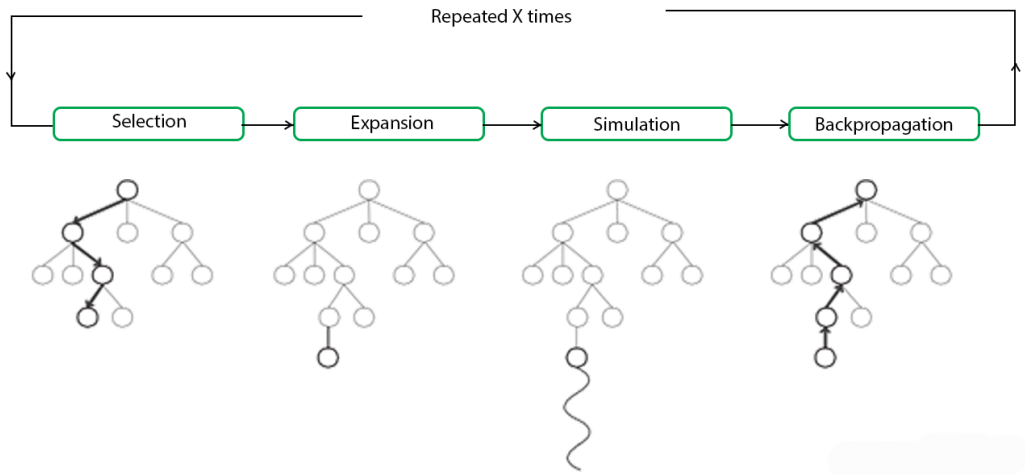
電腦對局理論 Project3 Report
0716120 林怡安

1. Introduction
   這次作業主要在做 mcts，要實作出 mcts 並在 NOGO 上來呈現最終結果。

2. Implementation
   MCTS 為蒙地卡羅搜尋法的簡稱(Monte Carlo Tree Search)，原理如下圖。




(1) Selection
我們要開始搜尋最好的路徑直到 leaf，而最好的路徑則使用 UCB 的公式，讓我們可以不會一直只找同一條路。

$$\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$$

Q(v')為此 node 的勝場，N(v')為此 node 被拜訪過幾次，而 N(v)為此 node 的 parent 被拜訪過的次數。

(2) Expansion
當找到我們最好的 leaf 時，接下來要做 expansion 的部分。在此次作業我是一次性的將所有能夠下的位置一次都 expand 完，若發現沒有新增任何 node，即將此 leaf 的 is_terminal 設為 true，就可以免去之後若再次找到此 node 而又繼續做 expand 的動作。

(3) Simulation
Expand 完之後，我們要來使用新增的這個 node 做一次 game 的模擬。由

於我在 expansion 時是把所有可以下的 node 一起 expand，所以我是使用隨機抽選一個 node 來做 simulation。Simulation 的方法是 random 下棋直到有一方輸。

(4) Backpropagation

Simulation 完後就會得到到底是輸還是贏，若贏的話即 1，輸的話為 0，而 backpropagation 就是要將這個值更新給上面的節點。我們需要更新這個節點的 visit_count 以及勝利的次數，才能在 Selection 的 UCB 中計算出適合的 node。

3. Code

```
struct node{
        node *parent=nullptr;
        std::vector<node*> childrens;
        int wi=0;
        int si=0;
        bool isleaf=true;
        board state;
        board::piece_type self;
        action::place move_placed;
        bool is_terminal=false;
        node(node *p, board s, board::piece_type me){
            parent=p;
            state=s;
            self=me;
        }

};
```

首先創建最基本的元素，node。

(1) Selection

```
while(!current->isleaf){
    float best=0;
    int best_index=0;
    for(int i=0;i<current->childrens.size();i++){
        if(best<best_child(current->childrens[i])){
            best=best_child(current->childrens[i]);
            best_index=i;
        }
    }
    current=current->childrens[best_index];
}
```

```
if(who==child->self){
    if(child->si==0){
        return double(1000000);
    }
    else{
        return double(child->wi)/double(child->si)+exploration_c*sqrt(log(double(child->parent->si))/double(child->si));
    }
}
else{
    if(child->si==0){
        return double(1000000);
    }
    else{
        return (1.0-double(child->wi)/double(child->si))+exploration_c*sqrt(log(double(child->parent->si))/double(child->si));
    }
}
```

這邊 selection 我們先用迴圈的方式找到最佳的 node。至於 UCB 的算法，因為我們的棋子有兩種(黑及白)，因為我們都是一起更新勝率的，所

以若是對方的 node 勝利的次數也會被加上去，但由於這對於對方做選擇時及不合理(=挑讓我方能贏的最佳解)，所以只要不是我方的勝率都須先 1-勝率，換成對方的最佳解。

