

四子棋报告

2024-05-28

张瀚宸

1 理论与基础实现

本次实验最基本的算法就是 MCTS。虽然好像推荐的是 α - β 剪枝但是好像并没有太多人采用。MCTS 的四个基本过程就是选择、扩展、模拟、回溯。一个典型的示意图如 Figure 1 所示。

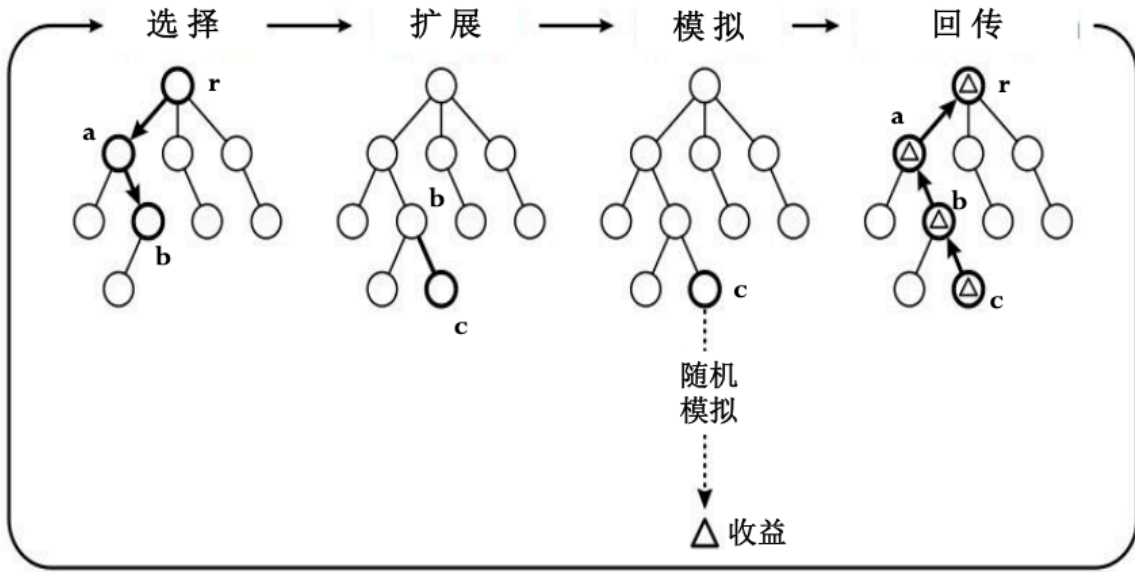


Figure 1: MCTS 示意图，摘自课件

1.1 选择

在选择过程中，最关键的步骤就是 UCB 算法。公式是

$$V = \frac{w_c}{v_c} + C \cdot \sqrt{\frac{2 \ln(v_p)}{v_c}}$$

其中 V 是最终 UCB 给中的评估结果， w_c 代表子节点下一步落子方在模拟中获胜字数， v_c 代表子节点的总访问次数也就是总模拟盘数， C 是平衡探索和利用的一个超参数， v_p 是父节点也就是正在选择者的访问次数。

在实践中我没有过多地尝试不同超参 C ，只是根据经验设置了 $C = 1$ 。

1.2 扩展

扩展步骤其实有几种不同的变体，不过一般最典型的方式就是持续选择直到遇到一个没有完全展开的节点，然后随机选择一个方向扩展出子节点，然后对这个新的子节点进行模拟。如果这是本节点最后一个可扩展的方向，那么就将这个节点标记为完全扩展，下次访问到这个节点时就进行只进行选择。

还有一些扩展方式，比如遇到未扩展的节点时立刻将所有子节点扩展，AlphaGoZero 采用的就是这样的方式。AlphaGoZero 这么做的理由是他可以一次性得到所有方向的先验概率，我不这么做的理由是如果我提前扩展全了但不进行模拟那么就浪费了很多内存，都进行模拟的话可能会浪费一些时间，可能会减少最终探索的深度。

1.3 模拟

模拟过程其实是整个算法的重中之重。模拟算法的效率和性能直接决定了整个算法的效率和性能。不难注意到，从效率角度来讲，整个算法性能热点就在模拟；从性能角度来讲，算法获得的所有价值评估信息都最终来自模拟过程，也就是直接影响了算法对局面的判断。

