Solving the Path Finding Problem in Tower of Saviors Using Genetic Algorithms

張聖寵
61247041s@gapps.ntnu.edu.tw

張睿恩
40947075s@gapps.ntnu.edu.tw

I.問題定義

《神魔之塔》（Tower of Saviors, ToS）是一款由香港手機遊戲公司 MadHead 所開發的轉珠遊戲。遊戲主要由一個5×6的棋盤與六種不同屬性的符石所組成。玩家需要透過選定起始位置，並移動符石來將相同屬性的符石以直排或橫排的方式連成三粒或以上來消除符石（圖 1）。

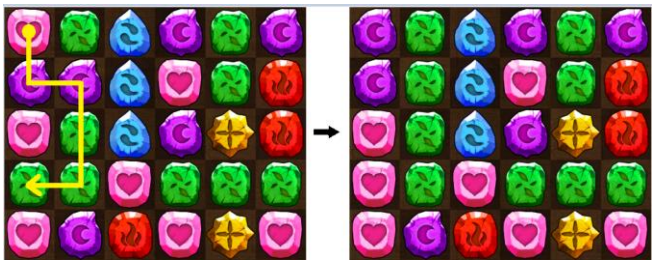

圖 1：神魔之塔遊戲盤與符石移動

根據消除的符石數量與所形成的連擊（combo）數，玩家的對伍將會對敵人進行不同程度的攻擊。如何在有限的時間內進行最佳化的符石移動，以獲得最高的攻擊分數，是玩家取得勝利的關鍵。

在本研究中，我們將 ToS 的路徑搜尋問題拆分成兩個部分：搜尋最佳盤面（Board Optimization）與搜尋到達最佳盤面的路徑（Path Finding）。

首先，我們將 Board Optimization 問題定義為找到一種符石排列方式，使得符石消除後的攻擊分數最大化。這將涉及到盤面上所有可能的符石排列組合以及消除分數的計算。

其次，Path Finding 問題則被定義為找到一個從初始盤面移動符石到達最佳盤面的最佳移動序列。這涉及在有限步數內，通過一系列移動操作將初始盤面轉換為最佳盤面。

本研究旨在基於基因演算法（Genetic Algorithm, GA），設計一套有效的演算法來解決上述問題，目標是在有限的步數內，找到一個最佳的移動符石序列，以獲得最高的攻擊分數。

II.基因演算法

基因演算法於 1975 年由 J. H. Holland 提出[1]，其概念源自於自然界中生物族群的演化過程。GA 將問題解進行編碼，視為個體的基因，然後透過模擬物種選擇（Selection）、交配（Crossover）以及突變（Mutation）的機制來進行族群進化（Population evolve），從而使演算法可以有方向地探索解空間並逐步接近最佳解。

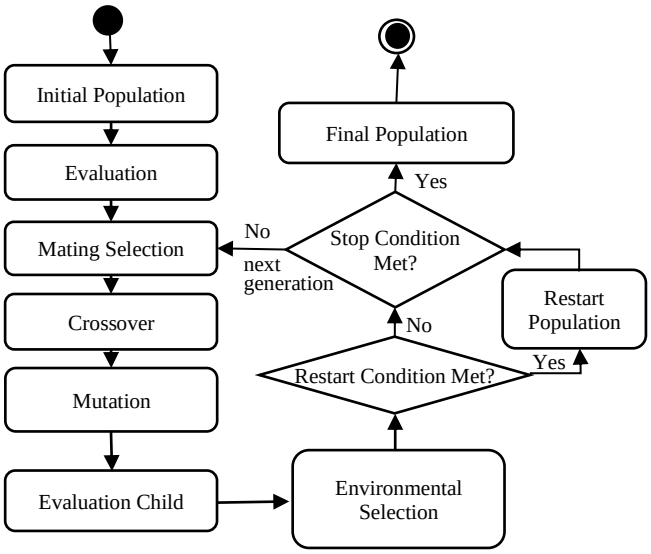

圖 2: 基因演算法流程圖

III. 最佳盤面搜尋演算法

在本章中，我們將說明我們是如何基於基因演算法，來設計我們的最佳盤面搜尋演算法。

A. 解表示法與適應度計算

1. 解表示法（Representation）

我們使用一個5×6的字元陣列來表示 ToS 的遊戲盤面，以做為 Board Optimization 問題的解，其中六種不同的字元分別表示了六種不同屬性的遊戲符石（如圖 3）。

