报告四

09016428　宋子星

1. 回顾

在上一篇报告里，我着重分析了 NegaMax 和 MiniMax 算法的性能差异，得出了 NegaMax 算法在空间复杂度上更胜一筹的结论。同时，引出了启发式搜索函数，将所以节点按照评价函数的大小进行升序或者降序的排列，从而进一步提高剪枝效率。本次报告，我将进一步优化

2. 启发式搜索算法的改进

2.1 启发式搜索算法

alpha–beta 剪枝有严重依赖于每一层节点扫描的顺序，因为前面节点生成的 alpha 直接决定了后面节点剪枝的多少。如果我们针对每一层生成的节点事先排好序，那么程序的速度将极大地提升。然而，对生成的节点进行绝对准确地排序是不可能的，因为需要巨大的运算量（等同于再进行一次 alphabeta 剪枝的搜索），所以我们只能对节点进行粗略地排好序。我之前已经在先前的报告中讨论了如何对棋盘上的位置进行评分，我们可以利用这个评分，对生成的可能的点进行排序。排序开销小，然而对整体的搜索速度带来巨大的提升。加入此优化之后，五子棋可以轻松做到 4 层搜索。

2.2启发式搜索算法的改进

在进行了 AlphaBeta 剪枝和启发式搜索之后，搜索依然只能进行到 4 层左右。6 层还非常慢。

在启发式式搜索的优化中，已经对下一步可能出现的位置粗略地排好了序，极大地提高了 AlphaBeta 剪枝的效率。但是由于每层产生的点太多，导致计算机仍然需要搜索数量巨大的节点。

事实上，只要对位置进行评分的函数足够好，我们就可以保证最优解基本出现在排好序的节点的前 15 个之中。所以我们只需要扫描产生的排好序的点的前十个点就可以保证比较好的效果了。这个不起眼的优化提升非常巨大，有了这个小优化之后，在五子棋 AI 的搜索层数可以达到 6 层，棋力迅猛提升。

2.3代码实现

//改进后的启发式搜索

```
1.  MoveValue Gomoku::negaMax(int depth, double alpha, double beta, int color)
2.  {
3.   if (depth == 0 /*|| isGameOver()*/) {
4.            double res = EvaluateState();
5.            return MoveValue(res);
6.   }
7.   MoveValue bestMove(-DBL_MAX);
8.
9.   //vector<MOVE>LegalMoves = GenerateLegalMoves();
10.
11.  vector<MOVE>LegalMoves = GenerateSortedMoves(color);
12.  int count = 0;
13.  vector<MOVE>::iterator movesIterator = LegalMoves.begin();
```

```
14.  while (movesIterator != LegalMoves.end()) {
15.          MOVE currentMove = *movesIterator;
16.          MOVE newMove = currentMove;
17.          movesIterator++;
18.          applyMove(currentMove);
19.          double val = -(negaMax(depth - 1, -beta,-alpha,color%2+1).returnValue);
20.          Grid[currentMove.x][currentMove.y] = 0;
21.          CurrentColor = CurrentColor % 2 + 1;
22.          //UnmakeMove();
23.          if (val > bestMove.returnValue) { bestMove.returnValue = val; bestMove.returnMove
     = currentMove; }
24.          if (val > alpha) { alpha = val; }
25.          if (alpha >= beta) break;
26.          if (count++ >= 15) break;
27.
28.  }
29.  return bestMove;
30. }
```

3. 算杀

3.1 alpha–beta 剪枝的缺陷

alpha–beta 剪枝最多只能搜索有限层，目前优化之后我可以搜索到 6 层。即电脑 3 步，人 3 步后的情况。但是还是不够的。看不到有限步数之后的棋。当遇到一些高手就可以看出棋力的不足。

