Artificial Intelligence HW3 – EinStein Wurfelt Nicht

TUDENT NO: 90799210Y NAME: 方志哲

user
[公司名稱] | [公司地址]

內容

1. 基本背景資訊

a. 機器規格

- 處理器
  Intel® Core™ i5 7300U 處理器(@2.50GHZ, @ Turbo up 2.71GHZ)
- 作業系統
  Windows 10 Home
- 記憶體
  8 GB DDR4 2133MHz SDRAM Onboard Memory, 1 x SO-DIMM socket for expansion, up to 12 GB SDRAM
- 螢幕
  14.0" (16:9) LED backlit FHD (1920x1080) 霧面螢幕 Panel with 72% NTSC with 178˚ wide-viewing angle display
  Support ASUS Splendid Technology
- 顯示晶片
  NVIDIA GeForce 940MX , with 2GB GDDR5 VRAM
- 鍵盤
  Chicklet keyboard

- 讀卡機
  Multi-format card reader (SD/SDHC/SDXC/MMC)
- WebCam
  VGA 網路攝影機
- 網路功能
  Wi-Fi
  Integrated 802.11 AC (2x2)
  Bluetooth
  Built-in Bluetooth V4.2
- 介面
  1 x Type C USB3.0 (USB3.1 GEN1)
  1 x Type A USB3.1 (GEN1)
  2 x USB 2.0 port(s)
  1 x RJ45 LAN Jack for LAN insert
  1 x HDMI
  1 x SD card reader
  1 x audio jack COMBO
- 音效
  Built-in Stereo 2 W Speakers And 麥克風
  Support Windows 10 Cortana
  ASUS SonicMaster Technology
- 電池
  3 Cells 42 Whrs Battery
- 電源 AC
  輸出：
  19 V DC, 3.42 A, 65 W
  Plug Type :ø4 (mm)
  輸入：
  100 -240 V AC、50/60 Hz 通用
- 產品尺寸
  326.4 x 225.5 x 18.75 ~20.15 mm (寬 x 長 x 高) (w/ 3cell battery)
- 安全鎖
  Security lock
- 備註
  * 資料儲存應用:
  1TB HDD+128GB

  *重量:1.3kg(規格視地區而異)

b. 開發軟體版本

Visual Studio 2017 (C++/C)

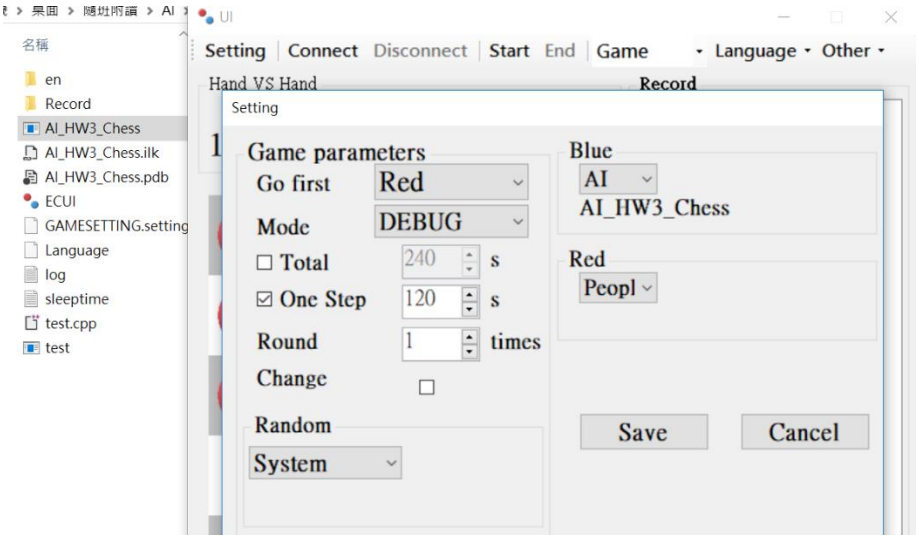
c. Contact Telephone:

0919528031

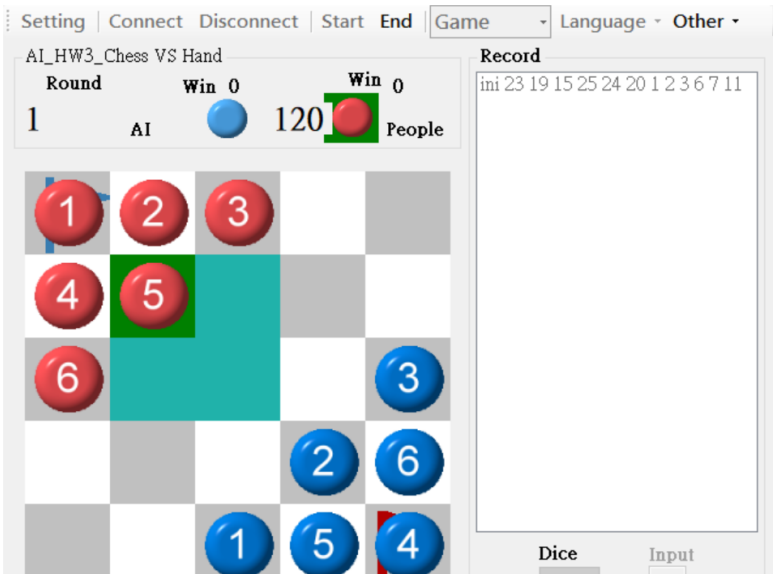
2.如何執行程式

a.下棋界面 [Reference 4, 5]

