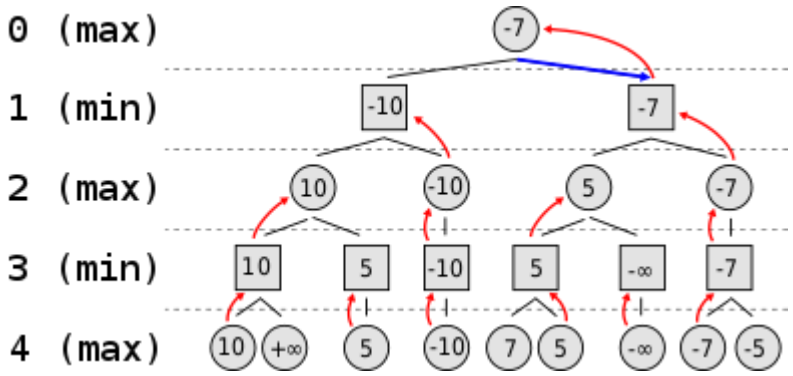
Final Project Report

104000054 董庭瑋

- ## Method

主要是利用 minimax 去算出所有可能性之中的最佳解。



(wiki 的圖)

其實助教有解釋過了所以我就大概講一下就好。Max 的部分是 player，min 則是 opponent，每一次 player 的回合，理論上都會儘可能地去找所有 valid move 裡面最好的一個，也就是這一步下去 player-opponent 的棋子數量會是最大值。而 opponent 的回合則是相反，他會去找”對他來說的最大值”，也就是對 player 來說的最小值。而如果只考慮一步的話(greedy)，在面對比較強的 AI 時很容易被打敗，所以 minimax 就是要從當前這一步一路算到最後，再從所有的可能性中去找一個最好的。

但因為題目有限制時間，而且如果每一步都把所有的可能都算出來，整棵樹會長得很大會花很多時間，反正至少是會超過題目的時間限制，所以要

用 alpha-beta pruning 來降低時間。大致上的概念就是把重複運算或是明顯沒有必要運算的分枝給剪掉，用一組 alpha-beta 值來訂定一個區間，只有在這個區間裡的值才會被考慮進去，同時這組值也會在 recursive 的過程裡不斷被更新。

- ## Implement

首先為了方便操作和設計，我把 main 裡面的 Point struct 和 OthelloBoard class 弄了進來。

```
struct Point {
  int x, y;
  Point() : Point(0, 0) {}
  Point(float x, float y) : x(x), y(y) {}
  bool operator==(const Point &rhs) const { return x == rhs.x && y == rhs.y; }
  bool operator!=(const Point &rhs) const { return !operator==(rhs); }
  Point operator+(const Point &rhs) const {
    return Point(x + rhs.x, y + rhs.y);
  }
  Point operator-(const Point &rhs) const {
    return Point(x - rhs.x, y - rhs.y);
  }
};
```

(Point struct)

```
class OthelloBoard {
public:
  enum SPOT_STATE { EMPTY = 0, BLACK = 1, WHITE = 2 };
  static const int SIZE = 8;
  const array<Point, 8> directions{
      {Point(-1, -1), Point(-1, 0), Point(-1, 1), Point(0, -1),
       /*{0, 0}, */ Point(0, 1), Point(1, -1), Point(1, 0), Point(1, 1)}};
  array<array<int, SIZE>, SIZE> board;
  vector<Point> next_valid_spots;
  array<int, 3> disc_count;
  int cur_player;
```

(OthelloBoard class，太長了我意思一下就好)

基本上就是完全沿用，怕做了更動會讓後面的處理很麻煩所以沒有做什麼改變，

只有把後面 encode 會造成 bug 的地方拿掉(我懶得修，反正也用不到)。

```
int main(int, char **argv) {
  // cout << "start random" << endl;
  ifstream fin(argv[1]);
  ofstream fout(argv[2]);
  initScoreBoard();
  read_board(fin);
  read_valid_spots(fin);
  write_valid_spot(fout);
  fin.close();
  fout.close();
  return 0;
}
```

我是拿 random 來改的，大致上的架構沒有差太多，我會先把流程講一

次，優化跟改動的部分後面再說。

```
248
249 ▼ void read_board(ifstream &fin) {
250     fin >> player;
251 ▼   for (int i = 0; i < SIZE; i++) {
252         for (int j = 0; j < SIZE; j++) {
253           fin >> board[i][j];
254       }
255     }
256   }
257
258 ▼ void read_valid_spots(ifstream &fin) {
259     int n_valid_spots;
260     fin >> n_valid_spots;
261     int x, y;
262 ▼   for (int i = 0; i < n_valid_spots; i++) {
263       fin >> x >> y;
264       valid_spots.push_back({static_cast<float>(x), static_cast<float>(y)});
265     }
266   }
267
```

