拼音输入法实验报告

高敬越 计 92 2019011230

May 2, 2021

目录

1 策略的思路与实现

1.1 基本框架

1. 选用了信心上限树 (UCT) 算法来进行模拟；
2. 采用面向对象的设计方式，建立了作为 ai 引擎的 uct 树类 engine 和 uct 树的结点类 node；
3. 从当前局面开始拓展、模拟，在计算 2.6s 后，按信心上限选择下的位置；
4. 对必赢以及不下就输的点进行了判断；
5. 选择了以一定系数 $\gamma < 1$ 不断衰减至 1 的 reward 值来代替信心上限中的胜利次数，根据下子后的局面进行一定的引导。

1.2 Engine 类

Engine 类实现了函数 UCTS()、getPoint()，存储了棋盘的大小和根结点 root 的指针。为了代码实现的简洁，将主函数中的 treePolicy 放在 UCT 类中实现，模拟拓展的 defaultPolicy 和传导函数 backup() 放到了结点类。

```
class Engine {
    int m, n;
    Node* root;

public:
    Engine();
    Engine(const int m, const int n, int player,
           int **board, const int *top, Point last);
    void UCTS();
```

```
    ~Engine();
    Point getPoint();
    Node* treePolicy();
};
```

1. UCTS() 拓展过程的函数，算法按讲义伪代码；算法结束后按信心上限选择 bestChild。

```
void Engine::UCTS() {
    clock_t start = clock();
    while (clock()-start<= TIME_LIMIT * CLOCKS_PER_SEC) {
        Node *node = treePolicy();
        int delta = node->defaultPolicy();
        node->backup(delta);
    }
}
```

2. treePolicy()

```
Node* Engine::treePolicy() {
    Node *now = root;//从当前的根结点开始
    while (now && !now->terminal()) {//如果未到终结点就继续扩展
        Node *tmp = now->expand();
        if (tmp)
            return tmp;
        else
            now = now->bestChild();//扩展完毕就选取最好的子节点
    }
    return now;
}
```

1.3 Node 类

在 Node 类中定义了信心上限树所需要的信息，包括当前结点对应的玩家编号、棋盘大小、棋盘当前局面等。同时实现了函数，例如拓展函数 expand()，最佳子节点 bestChild()，传播 reward 函数 backup(int)，特判当前局面的 immediateWin()、nearlyLose() 等。

```
struct Node {
    enum Status {NOT_TERMINATED, WON, TIE, LOST, UNKOWN};

    int m, n;
    int R, N;
    Status status;
    Point pos;
    int _player;//结点对应
    int nxt_top_index;

```

```
    static int invalid_x, invalid_y;

    double c = 0.7;//信心的参数
    double gamma = 0.1;//reward的参数

    int board[maxn][maxn];//当前结点对应局面的board数组
    int top[maxn];//当前结点对应局面的top数组
    Node* par;
    std::vector<Node*> children;

    Node(Point pos, int player, int **board
        , const int *top, int m, int n);
    Node(Point pos, int player, int board[][maxn]
        , const int *top, int m, int n);
    ~Node();

    Node* expand();
    Node* bestChild(int last=1);
    int defaultPolicy();
    bool terminal();

    void backup(int);

    int nearlyLose();
    int immediateWin();
};
```

1. expand() 函数从棋盘最左侧开始，不断判断当前列是否可以下棋，如果可以就将其扩展。在最初为了促进 ai 多向已经下棋的部分下子，将 expand() 函数开始拓展的位置改到了对方下棋的位置，但发现效果不佳。于是在参考资料后，将这一部分加到了选择落子的函数 bestChild() 中。

```
Node* Node::expand() {
    while (nxt_top_index<n && top[nxt_top_index]==0)
        ++nxt_top_index;
    if (nxt_top_index>=n) {
        return nullptr;
    }
    Point new_pos(top[nxt_top_index]-1, nxt_top_index);
    Node *node = new Node(new_pos, rival(_player),board,top,m,n);
    ++nxt_top_index;
    node->par = this;
    children.push_back(node);
    return node;
```

```
13 }
```

2. bestChild() 函数选择当前结点的子节点中信心最高的结点。在选择时，首先判断有无已经胜利的结点；然后再对信心进行排序，其中若出现不下就要输的结点，便选择这一结点。在进行拓展时，信心函数选择为，$c\sqrt{\frac{2\log(N_{node})}{N_{total}}}+\frac{\sum reward}{N_{total}}+0.1\cdot\frac{N_{rival}}{N_{total}}$，与常规相比增加一个 reward，用来鼓励 AI 在开始向对方棋子个数 N_{rival} 多的地方下棋；在进入中后期时这个鼓励也会较小，不会过多影响棋局。
3. defaultPolicy() 函数是从当前 this 这个结点开始模拟棋局，并在结束后返回 reward 值。在计算胜场时，用 $\sum\gamma^r reward$ 来代替每次加 1（衰减到最低为 1），用来为棋局增加几步的模拟出的未来信息，指导棋局不要向过于坏的方向下棋，而要向真正利于自己的方向。例如可以避免对方已连接 3 个，ai 下的棋刚好为对方第四个垫了位置，使得对方立刻就赢了的情况。
4. backup() 函数将 defaultPolicy() 返回的 reward 向上传播，传播到根结点。在传播时不断交换正负号，并对绝对值大于 1 的值进行衰减。

2 统计结果

在 saiblo 网站中进行了多次批量测试，测试 id 分别为 #11058,#10859，#10348，分别为

批量测试 #11058

批量测试 #10348

平均每百场胜场 95.3，负场 4.7，平均胜率 95.3%。由于本机 MacBook 性能原因，未进行完整的本机测试。

3 总结和改进

1. 在实现了重力四子棋 AI 后，我对蒙特卡洛方法、信心上限树搜索方法有了更深刻的认识和理解，也提升了我对于 ai，特别是棋类 ai 的认识；
2. 在实现过程中，最初设计了 engine,uct,treenode 三个类来进行实现，发现实现得不够自然、简洁，并经常会遇到析构错误等问题；经过查阅资料和相关代码，融合了 engine 和 uct 两类，设计实现了现在的模型，提高了我的编程能力和对面向对象的理解；
3. 在最初实现后，发现胜率并不高。在经过学习相关代码资料和阅读科协推送后，增加了对特殊情况、信心函数、reward 值等改进，有效提高了胜率；
4. 感谢老师和助教的讲解。