下棋界面我使用已開發平台 ECUI, 只要將我的執行檔 AI_HW3_Chess.exe 拉入右方 ECUI 的設定介面, 儲存設定後即可開始下棋



點選介面上方 start, 即可開始利用架設好的平台與 AI 進行對弈



綠色代表紅方合法的走步, 你可以任選一個點作走步, 更詳細說明可以參考 Refrence 4 ,5 林順喜老師所提供的 ECUI 教學文件

b. Debug 介面

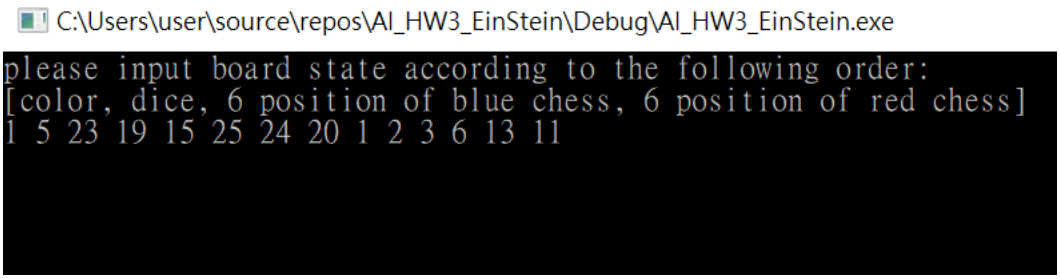
為了更方便各種盤面的測試與開發, 我自己額外寫了一個程式來測試 AI 對盤面的判斷

- 輸入棋盤資訊

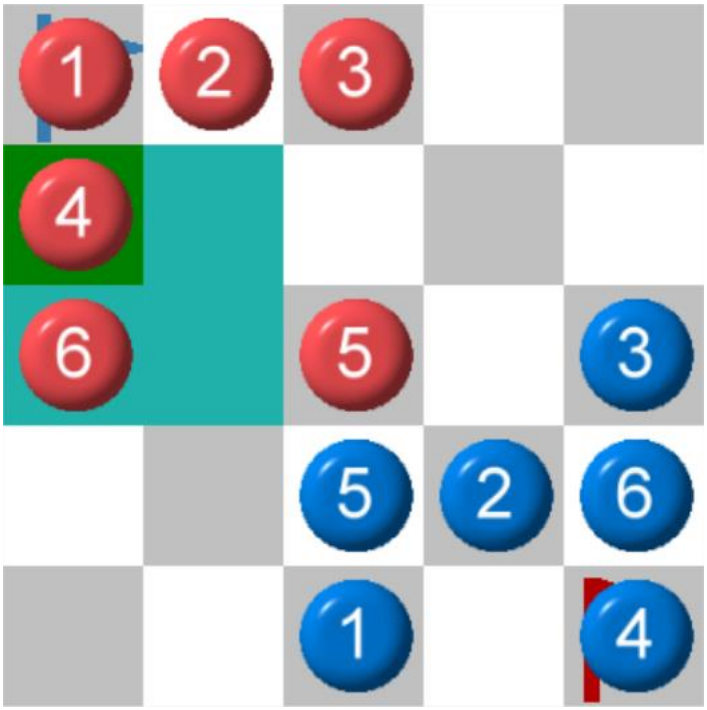
開啟程式 AI_HW3_EinStein.ext, 一開啟後便會出現資訊要求使用者輸入棋盤資訊, 棋盤資訊的格式是按照 ECUI 內的定義: (唯一不同的是將 Blue 用 0 代表, Red 用 1 代表)

- **輸入：**
- **get 你的顏色 骰子點數 藍棋位置（6 個） 紅棋位置（6 個）**
  - get B 2 23 0 15 13 24 20 1 2 9 6 0 11

程式執行畫面如下, 1 5 23 19 15 25 24 20 1 2 3 6 13 11 為我們輸入之棋盤之訊

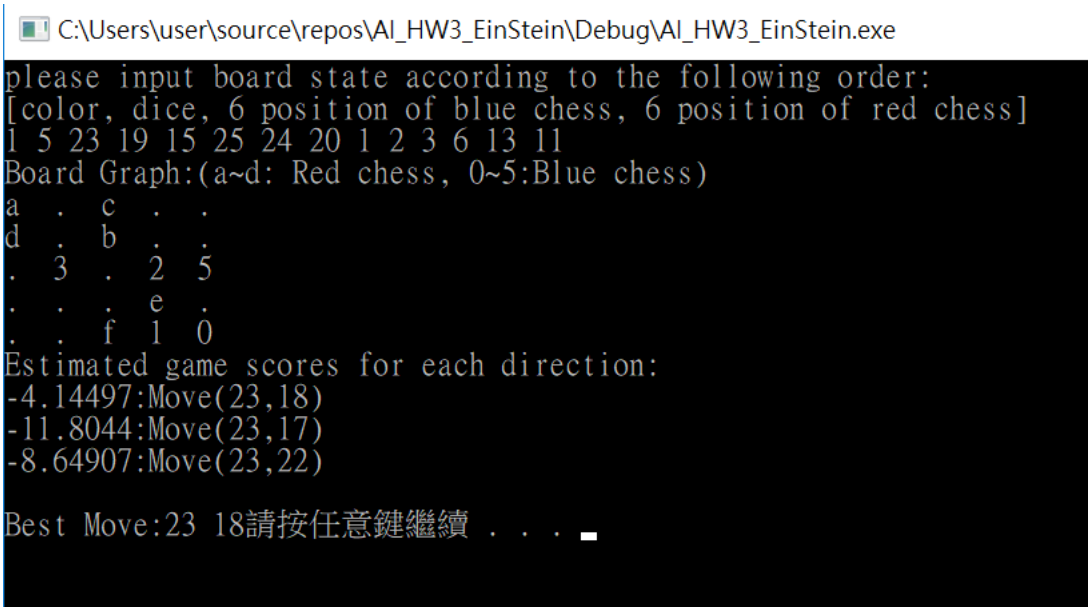


按照上述棋盤定義, 上述 input 即為下列盤勢, 輸入後讓 AI 判斷局勢, AI 會計算出 best Move



- 輸出棋盤資訊, Estimated Game Scored 以及最佳 Move

輸入棋盤資訊後, 程式會將棋盤資訊視覺化, 並且計算出棋子各種走步所得之期望賽局分數(分數越大代表執子方棋局越佔優勢, 負數代表敵方較佔優勢), 可以看到下圖三種走法期望分數分別為-4.14497, -11.8044, -8.64907, AI 會挑選最大期望賽局分數對應到的走步, 作為下一步, 因此 AI 選擇 Move(23, 18)這一個走步