W	W	W	L	L	L
W	W	W	L	L	L
F	F	F	D	D	D
F	F	F	D	D	D
E	E	E	H	H	H

圖 3: 盤面示意圖

2. 適應度計算（Fitness Evaluation）

在基因演算法中，適應度被用於評估解的好壞程度，對於 Board Optimization 問題，除了最終的攻擊分數外，我們還額外將盤面的符石分佈情形納入考量，包含解盤面與初始盤面間的差異度，以及在消除符石後，剩餘盤面符石的聚集度，其完整的適應度計算公式如公式(1)：

$$Fitness = w_s \times Score - w_d \times Distance - w_c \times Density \quad (1)$$

其中，Score 的計算公式為：

$$Score = \frac{N_{stones} + Combo}{30 + 10} \quad (2)$$

- w_s、w_d、w_c：權重參數。
- N_{stones}：消除的總符石數量。
- $Combo$：連擊數。

Score 的計算方式源自於 ToS 遊戲中基礎攻擊的傷害計算公式，而在這之上我們額外對其進行了標準化處理，以便與後續不同的數據進行合併。

在適應度的計算公式中，Distance 的計算方式如下：

$$Distance = \frac{D}{30 \times 4.5} \quad (3)$$

$$D = \sum_{(i,j)\in B} \min_{(i',j')\in T} \left(\left(dist\big((i,j),(i',j')\big) \middle| B(i,j) = T(i',j') \right) \right) \quad (4)$$

$$dist\big((i,j),(i',j')\big) = |i - i'| + |j - j'| \quad (5)$$

- $B(i,j)$：解盤面在位置(i,j)的符石類型。
- $T(i,j)$：初始盤面在位置(i,j)的符石類型。

Distance 的計算思路是統計解盤面上的符石與初始盤面同種符石間的最小曼哈頓距離（Manhattan Distance）。

而之所以在計算 Fitness 時要扣除 Distance 的用意在於，由於我們的目標是希望能在較短的路徑中取得較高的得分，所以若解盤面與初始盤面的差異越小，則表示其到達的可能路徑也就越短，作為解盤面來說也就更好。

最後，我們還在 Fitness 中考慮了消除符石後，盤面剩於符石的聚集程度，其計算方式如下：

$$Density = \frac{1}{6 \times 9} \sum_{k} Avg\_Dens_k \quad (6)$$

- 對於每種符石屬性k，計算該屬性所有剩餘符石之間的曼哈頓距離平均值Avg_Dens_k。
- 若某種符石無法成對，則設其平均距離為 1。

之所以在計算 Fitness 時考慮 Density 的用意在於，當盤面在計算完攻擊分數、消除完符石後，對於盤面上剩餘無法消除的符石，我們認為其同屬性符石間的聚集程度越高，也就是Avg_Dens_k越小，則應表示盤面越好。

B. 族群初始化（Initial Population）

對於 Board Optimization 問題，我們採用了隨機初始化方法，即透過對初始盤面進行隨機重排，來產生族群個體。

C. 選擇、交配與突變方法

1. 親代選擇（Mating Selection）

在親代選擇的階段，我們採用了 Two Tournament Selection（TS）方法。TS 會隨機從族群中選擇兩個個體，並選擇 Fitness 較大者作為一名親代。

2. 交配（Crossover）

在交配的階段，我們採用了一種改良的 Linear Order Crossover（LOX）方法，來處裡二維陣列形式的解。此方法的核心概念是將兩個親代的一部份盤面進行交換，以產生新的子代個體。

