人工智能导论
四子棋作业报告

张晨，计71，2017011307

1 算法思路

我采用了“蒙特卡洛搜索”和“信心上限树算法”，基本按老师课件中的伪代码实现。

在内存中维护一棵UCT，每轮开始前依据自己和对手的上一个决策对树进行换根，之后依据经验公式进行棋局的随机模拟至分出胜负，将胜（1分），平（0分），败（-1分）作为收益回传至根，最后选取模拟次数最大的点。

2 优化

2.1 内存池

算法运行过程中，会出现大量的新建节点与删除节点操作，如果每次都调用new和delete运行速度会十分不理想，因此我选择在开始时建一个大数组作为内存池，并使用一个队列维护可以使用的节点。新建节点时，直接从队列中取一个节点即可。删除节点则略有复杂。以图1为例，1号节点为根节点，4号节点为上一轮的决策。，现在要将树根换成4号节点，则1号节点，2/3/5号节点及其子树都需要被删除。

删除子树可以采用递归dfs的方法，但由于涉及大量的堆栈操作，效率十分低，会成为整个程序的瓶颈。删除子树也可以用对子树进行bfs的方法进行实现，不过bfs队列与内存池队列极为相似，单独实现总显得有一些累赘。

我选择了将一号节点的孩子信息清空，把它放入内存池队列。之后将2/3/5号点（不包括其子节点）放入内存池队列。新建节点时，从内存池中取出一个节点后，要将它的孩子加入内存池。

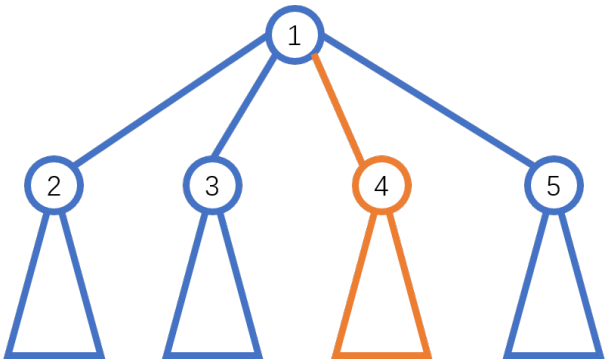

图 1: 一棵UCT

2.2 剪枝

2.2.1 一些定义

首先引入Vector Allis的四子棋必胜策略论文中的threat概念。

定义 1 如果一个点被A的对手B占领后，B会连起4个子，则这个位置是A的*threat*

此外，为叙述方便，还需定义urgent threat。

定义 2 如果这一手棋可以下到某个*threat*的位置，则称这个*threat*为*urgent threat*。

2.2.2 落子策略

按以下顺序执行落子决策

1. 如果对手有urgent threat，则下在那里即可获得胜利。
2. 如果自己有urgent threat，则必须下在那里，否则下一步对手可以下在那里以获胜。
3. 按UCT进行落子，但不能下在对手的threat的下面一格。

2.2.3 必胜必败态的判定

如果在玩家落子前，他的对手有urgent threat，则玩家可以将子下在那里直接获得胜利。下面的判据假设对手没有urgent threat。

1. 如果一个玩家拥有两个urgent threat，则无论他将子下在哪个位置，对手都可以将子下在另一个位置从而获得胜利。因此，如果下完一手棋后对手出现了两个urgent threat，且自己没有urgent threat，则玩家必胜。如果玩家落子前，自己有两个urgent threat，且对手没有urgent threat，则玩家必败。
2. 如果玩家有一个urgent threat，且这个位置上面也是自己的threat，则
   (a) 若玩家不把子下在urgent threat上，则对手可以将子下在那里以获得胜利。
   (b) 若玩家把子下在urgent threat上，则这个位置上面的threat成了新的urgent threat，对手可以将子下在这个新的urgent threat上以获得胜利。

