自學JavaScript並學習撰寫西洋棋引擎

國立新竹高中

蔣忻勳

目錄

摘要

在本次自主學習中，我以了解西洋棋引擎演算法原理為目標，觀看教學影片學習使用JavaScript撰寫西洋棋引擎。在過程中，我學到了：

1.學會西洋棋引擎的運作原理

2.學習真正的工程師寫程式的方法、習慣

3.體驗製作網頁、運用CSS、HTML檔

途中，我花費了許多心力，最終了解西洋棋引擎是使用AlphaBeta演算法運算最佳步數。因為AlphaBeta的效率與計算順序相關，我們須加入其他技術增加其效率。

成品網址：

https://ian0520.github.io/JSChess/


```
Game Board:
8   r  n  b  q  k  b  n  r
7   p  p  p  p  p  p  p  p
6   .  .  .  .  .  .  .  .
5   .  .  .  .  .  .  .  .
4   .  .  .  .  P  .  .  .
3   .  .  .  .  .  .  .  .
2   P  P  P  P  .  P  P  P
1   R  N  B  Q  K  B  N  R

    a  b  c  d  e  f  g  h
side:b
enPas:45
castle:KQkq
key:21b7222e
D: 1 Best: d7d5 Score: 0 nodes: 25 Pv:  d7d5
D: 2 Best: d7d5 Score: -25 nodes: 123 Pv:  d7d5 f1d3 Ordering:86.67%
D: 3 Best: e7e5 Score: 0 nodes: 1231 Pv:  e7e5 d2d4 d7d5 Ordering:88.32%
D: 4 Best: e7e5 Score: -25 nodes: 5767 Pv:  e7e5 d2d4 d7d5 c1e3 Ordering:86.58%
D: 5 Best: e7e5 Score: -5 nodes: 33526 Pv:  e7e5 d2d4 b8c6 g1f3 f8d6 Ordering:86.79%
D: 6 Best: e7e5 Score: -25 nodes: 221637 Pv:  e7e5 g1f3 b8c6 d2d4 f8b4 Ordering:85.27%
D: 7 Best: e7e5 Score: -5 nodes: 1075446 Pv:  e7e5 Ordering:86.30%
```

學習動機

西洋棋是我最主要的休閒活動之一，我非常喜歡利用休閒時間研究西洋棋開局和線上與其他玩家遊玩。遊玩後，我總會使用西洋棋引擎Stockfish分析棋局，看我哪裡下得不好。我經常好奇，這些西洋棋引擎是如何在短時間內分析15甚至20步之後的可能性，並判斷每一步的價值、選出最好的一步，於是我想透過學習寫西洋棋引擎了解其背後的運作原理，同時學習JavaScript、練習寫程式技巧。

Stockfish

程式語言方面，我選用JavaScript，原因是我希望能將我的成果透過網頁與其他人分享，而JavaScript搭配HTML、CSS即能簡單架設網站。此外，JavaScript也是鮮少有機會在學校學習的語言然而卻十分泛用，因此我想透過這次學習，練習使用JavaScript，也練習架設簡單的網站，使JavaScript成為未來能使用的程式語言。

學習計畫

學習基礎JavaScript語法

準備

為了避免之後太多語法不懂
先觀看YouTube影片學習
JavaScript常用的語法

學習撰寫西洋棋引擎 → 遇見困難

學習

跟著影片打程式
盡可能理解每一行
程式碼的意義

西洋棋引擎方面：
參考
Chess Programming Wiki

JavaScript方面：
上網查詢 或
回到上一步的影片

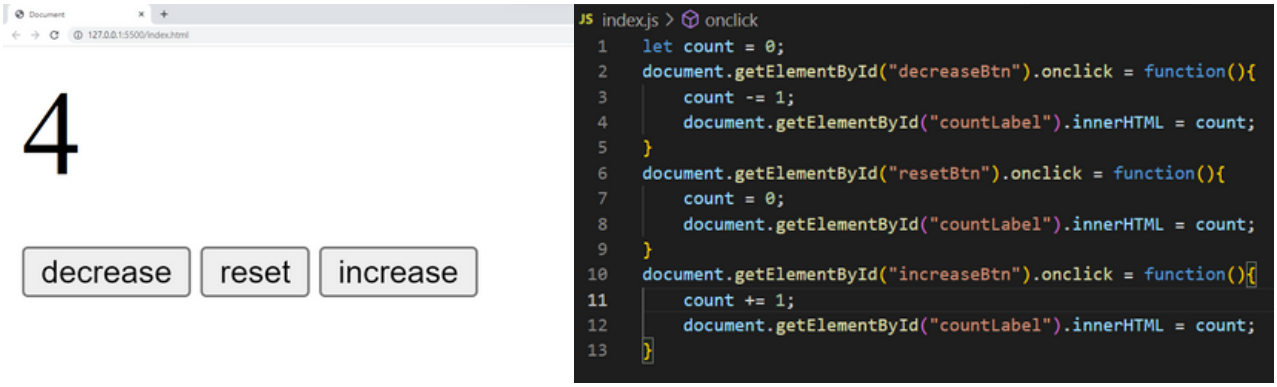
準備

為了在寫西洋棋引擎時不會太多語法看不懂，我決定觀看教學影片，先對JavaScript有一定的了解。

我選用Youtube上Bro Code的：
JavaScript Full Course for free　(2023)
https://www.youtube.com/watch?v=8dWL3wF_OMw&ab_channel=BroCode

我選這部影片是因為我認為其內容十分完整，且作者講話滿生動的，可以讓我學習語法的同時不感到枯燥乏味。

此外，此影片有分明的時間標籤，因為影片長達八小時，如果看完會花費過多時間，因此我決定先看前1小時熟悉最基本的語法，等之後遇到新的語法再使用時間標籤跳到需要的地方。