(2) Expansion

```
node* expand(node *current){
    if(current->is_terminal){
        return current;
    }
    else{
        board::piece_type op=my_opponent(current->self);
        if(op==who){
            for(int i=0;i<space.size();i++){
                board temp=current->state;
                board t=current->state;
                action::place placement(i,op);
                if(placement.apply(t)==board::legal){
                    placement.apply(temp);
                    node* newnode=new node(current,temp,op);
                    newnode->move_placed=placement;
                    current->childrens.push_back(newnode);
                }
            }
        }
```

這邊貼上最主要的部份。當輪到我方要 expand 時，就從我方能下的 space 中挑選所有能夠走的步數(board::legal)並將所有步數都 expand 到現在的 node 當中。輪到對方時也是做同樣的事情，只是不同的地方是從對方的 space 中挑選。

(3) Simulation

```
while(1){
    int terminate=1;
    if(myop==who){
        current_space=my_space;
        next=opponent;
    }
    else{
        current_space=op_space;
        next=who;
    }

    for (const action::place& move : *(current_space)) {
        board t=temp;
        if (move.apply(t) == board::legal){
            move.apply(temp);
            terminate = 0;
            break;
        }
    }
```

這邊為最重要的部分，我們需要先去判斷現在下的到底是何方，然後再用那一方能走的步數做 random 挑選。

```
    if(terminate==1){
        if(myop==who){
            score=0;
        }
        else{
            score=1;
        }
        break;
    }
    myop=next;
}
return score;
```

當有一方無法下時，我們要來判斷這場到底是誰勝利。若遊戲還可以繼續的話，也要記得換對方下。

(4) Backpropagation

```
void backpropogation(node *current, int score){
    while(current!=nullptr){
        current->wi+=score;
        current->si++;
        current=current->parent;
    }
```

Backpropagation 的部分分常簡單，只要一直更新上面節點的資料即可。

(5) Tack_action

```
while(1){
    simulation_count++;
    node *best_leaf=selection(root);
    node *new_leaf=expand(best_leaf);
    int score=rollout(new_leaf, &my_space, &opponent_space);
    backpropogation(new_leaf,score);

    if(simulation_count%500==0){
        end=clock();
        if((end-start)/CLOCKS_PER_SEC>0.999999){
            break;
        }
    }

}
```

這邊放上每走一步之前就做 mcts 模擬的 code，我是使用 clock 來計算是否會超時，以之前下的時候看到的資料，大概每一方最多會下到 33-34 步，所以將時限(40sec/35)就是限制 mcts 每次能跑的時間。

```
int bestcount=-1;

for(int i=0;i<root->childrens.size();i++){
    if(root->childrens[i]->si>bestcount){
        bestcount=root->childrens[i]->si;
        best_move=root->childrens[i]->move_placed;
    }
}


deletenode(root);
return best_move;
```

做完模擬後要決定到底要下哪一步，選擇 mcts 模擬出來走過最多次的 children。

後面要記得刪除 mcts tree，不然會因為記憶體空間問體導致程式 crash。

4. Problem Encountered

這次作業在實作時其實沒遇到很大的問題，不過因為我忘記 UCB 的公式分母不能為 0，而且也忘記去探索其他沒被走過的 node，導致我連 random 都打不過。後來發現自己 UCB 算錯改進後，終於能夠全剩 weak 以及跟 strong 打到 30 場贏 20 場了。

```
GoGui-TwoGTP Launcher V20221101
========== PLAYERS ==========
P1B: ./nogo --shell --name="Hollow-Black" --black="mcts T=1000"
P1W: ./nogo --shell --name="Hollow-White" --white="mcts T=1000"
P2B: ./nogo-judge --shell --name="Judge-Weak-Black" --black="weak"
P2W: ./nogo-judge --shell --name="Judge-Weak-White" --white="weak"
============ GAMES ============
Storage: gogui-twogtp-20221128034430
Monitor: ./gogui-twogtp-20221128034430.mon
P1B vs P2W: ##### 5:0
P2B vs P1W: ##### 0:5
========== RESULTS ==========
P1: (5+5)/10 = 100.0%
P2: (0+0)/10 = 0.0%
```

```
GoGui-TwoGTP Launcher V20221101
========== PLAYERS ==========
P1B: ./nogo --shell --name="Hollow-Black" --black="mcts T=1000"
P1W: ./nogo --shell --name="Hollow-White" --white="mcts T=1000"
P2B: ./nogo-judge --shell --name="Judge-Weak-Black" --black="strong"
P2W: ./nogo-judge --shell --name="Judge-Weak-White" --white="strong"
============ GAMES ============
Storage: gogui-twogtp-20221128040024
Monitor: ./gogui-twogtp-20221128040024.mon
P1B vs P2W: ############### 9:6
P2B vs P1W: ############### 4:11
========== RESULTS ==========
P1: (9+11)/30 = 66.67%
P2: (4+6)/30 = 33.33%
```