最优的那个。
  - **蒙特卡洛树搜索（MCTS）**：
    - 对每个候选落子进行大量随机模拟（默认总模拟次数 400000 次，均摊到每个空位），利用多进程并行计算每个落子在随机游戏中最终胜率，然后选取胜率最高的落子。
- **交换规则判断**
  - 当人类先手时，AI 会评估两种局面：
    - **保持当前身份的局面**：直接用 `evaluate(ai_player)` 得到局面评分。
    - **交换身份的局面**：将人类的第一步视作 AI 的落子，再调用 `evaluate(ai_player)` 得到另一种评分。
  - 如果交换后的局面评分明显更高（如超过当前局面的 1.2 倍），则执行交换，即调用 `swap_move` 得到新的落子位置，移除原有落子并落下交换棋子。

---

3. 运行流程（游戏主循环）

- **主循环**
  - 在每个回合开始前，首先调用 `is_winner(player)` 判断当前局面是否已经有一方连通获胜（例如玩家 1 是否从上到下连通）。
  - **人类回合**：
    - 读取输入，解析后调用 `parse_move` 转换为棋盘坐标，检查位置是否有效且为空，然后将人类棋子放置在该位置。
  - **AI 回合**：
    - 调用 `get_best_move(player, depth, use_mcts)`，根据设定选择使用 MCTS 或 Minimax 来计算最佳落子，然后将 AI 的棋子落下，并将落子的坐标格式化输出（如 `"a1"` 格式）。

