Theory of Computer Games

Homework 2 Report (Fall 2023)

學生：許家銓；學號：r12922080

---

0. Complilation and Execution

- Compile

```
$ cd r12922080
$ make
```

The generated execution file is called **agent**, you can then play with it.

- Execute

```
$ ./game -p0 ../r12922080/agent -p1 ../baseline/hard -r 20
```

---

1. MCTS Algorithm and Implementation

MCTS基本分為四個部分：FindPV，Expand，Simulate，Backpropagation。因此整個大架構就如下：

```
while (clock() < time_limit)
{
    pvb.copy(game);
    ptr = FindPV(root, pvb);

    rtover = pvb.is_over_rtresult();
    if (rtover != 0) {
        simulate_is_over(rtover);
        backpropagation(ptr);
    }
    else {
        expand(ptr, pvb);
        simulate(ptr, pvb);
        backpropagation(ptr);
    }

    if (nodes[root].Ntotal > 1000000000000)
        break;
}
```

關於上課已經提到的MCTS與function的細節與講義基本無異，因此接下來會以一些課本上並未提供太多細節的simulation和我對原有MCTS的些微改動做解釋。

(a) Simulation

Simulation的主要目的是透過simulation去得到各分支的win rate並進行update，流程大致如下：

- 取得現在child node的盤面
- 進行20次random walk，每做完一次就同時記錄勝敗並回復到第一步的盤面

- 將勝場與總模擬次數給backpropagation去update上面的nodes

20次這個數值是因為在考量平均狀況之後所設下的數字。假設考慮未來兩步，第一步你有最多4個方向能走，第二步對手有最多4個方向能走，那兩步內總共的可能性就是16種，為了避免一些必輸或必贏的步數因為運氣不好或simulation次數太少，因此我選擇略大的20次使得兩步內的可能性都能被simulate到的期望值略大於1。

至於為何不再往上設則是想避免simulate過多次數在無用的branch上面，因此只要確保這個branch有高機率搜到每個可能性就好，至於要更多的simulation就讓FindPV去決定。

(b) If FindPV return an end game

在FindPV之後，我會先確認這個state是否結束。由於一個已經結束的node無法被expand跟simulate，我會根據其勝或負來直接加上全勝或全負的simulation次數，藉此增加他的權重。

```
void simulate_is_over(int &rtover) {
    simulate_deltaN = SIMULATION_PER_OVER_BRANCH;
    simulate_deltaW = ((self_color == rtover) ? SIMULATION_PER_OVER_BRANCH : 0);
}
```

(c) If simulation times of root is over 10^12

在MCTS的最後我會判斷simulation的次數是否大於一定數值，這個理由是因為在後期解殘局的時候，過多的simulation會導致long long overflow，使得它容易下錯步數。因此設置了一個數量上限作為避免overflow的方法。雖然使用string或double可以增加其上限，但實際上string與double所產生的overhead更高，因此不採用。

(d) UCB Variance

為了讓UCB不要過度集中，因此我根據講義內容也重新implement了有variance版本的UCB。

Variance由於會在每次FindPV中call UCB時使用，因此在simulate完那個node後就必須先儲存：

```
void simulate(int &ptr, min_board &pvb)
{
    ...
    for (i = 0; i < nodes[ptr].Nchild; i++) {
        ... // after simulation on ith child of nodes[ptr]
        nodes[id].Average = (float)nodes[id].sum1 / l_2_f;
        nodes[id].Variance = nodes[id].Average * (1 - nodes[id].Average);
        ...
    }
}
```

而後要根據講義修正UCB的公式，在最後乘上開根號的min(V, 0.25)：

```
float CsqrtlogN_div_sqrtN;
inline float UCB(int &id) {
    CsqrtlogN_div_sqrtN = nodes[parent(id)].CsqrtlogN / nodes[id].sqrtN;
    return ((nodes[id].depth%2) ? nodes[id].Average : (1.0-nodes[id].Average)) + \
            (CsqrtlogN_div_sqrtN * \
             sqrtf(min(nodes[id].Variance + 1.414214 * CsqrtlogN_div_sqrtN, 0.25)));
}
```

---

2. Tricks and Experiments

我實際上並沒有對這次的作業演算法進行太多變動，因此接下來的內容主要聚焦於我使用了哪些方法讓程式碼能夠快上一些。至於判斷的方式，我會以前五次simulation的速度平均與winning rate為主，這也是我評估各種方式速度的標準。