   因此，出现这种情况后，玩家必败。若玩家落子后，对手出现这种情况，则玩家必胜。
3. 玩家A落子后，对手B无法找到上面不是自己的threat的落子点，则玩家A必胜。

在蒙特卡洛搜索的过程中，倘若可以判定某个状态是必胜/必败态，则直接返回这个状态的收益，而不需要模拟至游戏真正结束。直接返回可以节约运算时间，但更重要的是，较蒙特卡洛的随机落子，直接返回时对这个局面的更为真实的评价。

2.3 剪枝过程的加速

当且仅当玩家把子下在了对手的threat的位置上，玩家可以获得胜利；找到threat后，可以直接通过它所在列的子的个数判定它是否是urgent threat；出现新的urgent threat后，判定它是否会导致必胜/必败态的运算量也很小，因此剪枝过程中最大的瓶颈在于找到threat的位置，找threat的过程占到了我最初版的程序的2/3以上的时间。为提升每轮的迭代次数，我使用状态压缩的方法对找threat的过程进行了优化。

每个位置有空位、不可落子点、有A的子、有B的子四种情况，可分别用两位二进制数00,01,10,11进行编码。每一行的位置数m满足$m \leq 12$，因此可以使用此种编码方式编为一个$2m$位二进制数，并在实现时用一个int表示。

为了实现方便，我在棋盘的最左边、最右边分别额外加一列全是不可落子点的位置，将每一行编码为一个$2m+4$位二进制数。这样做的好处是，若同一行的连续四个位置中存在超出原棋盘范围的点，则这四个位置中一定至少有一个不可落子点，因此不需要特判出界的情况。

	A	B	B		X	B	A		A		
01	10	11	11	00	01	11	10	00	10	00	01

图 2: 编码示例

落子后，可通过右移及按位与操作取出落子点及其周围的连续四个位置的编码，若为"00111111"、"11001111"、"11110011"、"11111100"中的一个，则出现了一个B的threat，若为"00101010"、"10001010"、"10100010"、"10101000"中的一个，则出现了一个A的threat。落子点可以位于取出的位置中的任一位置，故总共需要取四次数。

斜向的操作与之类似，两种斜向可分别视为把$x+y$相同的归为一行及把$x-y$相同的归为一行，通过在棋盘外部加不可落子点来限定落子的范围。

竖向则略有不同。四个在竖向上连续的棋子，必定是按照从低到高的顺序落下的，因此如果落子后产生threat，则threat一定是落子点之上的一个点。利用这个性质，仅需要做一次取数和一次判定。

通过状态压缩，我把局面判定花费的时间缩小到了原来的1/3，每轮迭代次数变为原来的两倍。

2.4 其他常数优化

我使用visual studio中的性能分析工具统计了每一行代码的执行时间，并重写了一些编译器无法优化的热点代码，使得我的模拟次数有小幅提升。在我的笔记本电脑上每轮可进行约5e5次模拟。

3 实验结果

与2.dll至100.dll分别进行了5轮对战，每轮对战包含一次先手与一次后手，总胜率97%。下表是胜率的统计，表中未列出的均取得全胜。

编号	100	96	94	92	90	88	86	84	78	42
胜率	80%	80%	80%	90%	80%	90%	90%	90%	80%	90%

4 其他尝试

我阅读了Vector Allis的四子棋必胜策略论文，但是发觉其理论建立在棋盘小而特殊的规模上，对优化蒙特卡洛算法的作用有限，虽然这篇论文对我形成最终的算法几乎没有影响，但读这篇论文本身就是一件有趣的事情。

此外，我还尝试了通过一些判据判定“很有可能赢”之后就返回，但实战效果变差，我认为这是因为这些判据不完全准确，从而自己无法搜出一些隐藏较深的赢法，且较强的测例能够抓住我因把失败错判成胜利而出现的失误。

5 总结

完成此次大作业的过程中，我更加深入理解了蒙特卡洛搜索与信心上限树算法，领略到了蒙特卡洛的威力。