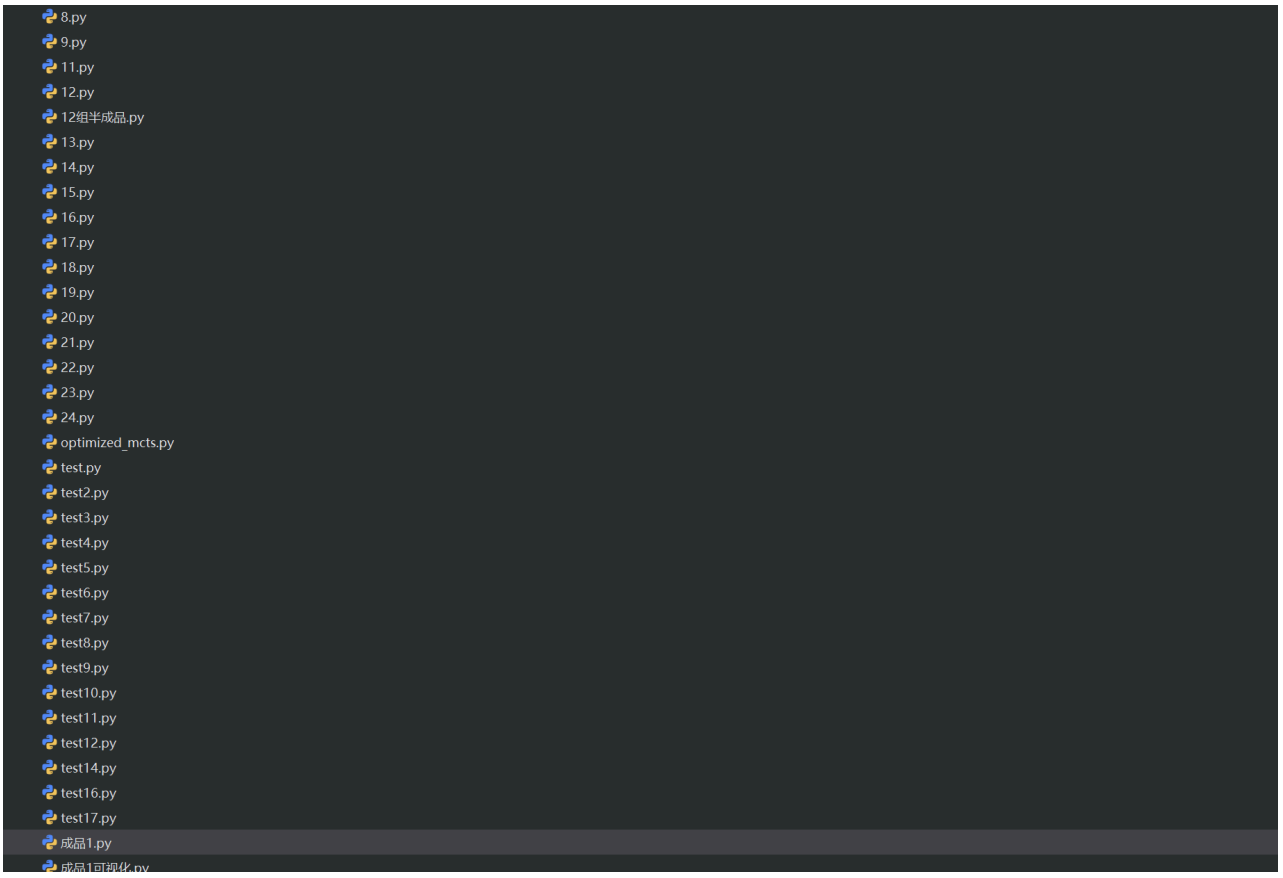
---

4. 总结

整个代码的运行基于以下核心逻辑：

- **棋盘状态管理**：通过 `HexAI` 类维护棋盘数据、棋子分布和各种方向信息。
- **局面评估**：依靠多个子模块（连通性、最短路径、桥接模式、对手威胁）为局面打分，帮助判断当前哪个局面更有利。
- **搜索决策**：通过 Minimax 与 MCTS 两种搜索方式，在多个候选落子中选择最优解。
- **判断胜负**：在每步落子后使用 DFS 判断是否存在从一边到另一边的连通路径，从而确定游戏胜负。
- **交换规则**：在合适时机比较交换前后局面，判断是否应执行交换，平衡先手优势。

9. 思路过程



> 一共迭代了近50个版本，从minmax算法到蒙特卡洛到剪枝算法优化，有很多版本被我们抛弃，这里就不一一赘述了。最后，我们修改了算法重复次数，计算机预测步骤个数，以及多个cpu同时运行来优化ai性能。还是很有收获的。

Bonus

Bonus 1

在我们的代码中，判断棋局结束（胜利或失败）的功能主要是通过函数 `is_winner` 实现的。这个函数采用了深度优先搜索（DFS）的方式，从目标起始边开始探索，检查是否存在一条连续的己方棋子通路延伸到目标边，从而确定是否达成了连通条件。

下面是该算法的设计思路以及在代码中的实现：

设计思路

1. **明确目标边**
   - 对于玩家1（通常负责连接上边到下边），我们从棋盘上方（第一行）的所有己方棋子出发，检查是否能延伸到下边（最后一行）。
   - 对于玩家-1（通常负责连接左边到右边），则从棋盘左侧（第一列）的己方棋子出发，检查是否能延伸到右侧（最后一列）。
2. **使用深度优先搜索（DFS）**

- 从目标起始边的每个己方棋子开始，递归地沿着所有相邻的方向（六个方向）探索。
- 在探索过程中维护一个“已访问”集合，防止重复搜索和无限循环。
- 如果在搜索过程中达到了目标边，则说明该玩家已经构成连通，游戏结束，对应玩家获胜。

3. **判断结束情况**
   - 在每次落子后（无论是人类还是 AI），程序会调用该算法来检查是否有玩家获胜。
   - 若存在一方满足连通条件，则结束当前对弈，自动停止游戏流程。

代码实现

下面的代码块展示了我们如何通过 DFS 判断棋局结束的情况：

```
def is_winner(self, player):
    visited = set()

    def dfs(x, y):
        if (x, y) in visited:
            return False
        # 对于玩家1，检查是否到达棋盘的下边；对于玩家-1，则检查是否到达右边
        if player == 1 and x == self.size - 1:
            return True
        if player == -1 and y == self.size - 1:
            return True
        visited.add((x, y))
        # 遍历六个可能的相邻方向
        for dx, dy in [(1, 0), (0, 1), (1, -1), (-1, 1), (-1, 0), (0, -1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < self.size and 0 <= ny < self.size and self.board[nx, ny] ==
player:
                if dfs(nx, ny):
                    return True
        return False

    # 对于玩家1，从第一行的所有己方棋子开始
    for i in range(self.size):
        if player == 1 and self.board[0, i] == 1 and dfs(0, i):
            return True
        # 对于玩家-1，从第一列的所有己方棋子开始
        if player == -1 and self.board[i, 0] == -1 and dfs(i, 0):
            return True
    return False
```