練習用的Counter程式輸出結果 / 程式碼
我從中練習了簡單的JavaScript、CSS、HTML連動

學習

Youtube上以JavaScript製作西洋棋引擎的教學其實不算少，經過瀏覽之後，我選擇
Bluefever Software的：
Programming A Javascript Chess Engine播放清單
https://www.youtube.com/playlist?list=PLZ1QII7yudbe4gz2gh9BCI6VDA-xafLog

他是我找到的影片教學中把程式碼解釋的最完整的，雖然他在影片說他的專業是C++，但我認為身為一個初學者，向不是最專業的人學習反而更能跟上腳步，也更清楚什麼地方容易犯錯。雖然影片註解都有提供那部影片的程式碼，但我選擇跟著影片一起打程式碼，以更好理解每行程式碼所代表的意義。

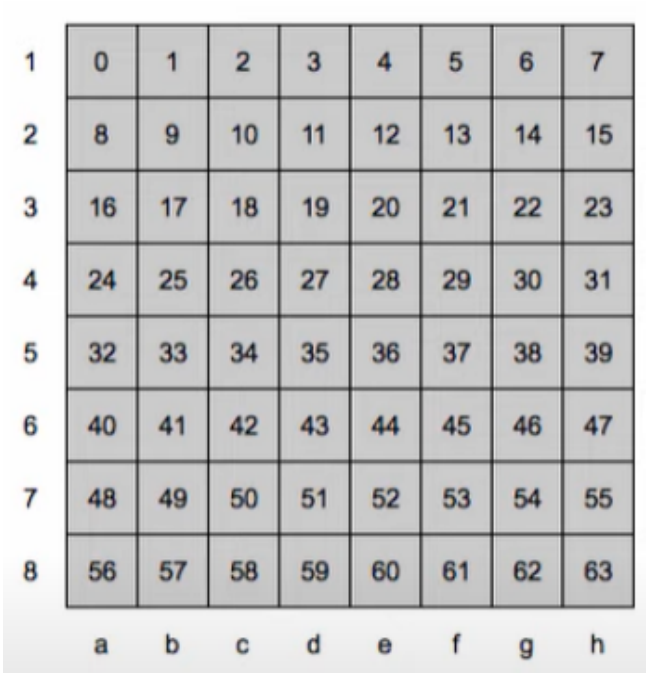
接下來我會以：
1.設置棋盤、規則、步數
2.尋找最佳步數的演算法
3.GUI
的順序記錄我的所學，其中我希望將重心放在第二步驟，因為這也是我最想了解的地方。

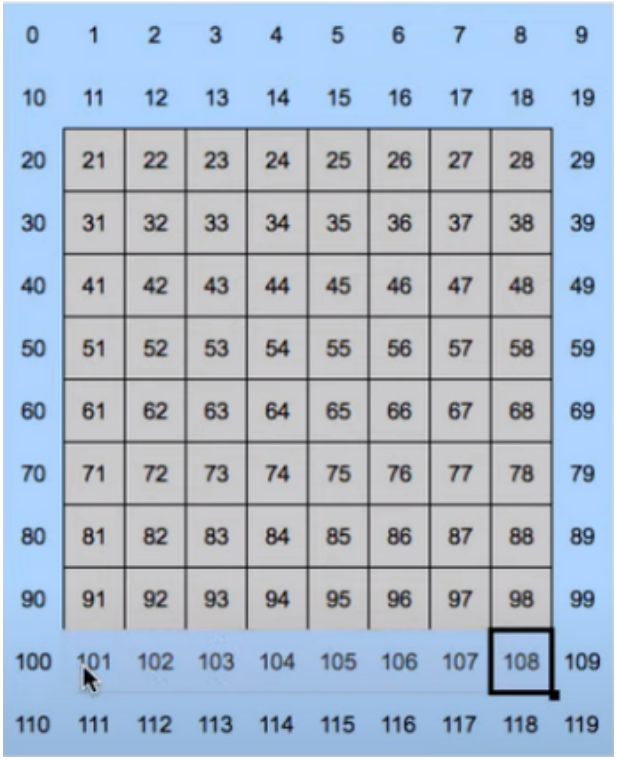
設置棋盤、規則、產生步數

一、設置棋盤：

一開始我們將格子分為兩種形式：

1. 編號0~63，如真實的棋盤(左圖)
2. 編號0~119，將上下邊界多出兩列，一列改為10格(右圖)
   再將格子分為棋盤內(灰色)、棋盤外(藍色)





使用120格是因為稍後要產生騎士的步數(前進兩格，往左或往右一格)時是使用：

當前的格子 + (-8、-19、-21、-12、8、19、12或21)

再判斷那個格子是否是空的，若使用0~63制，可能會判斷到不存在的格子，容易出錯，因此我們改用右邊的方法，再判斷目標格子是否在棋盤範圍內。

之後，我們要將西洋棋初始位置每個格子的各種資訊儲存至GameBoard的不同陣列裡，這裡用到了西洋棋FEN(Forsyth-Edwards Notation)紀錄棋子位置技巧。其實在這之前，我也不知道這樣的記法。

FEN：

上網查詢後，得知這是一種用一行字記錄空格、各棋子位置、50步規則、入堡資格、吃過路兵資格的方法，例如起始位置的記法：

rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1

代表了

第8列有黑城堡(小r)、黑騎士(小n)、...

第7列每格皆為黑小兵(小p)

第6~3列皆為8格空格，以數字表示

第2列每格皆為白小兵(大P)

第1列有白城堡(大R)、白騎士(大N)、...