1.4 回溯

回溯过程在理论上没什么复杂的，只需要将模拟结果告知所有父级节点即可。但是工程上极容易出现的一个问题是由于树种节点奇数层和偶数层的落子方是不一样的，导致胜率算反。

2 树根移动

这是一个相对基础的想法，我发现不少人都想到了，因此不多说。核心就是在对方实际落子后移动树根而不是重建整棵树。我的初版实现中就自带了这个。

3 必胜推导

这个好像想到的人或者想到且实际做了的人就比较少了。我们可以考虑一下，在接近终局的时候，有些节点的胜负就是已经确定了的。那么我们就可以将这些节点标记为必胜。当然这里必胜包括我方必胜、对方必胜和平局。

那么有了这个必胜标记后，那么我们就可以进行必胜推导了。对于某个节点，可以分几个情况讨论：

1. 轮到 A 下了，存在一个子节点使得 A 必胜：那么此时显然这个节点也是 A 必胜的，因为 A 只需要落子到对应子节点即可保证必胜。
2. 轮到 A 下了，所有子节点都是 B 必胜：那么此时这个节点也是 B 必胜，因为 A 无论如何下哪都是 B 赢。
3. 轮到 A 下了，一部分节点是 B 必胜，剩下所有都是平局：那么此时这个点应该也是平局，因为 A 与其输会更希望逼平，但在实际中这种情况其实是几乎不存在的。

我的必胜推导就是在选择的过程中顺便进行的。有了这个必胜推导，其实在 MCTS 选择过程中就可以跳过所有必胜节点，因为我们假设双方都算出了这些是必胜的，那大家必然就尽可能不会让对方选到这些点，那么我们在选择的时候就可以直接跳过这些确定性局面了。这在搜索中有助于剪掉不少分支。

不过有个问题，如果跳过了必胜节点，那么父级节点由于永远选择不到这些对自己非常有利（或非常不利）的节点，那么他们的胜率可能会存在一些偏差，我们后续会讨论这个事情。

在实践中，我目前只发现了榜一 `lionjump` 明显也使用了这个策略，一个明显的特征是在终局前十到二十步进入“必杀时刻”，也就是秒响应，因为已经有必胜策略了，无需再做多余计算了。

4 压榨效率

其实不难发现, 这个算法的性能和效率有很强的相关性。效率越高, 在同样时间内模拟就能越多, 算法性能也就越好。这个效率其实包含了内存效率和时间效率两方面。从内存上粗略计算一下, 如果我们按照框架给的 `int[12][12]` 来存棋盘 (实际上这还是少算了 12 个指针), 那么一个棋盘就要占用 $12 \times 12 \times 4 = 576$ bytes, 那么对于 1G 内存, 最多只能存储 1864135 个棋盘, 这还不算节点需要存储的子节点指针等等一系列信息。不管从哪个角度来讲, 用 32bit 来存一个实际上只有 1.58bit 的信息是非常非常奢侈的。另外, 对于这个这么大的数组, 访存也需要不少时间。

4.1 内存优化——位图

对于棋盘, 我们可以采用一个更高效的存储方式, 用 2bit 来存储一个位置。一行最多可能需要 24bit, 可以存到一个 `int` 里。也就是说一个棋盘实际上可以用 `int[12]` 来存, 这一下子就高效了不少。

4.2 效率优化——位运算

可以看到, 标准的 `Judge.cpp` 里面的实现是相对低效的。我们既然有了位图, 那也可以采用位运算的方式来判断胜负, 这可比循环访存高效多了。这里首先要有个前提, 那就是为了方便运算, 双方的棋子分开存会更好。我具体的实现方式是低十六位通过 0 和 1 表示有无 1 号玩家的棋子, 高 16 位存 2 号玩家。

考虑如下一行

```
1  /*
2   * 0000ABCXDEF0
3   */
```

假设 X 是最后一次的落子, 那么从行角度来讲, 可能与赢有关的就是 A-F 六颗子。我们此时要看的其实就是 ABCXDEF 七颗子中间有无连四。

那么考虑通过移位将棋盘叠加起来

```
1  /*
2   * 0000ABCXDEF0
3   * 000ABCXDEF00
4   * 00ABCXDEF000
5   * 0ABCXDEF0000
6   */
```

如果有连四的话, 当且仅当中间这某一列应该全是 1, 也就是连四当且仅当这四行位与起来以后非零。¹ 用程序来讲, 就是这样

```
1  bool rowWin(const short row) {
2      const short row1 = row & (row << 1);
```

¹这个算法最初是由我的舍友谢濡键想到的

```

3 |     const short row2 = row1 & (row1 << 2);
4 |     return row2;
5 | }