3.2 算杀方法的引入

所谓算杀就是计算出杀棋，杀棋就是指一方通过连续的活三和冲四进行进攻，一直到赢的一种走法。

很显然，同样的深度，算杀要比前面讲的搜索效率高很多。因为算杀的情况下，每个节点只计算活三和冲四的子节点，也就是计算那些己方能够迫使敌方在一步之内作出防守或者确定性的一步。所以同样是 1 秒钟的时间，搜索只能进行 4 层，而算杀我经过尝试可以进行到 10 层左右。

而且很容易想到，算杀其实也是一种极大极小值搜索，具体的策略是这样的：

- MAX 层，只搜索己方的活三和冲四节点，只要有一个子节点的能赢即可
- MIN 层，搜索所有的己方和对面的活三和冲四节点（进攻也是防守），只要有一个子节点能保持不败即可。

3.3 算杀算法的集成

有了算杀模块之后，我们可以直接对当前棋局进行算杀，但是显然更好的做法是在搜索中进行算杀，通过 N 层搜索结合 M 层算杀，我们可以最多搜索到 N+M 层。

3. 4 代码实现

//找到杀棋

```
vector<MOVE> Gomoku::FindKillMove(int color)
{
	vector<MOVE>res; int i, j;
	for(i=0;i<15;i++)
		for (j = 0; j < 15; j++) {
			if (Grid[i][j] == 0)
			{
				Grid[i][j] = color;
				if (WinFive(i, j, color)) { res.push_back(MOVE(i, j)); Grid[i][j] = 0; return res; }
				if (AliveFour(i, j, color)) { res.push_back(MOVE(i, j)); Grid[i][j] = 0; return res; }
				if (DeadFourA(i, j, color) || DeadFourB(i, j, color) || DeadFourC(i, j, color))
{ res.push_back(MOVE(i, j)); }
				if (AliveThree(i, j, color)) { res.push_back(MOVE(i, j)); }
				Grid[i][j] = 0;
			}
		}
	return res;
}
```

//找出防守策略

```
vector<MOVE> Gomoku::FindSafeMove(int color)
{
	vector<MOVE>res; int i, j;
	vector<MOVE>SelfFive;
	vector<MOVE>RivalFive;
	vector<MOVE>SelfFour;
	vector<MOVE>RivalFour;
	vector<MOVE>SelfDeadFour;
	for (i = 0; i<15; i++)
		for (j = 0; j < 15; j++) {
			if (Grid[i][j] == 0)
			{
				Grid[i][j] = color;
				if (WinFive(i, j, color)) { SelfFive.push_back(MOVE(i, j)); Grid[i][j] = 0; return
SelfFive; }//若己方有直接赢的可能，显然直接赢
				Grid[i][j] = color % 2 + 1;
				if (WinFive(i, j, color % 2 + 1)) { RivalFive.push_back(MOVE(i, j)); Grid[i][j] = 0;  }//若
对方下一步就赢，己方这一步赶紧堵上
				Grid[i][j] = color;
				if (AliveFour(i, j, color)) { SelfFour.push_back(MOVE(i, j)); Grid[i][j] = 0;
res.push_back(MOVE(i, j));}//对方不会一步之内就赢，而且己方有活四
				Grid[i][j] = color % 2 + 1;
```

```
                    if (AliveFour(i, j, color % 2 + 1)) { RivalFour.push_back(MOVE(i, j)); Grid[i][j] = 0;
res.push_back(MOVE(i, j));}//对方不会一步之内就赢，而且己方无活四
                    Grid[i][j] = color;
                    if (DeadFourA(i, j, color) || DeadFourB(i, j, color) || DeadFourC(i, j, color))
{ SelfDeadFour.push_back(MOVE(i, j)); Grid[i][j] = 0; res.push_back(MOVE(i, j));
                    }
                    Grid[i][j] = 0;
                }

            }
                    if (RivalFive.size() > 0) return RivalFive;//若对方下一步就赢，己方这一步赶紧堵上
                    else if (SelfFour.size() > 0) return SelfFour;//若对方不会一步之内就赢，而且己方有
活四
                    else if (RivalFour.size() > 0) { if (SelfDeadFour.size() > 0) return SelfDeadFour; else
return RivalFour; }//对方不会一步之内就赢，而且己方无活四但有死四，否则还是堵起来吧

                //    else return res;
                    else return vector<MOVE>();
}
```