之後w代表輪到白方

KQkq分別代表：

白方有王、后邊入堡資格

黑方有王、后邊入堡資格(國王、城堡未移動過)

0、1紀錄了現在為50步規則的第幾步

(若連50步雙方無吃子、移動小兵，則平手)

於是我們用這樣的規則，定義函式ParseFen(fen)，從左至右用count記錄第幾個字，使用switch語法，遇到不同的字元就做不同的指令。

```
while((rank >= RANKS.RANK_1) && fenCnt < fen.length){
    count = 1;
    switch (fen[fenCnt]){
        case 'p': piece = PIECES.bP; break;
        case 'r': piece = PIECES.bR; break;
        case 'n': piece = PIECES.bN; break;
        case 'b': piece = PIECES.bB; break;
        case 'k': piece = PIECES.bK; break;
        case 'q': piece = PIECES.bQ; break;
        case 'P': piece = PIECES.wP; break;
        case 'R': piece = PIECES.wR; break;
        case 'N': piece = PIECES.wN; break;
        case 'B': piece = PIECES.wB; break;
        case 'K': piece = PIECES.wK; break;
        case 'Q': piece = PIECES.wQ; break;
```

每個格子上有什麼棋子 儲存於GameBoard.pieces[]
入堡資格 儲存於GameBoard.castlePerm[]
50步規則記錄 儲存於GameBoard.fiftyMove
...

這是我第一次接觸到switch語法：
偵測目標的值，根據值的不同做出不同的動作。
我覺得這是非常方便的語法，可以取代一長串的if、else
我想未來應該會很常用到。

這個步驟中，我最印象深刻的是紀錄入堡資格的方法，作者用四位2進位數字表式白/黑方 王/后入堡資格，白王0001，白后0010，黑王0100，黑后1000，最後再與實際狀況bitwise就能得知哪些正確，如此一來，只需一個Integer就能記錄4件事，算是讓我大開眼界。

我注意到這個作者的方法是，即使要多寫幾個函式，也要盡量讓寫程式的人方便一點，例如：

```
function FR2SQ(f,r){
    return ( (21 + (f) ) + ( (r) * 10));
}
```

這個函式是輸入行數、列數，轉換為120制格子的代號，有了這個函式，我們就不用不斷對照圖片，找到目標格子的代號，而只需直覺地輸入行數、列數。

```
var FILES = { FILE_A : 0, FILE_B : 1, FILE_C : 2, FILE_D : 3, FILE_E : 4, FILE_F : 5,
var RANKS = { RANK_1 : 0, RANK_2 : 1, RANK_3 : 2, RANK_4 : 3, RANK_5 : 4, RANK_6 : 5,
```

這個dictionary將行數、列數轉為西洋棋玩家熟悉的File A、B、C ，Rank 1、2、3制，如此一來，寫程式提到行、列時，就只需要寫FILES.FILE_A、FILES.FILE_B...就好

隨後的函數還有在console輸出目前棋盤PrintBoard、判斷格子是否被攻擊(入堡用途)的SqaureAttacked...等等，但我希望能將重點篇幅放在演算法的部分，所以想省略記錄這些我認為較為簡單的函式。

```
Game Board:

8   r  n  b  q  k  b  n  r
7   p  p  p  p  p  p  p  p
6   .  .  .  .  .  .  .  .
5   .  .  .  .  .  .  .  .
4   .  .  .  .  .  .  .  .
3   .  .  .  .  .  .  .  .
2   P  P  P  P  P  P  P  P
1   R  N  B  Q  K  B  N  R

    a  b  c  d  e  f  g  h
```

console輸出起始位置結果

二、產生步數GenerateMoves：

作者這裡同樣利用了bitwise的技巧表示每個Move來自哪個格子、去哪個格子、是否吃子(及目標棋子的價值)、是否為入堡、是否為升變(及變成目標的價值)。

```
var MFLAGEP = 0x40000;
var MFLAGPS = 0x80000;
var MFLAGCA = 0x1000000;

var MFLAGCAP = 0x7C000;
var MFLAGPROM = 0xF00000;
```

這裡用的是16進位(0x)，因為儲存：

來自的格子需7個bit

去的格子需7個bit(因為有64個格子)

吃子需4個bit(紀錄目標的價值)，

是否為過路兵各需1個bit

是否為小兵第一次移動(能動兩格)需1個bit

小兵升變需4個bit(紀錄升變目標價值)

是否入堡/入堡種類需4個bit

共28個bit，用16進位則只需7位數字

```
function FROMSQ(m) { return (m & 0x7F); }
function TOSQ(m) { return ((m >> 7) & 0x7F); }
function CAPTURED(m) { return ((m >> 14) & 0xF); }
function PROMOTED(m) { return ((m >> 20) & 0xF); }
```

利用這些函數判斷這個Move的性質

接下來我們要求電腦產生當前所有合法的步數，並儲存至GameBoard.moveList

```
var KnDir = [ -8, -19, -21, -12, 8, 19, 21, 12 ];
var RkDir = [ -1, -10, 1, 10];
var BiDir = [ -9, -11, 11, 9];
var KiDir = [ -1, -10, 1, 10, -9, -11, 11, 9 ];
```

先記錄所有棋子移動的方向(皇后與國王方向相同)，再將棋子區分為：

1. SlidePiece：無距離限制的棋子

不斷向旗子移動的方向增加格數，直到遇到另一個棋子或到棋盤外，新增移動或吃子的步數。

```
for(index = 0; index < DirNum[pce]; index++){
    dir = PceDir[pce][index];
    t_sq = sq + dir;
```

```
while(SQOFFBOARD(t_sq) == BOOL.FALSE){

    if(GameBoard.pieces[t_sq] != PIECES.EMPTY){
        if(PieceCol[GameBoard.pieces[t_sq]] != GameBoard.side){
            AddCaptureMove( MOVE(sq, t_sq, GameBoard.pieces[t_sq], PIECES.EMPTY, 0 ));
        }
        break;
    }
    AddQuietMove( MOVE(sq, t_sq, PIECES.EMPTY, PIECES.EMPTY, 0 ));
    t_sq += dir;
}
```