总结

- **是否结束**：该算法在每次落子后自动判断棋盘中是否存在从起始边到目标边的连续连通路径。
- **设计关键点**：
  - 利用 DFS 遍历相邻棋子，确保遍历过程中避免重复检查。
  - 根据不同玩家的目标边（玩家1：上到下，玩家-1：左到右）设置起始搜索点。

    - 一旦搜索过程中发现连通到目标边，则返回 True 表示游戏结束，对应玩家获胜。
- **集成到程序中**：在主循环中，程序会在每次落子后调用 `is_winner`，如果返回 True，则输出相应信息并停止游戏。

Bonus 2

> 这部分我并没有写进代码，不过我有大体的思路

---

1. 主要特殊棋形分类

(1) 无效点位（Dead Spots）

- **定义**：某些点位无论谁落子都不会影响胜负，因此是无效的。
- **示例**：
    - 被四颗相同颜色棋子围住的内部点（死角）。
    - 两个已经被敌方控制的区域间的独立点位，不可能改变局势。
- **策略**：
    - 预处理棋盘，标记这些点位，搜索时直接跳过。
    - 通过 **连通性分析** 或 **拓扑判断** 来确定无效点。

---

(2) 边界点位（Border Stability）

- **定义**：某些靠近边界的棋子形成的结构，已经确保连通，不需要额外落子。
- **示例**：
    - 蓝色棋子在 **左侧或右侧边界** 已经形成连通路径，则该路径上的额外点位不再需要考虑。
    - 红色棋子在 **上侧或下侧边界** 形成稳定结构。
- **策略**：
    - 计算边界连通性，若某方已锁定边界，则避免不必要的防守或进攻。
    - 通过 **Flood Fill（洪水填充）** 或 **并查集** 快速判断边界连通情况。

---

(3) 桥形（Bridge）

- **定义**：棋盘上有一些隐形的“桥”，即两个棋子之间只要再下一步，就可以形成稳固的连接。
- **示例**：
    - **远距离桥**：形如 “× ×” （隔一个空位）
    - **斜桥**：斜向 2 个棋子相隔一格
- **策略**：
    - 提前计算“关键桥”位置，优先考虑落子。
    - 遇到敌方关键桥，进行阻断（如 MCTS 提前探索这些点）。

---

(4) 连接关键点（Critical Connection）

- **定义**：某些点如果被对手抢先落子，会极大影响局势，属于“关键连接点”。
- **示例**：
  - 连接两个大块己方区域的唯一空位。
  - 如果被对手占据，己方必须绕路增加额外步数。
- **策略**：
  - 在评估函数中赋予更高的权重。
  - 在搜索过程中，优先考虑关键连接点的落子。

---

2. 具体实现方式

为了将这些特殊棋形应用到模型中，可以使用以下技术：

(1) 预计算并存储

- 在 **游戏初始化** 时，扫描棋盘，存储无效点、桥、边界点等。
- 以 **哈希表 / 字典** 形式存储，减少运行时计算量。

(2) 结合搜索算法

- 在 **MCTS 或 Minimax 搜索时**，
  - 若某个落点属于 **无效点**，直接跳过。
  - 若某个落点属于 **桥、边界点、关键连接点**，则赋予 **额外评分**，影响搜索决策。

