Theory of Computer Games

HW2 – Othello report

NTUST / M10415096 / Ze-Hao Wang

1. 本文名詞定義與前言

- ROTP: 基礎為原範例程式中的 OTP 類別。
- OTP: 繼承了 ROTP 類別的子類別，其中特別是 override 了 genmove 的函數。

本次作業實作的項目有 **UCB、UCT 以及 Progressive pruning** 等。依規定僅修改 OTP.h 檔案，並進行編譯與執行等操作。

2. 初始版本的 UCB

在這一版本中，我將 UCB 仍大致上劃分與 Monte Carlo Tree Search 近似的四個步驟外，加上了建置初始 root 盤面的前置工作。為了保留範例程式中的亂數選步，我將原先的 OTP 更名為 ROTP，讓經過 override 的類別稱作為 OTP，會這樣的主因是本次操作不得修改 OTP.h 的限制下，外部程式仍會調用 OTP.h 當中的 OTP 類別。在本節中並不會介紹我在 UCB 使用的資料結構，原因是與 UCT 相當近似，可直接參考 3-1 小節的 UCT 資料結構。

2-1. Preprocess in UCB

利用了範例程式中 *ML* 與 *MLED* 的概念，取得目前的所有合法步後，將其列為評定的選項之一。然而每一次的迭代都會需要 root 盤面，此處所指的 root 並非遊戲開始的四子盤面，而是在當前回合中，該玩家在進行下一步棋的先行盤面。在每一輪的迭代模擬中，皆會宣告一虛擬盤面 (即 ROTP 類別)，並加以改良範例程式中的 *History(H)* ，藉由依序執行 OTP 界面中的 *play* 指令，重現出該回合的 root 盤面。

2-2. Selection in UCB

在上一小節提及的選項，便是該回合中的合法步，在這些節點中選擇最高的 UCB score 接續下一步驟。其中 UCB 使用下述公式：

其中的 N 是目前的迭代次數；c 是一常數，主要使用 1.414。

child.wins / child.visits + c * sqrt(log(N) / child.visits)

2-3. Expansion in UCB

然而在 UCB 中並不需要再衍生分支，於是對 selection 步驟中所選之節點執行 *play* 指令，確立該棋步以加入至盤面之中。

2-4. Simulation in UCB

board.h 中的 *is_game_over()* 包裝至 OTP 類別中使用。藉此成為模擬迴圈的跳出判斷條件，反覆執行 *genmove* 指令，而亂數模擬採用原先範例程式的亂數方式（C++ 之 *uniform_int_distribution*）。

2-5. Propagation in UCB

利用 *board.h* 中的 *get_score()*，包裝至 OTP 類別中調用。並在各選項中更新其 wins 與 visits 的數值，其中 wins 的部份為，勝利 +1；和局 +0.5；失敗 +0。但事實上 *get_score()* 是以先手的角度去計算，在這裡會將自身 tile 加權進去，達到取用自身的正確得點。

2-6. 選擇最高勝率者

以 wins / visits 最高分數者作為本步驟的最佳解。

3. 基於樹狀結構的 UCT

保留了前一章節 UCB 的評分機制，在此章節除了 simulation 的步驟外，有小幅度以上的修正。

3-1. 資料結構

UCB、UCT 基本上使用的結構大致相同，但做了些微修改，這裡將整合說明 UCT 的資料結構。

```
#define Branches std::vector<Visitation*>

struct Visitation {
    // Is still available due to different round.
    bool isAvailable;
    // Next selected position.
    int nextXY;
    // Times of visiting and winning.
    double visits, wins;
    // Parent node.
    Visitation *parent;
    // Is still can be expand the children nodes.
    bool isExpandable;
    // Children nodes.
    Branches branches;
};
```

3-2. Preprocess in UCT

與前章節的概念大致雷同，但因為目前為樹狀結構，可能會遇到未被模擬過的分支，所以在重現盤面遇到該情況時必須當下新增出該子節點。

```
do
  play 該 history 的座標

  found = 尋找下一層子節點中與 history 符合者

  if 剛才搜尋的 found 存在
    更動 snPtr, root 的 pointer 至此
```

```
  else
    新增一節點，並且初始化
    更動 snPtr, root 的 pointer 至此

while history 的元素
```