具體步驟如下：

a. 選定交叉點：

使用兩個交叉點（cross_point1、 cross_point2）分別做為盤面的左上座標和右下座標，以確定要交換的子盤面範圍。

b. 交換子盤面區域：

將兩個親代（parent1、parent2）的子盤面區域交換，以分別生成新的子代個體。

c. 填充剩餘盤面：

將親代扣除交換後的子盤面中的符石，將剩餘符石由左至右、由上到下依序填入子代中，以產生完整的子代。

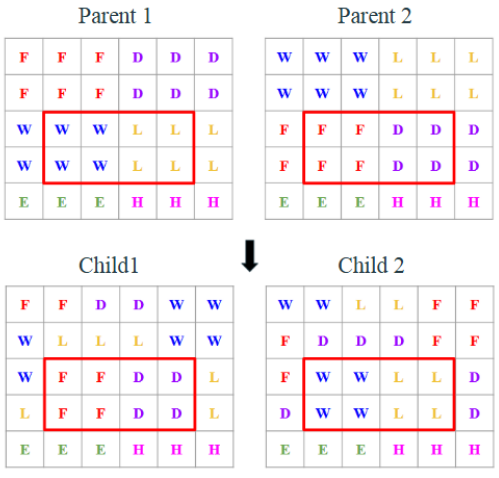

圖 4: LOX 示意圖

表 1: Pseudo code for LOX

```
LinearOrderCrossover(parent1, parent2, cross_point1, cross_point2)
    Initialize child1, child2 as empty TosBoard
    # Extract sub-boards from parents based on crossover points
    sub_board1 = ExtractSubBoard(parent1, cross_point1, cross_point2)
    sub_board2 = ExtractSubBoard(parent2, cross_point1, cross_point2)

    # Count the types of stones in the sub-boards
    sb1_stone_counts = CountStoneTypes(sub_board1)
    sb2_stone_counts = CountStoneTypes(sub_board2)

    # Fill stones list based on stone counts in the sub-boards
    fill_stones1 = []
    fill_stones2 = []
    For each (row, col) in parent1
        stone_type1 = parent1[row][col]
        stone_type2 = parent2[row][col]

        # Check if stone_type1 is needed for child1
        If sb2_stone_counts[stone_type1] > 0
            sb2_stone_counts[stone_type1] -= 1
        Else
            Append fill_stones1 with stone_type1

        # Check if stone_type2 is needed for child2
        If sb1_stone_counts[stone_type2] > 0
            sb1_stone_counts[stone_type2] -= 1
        Else
            Append fill_stones2 with stone_type2

    # Swap sub-boards between parents to create children
    For each (row, col) in range(cross_point1 to cross_point2)
        child1[row][col] = parent2[row][col]
        child2[row][col] = parent1[row][col]

    # Fill the remaining parts of the boards with the fill_stones lists
    For each (row, col) in child1
        If child1[row][col] is NONE
            child1[row][col] = pop(fill_stones1)

        If child2[row][col] is NONE
            child2[row][col] = pop(fill_stones2)
    Return child1, child2
```

3. 突變（Mutation）

對於每一個新產生的子代，它們會有一定的機率進行突變，而在突變的階段，我們採用了 Board Shift Mutation 方法，該方法會隨機選擇子代盤面中的一個矩形區域，並對其進行隨機的位移操作。

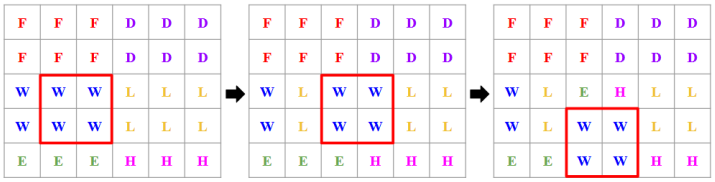

圖 5: Board Shift 示意圖

4. 環境選擇（Environmental Selection）

在環境選擇階段，我們採用了$n+n$環境選擇策略來確保族群的穩定發展。在$n+n$方法中，算法會維持一個恆定大小的族群，其中n表示族群中的個體數量。每一代中，我們將生成n個新的子代個體，然後從合併後的$2n$個個體（即n個親代和n個子代）中選出 Fitness 最高的n個個體來做為下一代的族群。