2. NonSlidePiece：騎士

向旗子的移動方向增加一格，並判斷這格是否有棋子(棋子為哪邊)、是否在棋盤外新增移動或吃子的步數。

```
while (pce != 0){
    for(pceNum = 0; pceNum < GameBoard.pceNum[pce]; ++pceNum){
        sq = GameBoard.pList[PCEINDEX(pce, pceNum)];

        for(index = 0; index < DirNum[pce]; index++){
            dir = PceDir[pce][index];
            t_sq = sq + dir;

            if(SQOFFBOARD(t_sq) == BOOL.TRUE){
                continue;
            }

            if(GameBoard.pieces[t_sq] != PIECES.EMPTY){
                if(PieceCol[GameBoard.pieces[t_sq]] != GameBoard.side){
                    AddCaptureMove( MOVE(sq, t_sq, GameBoard.pieces[t_sq], PIECES.EMPTY, 0 ));
                }
            }

            else{
                AddQuietMove( MOVE(sq, t_sq, PIECES.EMPTY, PIECES.EMPTY, 0 ));
            }
        }
    }
```

3.小兵

(1)判斷是否為升變，看是否要變換為皇后

(2)判斷前方是否有棋子、是否為第一步，新增移動步數

(3)判斷右前、左前是否有對方棋子、

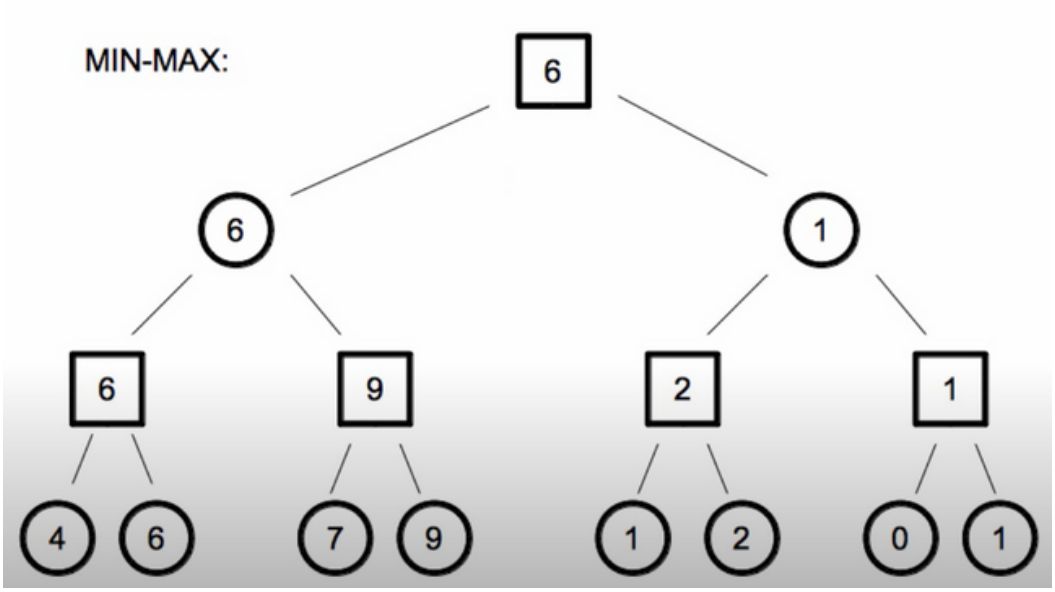
是否能吃過路兵，新增吃子步數

```
if(GameBoard.side == COLORS.WHITE) {
    pceType = PIECES.wP;

    for(pceNum = 0; pceNum < GameBoard.pceNum[pceType]; ++pceNum) {
        sq = GameBoard.pList[PCEINDEX(pceType, pceNum)];

        if(SQOFFBOARD(sq + 9) == BOOL.FALSE && PieceCol[GameBoard.pieces[sq+9]] == COLORS.BLACK) {
            AddWhitePawnCaptureMove(sq, sq + 9, GameBoard.pieces[sq+9]);
        }

        if(SQOFFBOARD(sq + 11) == BOOL.FALSE && PieceCol[GameBoard.pieces[sq+11]] == COLORS.BLACK) {
            AddWhitePawnCaptureMove(sq, sq + 11, GameBoard.pieces[sq+11]);
        }

        if(GameBoard.enPas != SQUARES.NO_SQ) {
            if(sq + 9 == GameBoard.enPas) {
                AddEnPassantMove( MOVE(sq, sq+9, PIECES.EMPTY, PIECES.EMPTY, MFLAGEP ) );
            }

            if(sq + 11 == GameBoard.enPas) {
                AddEnPassantMove( MOVE(sq, sq+11, PIECES.EMPTY, PIECES.EMPTY, MFLAGEP ) );
```

尋找最佳步數的演算法

一、MinMax 和 AlphaBeta

MinMax:

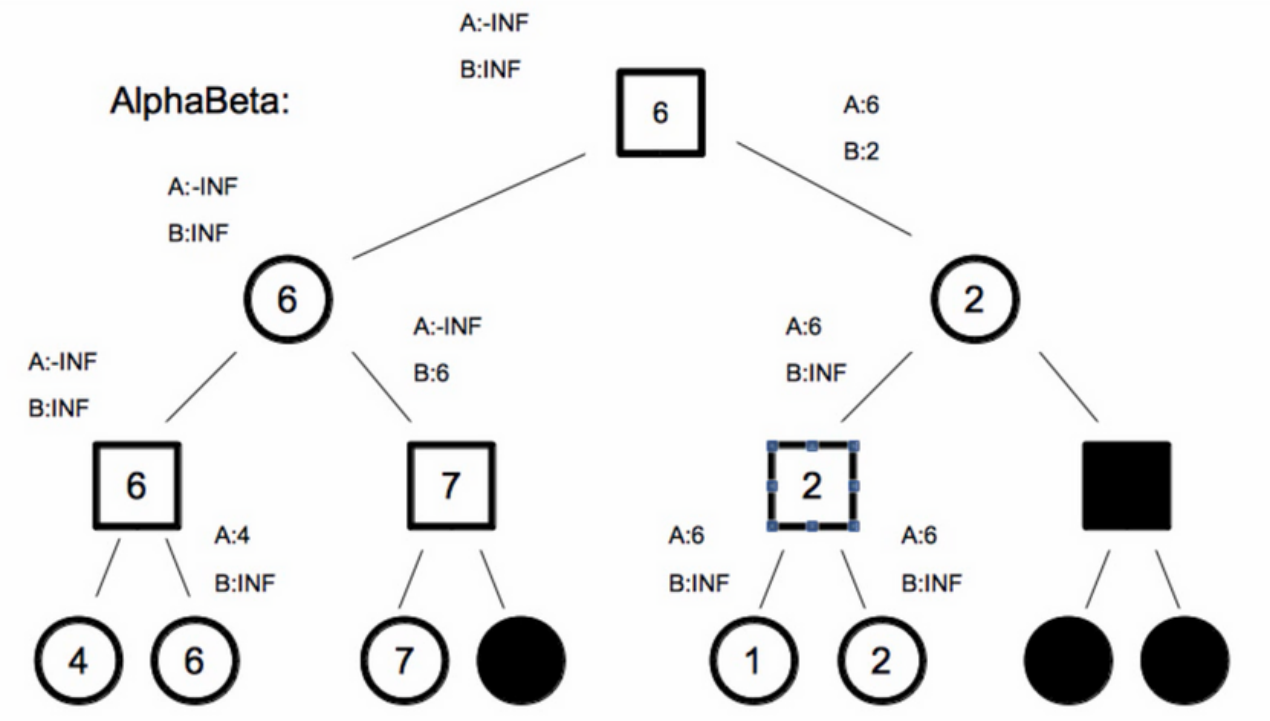


假設方形為白方，圓形為黑方，以MinMax搜尋法來看，
方形取分數最高的一項，也就是Maximizer；
圓形想取分數最低的一項，也就是Minimizer。
如上圖，由最下層開始看：
白方選擇了最高分的步數，紀錄在第二層；
黑方又選擇了分數最低的步數，記錄在第三層；
計算後，白方選擇了高分的左邊，這步即為6分。

這個方法不僅考慮了自己的選擇，更考慮了對方的選擇，可以有效決定哪個步數實際上最有效。

但此方法需要計算所有可能的步數，是十分龐大的計算量，我們應用AlphaBeta法減少搜尋量。

AlphaBeta :



在AlphaBeta法中，Alpha表示Maximizer能取得的最高分數，Beta表示Minimizer能取得的最低分數。

此方法中，白方一樣在第一層第一個分支選擇了分數較高的步數，但到第二個分支時，第一個選擇7已經大於剛剛得到的6(>Beta)，因此黑方在下一步不可能選擇此分支，所以就不用計算剩下的可能了。

另一邊，在右邊分支的第二層，我們已經知道有一可能為2，小於左邊得到的6(<Alpha)，因此白方不可能選擇右方的2，剩下的可能也同樣不需再執行。

這種方法可以減少需要搜尋的步數，但搜尋步數的排序也變得十分重要，從對自己最高分的步數開始搜尋可以最有效率地減少需搜尋的數量。此方法為我們選用的演算法。

二、評估場面的分數Evaluation

若要使用AlphaBeta，我們必須定義每一步的價值，但如果這一步不是吃子，我們就沒辦法依場上棋子的數量量化這步的分數，因此我們應衡量每個棋子在不同位置的價值。

我們用西洋棋原則為棋子在不同位置設定不同價值，例如：

```
var KnightTable = [
        0   ,   -10 ,   0   ,   0   ,   0   ,   0   ,   -10 ,   0   ,
        0   ,   0   ,   0   ,   5   ,   5   ,   0   ,   0   ,   0   ,
        0   ,   0   ,   10  ,   10  ,   10  ,   10  ,   0   ,   0   ,
        0   ,   0   ,   10  ,   20  ,   20  ,   10  ,   5   ,   0   ,
        5   ,   10  ,   15  ,   20  ,   20  ,   15  ,   10  ,   5   ,
        5   ,   10  ,   10  ,   20  ,   20  ,   10  ,   10  ,   5   ,
        0   ,   0   ,   5   ,   10  ,   10  ,   5   ,   0   ,   0   ,
        0   ,   0   ,   0   ,   0   ,   0   ,   0   ,   0   ,   0
        ];
```

這個Table表示了騎士在每個格子的價值，因為騎士在棋盤中央時能到達的格子比較多，因此價值較高。

值得注意的是，這個Table是給白方的，也就是為什麼第一列第二行的數字為-10(因為那是騎士原本的位置，以此鼓勵電腦移動騎士)，還需另一函數將Table顛倒供黑方使用。

另外還有一個被公認為優勢的因素，雙主教BishopPair，因為主教擁有比騎士更高的機動性，如果一方擁有雙主教，通常被認為是有小優勢的一方。

```
var BishopPair = 40;
```

三、應用AlphaBeta於程式中

1. 最基本的AlphaBeta：

如下兩張圖，我們先利用遞迴函式，找出最底層的分數(Score為負的且Alpha與Beta交換位置是輪替讓Maximizer和Minimizer選擇)

```
function AlphaBeta(alpha, beta, depth){
```

```
for(MoveNum = GameBoard.moveListStart[GameBoard.ply]; MoveNum < GameBoard.moveListStart[GameBoard.ply + 1];

    PickNextMove(MoveNum);

    Move = GameBoard.moveList[MoveNum];
    if(MakeMove(Move) == BOOL.FALSE){
        continue;
    }
    Legal++
    Score = -AlphaBeta( -beta, -alpha, depth-1);
```

```
if(depth <= 0) {
    EvalPosition();
}
```