3-2. Preprocess in UCT

與前章節的概念大致雷同，但因為目前為樹狀結構，可能會遇到未被模擬過的分支，所以在重現盤面遇到該情況時必須當下新增出該子節點。

```
do
  play 該 history 的座標

  found = 尋找下一層子節點中與 history 符合者

  if 剛才搜尋的 found 存在
    更動 snPtr, root 的 pointer 至此
  else
    新增一節點，並且初始化
    更動 snPtr, root 的 pointer 至此

while history 的元素
```

3-3. Selection in UCT

使用的公式與前章節相同，但因為這一次建立的 min-max tree 的關係，當屬於自身回合將選擇最大 UCB score（對自己最有利），屬於敵方回合將選擇最小 UCB score（對自己最不利）。

```
do
  isMyTurn = 目前的回合數 % 2 == 我所持的棋

  if isMyTurn == true
    max = 目前所有子節點中 UCB score 最大者
  else
    max = 目前所有子節點中 UCB score 最小者

  更新 snPtr 的 pointer 至 max 的 address
```

```
  play 該 max 的座標

  目前的回合數 +1

while 當前節點的 isExpandable == false 且 非分支數量 > 0
```

3-4. Expansion in UCT

在 UCT 中的 expansion 會檢查目前的合法步是否都以納入目前的節點之中，若沒有新建該節點並執行 *play*，讓盤面往後推移一回合。

```
ptr = ML + 目前所有子節點的數量

if ptr != MLED
  新增一節點，為 ptr ( ML 是一陣列，ptr 為其指標 )
  將新節點插入至目前節點的子節點

  if ptr+1 == MLED
    將目前節點的 isExpandable = false

  更新 snPtr 的 pointer 為新節點

  play 該新節點的座標
```

3-5. Simulation in UCT

與前章節之 simulation 完全相同。

```
do
  genmove

while 遊戲結束
```

3-6. Propagation in UCT

與前章節不同處，在更新得分時將會同時更新其所有父節點的 visits 與 wins。

```
score = 經過加權後的總分

weight = 依照 score 的正負決定勝負

do
  visPtr 的 visits +1
  visPtr 的 wins 加上 weight

  更新 visPtr 的 pointer 為其父節點

while visPtr != NULL
```

4. 剪枝優化

在計算 UCB score 時，加入判斷該節點的 *isAvailable* 是否為 *true*，若為 *false* 則讓得分變成負無窮大 (-INFINITY) 。然而，在 selection 的階段時，可以設定某迭代次數之倍數時進行剪枝。在本次作業中，我採用的是每 1,000 次迭代，將選擇數量不超過平均「目前跌代數 ÷ 選擇數」種者。將其 *isAvailable* 設定為 *false*，藉此達到減少不必要資源的模擬計算，以分配其它計算資源給勝率較高者。

5. UCB 參數調整

本章節針對 UCB score 的常數 *c* 進行調整，就本次的實驗環境比較難以就遊戲結果之數據觀察出明顯現象，因為雙方的 AI 使用了相同的演算法，若能將其一方固定唯一不同的演算法，方可比較之。但仍可知道的是，若 *c* 較高，將會讓期望勝率較低者分配到愈多的模擬機會，反之亦然。

6. 實驗環境

- CPU: Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz
- RAM: 8GiB DIMM Synchronous 2133 MHz x2

7. 編譯與執行方式

請參照 code/shell.sh 或 code/makefile，內已包含**編譯指令**、**執行方式**。

7-1. 編譯方式

```
$ cd ./code
$ make

or

$ cd ./code
$ g++ ./src/judge.cpp -std=c++11 -O2 -Wall -o ./debug/judge
$ g++ ./src/search.cpp -std=c++11 -O2 -Wall -o ./debug/search
```

7-2. 執行方式

```
$ ./debug/judge 7122 > ./debug/log_ju.txt &
$ ./debug/search 127.0.0.1 7122 > ./debug/log_p1.txt &
$ ./debug/search 127.0.0.1 7122 > ./debug/log_p2.txt &
```

7-3. 開啟除錯模式

請將 code/src/OTP.h 的 // #defined DEBUG true 的註解移除，並重新編譯。

7-4. 備註部份

若有需要修改模擬次數、UCB 常數或剪枝週期等，請參考 *OTP.h (Line 159~161)*。