而走步的定義則沿用 ECUI 平台的定義: Move(a,b) 代表從移動位置 a = 23 的棋子到位置 b = 18

- **輸出：**
- **移動起點 移動終點**
  - 23 18

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

3. AI 使用方法

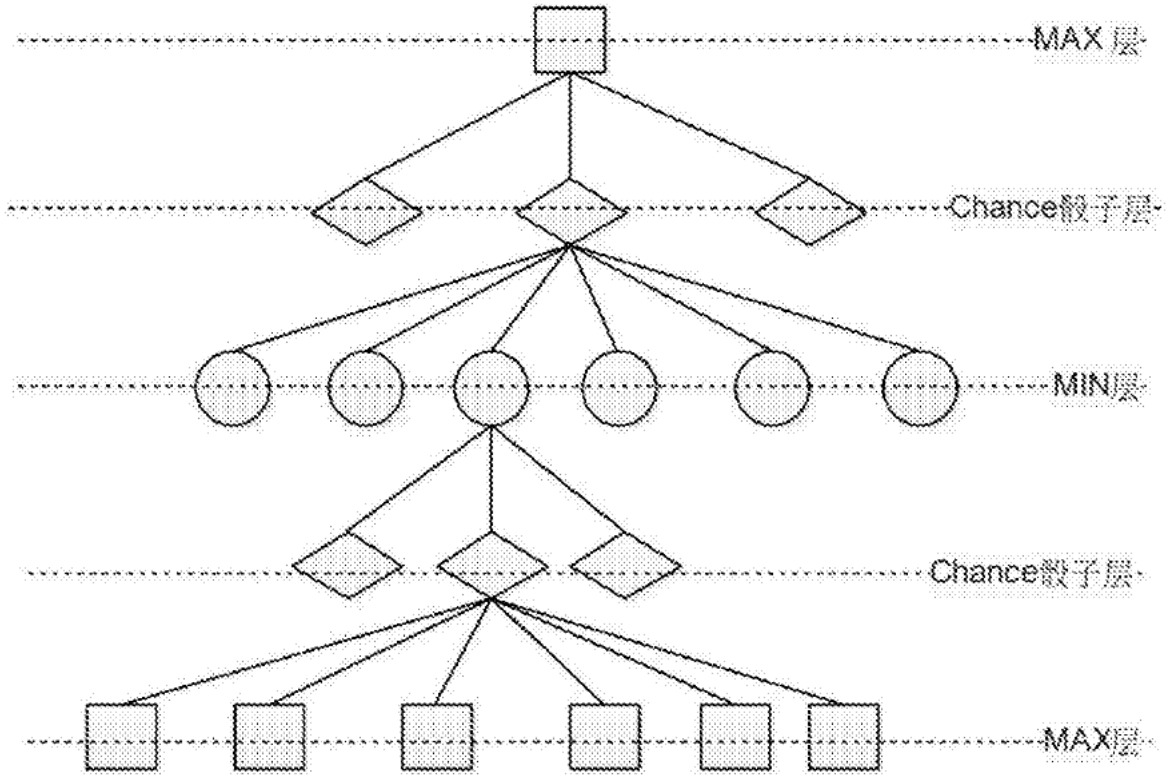
- 核心演算法

核心演算法是使用 Expect Minimax Algorithm 作盤面的操練, 但我依照 Negamax Algorithm 的精神對此演算法做了部分修正, 讓程式碼不用再針對 min/max 方分別寫函式, 寫下來教為簡潔, 操練演算法虛擬碼如下:

```
float expected_minmax(int depth, bool chanceNode) {
如果為 chance node:
        If(判斷賽局結束 or 需要 Cut-OFF){
                回傳審局函數評分
        }else{
                Best = 六種擲骰子情況下, 各別盤面最後的賽局分數期望值
                儲存該盤面至 hash Map
                Return Best;
        }
如果為 min/max node:
```

```
        令 best = -INFINITY
        已經骰子為 d 的情況下
        For 當前盤面 min/max 方所有可能走法{
                If(非重複盤面){
                        Value(i) = (-1) * 遞迴呼叫 expected_minmax 回傳此走步下, 最後盤面期望值
                }
        }
        Return 最大的 Value(i);
}
```

操練演算法圖示(Game Tree)如下:



[Reference 3]

- 時間複雜度分析

Average Case: $O(b^d) = O((3*6)^d) = O(18^d)$, 其中 b = number of branch, d = Depth of Game Tree

Average Case 考慮當丟出來的骰子都有其對應的棋子, 則為了計算期望值, 我們要計算每一個可能骰子下之賽局分數, 又因為每個骰子最多有 6 個可能點數(1~6), 而且丟完骰子後可以決定要動哪一顆棋子, 而每一顆棋子有 3 個可能走向(以藍棋為例就是左, 左上, 上), 所以每一層 game tree 都有 18 個分支