我們知道，如果輪到Maximizer選擇，Alpha>Beta，這支分支就不必繼續計算，因此直接回傳Beta，否則，紀錄Alpha。

其中如果是第一步就搜到能剪去的分支，fhf(fail high first)就+1如果不是，fh+1，最後以fhf/fh計算效率。

```
if(Score > alpha) {
    if(Score >= beta) {
        if(Legal == 1) {
            SearchController.fhf++;
        }
        SearchController.fh++;

        return beta;
    }
    alpha = Score;
    BestMove = Move;
```

當計算完後，我們得到這一步的分數。

```
    return alpha;
}
```

---

2. 偵測是否被將軍：

當一方沒有合法步數的時候有兩種可能：

1. 被將軍，對方獲勝
2. 無子可動，平手

這兩種情況就不須評估場面，而是被將軍直接設為負無限大(這裡設為-29000)，無子可動設為0。

```
if(Legal == 0){ //Mated or not
    if(InCheck == BOOL.TRUE) {
        return -MATE + GameBoard.ply;
    } else {
        return 0;
    }
}
```

此外，當一方被將，很有可能接下來的步數會出現將軍，所以通常此時會提高搜尋的深度，提高贏的機會。

```
if(InCheck == BOOL.TRUE){
    depth++;
}
```

3. Quiescence Search：

經過前幾個步驟，並處理好儲存最佳步數(省略)後，我們已經可以讓引擎進行簡單的搜尋了：

```
D:1 Best:d2d4 Score:30 nodes:21 Pv: d2d4
D:2 Best:d2d4 Score:0 nodes:224 Pv: d2d4 d7d5
D:3 Best:d2d4 Score:30 nodes:2245 Pv: d2d4 d7d5 e2e4
D:4 Best:b1c3 Score:-10 nodes:26842 Pv: b1c3 d7d5 d2d4 e7e5
D:5 Best:e2e3 Score:85 nodes:230351 Pv: e2e3 e7e5 d1g4 f8d6 g4g7
```

然而由上圖可以看到，深度5的搜尋中，電腦想要盡快出皇后並在最後一步盡量吃一子，而不管對方的下一步，且評估的分數有很大的誤差。

上圖深度五搜尋後版面

這是視界限制效應(Horizon Effect)，意思是電腦到第五步就停止搜尋了，沒有考慮之後的可能性，就只評估五步後的分數，因此電腦想在第五步盡量吃一子。然而這樣非常危險，因為如果電腦在第五步用皇后吃了小兵而認為贏了一隻小兵，誤以為自己是優勢，皇后卻在第六步被吃了，因此實際上電腦是損失了一隻皇后。

為了防止這種情況發生，我們得使用Quiescence Search，也就是只用AlphaBeta搜尋吃子的步數，直到沒有吃子的情況發生，判斷那個情況的分數。

如此一來，如果電腦在第五步用皇后吃了小兵，他會再繼續看有沒有子能被吃，就不會忽略了自己的皇后也被吃的情況。

Quiescence Search的程式碼與AlphaBeta大同小異，只差在搜尋吃子，因此我不在此展示。

```
D:1 Best:d2d4 Score:30 nodes:21 Pv: d2d4
D:2 Best:d2d4 Score:0 nodes:207 Pv: d2d4 d7d5
D:3 Best:d2d4 Score:25 nodes:2680 Pv: d2d4 d7d5 c1e3
D:4 Best:d2d4 Score:0 nodes:17507 Pv: d2d4 d7d5 c1e3 c8e6
D:5 Best:e2e4 Score:25 nodes:217789 Pv: e2e4 e7e5 d2d4 d7d5 c1e3
```

可以看出搜尋正常許多了。

---

4. MVV-LVA：

MVV-LVA全名為：
Most Valuable Victim - Least Valuable Aggressor
用來設定，當被吃的子價值越高、吃子的子價值越低，這步的搜尋順序就越前面。

這樣能讓AlphaBeta搜尋時能計算更少分支，原因是在西洋棋中，大部分的時候，以越低價值的子吃越高價值的子，通常會越有優勢，因為吃子的子很有機會接下來也被對方吃，此時以低價值的子吃就代表放棄越少價值。

如下圖，在陣列MvvLvaValue中，每種棋子都有自己的MvvLva價值(索引值代表棋子種類)，另外有一陣列儲存每一種Attacker-Victim組合代表的價值。

```
var MvvLvaValue = [ 0, 100, 200, 300, 400, 500, 600, 100, 200, 300, 400, 500, 600 ];
var MvvLvaScores = new Array(14 * 14);
```

```
var PIECES =  { EMPTY : 0, wP :1,wN : 2, wB : 3,wR : 4,wQ : 5, wK : 6,
bP : 7, bN : 8, bB : 9, bR : 10, bQ : 11, bK : 12};
```

隨後我們使用函式InitMvvLva，設定每種組合的分數=Victim價值 + 6 - (Attacker價值 / 100)
Attacker/100 是因為被吃的子的價值還是比較重要，如果不分辨，那主教吃皇后和小兵吃主教就同分了。

```
function InitMvvLva(){
    var Attacker;
    var Victim;

    for(Attacker = PIECES.wP; Attacker <= PIECES.bK; ++Attacker){
        for(Victim = PIECES.wP; Victim <= PIECES.bK; ++Victim){
            MvvLvaScores[Victim * 14 + Attacker] = MvvLvaValue[Victim] + 6 -(MvvLvaValue[Attacker]/100);
        }
    }
}
```

