基因演算法 期末報告

1. 介紹

隨著電腦的計算能力大幅的提升，電腦傳統的棋類、紙牌遊戲的能力也漸漸的超越了人類，人們開始思考：是否有暴力法以外的方法，能夠快速的提升遊戲的勝率呢？

這次的期末專題，我們將一些方法實作在跳棋上，由於跳棋的規則簡單，沒有許多限制，很適合拿來驗證一些方法的可行性，因此在接下來的幾個章節，我們描述了跳棋相關的背景知識，說明我們實作的方法，再來是這些方法的模擬結果。

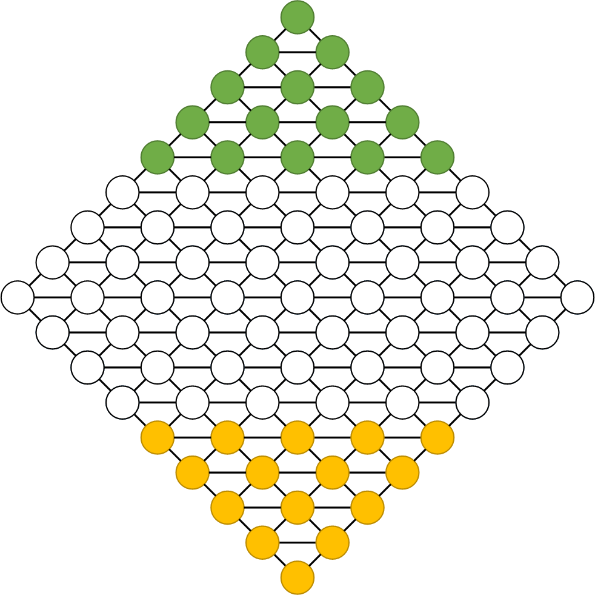
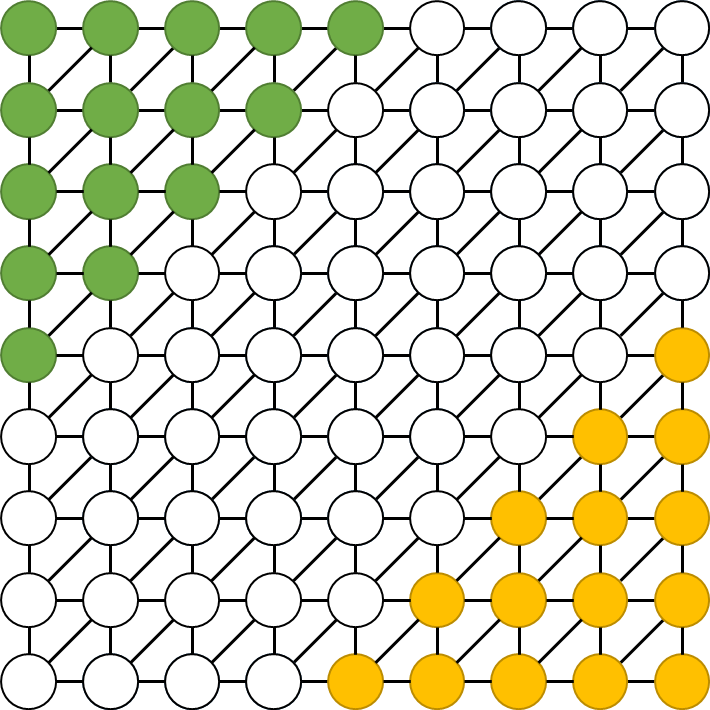
和大多數的棋類、紙牌遊戲的程式不太相同的是，我們並沒有使用棋譜，而是讓電腦從0開始進行對弈，想知道在沒有初始棋譜的情況下，這些方法對於訓練結果的差異，以及他們各自的優缺點。在最後的討論與結論，我們分析了一些結果，並且討論造成這些結果的原因，以及在實作過程中，碰到的一些問題、想法與解決方法。

1. 背景知識
   1. 遊戲規則

關於雙人跳棋，雖然主要的規則相同，但在細節上有著為數不少的差異，因此我們選擇了其中的一類規則，作為我們實作的參考。

* + 1. 初始盤面

如(圖一)所示，我們從三人的六芒星棋盤中，選擇了對角的菱形區域作為我們的盤面，若將他旋轉並稍作變形，他是一個9\*9的棋盤，其中雙方分別在其中的一個角落，各自有15顆棋子。