PS: 這裡定義的深度和一般的 tree 定義不太相同, 我是把 min/max node + chance node 合併算作一層下去作計算

Worst Case: $O(b^d) = O((2*3*6)^d) = O(36^d)$, 其中 b = number of branch, d = Depth of Game Tree

Worst Case 考慮當丟出來的骰子沒有其對應的棋子, 則按照按愛因斯坦棋的規則會選擇兩顆和骰子編號最接近的棋子, 且我們假設這 2 顆最接近骰子編號之棋子均存在(Ex. 骰子丟到 3 號, 則可以移動 4 號 or 2 號棋子)

則為了計算期望值, 我們要計算每一個可能移動之棋子下之賽局分數, 又因為每個骰子最多有 6 個可能點數(1~6), 而且丟完骰子後可以決定要動哪一顆棋子, 而每一顆棋子有 3 個可能走向(以藍棋為例就是左, 左上, 上), 所以每一層 game tree 都有 2*3*6 = 36 個分支

- 空間複雜度分析

Average Case:

延續時間複雜度分析對 average case 的定義, 加上 Expect minimax 採用 death first search, 其空間複雜度為 O(b*d) = O(18*d), 其中 b = number of branch, d = Depth of Game Tree

Worst Case:

延續時間複雜度分析對 worst case 的定義, 加上 Expect minimax 採用 death first search, 其空間複雜度為 O(b*d) = O(36*d), 其中 b = number of branch, d = Depth of Game Tree

4. 操練要項

a. 資料結構

- 棋盤表示

自行定義 Board 類別如下:

```
class Board
{
public:
	int dice; // Number of dice
	bool color; //color (1:Red  0:Blue)
	int ChessPos[12] = {}; //Chess Position for each chess, Ex. ChessPos[0] = i means chess 0 is
in position i
	int * BChessPos = ChessPos; //Chess Position for blue chess
	int * RChessPos = ChessPos + 6; //Chess Position for red chess
	unsigned long BChess = 0; //bit borad for blue chess
	unsigned long RChess = 0; //bit borad for red chess
	int pBoard[25]; //1-D Representation of board
	Board(int * state) {//Constrouctor of Board
		略
	}
};
```

這裡為了加快處理速度, 用一維來表示盤面: (方格內值代表位置編號)

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

紅藍棋子的編號表示法如下: (藍棋用 1~6 號表示, 紅棋用 7~12 號表示)

7	8	9		
10	11			
12				6
			5	4
		3	2	1

根據上述圖示, chessPos[i]儲存了每個編號 i 棋子的位置, Ex. chessPos[i] = 19 代表編號 i 的棋子在第 19 號棋盤位置, pBoard[i]儲存了棋盤每個位置上的棋子編號, Ex. pBoard[i] = 6 代表棋盤位置 i 存放了藍棋編號 6 的棋子

其中值得一提的是 `unsigned long BChess` 與 `unsigned long RChess`, 這兩個 bit board 分別用來儲存當前盤面所有紅/藍棋的位置, 嘗試利用一個變數來 encoding 上述資訊, 目的是為了達到 O(1)的走步&吃子&選子處理, 還有另一個目的是為了方便加速後續 O(1)重複盤面搜尋(後面關於重複盤面判斷的章節會有詳細介紹, 這邊先介紹資料結構表達方式), 這邊宣告成 unsigned long 的原因是因為一維的棋盤表示總共有 1~25 個位置, 而 long 有 32 個 bit 足夠讓我們存放棋子, 宣告 unsigned 是為了讓 bit operation 邏輯運算時, 不會出現負數這種錯誤

Ex.

RChess = 0000100000…0000000010, 令 bit 0~31 (25~31 bit 空著不用) 分別代表棋盤的位置, 則 RChess 第 i 個 bit 為 1 代表棋盤位置 i 有一顆紅棋

BChess = 0000100000…0000000010, 令 bit 0~31(25~31 bit 空著不用) 分別代表棋盤的位置, 則 BChess 第 i 個 bit 為 1 代表棋盤位置 i 有一顆藍棋

- 盤面的表示/判斷重複盤面

要辨別每一個盤面是否重複, 我們會需要三個資訊:

1.當前盤面所有藍棋的位置　2.當前盤面所有紅棋的位置　3. 當前盤面輪到誰先下

如果有一個盤面上述三個資訊都一樣, 則其最後的期望收盤分數必定相同, 有了上述上個資訊我們就可以辨別各種特殊盤面, 因此這邊使用了一個 HashMap 來儲存重複盤面: (為了達到 O(1) Search time, 加快判別重複盤面

的時間)

```
// tuple[0]: BChess tuple[1]: RChess tuple[2]: color

unordered_map<tuple<unsigned long, unsigned long, bool>, float, tuple_hash> nodeMap;
```

tuple[0] = 當前所有藍棋的位置, tuple[1] = 當前盤面所有紅棋的位置 tuple[2] = 當前執子顏色

其中當前所有藍棋的位置即前面提到的 Board.BChess, 當前盤面所有紅棋的位置即前面提到的 Board.RChess

自行定義 hash function 計算函式如下:

```
//Define hash of each node 的計算, 用來計算與儲存重複盤面
typedef std::tuple<unsigned long, unsigned long, bool> key_t;
struct tuple_hash : public std::unary_function<key_t, std::size_t>{
	std::size_t operator()(const key_t& k) const
	{
		return std::get<0>(k) ^ std::get<1>(k) ^ (unsigned long)std::get<2>(k);
	}
};
```