D. 重啟族群與演化停止條件

1. 重啟族群（Restart Condition）

在最佳盤面搜尋演算法中，我們並沒有設定族群的重啟條件，所以這個階段我們將會留待 Path Finding 的部分再做說明。

2. 演化停止條件（Stop Condition）

在最佳盤面搜尋演算法中，我們設立了三個停止條件：

- Achieve max generation：演化達到最大代數。
- Low diversity：當族群的 Fitness 標準差小於一定值時，代表族群已經收斂、個體多樣性不足，故提前終止演化。
- Evolution stagnation：當族群中 Fitness 最大值在連續數代都未更新時，代表族群已陷入局部最佳解中，已經難以再有更好的個體產生，故提早終止演化。

IV.路徑搜尋演算法

在本研究中，我們將 Path Finding 問題視為類似 N-puzzle 問題來處裡。N-puzzle 問題是一個經典的組合問題，該問題由一個$m \times m$的盤面和N塊有序的滑動方塊組成，其中：

$$N = m^2 - 1 \tag{7}$$

盤面上留有一個空格，允許相鄰的方塊滑動到空格位置。N-puzzle 問題的目標是找到一條滑動路徑，將盤面從初始狀態移動到目標狀態。

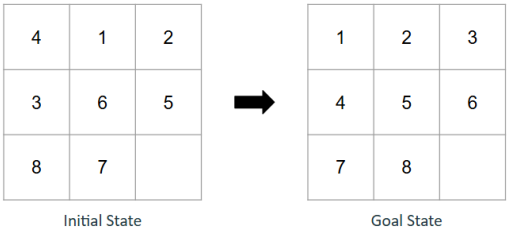

圖 6：8-puzzle 問題示意圖

而這與我們 Path Finding 問題的目標相當類似，以下，我們將說明我們是如何基於基因演算法與 N-puzzle 問題，並參考了 Harsh Bhasin 等人的研究[2]，來設計我們的路徑搜尋演算法。

A. 解表示法與適應度計算

1. 解表示法（Representation）

對於 Path Finding 問題，我們設計了一種專門的解表示方法，包含步數、起始位置和搜尋路徑：

- 步數：一個介於 0 到最大步數限制之間的整數，表示此個體的路徑長度。
- 起始位置：表示起始符石在盤面上的網格座標。
- 路徑：路徑由一連串 0 到 1 之間的實數組成，這些實數表示在搜尋空間中的移動方向，每個實數對應一次移動操作，並指示符石在盤面上的具體移動方向。