```

对于列，其实原理和行是一样的，只不过我们需要再存一遍列取向的位图。虽然这要存两个 `int[12]`，但速度却快多了。对于斜向，依然类似，但如果再存我的内存有点不够，通常会导致搜索速度快过内存能支持的，因此平衡角度考虑我没有再存斜向。

5 智慧模拟

正如上文所说，模拟对于整体的性能至关重要。可以观察到，不少情况下模拟其实并不能很好评估准确率，一个更智能的模拟算法可以使得整体局面评估准确不少。受到学长启发²，在模拟中我加入了先验概率，也就是使得落子到中央的概率更高。不过就我实际观察而言这个提升效果好像并不明显。我认为有用的是另一个优化，就是考虑“冲三”和“跳三”的威胁。如下所示

```

1 | 001X1100
2 | 02111X00

```

示意中 X 位置是 2 号必走的。其实还有一种必堵是“活二”情况，不过这个相对来说比较复杂，我并没有采用。

5.1 冲三侦测

冲三由于是连续的，所以其实和连四判断是一样的。所有冲三必堵位置可以由下给出

```

1 | int hot(const int row) {
2 |     return row & (row << 1) & (row << 2);
3 | }

```

分析一下

```

1 | 00001110
2 | 00011100
3 | 00111000
4 | &-----&
5 | 00001000
6 | --hot---
7 | 000X100X

```

hot 位置和必堵位置 X 的关系如上所示。也就是判断 y 位置是否是必堵的方法如下所示

```

1 | constexpr int mask = 0b10001;
2 | const int off = (p - 1) * 16;

```

²<https://github.com/Guangxuan-Xiao/Connect4/blob/master/report.md>

```
3 | return charge_r[x] & (mask << (y + off) >> 1);
```

其中 `charge_r[x]` 就是 `hot` 返回的。

5.2 跳三侦测

对于跳三情况，也可以分析一下

```
1 | /*
2 |  *1011000
3 |  *0001011
4 |  *0010110
5 |  *0101100
6 |  *1011000
7 |  */
```

跳三的特征是对于四个连续棋子里面，头和尾有棋，且中间某一个点里有棋，此时另一个点就是必堵了，那么程序写出来应该就是这样子的

```
1 | int jump(const int row) {
2 |     const int match = row & (row << 3);
3 |     const int A = match & (row << 2);
4 |     const int B = match & (row << 1);
5 |     return (A >> 1) | (B >> 2);
6 | }
```

这个函数所有标 1 的位置应该就是需要堵的地方了。

5.3 综合使用

综合以上两点，我们可以写出完整的程序了。其实对于需要使用这么复杂的判断的只有模拟过程，那么我们此时其实完全就不用考虑内存的事情了，因为谁要模拟谁就来维护一个复杂的表格就可以了，模拟完用完就删就行了。完整的判断如下所示

```
1 | bool mustWin(const int x, const int y, const char p) const {
2 |     constexpr int mask = 0b10001;
3 |     const int off = (p - 1) * 16;
4 |     return charge_r[x] & (mask << (y + off) >> 1)
5 |         || charge_c[y] & (mask << (x + off) >> 1)
6 |         || charge_sl[x + y] & (mask << (y + off) >> 1)
7 |         || charge_sr[x - y + 11] & (mask << (y + off) >> 1)
8 |         || jump_r[x] & (1 << (y + off))
9 |         || jump_c[y] & (1 << (x + off))
10 |        || jump_sl[x + y] & (1 << (y + off))
11 |        || jump_sr[x - y + 11] & (1 << (y + off));
12 | }
```

r c sl sr 后缀分别代表 row, col, slanted left, slanted right。也就是横竖和左右斜。更新函数应当是

```
1 void update(const int x, const int y, const char p) {
2     set(x, y, p);
3     charge_r[x] = hot(rows[x]);
4     charge_c[y] = hot(cols[y]);
5     charge_sl[x + y] = hot(slanted_left[x + y]);
6     charge_sr[x - y + 11] = hot(slanted_right[x - y + 11]);
7     jump_r[x] = jump(rows[x]);
8     jump_c[y] = jump(cols[y]);
9     jump_sl[x + y] = jump(slanted_left[x + y]);
10    jump_sr[x - y + 11] = jump(slanted_right[x - y + 11]);
11 }
```