(3) 通过博弈数据训练策略

- 让 AI 自己多次对弈，分析高胜率对局，发现哪些点更关键，自动优化策略库。

Bonus 3

> 这部分同样来不及写了，但是有大致的简单的不成熟的想法
>
> > **Hex 棋开局库的算法思路**
> >
> > 1. **为什么要有开局库?**
> >   - Hex 棋开局时，可能的落子点太多，计算量大，AI 运行会变慢。
> >   - 预先存一些“好”开局，减少计算量，让 AI 更快、更稳定。
> > 2. **基本思路**
> >   - 先把一些**常见的好开局**存下来，比如：
> >   - 先手优选 `f5`（棋盘中心）
> >   - 对手下 `f5`，我们下 `e6`（对角扩展）
> >   - 开局时，AI 先查**开局库**，如果有对应的走法，直接使用。
> >   - 如果对手的落子不在库里，就进入普通搜索模式。

Bonus 4

> 其他优化

1. 搭桥（Bridge）

思路：

- 在 Hex 棋中，两个棋子如果间隔一个空位，并且这个空位不会被对方轻易阻断，那么它们可以视为“连通”状态。
- AI 需要优先占据这些关键点，保证自己的连通性，或阻断对方的连接。

实现方式：

- 预判两个棋子之间的空位是否是一个“安全连接点”
- 如果是，优先考虑落子，强化自己的连通性

---

2. 虚拟连接（Virtual Connection）

思路：

- 某些棋型结构天然具备强连通性，例如“桥接点”或“边界连接点”。
- AI 可以在评估时，假设这些位置最终会连接，而不必真实模拟每一步。

优化效果：

- 让 AI 评估更精准，避免浪费计算资源去模拟不必要的落子。

---

3. BFS / 并查集（Union-Find） 判断连通性

思路：

- 用 **BFS** 结构，动态维护当前棋子的连通情况。
- 每次落子后，快速检查是否形成了一条完整的路径（从一侧到另一侧）。

优化效果：

- **快速判断胜负**，避免多余计算。
- **实时评估形势**，AI 可以知道自己是否已经接近胜利或需要防守。

---

4. 关键点 Heuristics（启发式策略）

思路：

- **优先考虑边界附近的棋子**（因为靠边的棋子连接路径更短）。
- **避免无效点**（比如被包围的死角，走了也没有作用）。
- **防止被对手切断关键路径**（计算哪些点是对手的“必走点”）。

