Report of Theory of Computer Games project 3, 2022

109550027 紀竺均

a. Implementation

In this assignment, I create two classes – “MCTS” and “mcts_agent” which inherited from class “agent”.

In MCTS class, there are four mainly function for constructing MCTS tree – select, expand, simulate, and update. And a function “best_action” that decide the action based on the tree.

In class “MCTS”:

1. select():

   In function “select”, I select the current node’s child with the highest uct value. I’ll discuss the way I calculate uct value in section b.

```
Node* select(Node* curnode, bool myturn){
    float bestvalue=-10000;
    //Node* bestnode = new Node();
    int bestchild = 0;
    for(int i=0;i<(int)curnode->childs.size();i++){
        //Node* child = curnode->childs[i];
        //double val = uctvalue(*child, curnode->visittime);
        double val = uctvalue(*curnode->childs[i], curnode->visittime, myturn);
        if(bestvalue < val){
            bestvalue = val;
            //bestnode = child;
            bestchild = i;
        }

    }
    return curnode->childs[bestchild];
}
```

2. expand():

   In function “expand”, I append every legal move in current space to the node’s child list.

```
void expand(Node* node, bool myturn) {
    std::vector<Node*> children;
    std::vector<action::place>& tmpspace = (isblack(myturn)) ? blackspace : whitespace;
    for(int i=0; i < (int) tmpspace.size();i++){
        //cout << "inside 155 for loop\n";
        action::place& nextmove = tmpspace[i];
        board cur = node->position;
        if (nextmove.apply(cur) == board::legal)
            children.push_back(new Node(cur));
    }
    node->childs = children;
}
```

3. simulate
   In function “simulate”, I apply the legal moves based on rand_move function, which return a random legal movement drawn from current space, and return the win/lose result.

```
int simulate(const board& state, bool myturn){
        int iswin = 1;
        board tmp = state;
        action::place p = rand_action(tmp,myturn);

        while(p.apply(tmp)==board::legal){
            myturn = !myturn;
            p = rand_action(tmp, myturn);
            iswin ++;
            iswin = iswin % 2;
            cout << myturn << " " << iswin << endl;
        }
        return !myturn;
        //TODO: should return win or lose
    }
```

```
action::place rand_action(board& state, bool myturn){

    std::vector<action::place> tmpspace = isblack(myturn)? blackspace : whitespace;
    std::shuffle(tmpspace.begin(), tmpspace.end(), engine);
    for (const action::place& move : tmpspace) {
        board after = state;
        if (move.apply(after) == board::legal){
            return move;
        }
    }
    action::place t = tmpspace[0];
    tmpspace.clear();
    tmpspace.shrink_to_fit();
    return t; //illegal move
}
```

4. update():
   update current node’s visit time and the count of win time.

```
void update(Node* node, int iswin){
    node->visittime++;
    node->wintime += iswin;
}
```

5. best_action():
   If the childs list is empty, it means that there isn’t any legal move for current root, so we simply return a random action.
   If there are childs for root, we could choose the child with greatest visit time to be our next position.
   Once we have the next position, we can traverse thru current space to find what is the legal move that can lead to that position, and

finally return that legal move.

```
action::place bestaction(){
    int best = 0;
    if(root->childs.empty()){
        //cout << "weird" << endl;
        //return action();
        action::place p  = rand_action(root->position, true);
        board tmp = root->position;
        if(p.apply(tmp)==board::legal){
            return p;
        }
    }
    else{
        Node* bestchild = root->childs[0];
        for(int i=0; i<(int)root->childs.size(); i++){
            Node* child = root->childs[i];
            if(child->visittime > best){
                best = child->visittime;
                bestchild = child;
            }
        }
        std::vector<action::place>& tmpspace = (who==board::black)? blackspace:whitespace;
        for(int i=0;i<(int)tmpspace.size();i++){
            action::place next = tmpspace[i];
            board& nextboard = bestchild->position;
            board curposition = root->position;
            if(next.apply(curposition) == board::legal && nextboard == curposition){
                return next;
            }
        }
    }
    cout << "no available action\n";
    return action();
}
```

b. Improvements

(1) UCT value calculation

Originally, I calculate uct value based on simple function:

```
if(child.visittime==0){
    return 100000000;  // devide by zero
}
float c = 1.414;
float exploitation = (float)child.wintime / (float)(child.visittime);
float exploration = sqrt(log(cur_visittime) / (float)(child.visittime));
return exploitation + c * exploration;
```

Then, I tried to improve the function by adding who's turn is it while calculating uct value because if it's the opponent's turn, their win rate should be our loss rate (1-winrate).

```
if(!myturn){
    exploitation = 1 - exploitation;
}
```

(2) Time distribution

Because there are total 73 spaces in the board, there are at most 73/2 =

32 times that we will do a mcts simulation. Originally, I let every simulation to run at most 0.99 seconds (so it won't exceed the time limit of 40 seconds per game).
My improvement is that, spend more simulations in the middle of the game.

```
void mcts_simulate(){
    clock_t start;
    start = clock();
    sims_count++;
    float clocktime;
    if(sims_count<=3) clocktime = 0.8;
    else if(sims_count<=8) clocktime = 1;
    else if(sims_count<=15) clocktime = 1.2;
    else if(sims_count<=20) clocktime = 1.5;
    else if(sims_count<=25) clocktime = 1.2;
    else clocktime = 1;
    while((float) (clock()-start)/CLOCKS_PER_SEC<clocktime){
        //cout << "simulation # " << endl;
        sim(root);
    }
```

c. Results

Origin mcts player against strong agent

```
============ GAMES ============
Storage: gogui-twogtp-20221120220452
Monitor: ./gogui-twogtp-20221120220452.mon
P1B vs P2W: ################ 3:12
P2B vs P1W: ################ 12:3
=========== RESULTS ===========
P1: (3+3)/30 = 20.0%
P2: (12+12)/30 = 80.0%
```

Improved mcts player against strong agent

```
============ GAMES ============
Storage: gogui-twogtp-20221121014846
Monitor: ./gogui-twogtp-20221121014846.mon
P1B vs P2W: ################ 6:9
P2B vs P1W: ################ 9:6
=========== RESULTS ===========
P1: (6+6)/30 = 40.0%
P2: (9+9)/30 = 60.0%
```

d. Problems encountered and solutions

bad_alloc error:

During the running time, I found out all the memory in workstation is depleted and thus get this error.

Solution: recursively “delete” every node after I take my action, clear and shrink_to_fit the vectors after I allocate them.