这里我们不需要更新完整棋盘, 因为新下的一颗子只会影响他所在的横竖斜。斜向存储与横竖是类似的, 只不过由于最多有 23 条对角线, 需要存成 `int[23]`。以上信息被我称作 `HeavyBoard`。我觉得如上的高效必堵搜索可以认为是我的算法的核心竞争力, 在应用这个策略以后, 我的 AI 在全局排行榜上排到了第 57 的位置。

6 激进剪枝

有了如上“必堵”的思路以后, 我们也可以把类似的思路运用在扩展的时候。对于某个节点而言, 他可能也存在一些必堵的情况。比如某个节点的棋盘是 1 号冲三, 轮到 2 下, 那么此时其实 2 是只有一个子节点是合理的。如果我们只采用必胜推导策略来剪枝, 那么最佳情况也需要扩展两层 24 个节点以后本节点才能推导出必走的子节点, 也就是把本节点 12 个可能方向都探索以后, 且 12 子节点每个都立马发现了 1 号必胜的孙节点, 此时通过必胜推导才能将本节点的 11 个节点标记为必胜从而使得本节点只选择必堵的方向。而实际上, 这一过程可能远远不止 24, 可能会有很多探索被浪费。

更激进的策略就是在新节点创建后将所有必堵情况和必赢情况筛出来, 如果有, 直接他的将其余子节点标记为 `nullptr`, 即不扩展。这里算出一个 `HeavyBoard` 以后恰好可以用到这个节点的模拟上。

总结一下, 激进剪枝发生在扩展出新节点时, 且对于新节点分三种情况:

1. 我方有冲三跳三: 标记必赢, 返回。
2. 对方有冲三跳三: 立即扩展必堵节点 (递归), 其余不扩展, 且本节点标记为已经完全扩展。
3. 对方有多个冲三跳三: 对方已经赢了, 标记对方必赢, 返回。

在应用了这个策略以后, 我的 AI 终于冲进了排行榜首页。实际上, 我能想到这个事情就是在和 `lionjump` 对局时发现他进入“必杀时刻”比我的 AI 早了大约三四个回合, 我从而意识到他搜索深度至少领先了我三四层, 从而促使我想到了我的必胜推导依然不够优。

7 内存管理

由于我主要使用的数据结构就是 `Node`, 所以为了更好地管理内存, 我自己开了一个内存池。为了卡 1G 内存, 我开了四百万个节点的内存池。从经验来讲, 一般中后期才会搜满。内存池有助

于更快地开和删节点,但其实从时间效率来讲我体感优化有限,更多来讲还是避免爆内存角度我觉得更有用。

8 效率分析

综合运用如上策略以后,我每次 2.5s 搜索时间内在我的电脑上大概可以搜索一百万到一百五十万个节点,在 saiblo 上每次大约六十万到一百二十万个节点。profile 一下我的程序可以得到如 Figure 2 所示的火焰图,可以看出效率热点还是在 simulate, 以及和 HeavyBoard 有关的事项。