這樣就符合被吃的子價值越高、吃子的子價值越低分數越高了。

```
D:1 Best:d2d4 Score:30 nodes:21 Pv: d2d4
D:2 Best:d2d4 Score:0 nodes:207 Pv: d2d4 d7d5 Ordering:60.00%
D:3 Best:d2d4 Score:25 nodes:2349 Pv: d2d4 d7d5 c1e3 Ordering:48.30%
D:4 Best:d2d4 Score:0 nodes:17903 Pv: d2d4 d7d5 c1e3 c8e6 Ordering:60.71%
D:5 Best:e2e4 Score:25 nodes:125503 Pv: e2e4 e7e5 d2d4 d7d5 c1e3 Ordering:67.33%
```

可以看到深度5的分支數量少計算了90000左右。

5. Killer Heuristic / Killer Move：

Killer Move是一種非吃子，但分數很高而能在AlphaBeta搜尋時剪裁掉其他分支的步，而這種步通常並存於些許不同的場面。例如：

分析左圖的場面，
電腦認為小兵到a6(箭頭)，
攻擊主教是好的步。

經過雙方下了：
小兵到d6、
小兵到d3後，電腦仍認為a6是分數很高

由上面例子可以看出，即使場面有變化，同樣的步數通常效果會差不多好，這就是Killer Move。但是這種非吃子步在AlphaBeta的順序十分後面，因此我們應該將這種步往前挪，這種方法稱為Killer Heuristic。

於是我們在尋找剪裁點時多加一個條件，如果這一步不是吃子，就把他加進SearchKiller陣列中，並把上一個Killer往後挪。

```
if(Score > alpha){
    if(Score >= beta){
        if(Legal == 1){
            SearchController.fhf++;
        }
        SearchController.fh++;
        if((Move & MFLAGCAP) == 0){
            GameBoard.searchKillers[MAXDEPTH + GameBoard.ply] =
                GameBoard.searchKillers[GameBoard.ply];
            GameBoard.searchKillers[GameBoard.ply] = Move;
        }
```

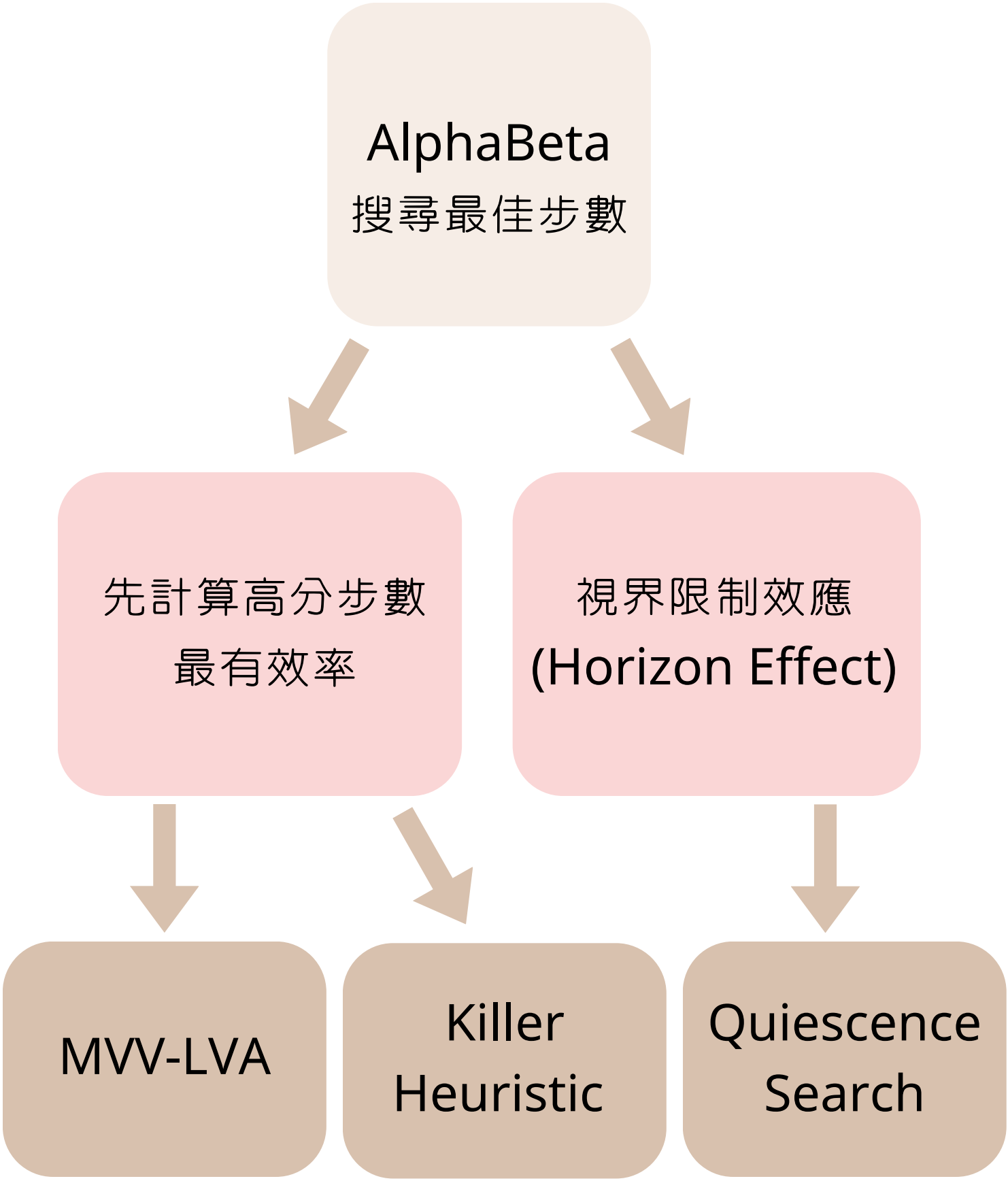
現在我們在尋找非吃子步時，就可以看這步是不是其中一個Killer Move，並把順序往前挪。

```
function AddQuietMove(move){
    GameBoard.moveList[GameBoard.moveListStart[GameBoard.ply+1]] = move;
    GameBoard.moveScores[GameBoard.moveListStart[GameBoard.ply+1]] = 0;

    if(move == GameBoard.searchKillers[GameBoard.ply]){
        GameBoard.moveScores[GameBoard.moveListStart[GameBoard.ply+1]] = 900000;
    } else if(move == GameBoard.searchKillers[GameBoard.ply] + MAXDEPTH){
        GameBoard.moveScores[GameBoard.moveListStart[GameBoard.ply+1]] = 800000;
    } else {
        GameBoard.moveScores[GameBoard.moveListStart[GameBoard.ply+1]] =
            GameBoard.searchHistory[GameBoard.pieces[FROMSQ(move)]* BRD_SQ_NUM + TOSQ(move)];
    }
    GameBoard.moveListStart[GameBoard.ply+1]++
}
```