优化方式：

- 设定一个评分系统，给不同点位打分（如中心分高，死角分低）。
- AI 选择分数高的位置下棋，提高决策质量。

---

5. 反转（Swap Rule）优化

思路：

- 在 Hex 规则中，后手有 **交换权**（即可以选择交换棋子）。
- AI 需要判断 **先手的落子是否太好**，如果是，则选择交换。

优化方式：

- 计算开局 **最优点**，如果对手落子在这个点附近，可能意味着换手更有利。

Report

文件夹介绍

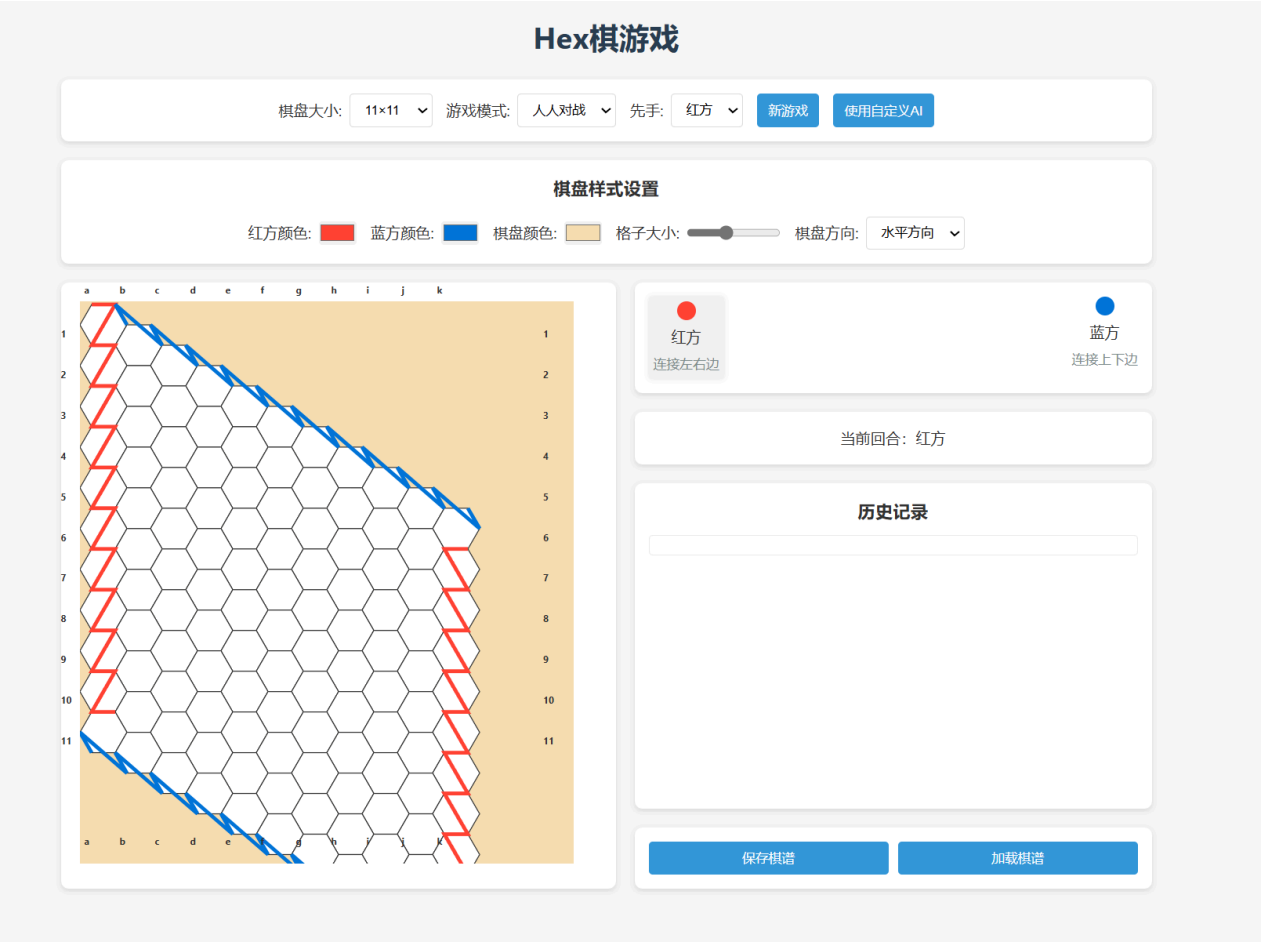
Frontend

- css文件夹
  - style.css --- 整体的样式设计
- js文件夹
  - board.js --- 棋盘整体设计
  - game.js --- 实现下hex棋的整体过程
  - history.js --- 实现记录步骤和回溯步骤的功能
  - main.js --- js代码的整体整合实现
  - save-load.js --- 下载导入棋谱
  - style.js --- 实现整体的风格调整列表
- index.html --- 整体前端网页的实现

Backend

- ai文件夹 --- 包含三种难度的ai
  - easy_ai.py
  - medium_ai.py
  - hard_ai.py --- 最初版本的我们的ai，后面优化的并未接入