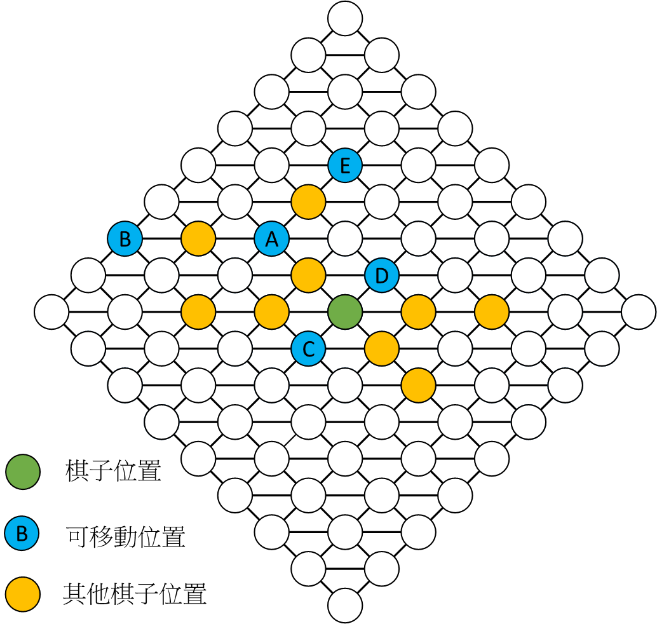
圖一：原始棋盤與轉換圖

* + 1. 行棋規則

雙方決定一方作為先手，輪流選擇一顆棋子進行移動，移動的方式分為平移與跳躍，移動後換對方進行移動，直到其中一方勝利。

平移：玩家選擇己方的一顆棋子，將棋移動到相鄰的位置，該位置不能有其他的棋子，以及不能超出棋盤。

跳躍：玩家選擇己方的一顆棋子，若該棋子相鄰有棋子(不限陣營)，則可將該棋子移動到對面方向的位置(如圖二所示)，對於該棋子而言，跳躍的次數不限。在一些規則中，有些可容許經過/停留在六芒星的任意區域，有些則是僅容許經過，也有不允許經過/停留的規則，在此我們選擇的是僅允許經過與停留在菱形的區域位置。



圖二：可跳躍位置說明圖

* + 1. 勝利方式

當有一方將己方的所有棋子移動到對方的初始位置時(順序不限)，該玩家即獲得勝利。然而在實作時，偶爾會遇到有顆棋子被卡在初始區域無法出去，因此我們增加了一個規則，當總回合數大於特定數字的時候，則判定遊戲結束。

* 1. 預期目標

對於跳棋而言，由於存在著先手必勝，因此我們期望訓練出來的結果，可以使先手的勝率增加，另外我們也從對弈的過程來看訓練的結果如何。

* 1. 先手必勝

首先是這個遊戲屬於完全公開資訊類，也就是說，雙方的所有資訊完全對等，再來是說，由於棋盤的大小有限，雙方的棋子數量也是固定的，因此棋盤的組合數也是有限。從這兩點，我們就可以將其畫成決策樹，起點只有一個，就是初始盤面，終點有很多個，由於跳棋並沒有和局一事，因此可以分成兩類：先手勝利與後手勝利。

在《博弈論》中，有一條定理叫做Zermelo’s theorem，他的內容是說：*在兩人遊戲中，如果棋盤的組合數有限，且棋盤的資訊完全公開，若其中不包含機率事件，那麼先手或後手其中一方必有必勝/必和的方法。*然而，由於跳棋並無和局，因此必有一方有必勝之法，然而後手的所有選擇，都可以被包含在先手的選擇之中，我們可以利用反證法，得到先手有必勝之法。

接下來是說明後手的所有選擇，都可以被先手包含。不論是先手或者後手，第一步可以分成兩類，一種是第一排的五顆棋子往前面的兩種方向移動一步，另一種則是第二排的四顆棋子往前面的兩種方向跳躍一步。而前者的移動方法，都可以藉由先往另一個方向移動後，再回到原本的位置；後者的移動方法，則是可以先將它移動方向路上的那顆棋子平移，再平移原本的那顆棋子。因此若後手有必勝的方法，那麼先手同樣有必勝的方法，因而得到僅有先手有必勝法。