利用 hash map 搜尋重複盤面的平均時間複雜度理論值 = O(1), 不會大幅度影響到處理速度

- 走步的表示

利用 Move Class 來記錄每一手的起點位置與終點位置, from 為移動棋子的當前位置, to 為當前棋子的移動終點

```
//Move Class is used to represent a valid chess move
class Move {
public:
	int from; //start position
	int to; //End position
	Move(int _from, int _to) { //Move constructor
		from = _from;
		to = _to;
	}
};
```

- **輸出：**
- **移動起點 移動終點**
  - 23 18

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

另外也事先定義了各種棋子可能的走步位移量(一維表示):

```
//direction[0] possible shift for blue chess => -1:左 -6:左上 -5: 往上
//direction[1] possible shift for red chess => 1:右 6:右下上 5: 往下
int direction[2][3] = { { -5, -6, -1 },{ 5, 6, 1 } };
```

- 審局函數表示與說明

審局函數公式:

紅棋: HeuristicValue(Red) - HeuristicValue(Blue)

藍棋: HeuristicValue(Blue) - HeuristicValue(Red)

HeuristicValue 越高代表越佔優勢, 對於藍其他的棋局分數就是自己的棋局分數減去敵方的棋局分數, 因此審局函數回傳的值越高代表勝算越大, 此種計算方式號處是能同時考慮攻守兩方的優勢, 而非單從己方兵力作審局函數的估算

而 HeuristicValue(Color) = 所有棋子的 Heuristic Table Value 加總

為了加快審局函式運算速度, 已經事先定義好各個棋子所在位置(一維表示法)所能得到的 Heuristic Table Value (分數越大代表越有利), 舉例 BEvalValue[i]代表藍棋在位置 i 能夠有 BEvalValue[i]的分數得分

```
//Heuristic value for Blue Chess
float BEvalValue[25] = { 16, 10, 5, 2.5, 1,
                         10, 8, 4, 2, 1,
                         5, 4, 4, 2, 1,
                         2.5, 2, 2, 2, 1,
                         1, 1, 1, 1, 0 };
//Heuristic value for Blue Chess
float REvalValue[25] = {0, 1, 1, 1, 1,
                         1, 2, 2, 2, 2.5,
                         1, 2, 4, 4, 5,
                         1, 2, 4, 8, 10,
                         1, 2.5, 5, 10, 16 };
```

Heuristic Table Value 計算方式如下: [Reference 3]

令 Distance = 為該棋子到終點的最短距離, 則 HeuristicValue (HV) = $2^{4-Distance}$, 這樣子的計算方式考慮到棋子越靠近終點越佔上風, 所以距離越近給了越大的分數

然而經過一些盤面測試這樣的計算方式仍然不夠完美, 因為越靠近邊界上的棋子越有優勢, 因為他難以被敵方吃子, 所以針對邊上的棋子我作了一個加乘 1.25 的動作, 反映出它的價值

最終發現在原點附近的棋子價值還是要在更低才能在大多數盤面反映他的真實價值, 因為真正影響戰局的通常是最靠進終點前幾名的棋子, 所以我將原點的棋子分數設為 0, 並且取消最靠近原點邊上的 1.25 加乘, 最終成了上述實測效果最好的 heuristic value

b. 走步/吃子步如何產生

- 決定移動棋子

如果骰子對應到的棋子存在, 則直接選定該顆棋子, 如果骰子對應到的棋子不存在, 則找存在且編號最接近骰子編號的棋子, 程式碼如下: (最壞狀況可能會檢查五顆棋子 = O(5))

```
void findPossibleChess(int * posChess) {
	int dice = (board.color == BLUE) ? board.dice : board.dice + CHESS_NUM;
	if (board.ChessPos[dice] >= 0) {
		posChess[0] = dice;
	}else {
		int lowerBound = (board.color == BLUE) ? 0 : CHESS_NUM;
		int upperBound = (board.color == BLUE) ? CHESS_NUM : 2 * CHESS_NUM;
```

```
                for (int i = dice + 1; i < upperBound; ++i) {
                        if (board.ChessPos[i] >= 0) {
                                posChess[0] = i;
                                break;
                        }
                }
                for (int i = dice - 1; i >= lowerBound; --i) {
                        if (board.ChessPos[i] >= 0) {
                                posChess[1] = i;
                                break;
                        }
                }
        }
}
```

- 嘗試各種走步(吃子)

每一次走步必須更新藍紅棋子的資訊(即 Board.RChess 以及 Board.BChess), 更新程式碼如下: (origPos: 棋子移動前的位置, nextPos: 棋子移動後的位置)

```
if (board.color == BLUE) {
        board.BChess &= ~(1 << *origPos);
        board.BChess |= (1 << *nextPos);
        board.RChess &= ~(1 << *nextPos);
}else {
        board.RChess &= ~(1 << *origPos);
        board.RChess |= (1 << *nextPos);
        board.BChess &= ~(1 << *nextPos);
}
```

以藍棋作說明, 藍棋移動要:

1. 清除原先位置的資訊 => board.BChess &= ~(1 << *origPos)

2. 在新的棋盤位置立標誌 => board.BChess |= (1 << *nextPos)

3. 將新位置的紅棋摘除(吃子) => board.RChess &= ~(1 << *nextPos);

另一方面也必須同時更新, 骰子編號對應到的位置(即 Board.ChessPos[i]), 同時記錄吃掉的棋子:

```
board.ChessPos[chessId] = *nextPos;

eatenChess = board.pBoard[*nextPos];

board.pBoard[*nextPos] = chessId;
board.pBoard[*origPos] = -1;
```

最後將棋子顏色作更換, 代表輪到另一顏色方下棋

board.color = !board.color;