Read_board 跟 read_valid_spots 的部分沒有改什麼，基本上就是沿用，

唯一有差別的地方是因為我 Point struct 改從 main 那邊拿，裡面多塞了 float，所以我在 read_valid_spots 最後在 push 的時候要轉成 float。

```
void write_valid_spot(ofstream &fout) {
  for (auto tmp : valid_spots) {
    if ((tmp.x == 0 && tmp.y == 0) || (tmp.x == 0 && tmp.y == 7) ||
        (tmp.x == 7 && tmp.y == 0) || (tmp.x == 7 && tmp.y == 7)) {
      fout << tmp.x << " " << tmp.y << endl;
      fout.flush();

      return;
    }
  }

  OthelloBoard now;
  for (int i = 0; i < SIZE; i++)
    for (int j = 0; j < SIZE; j++)
      now.board[i][j] = board[i][j];
  now.cur_player = player;
  now.next_valid_spots = valid_spots;

  int best = -1;

  Point ans;
  for (auto tmp : valid_spots) {
    OthelloBoard _try = now;
    _try.put_disc(tmp);
    int val = ABP(_try, 7, -99999999, 99999999, true);

    if (val > best) {
      best = val;
      ans = tmp;
    }

    fout << ans.x << " " << ans.y << endl;
    fout.flush();
  }
}
```

在 write_valid_spot 這邊，我先用一個新的 OthelloBoard “now” ，他基本上就是複製讀進來的 board，這樣做只是因為我不想動到原本的 board。

之後就是要去 call minimax 的部分，我又創了一個 OthelloBoard”_try”，用來跑 recursive，純粹是個人習慣，然後就是把每個 valid_spots 都跑過一遍。我用一個 integer variable”val”來存回傳的值，然後去 call ABP 這個 function(alpha-beta pruning，因為我直接用他取代 minimax)。之後會用這個 val 去跟一個 integer variable”best”比較，best 一開始是-1 因為我設定所有的 board value 都是正數，這樣可以確保我至少會找到一個解(但其實後來有遇到問題，後面再說)。

```
int ABP(/*node*/ OthelloBoard now, int depth, int alpha, int beta,
        bool maximizingPlayer) {

  if (depth == 0 || now.next_valid_spots.size() == 0){
    //return now.disc_count.size();
    int value = 0;

    for(int i = 0; i < SIZE; i++){
      for(int j = 0; j < SIZE; j++){
        if(now.board[i][j] == player)
          value += score_board[i][j];
      }
    }

    return value;
  }

  // if maximizingPlayer
  if (maximizingPlayer) {
    // value = -999999
    int value = -999999;
    // for each child of node do
    for (auto tmp : now.next_valid_spots) {
      // value := max(value, ABP(child, depth - 1, alpha, beta, FALSE))
```

接下來是重點的 ABP 部分。ABP 一共會吃五個參數：now 是當前的 board，對應到 ABP 演算法裡面的”node”。Depth 主要的功用是拿來

當 recurisve 的中止條件，每一次往下 call ABP，depth 都會減一，這樣 depth == 0 的時候就表示到 leaf node 了。Alpha、beta 則是用來 pruning 的參數，maximizingPlayer 是用來記錄目前是 player round 還是 opponent round。

就像大部分 recursive tree traversal 一樣，我習慣先寫好 leaf node 要回傳值的部分(其實 pseudo code 也是這樣)。當來到 leaf node 時，理論上表示到了要回傳值的時候，這邊的”值”指的是兩邊玩家的棋子相差的數量，我這邊的設計是我用了一個 score_board。

```
void initScoreBoard() {
  for(int i = 0; i < SIZE; i++)
    for(int j = 0; j < SIZE; j++)
      score_board[i][j] = 1;

  score_board[0][0] = score_board[0][7] = score_board[7][0] = score_board[7][7] = 9999;
}
```

這個 score_board 上面基本上全部都是 1，這樣可以很快速方便的紀錄目前的分數，當 recursive 到 leaf node 時，會有一個 integer value ” value” 來算出目前的分數，這個就可以來當成我們 leaf node 的值。

```
206   if (maximizingPlayer) {
207     // value = -999999
208     int value = -999999;
209     // for each child of node do
210     for (auto tmp : now.next_valid_spots) {
211       // value := max(value, ABP(child, depth - 1, alpha, beta, FALSE))
212       // alpha := max(alpha, value)
213       OthelloBoard _try = now;
214       _try.put_disc(tmp);
215
216       value = max(value, ABP(_try, depth - 1, alpha, beta, false));
217       alpha = max(alpha, value);
218
219       // if beta <= alpha
220       // break (*pruning beta*)
221       if (beta <= alpha)
222         break;
223     }
224     // return value
225     return value;
226   } else {
227     // else (*minimizing player*)
228     // value := 999999
229     int value = 999999;
230     // for each child of node do
231     for (auto tmp : now.next_valid_spots) {
232       // value := min(value, ABP(child, depth - 1, alpha, beta, TRUE))
233       // beta := min(beta, value)
234       OthelloBoard _try = now;
235       _try.put_disc(tmp);
236
237       value = min(value, ABP(_try, depth - 1, alpha, beta, true));
238       beta = min(beta, value);
239
240       // if beta <= alpha
241       // break (*pruning alpha*)
242       if (beta <= alpha)
243         break;
244     }
245     return value;
246   }
247 }
```