1. 實作方法與結果
   1. 盤面描述

在盤面的部分，由於棋盤的形狀為菱形，可以用一種轉換的方式將其變為9\*9的方形，使得整個棋盤可以用一個二維陣列來描述，而我們的紀錄方法分兩種：board和state，board的部分是紀錄棋盤的每個位置是哪顆棋子，0為無棋子、1為先手方、2為後手方，而state則是用一個一維陣列記錄兩個玩家所有棋子的位置，0為空、1~15為先手方、16~30則為後手方，如此一來，我們就可以記錄每一個盤面的詳細狀態。

至於移動的部分，每一筆資料包含三個值：棋子編號、目標x座標、目標y座標。利用這個方法，可以很方便的描述棋盤如何從一個狀態轉換至下一個狀態，不論是事後要重播，或是要進行計算都相當的容易。

* 1. 程式架構

我們將程式依照功能，分為幾個大的項目，分別是主程式、顯示部分、資料存取部分、遊戲規則、對戰與訓練，大致的流程如下：

func train():

init() #初始化相關數據

for iter in 1 to 10000

while True: #start battle

| findNextPoints()

| #搜尋所有可進行的動作

| calcScore()

| #計算這些行動的分數

| weightChoice()

| #依照分數進行選擇

| move() #進行移動

| recordAndShow()

| #紀錄並顯示

| mirror()

| #將玩家一、二的棋子互換

| if isEndGame():

| break

update() #依照結果更新評價

func battle():

init()

while True: #start battle

while True:

| getP1Move() #取得行動

| if isMovAble():#是否可移動

| | move() #進行移動

| | break

showState() #顯示結果

if isEndGame(): #是否遊戲結束

| break

while True:

| getP1Move()

| if isMovAble():

| | move()

| | break

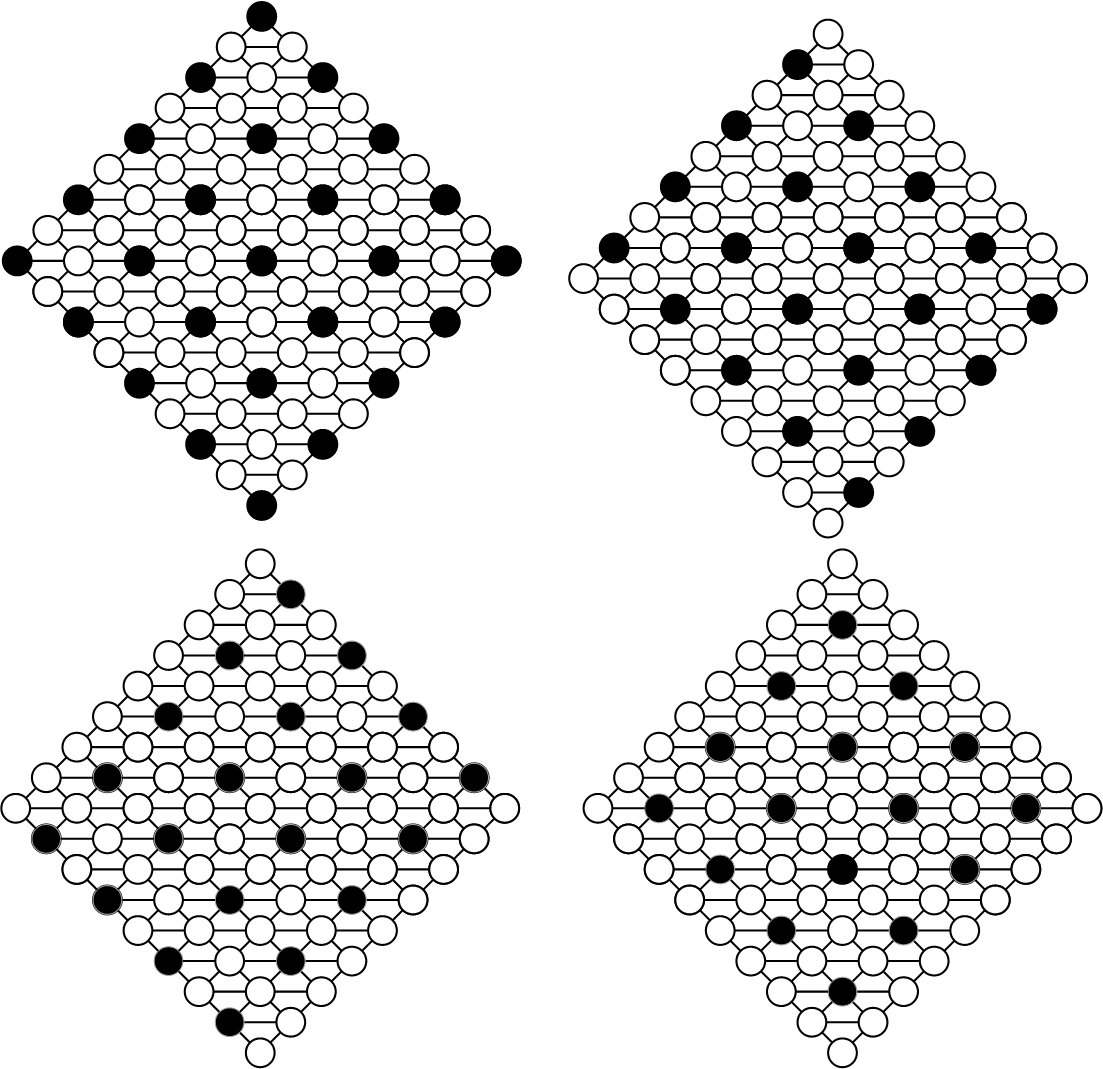
showState()