上述動作(移動子/吃子)可以於 O(1)時間複雜度內完成

- 決定最佳走步

因為每一顆子有三個可能走法, 所以每一層 min/max node 最多都會得到三個 Return Value, 此 Value 即每一種走法最後可能導致的賽局分數(對於該層 min/max node), 因此 min/max 方均會挑選最大的 Value 對應到的走步作為他的的最佳走步, 以藍棋為例, 如果往上, 左上, 左走得到的 Value 分別是 1, 2, 3, 則藍棋就會選擇走往左走

c. 要跑多深(Cut-Off 條件)

- 棋局結束判斷

第一種 Cut-Off 的條件為當棋局結束時, 而棋局結束有下列 2 種可能:

1. 藍/紅棋其中一方棋子全被吃光
2. 藍/紅棋其中一方到達終點

根據如上的條件以及先前定義的資料結構, 定義 Cut-Off 函式來判斷之: (紅字部分)

```
bool CutOff_Test(int depth) {
	return (board.BChess & 1) || (board.RChess & (1 << 24)) || (depth == 0) || (board.RChess == 0) || (board.BChess == 0);
}
```

其中`(board.BChess & 1) || (board.RChess & (1 << 24))` 判斷藍/紅棋是否有其中一方到達終點, 而 `(board.RChess == 0) || (board.BChess == 0)` 判斷藍/紅棋是否其中一方棋子全被吃光

- 思考時間限制

由於愛因斯坦棋的正式比賽有 30 秒思考時間的限制, 為了避免 AI 程式逾時作負, 我預先測試了不同深度所需要的時間, 讓 game tree 避免發展過深而運算時間超過 30 秒, 以下是實驗數據: (均以 worst case 下去測試運算時間, 也就是剛開局, 走步可能分支最多的情況下去測量)

Depth	Time (s)
4	<1
5	7
6	65

由於最大 Game Tree 設定在 5 和 6 的運算時間差距過大, 在 30 秒思考時間的限制下, 將最大深度設定在 5 來避免因 time out 而輸棋

d. 會不會逾時作負

根據上一個章節的分析, 如果將 game tree 的最大深度設定在 5 (也就是 AI 最多只考慮五步內局勢變化), worst case 下演算法平均只會需要 7~8 秒左右的時間, 所以實戰測試沒有遇過因逾時而作負的狀況

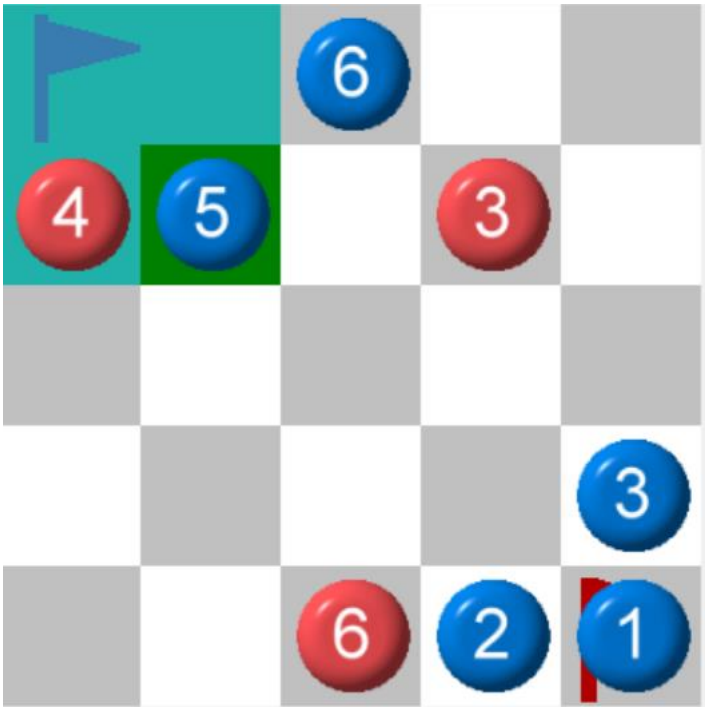
e.記憶體會不會爆掉

由於 Expect minimax 採用 depth first search, 空間複雜度為 O(b*d) = O(36*d), 遠優於 breadth first search 之空間複雜度 $O(36^d)$, 加上最大 game tree 深度設定在 5, 按理論來說不會爆掉, 實戰數場後也沒也遇到記憶體耗盡的問題, 以下是實驗數據: (均以 worst case 下去測試運算時間, 也就是剛開局, 走步可能分支最多的情況下去測量)