可以看到現在效率已經高達80%以上了

```
D:1 Best:d2d4 Score:30 nodes:21 Pv: d2d4
D:2 Best:d2d4 Score:0 nodes:89 Pv: d2d4 d7d5 Ordering:90.91%
D:3 Best:d2d4 Score:25 nodes:638 Pv: d2d4 d7d5 c1e3 Ordering:88.24%
D:4 Best:d2d4 Score:0 nodes:3693 Pv: d2d4 d7d5 c1e3 c8e6 Ordering:85.79%
D:5 Best:e2e4 Score:25 nodes:24250 Pv: e2e4 e7e5 d2d4 d7d5 c1e3 Ordering:84.24%
D:6 Best:e2e4 Score:5 nodes:114957 Pv: e2e4 e7e5 d2d4 b8c6 Ordering:81.89%
```

整理


AlphaBeta
搜尋最佳步數
先計算高分步數
最有效率
視界限制效應
(Horizon Effect)
MVV-LVA
Killer
Heuristic
Quiescence
Search

GUI

因為本次學習目標主要是研究西洋棋的演算法，所以我只如影片使用最簡單的界面。

這個部分，我跟著作者使用了許多JQuery的語法，例如

```
function DeSelectSq(sq){
    $(".Square").each( function(index){
        if(PieceIsOnSq(sq, $(this).position().top, $(this).position().left) == BOOL.TRUE){
                $(this).removeClass('SqSelected');
            }
    });
}
```

在這個取消選擇格子的函式中，我用到了$().each和$.()removeClass()來選擇

```
function ClearAllPieces(){
    $(".Piece").remove();
}
```

這個語法是我覺得最方便的，他把GUI上所有的棋子清除，卻只用到了一行程式碼。

經過搜尋，我學到JQuery就是用來幫助撰寫JavaScript程式的，而其中主打的就是很方便的選擇器，我也深有同感，因為在第一階段學習JavaScript語法的影片中，作者是用GetElementById、onclick...等等的語法，真的很長，有了JQuerry就不用再打這些了。

在設置格子、設置棋子、點擊功能、按鈕功能後，一個能與西洋棋引擎對弈的網頁就完成了！

成品連結：

https://ian0520.github.io/JSChess/

心得、收穫

這是我高中三年來最大的程式設計作品，從看影片學習JavaScript、寫西洋棋引擎程式到理解其所有原理，我花費了很多心思。

其中我覺得最大的收穫有：

1.學會西洋棋引擎的運作原理

西洋棋是我最主要的休閒活動之一，在使用網路上的引擎時，我總會好奇其運作的過程，我認為這次的學習算是解開了心中的一個謎。

2.學習真正的工程師寫程式的方法

這次學習的對象是個真正的工程師，而我在看影片中的講者寫程式時，我發現實際上他們會先將函式都定義好，最後再一一放進主程式裡，而且會將不同種類的函式分放到不同的檔案，更好整理、修改。

3.體驗製作網頁、運用CSS、HTML檔

這是我第一個在網路上發布的網站，其中的CSS、HTML都是第一次用，雖然沒有用到多高深技巧，但算是給了一次滿新鮮的經驗。

總結來說，這次製作完自己的網頁，並實際自身遊玩，真的有滿滿的成就感，我想未來我也會想繼續學習其他架設網頁、JavaScript的技巧，最終做出更精美、食用的網站。

參考資料

Chess ProGramming Wiki :
https://www.chessprogramming.org/Main_Page
Chess ProGramming Wiki - AlphaBeta:
https://www.chessprogramming.org/Alpha-Beta
Chess ProGramming Wiki - MiniMax:
https://www.chessprogramming.org/Minimax
Chess ProGramming Wiki - MVV-LVA:
https://www.chessprogramming.org/MVV-LVA
Chess ProGramming Wiki - Quiescence Search:
https://www.chessprogramming.org/Quiescence_Search
Chess ProGramming Wiki - HorizonEffect:
https://www.chessprogramming.org/Horizon_Effect
Chess ProGramming Wiki - Killer Move:
https://www.chessprogramming.org/Killer_Move
Chess ProGramming Wiki - Killer Move:
https://www.chessprogramming.org/Killer_Heuristic
jQuery是什麼，它跟JavaScript有什麼關係？它又有什麼能耐呢？
https://progressbar.tw/posts/6
電腦下棋的關鍵： Min-Max 對局搜尋與 Alpha-Beta 修剪算法:https://programmermagazine.github.io/201407/htm/focus3.html
Chess.com Forsyth-Edwards Notation (FEN)
https://www.chess.com/terms/fen-chess