if isEndGame():

| break

遊戲規則的部分實作方法有一些值得說明，首先是遊戲結束的判定方式，由於結束的方式只有一種，就是棋子全部到達角落，因此我們可以直接將每顆棋子的每個座標點加總，只要到達最大值(200)，就代表遊戲結束，使得判斷結果相當的快速。

另一個是尋找下一步的方法，如果針對每個棋子做DFS或BFS搜尋，會發現有相當大一部份的計算是重複的，因此我們改變了搜尋的方法。由於跳躍的規則相當的簡單，使得棋盤可以拆分成四個網路，如圖四所示。那麼尋找的方法就可以簡化，首先對於每個網路的每個節點，確認該節點是否與相鄰的節點相通(即中間有棋子)，再利用遞移性(若i可到j、且j可到k，則i可到k)，即可快速得到該網路的連通狀況。



↖net0 net1↗

圖三：棋盤分割成四個網路

↙net2 net3↘

* 1. 盤面評價

關於盤面評價，核心的概念是這樣，由於目標是從己方的初始區域移動到對方的初始區域，因此我們可以將己方的初始區域分數設為最低點，而對方的初始區域分數設為最高點，如此一來，前進、後退、平移都可以有各自的初始評價，當電腦依照這個初始評價對戰一些局數後，即可依照這個對戰結果更新這些盤面的分數。