Depth	Memory (MB)
4	1
5	4
6	15

5.盤面測試

- 盤面設定 1: 藍棋執子, 藍棋個數 = 4, 紅棋個數 = 3, 骰子 = 5, 藍棋大優, 有一步的解法可以贏賽局



耗用 Memory: 1MB

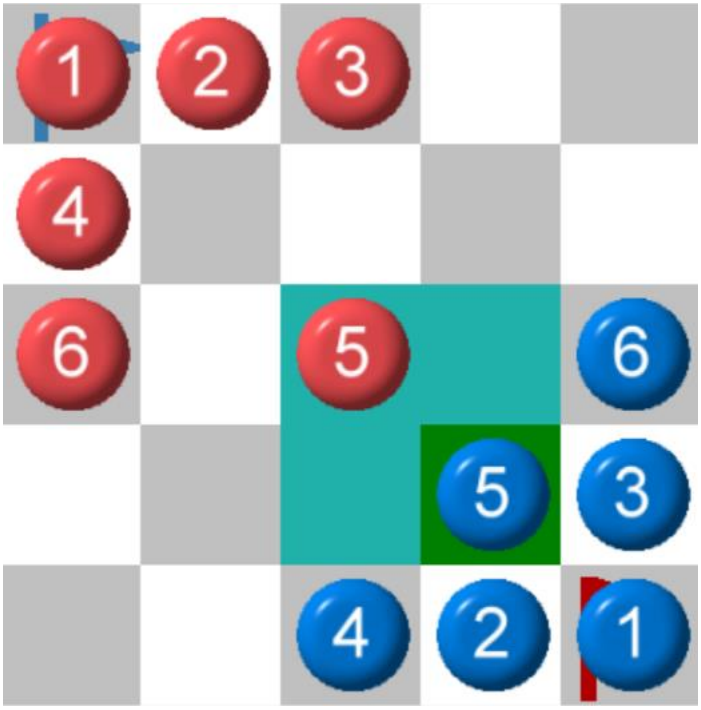
耗用 Time: 1s

預估期望賽局分數: 往上:9.57022　　往左上: 15　　往左: 11.7083

AI Decision: 移動棋子 5 往左上

說明: 藍棋只要往左上即可結束比賽, 所以應該往左上走, 與 AI 判斷的結果一致, 往左上方向可以獲得最高的期望賽局比分

- 盤面設定 2: 藍棋執子, 藍棋個數 = 6, 紅棋個數 = 6, 骰子 = 5, 剛開盤沒多久, 藍棋往左上看起來是最佳步



耗用 Memory: 2MB

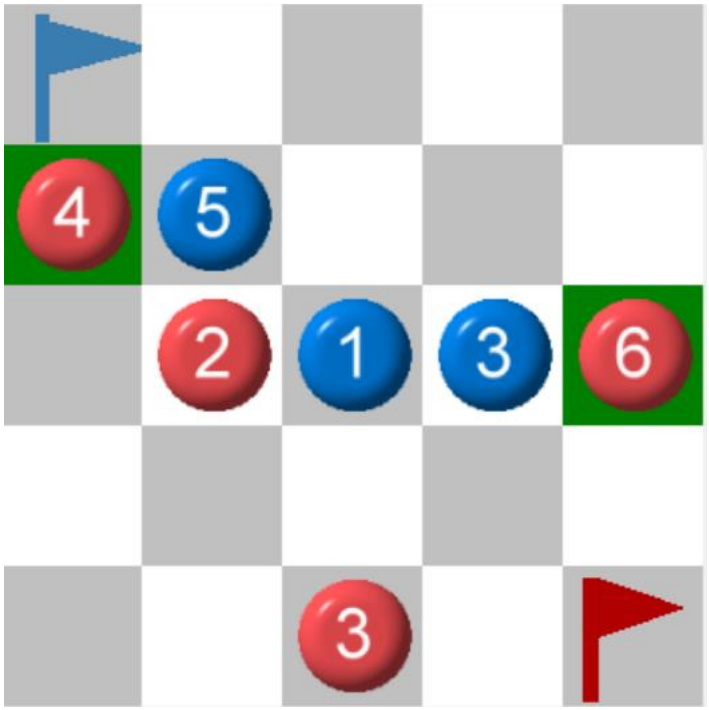
耗用 Time: 2s

預估期望賽局分數: 往上: -0.38233　　往左上: 4.04089　　往左: -0.340492

AI Decision: 移動棋子 5 往左上

說明: 藍棋只要往左上不但可以吃掉紅子, 還是最縮短與終點距離的一步, 所以可以看出我們往左上所得到的期望賽局分數最大, 籃子會往左上走, 合理

- 盤面設定 3: 紅棋執子, 藍棋個數 =3, 紅棋個數 = 4, 骰子 = 5, 接近尾盤, 紅方局勢迫在眉睫



耗用 Memory: 2MB

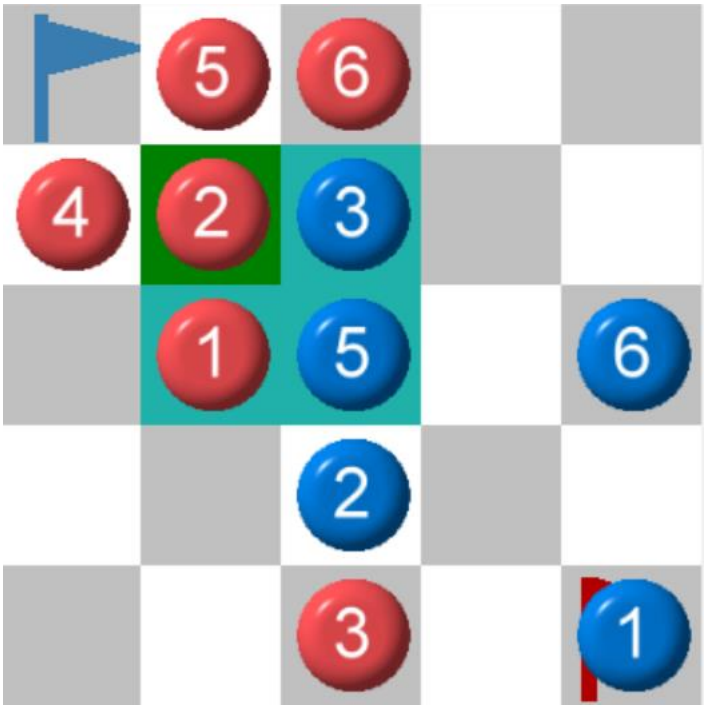
耗用 Time: 7s

預估期望賽局分數: 往下: -0.225284　　往右下: -1.38683　　往右: -0.128211

AI Decision: 選擇棋子編號 4 往右

說明: 由於骰子點數為 5, 但編號 5 的紅棋已經被吃掉, 因此紅方可以選擇移動編號 4 or 6 的棋子, 最後 AI 選擇了編號 4 往右, 吃去藍方編號 5 的棋子, AI 這個判斷相當合理, 因為如果不這樣做, 萬一下回合藍方擲出骰子 4,5,6 都會導致紅方輸棋, 因此紅方選擇移動編號四棋子吃掉藍方棋