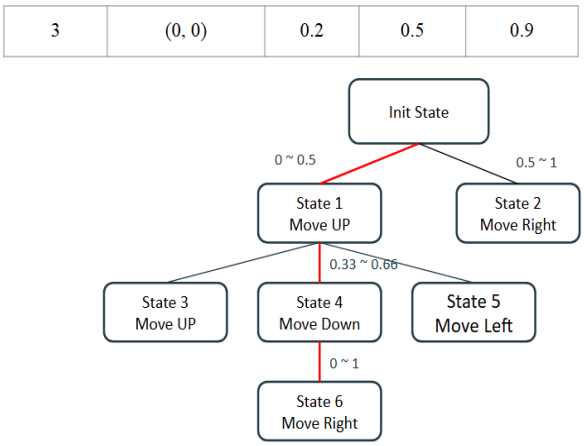

圖 7：Path Finding 解示意圖

在圖 7 的例子中，解的步數為 3，起始位置為(0,0)，搜尋路徑由後續的三個實數組成。

每個實數決定了從當前狀態到下一狀態的移動方向。我們的下一步搜尋狀態順序遵循上、右、下、左的順時針方向，這樣的順序能夠保證對所有可能的移動方向進行系統性的探索。

然而，具體的可行狀態會根據棋盤的邊界和避免回頭路的限制而有所不同。根據可行狀態的數量，我們會將 0~1 的實數均勻劃分為數個區間，再由此決定實數所表示的路徑。

2. 適應度計算（Fitness Evaluation）

對於 Path Finding 問題，我們沿用上一章中的盤面差異度公式(3)，將最佳盤面與路徑到達盤面之間的差異度作為解的 Fitness。

對於 Path Finding 問題來說，解路徑最終到達的盤面與最佳盤面之間的差異度越小，就表示解越好。

B. 族群初始化（Initial Population）

對於 Path Finding 問題，我們採用了隨機初始化方法，盤面上的所有位置都將做為起始位置，隨機產生固定數量的個體來做為初始族群。

C. 選擇、交配與突變方法

1. 親代選擇（Mating Selection）

在親代選擇的階段，和最佳盤面搜尋演算法相同，我們採用了 Two Tournament Selection 方法。

2. 交配（Crossover）

在交配的階段，我們採用了 Cut Splice Crossover（CSX）方法。CSX 方法會各自在兩個親代的路徑中選擇一個隨機交叉點，並將交叉點後的片段進行交換，以產生新的子代。

另外，若子代的長度超出最大步數限制，則直接將超出的路徑片段捨去。

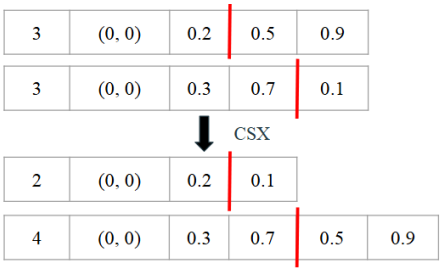

圖 8：CSX 示意圖

3. 突變（Mutation）

對於每一個新產生的子代，它們會有一定的機率進行突變，而在突變的階段，我們採用了 Segment Mutation 方法，該方法會隨機選擇子代路徑中的一個突變點，並將突變點後的所有實數替換為新的隨機實數值。

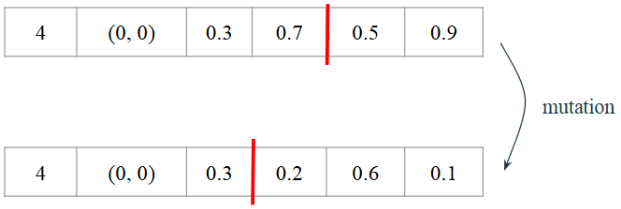

圖 9: Segment Mutation 示意圖

4. 環境選擇（Enviroment Selection）

在環境選擇階段，和 Board Opitization 相同，我們採用了$n+n$環境選擇策略。

D. 重啟族群與演化停止條件

1. 重啟族群（Restart Condition）

對於 Path Finding 問題，由於其很容易陷入局部解中，我們對其設置了一個族群重啟條件：當族群中的 Fitness 最大值和最小值在連續數代演化中都保持不變時，此時將會觸發族群重啟。

重啟方式為將族群中最差的一半個體替換為隨機產生的新個體，並且這些新個體的產生方式和初始化族群的方法相同，包含了等量的來自所有起始位置的個體。

2. 演化停止條件（Stop Condition）

在 Path Finding 問題中，我們設立了三個停止條件：

- Achieve max generation：演化達到最大代數。
- Low diversity：當族群的 Fitness 標準差小於一定值時，提前終止演化。
- Evolution stagnation：當族群中 Fitness 最大值和最小值在連續數代都未更新時，提早終止演化。

V.ToS 路徑搜尋演算法

綜合我們在第 III 和 IV 章所介紹的兩個演算法，就組成了我們完整的 ToS 路徑搜尋演算法（ToS Path Finding Algorithm, TSPF）：