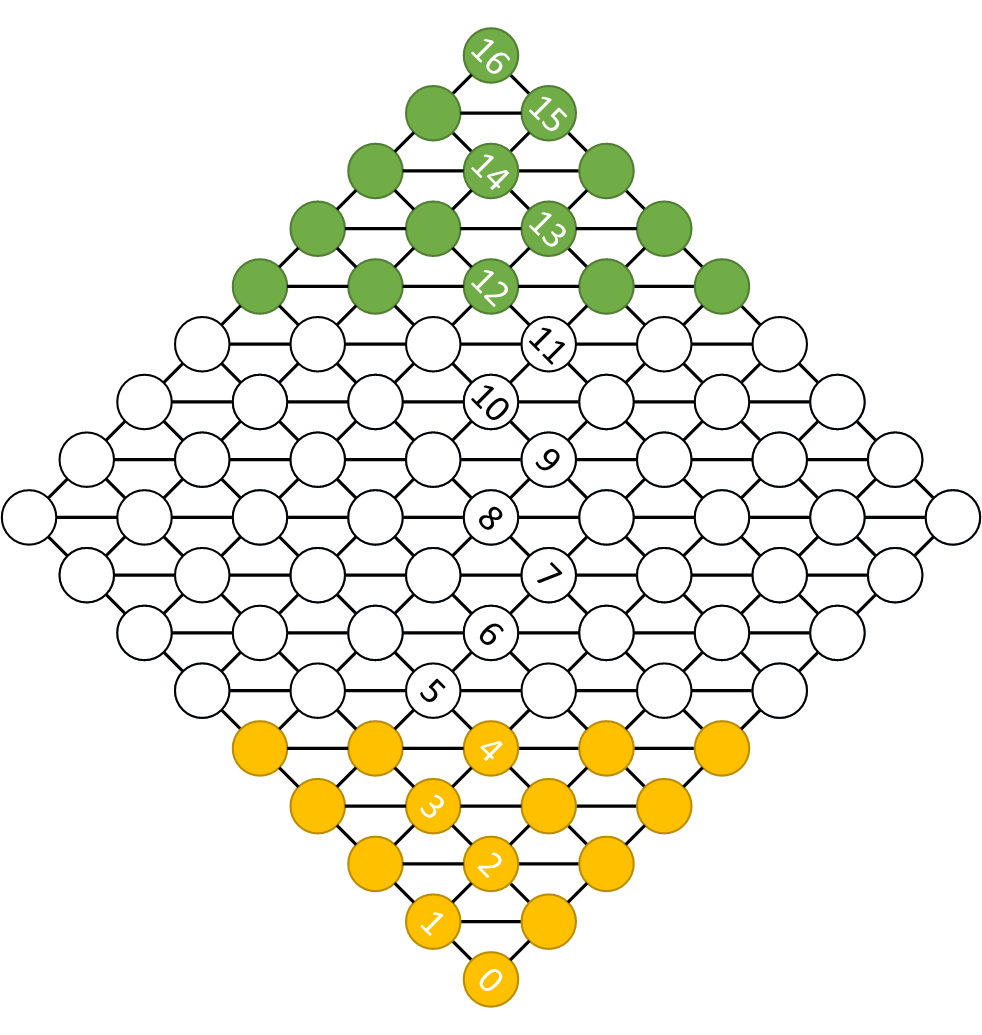
* + 1. 相關變數

round:總共進行的回合數(兩邊各走一次算一回合)

step:該盤面位於該次對戰的第幾回合

* + 1. 初始分數

如圖所示，從起點開始看，我們可以將每一橫排給予一個分數，當棋子在上面時，該玩家即獲得該分數，因此初始分數應該為0分\*1+1分\*2+2分\*3+3分\*4+4分\*5=40分，而終點分數則是16分\*1+15分\*2+14分\*3+13分\*4+12分\*5=200分，除了初始和結束，中途各盤面的初始分數也是如此計算。



圖四：分數計算方法說明

* + 1. 區域更新

在原本的概念中，每一次移動都會對盤面進行區域更新，如同螞蟻演算法，但在實做了之後，發現不論是對其增加、減少、或者讓他逼近某個數值，都會造成每一次的對戰回合數上升，因此在之後我們暫時取消了區域更新的部分。

* + 1. 全域更新

關於盤面數值的更新，我們嘗試了幾個方法，在下面一一進行說明：

類一：將數值分布在40~200之間

核心的概念就是讓贏家的分數往最終結果的分數邁進，而輸家的結果則是朝初始結果後退。除此之外，我們也嘗試過將逼近的分數從40~200變更為0~200，但不論是哪種方法，訓練的結果仍然不如預期，因此我們嘗試了不同的算法。

類二：依照步數與勝負從200往下安排

核心的概念就是讓每一盤的分數逼近總分扣掉距離，使得贏家會越來越好，輸家也可以貢獻部分結果，我們也嘗試過只更新贏家或者只更新輸家，就結果而言，勝率有稍微的提升，然而每次的模擬結果差異大於勝率的差異，因此我們再度嘗試了不同的做法。

類三：依照勝負增減

核心的概念就是將該場次的勝利/失敗平均分配給每一步，若該選擇並沒有那麼差，那麼他的勝率應該會超過50%，但是在多次的模擬結果後，發現最終結果的分數會大幅提升，使得在模擬對戰時，常常會使電腦出現亂流──即使距離終點僅僅兩三步，電腦卻遲遲無法到達終點。