4. 置换表

4.1 原理介绍

经过查阅相关资料，我了解到，置换表可以用来过去已经搜索过的棋盘状态。

主置换表是一个散列数组，每个散列项看上去像这样：

```
#define hashfEXACT 0

#define hashfALPHA 1

#define hashfBETA 2

typedef struct tagHASHE {

   U64 key;

   int depth;

   int flags;
```

```
    int value;
    MOVE best;
} HASHE;
```

这个散列数组是以“Zobrist 键值”为指标的。为求得局面的键值，除以散列表的项数得到余数，这个散列项就代表该局面。由于很多局面都有可能跟散列表中同一项作用，因此散列项需要包含一个校验值，它可以用来确认该项就是要找的。通常校验值是一个 64 位的数，也就是上面那个例子的第一个域。

从搜索中得到结果后，要保存到散列表中。如果打算用散列表来避免重复工作，那么重要的是记住搜索有多深。如果在一个结点上搜索了 3 层，后来又打算做 10 层搜索，就不能认为散列项里的信息是准确的。因此子树的搜索深度也要记录。

在 Alpha–Beta 搜索中，很少能得到搜索结点的准确值。Alpha 和 Beta 的存在有助你裁剪掉没有用的子树，但是用 Alpha–Beta 有个小的缺点，通常不会知道一个结点到底有多坏或者有多好，只是知道它足够坏或足够好，从而不需要浪费更多的时间。

当然，这就引发了一个问题，散列项里到底要保存什么值，并且当要获取它时怎样来做。答案是储存一个值，另加一个标志来说明这个值是什么含义。

4.2 代码实现（部分）

```
int AlphaBeta(int depth, int alpha, int beta) {
   int hashf = hashfALPHA;
   if ((val = ProbeHash(depth, alpha, beta)) != valUNKNOWN) {
      // 【valUNKNOWN必须小于-INFINITY或大于INFINITY，否则会跟评价值混淆。】
```

```
        return val;
    }
    if (depth == 0) {
        val = Evaluate();
        RecordHash(depth, val, hashfEXACT);
        return val;
    }
    GenerateLegalMoves();
    while (MovesLeft()) {
        MakeNextMove();
        val = -AlphaBeta(depth - 1, -beta, -alpha);
        UnmakeMove();
        if (val >= beta) {
            RecordHash(depth, beta, hashfBETA);
            return beta;
        }
        if (val > alpha) {
            hashf = hashfEXACT;
            alpha = val;
        }
    }
    RecordHash(depth, alpha, hashf);
    return alpha;
}

int ProbeHash(int depth, int alpha, int beta) {
    HASHE *phashe = &hash_table[ZobristKey() % TableSize()];
    if (phashe->key == ZobristKey()) {
        if (phashe->depth >= depth) {
            if (phashe->flags == hashfEXACT) {
                return phashe->val;
            }
            if ((phashe->flags == hashfALPHA) && (phashe->val <= alpha)) {
                return alpha;
            }
            if ((phashe->flags == hashfBETA) && (phashe->val >= beta)) {
                return beta;
            }
        }
        RememberBestMove();
    }
    return valUNKNOWN;
}
```

```
void RecordHash(int depth, int val, int hashf) {
    HASHE *phashe = &hash_table[ZobristKey() % TableSize()];
    phashe->key = ZobristKey();
    phashe->best = BestMove();
    phashe->val = val;
    phashe->hashf = hashf;
    phashe->depth = depth;
}
```

5. 棋力效果

经过多重的改进与测试，无论棋力还是思考速度都有了较大的提升，下面是测试最终版本程序和之前的最初我的版本进行对战的结果，可见黑子（最终）在回合不多的步数之内就可以战胜白子（最初）。