- server.py --- 实现前端和后端的连接



基础功能介绍

- 实现了基础的11 * 11棋盘界面，默认颜色为红蓝
- 实现了首步交换功能，即实现第一步后可选择进行交换

Bonus 2

棋盘样式控制

- 支持调整棋盘的格子数和棋盘的大小
- 支持实现棋子颜色的变化和棋盘朝向的变化

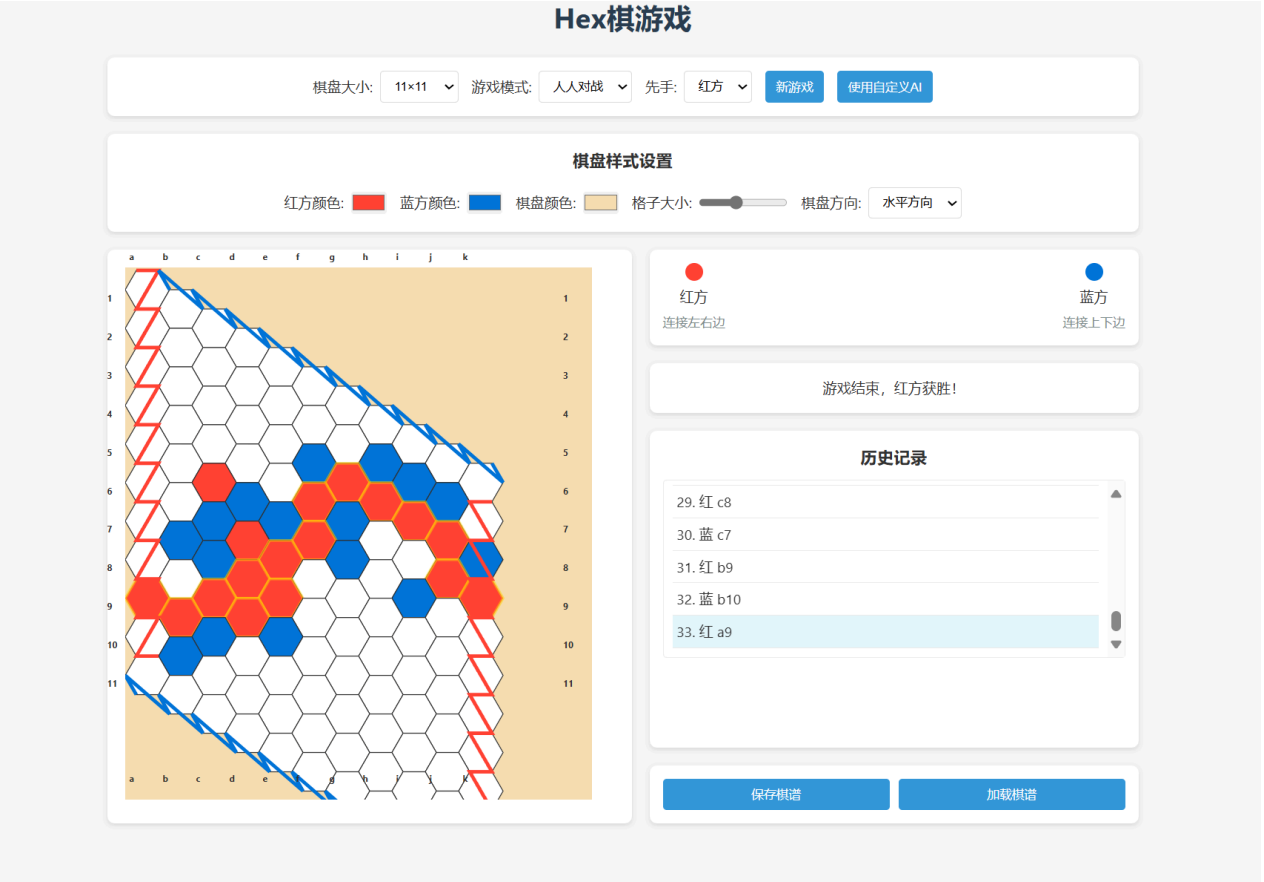
撤回列表

- 实现记录每一步的列表
- 能够通过列表进行回溯

保存代码

- 能够将当前棋谱进行保存

- 能够从本地读取棋谱进行棋局复现



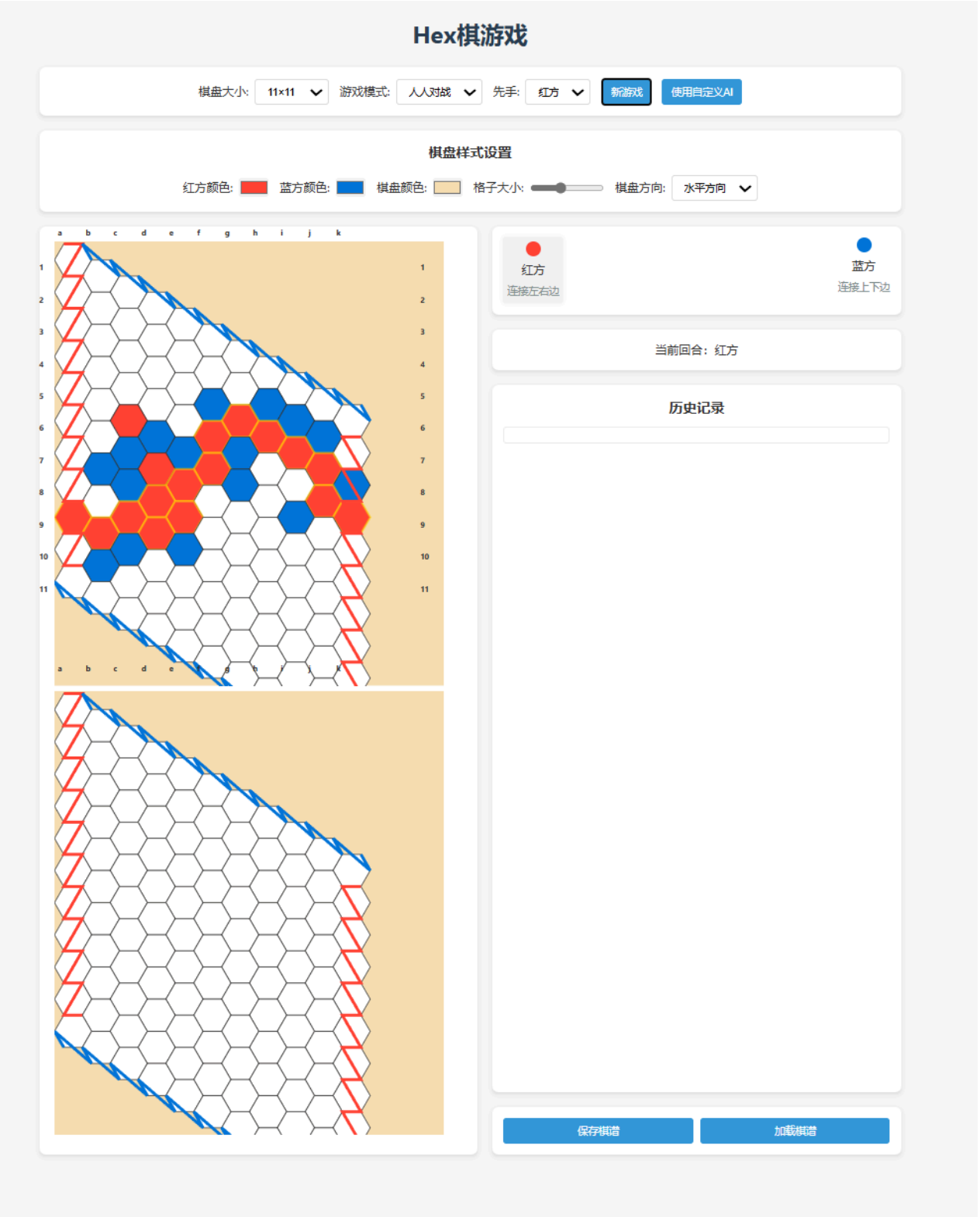
接入智能模型

- 提供人人对弈和人机对弈两种选项
- 电脑分为三种难度，均可与玩家进行对弈

其它优化

- 实现了新建棋盘后多张棋盘的集成显示
- 实现用户接入自定义ai进行博弈的功能
- 添加落子动画

- 添加胜利路径显示


Hex棋游戏
棋盘大小: 11×11
游戏模式: 人人对战
先手: 红方
新游戏
使用自定义AI
棋盘样式设置
红方颜色:
蓝方颜色:
棋盘颜色:
格子大小:
棋盘方向: 水平方向
a b c d e f g h i j k
1 2 3 4 5 6 7 8 9 10 11
红方
连接左右边
蓝方
连接上下边
当前回合：红方
历史记录
保存棋谱
加载棋谱