- 盤面設定 4: 紅棋執子, 藍棋個數 =5, 紅棋個數 = 6, 骰子 = 2, 中盤雙方目測勢均力敵



耗用 Memory: 2MB

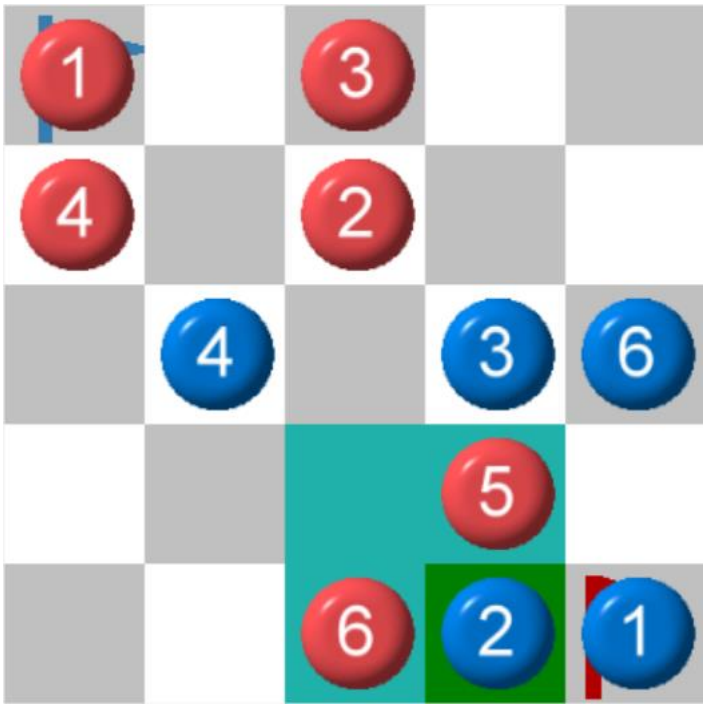
耗用 Time: 8s

預估期望賽局分數: 往上: -5.56711　　往右下: -1.66114　　往右: -3.83843

AI Decision: 選擇棋子編號 2 往右下

說明: AI 走的這步棋也是合理, 往右下可以讓編號 2 的紅子最大程度往終點靠, 還能吃掉編號 5 的藍子, 但 AI 似乎認為藍方較佔優勢, 原因是藍子較多的棋子靠近終點, 而紅方只有一顆編號 3 棋子較靠近終點

- 盤面設定 5: 藍棋執子, 藍棋個數 =5, 紅棋個數 = 6, 骰子 = 2, 中盤紅方略佔優勢



耗用 Memory: 6MB

耗用 Time: 13s

預估期望賽局分數: 往上: -4.14497　　往左上: -11.8044　　往左: -8.64907

AI Decision: 移動棋子 2 往上

說明: 藍棋只往左可以最大限度縮短與終點的距離, 但兩顆紅子已經兵臨城下, 如果不先吃掉其中一顆紅棋輸的機率非常大, 而 AI 選擇優先吃掉離終點只剩一步的紅棋 5, 看下來也是個合理的選擇

- 結論:

目前的 AI 下棋程式從測試盤面看來, 他不只會優先考量棋子與終點的距離, 也會考慮防守來避免輸棋, 也會想辦法減少對方棋子的數量(吃子獲勝), 這是由於審局函數同時考量了紅藍方的贏面, 兩者作相減, 目前我自己與自己開發的 AI 下棋, AI 勝率大概有 80%, 如果能夠再加快 AI 運算執行速度, 思考更多步棋後的盤面, 也許棋力可以再上升

6.遇到困難

1.程式執行速度不夠理想, 看了林品儒的論文發現要加快速度最好還要導入平行運算的 solution, 超頻 CPU, 以及對 expect mini-max algorithm 實行更複雜的 alpha-beta pruning, 並且可能最好能導入一些硬體運算的內建指令, 但由於我的經驗和時間受限, 尚無法仿效論文內的作法來進一步提升速度, 目前也只能針對演算法的時間複雜度作一些優化(利用 bit board), 導致程式效能不太理想, 比賽有 30 秒的限制, 所以我目前的程式只能考慮到五步以後的盤面, 因此棋力也受到限制

2. 由於我自己也是愛因斯坦棋的新手, 在 debug 與提升棋力的開發過程中, 有時難以判斷 AI 走出的棋子是好是壞, 導致我只能針對很誇張的爛棋作改良, 要再進一步提升 AI 的棋力, 我可能還要多找幾個人家開發的高棋力 AI 多多對局, 並且同時增進自己對盤面局勢的判斷, 再進一步去想棋力的改良與增進(審局函數的研究是很大關鍵)

3.這次的 debug 難度較作業二小蜜蜂又更高, 遞迴呼叫加上各種位元盤面以及加速運算, 導致程式碼更錯綜

複雜難以追縱, 也降低了可讀性, 造成開發速度的緩慢, 而且有許多 bug 難以被發現, 應該要想一些更有效率的開發手法來解決此問題

Reference

1. Minmax pseudo code - https://en.wikipedia.org/wiki/Expectiminimax
2. 林品儒碩士論文, 增強學習架構與期望最小最大算法之
3. 審局函數的改良 - 中國發明專利 CN105677923A 爱恩斯坦棋基于攻防兼备估值函数的博弈搜索方法
4. 棋局規則 - 林品儒愛因斯坦棋介紹 PTT
5. 棋局規則 - 愛因斯坦棋開發平台 – ECUI 說明手冊
6. Expect minimax 演算法 - 台師大資工系 人工智慧 CH5 上課投影片 (林順喜老師授課)