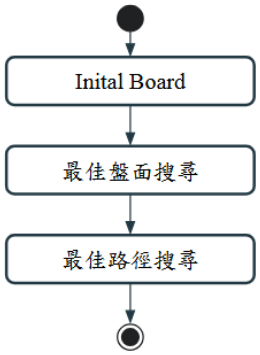

圖 10：ToS 路徑搜尋演算法流程圖

首先，我們會讀取一個初始盤面做為待解問題，接著，我們會使用最佳盤面搜尋演算法來找到初始盤面的最佳盤面。

接著，我們會透過路徑搜尋演算法來找到從初始盤面到達最佳盤面的路徑，而此路徑即為我們最終所要找尋的目標。

VI.實驗與結果

A. 測試問題集

為了評估 TSPF 的性能，我們設計了一組包含 15 個不同特性的測試盤面。這些測試盤面旨在涵蓋各種可能的遊戲情況，以全面測試演算法的效能和穩健性。

表 2: 測試問題集

測試盤面	說明
board01	盤面只有 1 種符石。
board02	盤面只有 2 種符石。
board03	盤面只有 3 種符石。
board04	盤面只有 4 種符石。
board05	盤面只有 5 種符石。
board06	盤面只有 6 種符石。
board07	每種符石皆有 5 顆。
board08	可以產生最大分數（全消+10 Combo）。 用於測試最佳解的搜尋能力
board09	包含 6 種符石，但有三種無法消除。 增加盤面搜尋的干擾性。
board10	對稱且全部同屬符石皆無相連。 增加路徑複雜度。
board11	包含五種符石各六顆，且全部聚集。 測試盤面搜尋跳出局部解的能力。
board12	符石聚集在中心位置。 測試盤面與路徑搜尋跳出局部解的能力。
board13	隨機生成的盤面。
board14	隨機生成的盤面。
board15	隨機生成的盤面。

B. 實驗設定

1. 計算環境

表 3：計算環境設定表

項目	內容
作業系統	Windows 10
開發與執行環境	Windows Subsystem for Linux 2
處理器	Intel Core i7-13700
記憶體	16 GB

2. 最佳盤面搜尋演算法參數設定

表 4：最佳盤面搜尋演算法參數設定

參數名	參數值	說明
max_gen	200	最大世代數
pop_size	500	族群個體數
mutation_rate	0.5	突變率
max_stagnation	60	演化停滯世代數閾值
stop_threshold	0.0001	族群收斂閾值
w_s	1.0	分數權重
w_d	0.2	差異度權重
w_c	0.2	聚集度權重

3. 路徑搜尋演算法參數設定

表 5：路徑搜尋演算法參數設定

參數名	參數值	說明
max_gen	1000	最大世代數
pop_size	3000	族群個體數
mutation_rate	0.3	突變率
max_stagnation	50	演化停滯世代數閾值
stop_threshold	0.0001	族群收斂閾值
max_steps	250	路徑最大步數

4. TSPF 參數設定

表 6：TSPF 參數設定

參數名	參數值	說明
algo_runs	10	完整執行算法的次數

C. 實驗結果

1. 突變率測試

在本節中，我們分別對最佳盤面搜尋演算法和路徑搜尋演算法進行了突變率的實驗測試，以找出最合適的突變率來用於後續的實驗。

對於每個測試問題，我們皆分別測試了 0.1、0.3、0.5、0.7、0.9 五種突變率，且每個測試皆完整執行了 20 次實驗，最終統計並分析 20 次的實驗結果。