根據觀察，遊戲在前五步中能否順利吃完自身棋與得到好的位置至關緊要，因此simulation的在這邊是最重要的。而到後期因為PV的Tree已經長得相對完整並接近min-max tree，simulation的次數會因為上面遇到end game後直接進行加法的處理而迅速膨脹，因此前五步的simulation會比較重要。

(a) From double to float

這份作業中預設的很多浮點數為double，但double所需要的計算量比float高出許多，因此我選擇將所有的double轉換成float，犧牲精度以換取更高的速度。而以下給出了使用float與double後的速度比較：

	win rate against hard (-r 20)	1st sim. times	2nd sim. times	3rd sim. times	4th sim. times	5th sim. times
float	0.95	268220	299690	436870	531670	813150
double	0.90	229820	262410	444980	519110	421120

從結果可知，犧牲精度的效果並沒有很顯著，但總體來說還是能得到略微的提升，進而稍稍提升winning rate。

(b) Pretransform long long to float

在處理long long與浮點數的運算時，程式必須先將long long轉成與float相同的底數才能做運算，因此我會在有long long轉成float的運算時先轉進一個數值，而不是每次運算都多做一次transform。實際例子如下：

```
void update_nodes(int &id) {
    ...
    l_2_f = (float)nodes[id].Ntotal;
    nodes[id].CsqrtlogN = 1.18 * sqrtf(logf(l_2_f));
    nodes[id].sqrtN = sqrtf(l_2_f);
    ...
    nodes[id].Average = (float)nodes[id].sum1 / l_2_f;
    ...
}
```

在update_nodes的function中，我實際上可能需要三次transform，而利用l_2_f這個變數的話就只做了一次。而以下就是使用pretransform後的實驗結果：

pretransform	win rate against hard (-r 20)	1st sim. times	2nd sim. times	3rd sim. times	4th sim. times	5th sim. times
w/	0.95	268220	299690	436870	531670	813150
w/o	0.90	265090	273070	394090	485370	331520

從結果可知，pretransform的方法對於整體速度的影響並沒有特別大，但還是能從這一次測試中略看出有pretransform後的結果確實比沒有做的程式快一些。

(c) UCB and UCB with variance

而除了基本的加速外，variance的implementations會使得總計算量增加，但卻可以讓UCB的表現更加穩定，因此根據講義重新修正後的UCB公式如下：

```
float CsqrtlogN_div_sqrtN;
inline float UCB(int &id) {
    CsqrtlogN_div_sqrtN = nodes[parent(id)].CsqrtlogN / nodes[id].sqrtN;
    return ((nodes[id].depth%2) ? nodes[id].Average : (1.0-nodes[id].Average)) + \
           (CsqrtlogN_div_sqrtN * \
```

```
            sqrtf(min(nodes[id].Variance + 1.414214 * CsqrtlogN_div_sqrtN, 0.25)));
}
```

因為CsqrtlogN / nodes[id].sqrtN的這個步驟會被重複計算，因此也先存到一個全域變數中進行處理。

variance	win rate against hard (-r 20)	1st sim. times	2nd sim. times	3rd sim. times	4th sim. times	5th sim. times
w	0.95	268220	299690	436870	531670	813150
w/o	0.70	318980	305060	316180	415880	623610

從結果可知，不使用variance的話在前面幾不的simulation確實比較快，但可能是因為他會更不容易發散找到合理的步伐，在第三手後的simulation times就開始略低於有variance的版本，這代表variance計算時間的犧牲是合理的，且也能夠在更具決定性的第4、5手中得到更多的simulation次數，並提高winning rate。

(4) Compilation Config

在我努力嘗試加速程式運算的時候，我了解到compile的config也會影響運算，因此我嘗試了下面不同的compilation config：

- -O3
- -O3 -ffast-math

ffast-math是g++內建的一個config，能夠在犧牲精度的情況下優化部分浮點數運算時間的config，因此我也嘗試了使用他的情況後simulation的上升情況，結果如下：

-ffast-math	win rate against hard (-r 20)	1st sim. times	2nd sim. times	3rd sim. times	4th sim. times	5th sim. times
w/	0.95	268220	299690	436870	531670	813150
w/o	0.70	219890	203740	243830	256500	335180

可以看見其實ffast-math帶來的浮點數運算效益是相當好的，能夠優化許多複雜或還沒處理的常數運算，並提升performace。

(5) Performace Conclusion

下面總結這份程式的performance，讓agent輪流對三個baseline進行50場的測試：

```
./game -p0 ../r12922080/agent -p1 ../baseline/{baseline} -r 50
```

round \ win rate	easy	normal	hard
Average	1.0	0.98	0.88