而在下面就是要去 call recursive 的地方了，一共會分成 player round 和 opponent round。其實兩邊要做的事情差不多，只是一個要去找最大一個要去找最小(minimax)，而一個要去更新 alpha 一個要更新 beta。在這裡面，因為也是要去 Call recursive，所以做的事情其實跟一開始 write_valid_spot 很像，用一個額外的_try 去跑 recursive，接住回傳的值，然後做比對之後更新。而 call 完 recursive 之後下面會有一個 beta

<= alpha，這個是在做 pruning 的部分。

```
Point ans;
for (auto tmp : valid_spots) {
  OthelloBoard _try = now;
  _try.put_disc(tmp);
  int val = ABP(_try, 7, -99999999, 99999999, true);

  if (val > best) {
    best = val;
    ans = tmp;
  }

  fout << ans.x << " " << ans.y << endl;
  fout.flush();
}
```

最後當 recursive 全部跑完，write_valid_spot 裡面的 ans 就會是最佳解，再把他的 x 跟 y fout 出去就好。

- ## Problems & solution

在設計的過程中我有遇到一些問題。像是最一開始的時候我雖然設計出了 ABP，但一直沒辦法很穩定的獲勝，後來我上網查了關於黑白棋的各種攻略，發現四個角非常的重要，因為這四個角一旦被拿走基本上就不可能被拿回來，而且拿到四個角就可以很容易拿到一整個邊，這樣獲勝的機率就很高。但是因為我一開始把 board 上的每一的點都設成一樣的 value 所以我的 AI 並沒有辦法了解到這件事。

```
void initScoreBoard() {
  for(int i = 0; i < SIZE; i++)
    for(int j = 0; j < SIZE; j++)
      score_board[i][j] = 1;

  score_board[0][0] = score_board[0][7] = score_board[7][0] = score_board[7][7] = 9999;
}
```

所以我在 initial score_board 的時候，把四個角的 value 拉到很大，這樣就可以讓我的 AI 盡可能地去拿這四個角。當然這樣也有缺點，因為 minimax 是去預測對方的動作，所以我知道四個角很重要對手當然也會知道他很重要，所以就會變成對手也會想辦法去搶角(後來我想 baseline 應該也有把這個設計進去，所以這個缺點應該就不存在了)。

而第二個問題是我在測我的 AI 時會一直下 invalid move，後來我把 timeout 拉長就有好一點，但一樣會在中間的某個步驟爆炸。我一開始想不太懂為什麼，因為我覺得我已經有進行 pruning 所以理論上應該是不會超時，除非題目設的時間根本就不夠。但後來我反覆測試之後發現，可能我設計的 pruning 不夠好，但的確在某些狀況下有可能會超時，因為我的設計是等全部的 tree 跑完之後回傳一個最佳解，但這個做法在某些狀況下會超時。

```
    Point ans;
    for (auto tmp : valid_spots) {
      OthelloBoard _try = now;
      _try.put_disc(tmp);
      int val = ABP(_try, 7, -99999999, 99999999, true);

      if (val > best) {
        best = val;
        ans = tmp;
      }

      fout << ans.x << " " << ans.y << endl;
      fout.flush();
    }
  }
```

因此後來我改成，在最外層的 loop 時，每跑一次 recursive 就會 fout 一次，這樣我在時限內一定會產生一組解，而如果我時間夠可以繼續往下跑，在產生更佳解時就會繼續 fout，因為題目設計是只會取最後一筆所以這樣可以確保我被取的解都是我在時限內能跑到的最佳解。

```
void write_valid_spot(ofstream &fout) {
  for (auto tmp : valid_spots) {
    if ((tmp.x == 0 && tmp.y == 0) || (tmp.x == 0 && tmp.y == 7) ||
        (tmp.x == 7 && tmp.y == 0) || (tmp.x == 7 && tmp.y == 7)) {
      fout << tmp.x << " " << tmp.y << endl;
      fout.flush();

      return;
    }
  }
```

另外我做了一點小小的優化(?)。我在 write_valid_spot 的最一開始，先檢查過整個 valid_spots，如果這當中存在四個角的話，我就直接取他當作答案，因為理論上他也應該是最佳解，就算不是，依照上面說的，四個角的重要性，他也應該被當成最佳解。而且這樣我可以確保如果我的

valid_spots 裡面有角的話一定會被選到，因為在某些狀況下，比如說我的 tree 跑不完，有可能 valid_spots 裡面有角但因為 tree 跑不完所以沒有選到，所以這樣做可以確保我一定會選到角。