除此之外，我們還有一些作法，他也是屬於類三，但有些不太相同：

當該玩家勝利時，會依照領先的步數進行加強，而失敗時，則是依照落後的步數進行削弱，因此當一個盤面不斷勝利時，他的分數也會漸漸提升，然而由於他並沒有上限，使得在模擬次數增加時，其數值就脫離控制，因此就結果而言，這樣的設計是失敗的。

這一個做法相當的不錯，由於他的上限與下限都已經被限定住了，因此不會有分數爆炸或消失的問題，但也伴隨了另一個問題：由於這樣的算法會使得即使某選擇並不差，但因為最後輸了，導致分數快速的下降，從此無法再進行翻身，因此在嘗試過這幾種方式後，還沒找到比較適合的方法。

* 1. 決策選擇

與更新方法類似，我們使用兩類的決策方法，來觀察他們之間的差異。

類一：傳統的輪盤法

首先我們用傳統的輪盤法去實作，發現由於每種選擇的分數相當的接近，造成實際的運作結果是每種選擇近乎隨機，因此我們增加了門檻值，我們嘗試過取最小值、平均值、最大值以及平均值±1~2個標準差，實測結果是平均值附近的效果最好，取最大值的結果是每場的對戰速度很快，然而學習的效果仍然不足，因此我們嘗試了不同的作法。

類二：冪次的輪盤法

由於傳統的輪盤法中，每一種選擇的分數差異都不大，因此我們嘗試的將權重改到冪次項，而基底則曾經用過1.1、1.5、2、4、8不同的基底，模擬的結果比原本傳統的輪盤法好，但在學習的情況仍然不佳。