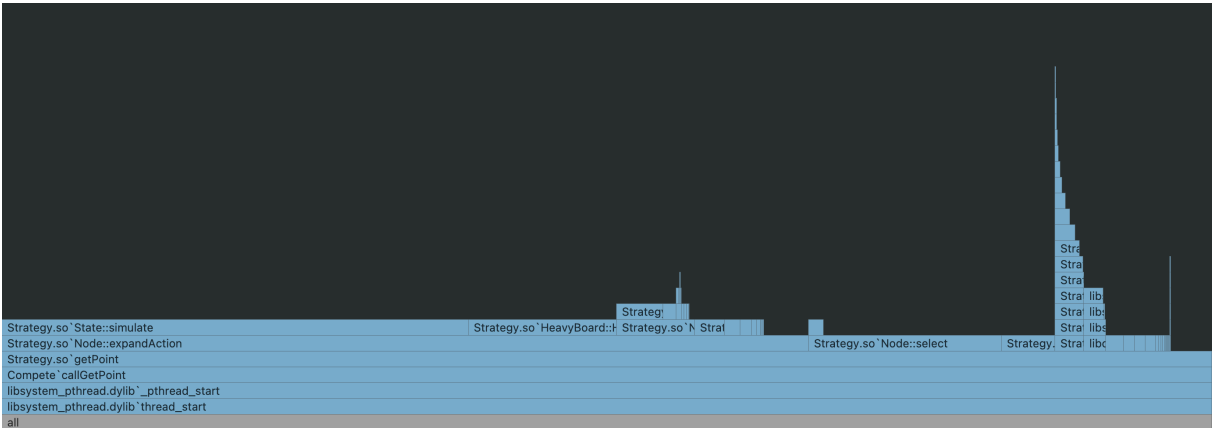


Figure 2: 使用 12*12 棋盘运行一分钟后的火焰图

9 实际结果

综合运用如上所述策略以后,在 saiblo 的批量测试结果如 Figure 3 所示。



Figure 3: saiblo 上五十个节点的批量测试结果

不过当然,这个是有运气成分在的,有时候我就没法把这个打满。这个我感觉和 saiblo 测试时候的负载有很大关系。

我将我的 AI 派遣到了排行榜上,在我写下报告时目前如 Figure 4 所示位列于全局第 16, 2024IAI 前缀第一。

游戏 > 四子棋 > 排行榜

四子棋的排行榜



































名次	积分	选手	AI
#1	1515pts.	 lionjump 欢迎香香小猪猪公主莅临参观	Genesis ? 版本30  快速人机对局
#2	1354pts.	 g4197 清华存储组欢迎您的加入 (storage.cs.tsinghua.edu.cn)	Catfish ? 版本3  快速人机对局
#3	1354pts.	 <2021IAI_2019011227>	connect3 C 版本2  快速人机对局
#4	1352pts.	 cheng	, C 版本1  快速人机对局
#5	1351pts.	 <2021IAI_2020040074> 不会下棋，连AI都下不过...	BetaCat C 版本17  快速人机对局
#6	1347pts.	 omegafantasy 你的梦中，我已不在	Ultima C 版本5  快速人机对局
#7	1345pts.	 <2021IAI_2019011264> 人就是江湖	*** C 版本15  快速人机对局
#8	1344pts.	 <2021IAI_2019013292>	就这。 C 版本27  快速人机对局
#9	1342pts.	 <2021IAI_2019011277> MOSFET	雷 C 版本1  快速人机对局
#10	1342pts.	 <2021IAI_2019013285> 一直漂亮的小花帽	try C 版本24  快速人机对局
#11	1341pts.	 CurtisSun	connect4 C 版本13  快速人机对局
#12	1340pts.	 <2023IAI_2022040114> dottie bot.	~ci-cpp~ ? 版本88  快速人机对局
#13	1340pts.	 <2021IAI_2019013287> 你的随从从演成了一个四连（	就这？？？ C 版本18  快速人机对局
#14	1339pts.	 <2023IAI_2021010723> 关门放狗	puppy ? 版本4  快速人机对局
#15	1339pts.	 LastMC	Reimu C 版本4  快速人机对局
#16	1336pts.	 <2024IAI_2022010776>	~ci-cpp~ ? 版本32  快速人机对局
#17	1334pts.	 <2021IAI_2019011342>	Example C 版本1  快速人机对局

Figure 4: 5 月 28 日中午四子棋全局排行榜截图

10 结语

本次实验总体来说感觉卷起来十分快乐。除了上述提到的主要优化策略外，还有不少局部的小技巧，比如缓存变量和分支优化之类的，这方面我也下了不少功夫，不过没有什么理论和原理上的创新就不提了。另外其实我的算法还有很多地方可以优化，不过实在是懒了。列举几个改进方向：

- Section 6 所讲的第一种情况好像在一些假设下是不存在的，可以优化掉，不过如果优化掉了在临近终局会有很多特判，细节很多。
- 非叶节点棋盘其实是没必要存的。
- 我现在棋盘由于遗留问题依然存了横竖双向，但由于我有了在扩展和模拟时动态开出的 `HeavyBoard`，横竖双向存储似乎是没必要的。
- 我的 AI 在搜到 root 必输的时候会开摆，但是很多时候有可能对手并没有搜到他已经赢了这件事，其实应该再坚持一下的。
- 很多更小的细节我都选择相信 -02 的 GVN 和 GCM 了，说不定还有不少小地方小细节可以优化。

以上只是目前能想到的一些，其实都不难做，只是实在懒得继续卷了。

说到这里,我还是很好奇 `lionjump` 究竟用了什么策略能遥遥领先。从榜上可以看出在他 1500 多分以后剩下的人分数很紧,可以说应该是没有什么很本质的创新。根据 ELo 公式

$$E_A = \frac{1}{1 + 10^{\frac{R_A - R_B}{400}}}$$

20 分以内可以说是互有胜负,但差 200 分的话基本上就是碾压了。

另外我在与我的 AI 对战时,可以说完全被碾压,基本五手以内我就会下出问题手然后胜率暴跌。我觉得我的 Elo 分数可能也就 1000 左右。