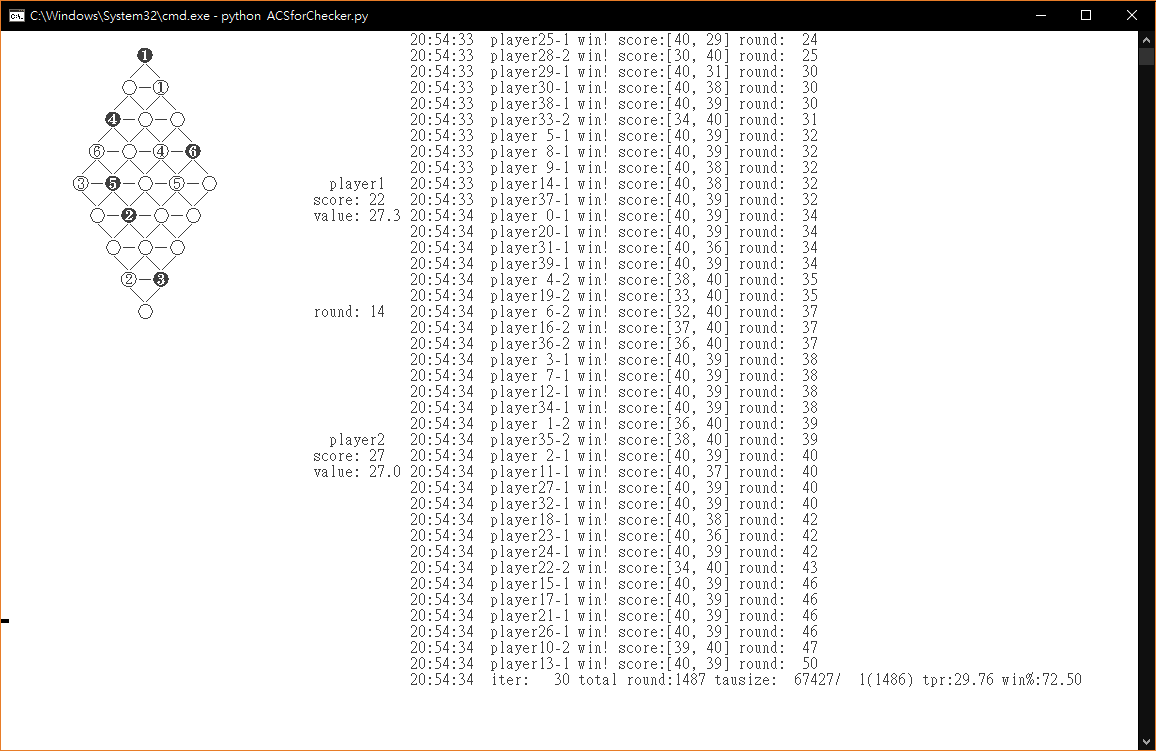
* 1. 模擬結果

在一開始，我們使用的是9\*9的棋盤，雙方各15顆棋子，但是在模擬的過程中，發現即使跑了數十萬場，其中重複的盤面非常稀少，因此先縮小棋盤與棋子的規模，先從5\*5，雙方各6顆棋子。當結果不錯時，再擴張其規模到7\*7，各10顆棋子，下面是我們其中一種模擬方法的過程截圖。

我們每一次都進行40場戰鬥，而左半邊的部分為其中某一場對戰的對戰過程，由於我們同時進行戰鬥，使得顯示結果並不會大幅影響計算時間，而中間分別是玩家一、玩家二的初始分數以及調整過的分數，右邊則是每一次對戰的結果，前面0~39代表這一世代的幾場戰鬥，而後面的1/2則是先手或後手勝利，接下來是他們的分數、以及回合數，與前面的不同，這裡是每個玩家移動結束就算一個回合，最下面則是該世代的總結，包含的幾個世代、該世代總共花費幾個回合、目前已經更新過的盤面、本世代新增的盤面數、本世代更新的盤面數、每個回合消耗的時間(ms)，以及玩家一的勝率。

玩家一與玩家二評分的方式有所不同，就是玩家一對盤面的評價會隨著對戰的次數進行更新，而玩家二則是依照初始的分數進行決策，除此之外，兩者的決策模式則是相同的。

整體而言，在眾多模擬方法中，區域更新取消的結果會比較佳，在全域更新中，第二類的方法也就是從200往下排會比第一類好，而第一類又比的三類方法更好，而決策選擇中，冪次的輪盤法表現則是較傳統的輪盤法優異，最後是門檻的選擇，以平均值作為門檻表現比以最大值、最大值減一個標準差、平均值減一個標準差與最小值都還要好，在5\*5的局面中，勝率已經可以到達70~80%，而局面的覆蓋程度也基本上完整達到，因此在眾多的模擬方法中脫穎而出，目前正在模擬7\*7的棋盤，看看在覆蓋率提升之後，勝率是否也能如此表現。



圖五：模擬過程截圖

* 1. 人機對戰

與模擬方法相似，但是操作玩家二的棋子是交給真人控制，而操作方法如圖，一開始先選擇要移動的棋子編號，接下來選擇方向，平移的六個方向分別是0~5，跳躍的六個方向分別是6~11，可以跳躍多次，每次輸入完一個數字按enter繼續，直到到達目標位置後輸入-1結束，若輸入的格式不正確，則會要求重新輸入。

1. 討論與結論
   1. 在大多數的模擬方法中，常常碰到一個問題，就是電腦往往不會選擇已經走過的盤面，在原本的規劃當中，是認為每一次對戰，都會有一些局面的分數被提升，這些比較有勝算的局面應該較容易被選擇，但之後的分析發現，雖然一方想要維持勝利的局面，但另一方則是會導向較沒勝算的局面，或者是未開拓的局面，使得模擬過程中，通常不會走到已經走過的局面。
   2. 因為1.的原因，使得我們不容易修正已經走過的路徑，因為通常不會走到，但從另一方面來說，由於各種走法都會被走過，因此盤面的覆蓋程度也是相當的大，然而時間上以及記憶體資源不足，使得目前的模擬結果並沒有太大的優勢，只有在5\*5的局面中，有一部分的模擬方法可以大幅的提高勝率，而將其改成7\*7的局面時，則是還在模擬中，因此尚未確定是否表現也能如此的優異。
   3. 在實作的時候，曾經考慮過使用Minimax建立搜尋樹，以及利用alpha-beta進行減枝，或者利用神經網路等方法，然而在這些方法中，前兩種還是回到了評分與更新的探討，而第三種方法則是不同的評分方式，因此在這次實作中，先不考慮這些方法，而是單純的只探討針對不同對戰過程進行盤面分析的影響。