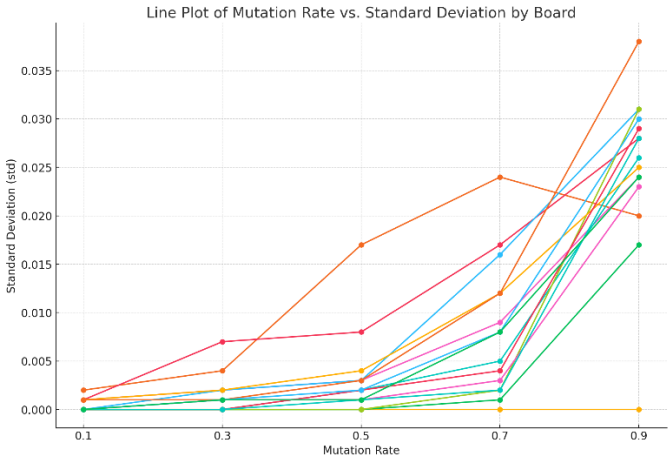

圖 11：最佳盤面搜尋演算法突變率與標準差關係圖

圖 11 中，x 軸為突變率，y 軸為 20 次實驗的 Fitness 平均標準差（Standard deviation），每一條折線即代表一個測試問題。

從圖中可以明顯觀察到，隨著突變率的上升，其平均標準差也就越高，即族群的收斂度越差。

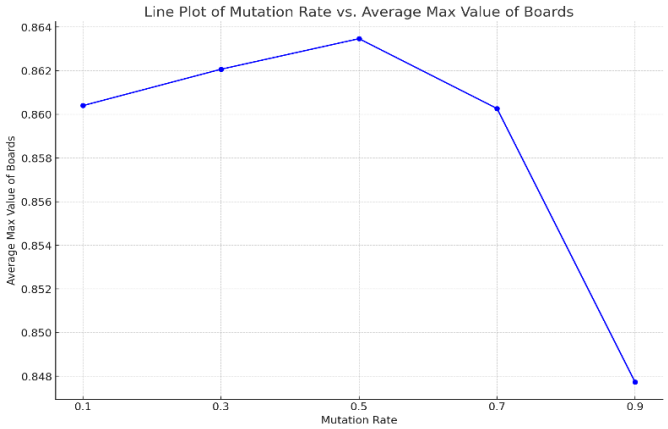

圖 12：最佳盤面搜尋演算法突變率與最佳解關係圖

圖 12 中，x 軸同樣為突變率，y 軸則為所有測試問題在所有實驗中，最佳解的 Fitness 平均值。

從圖中可以觀察到，當突變率為 0.5 時，有最好的最佳解 Fitness 平均值。我們的觀察是，對於最佳盤面演算法，突變率越高會增加族群的多樣性，使其更容易跳出局部最佳解，並有更高的機會可以找到更好的解盤面。

但同時，過高的突變率也會使得子代更加難以保留親代的特徵，也使得族群難以有效地收斂，這就導致了當突變率過高時，反而難以找到較好解的情形。

根據對實驗結果的觀察，最終我們選擇了以 0.5 做為最佳盤面演算法在後續實驗中的突變率。

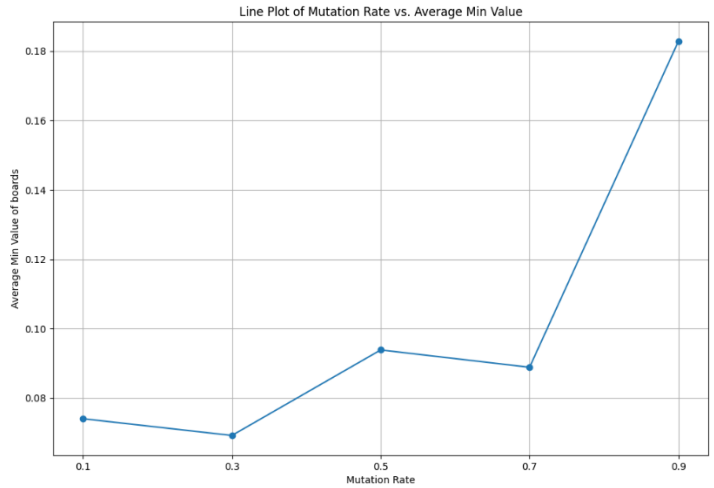

圖 13：路徑搜尋演算法突變率與最佳解關係圖

對於路徑搜尋演算法而言，由於在演化過程中具有族群重啟機制，故其族群最終的 Fitness 標準差並無參考性。

而在圖 13 中，x 軸為突變率，y 軸則為所有測試問題在所有實驗中，最佳解的 Fitness 平均值。由於 Fitness 的計算方式不同，對於路徑搜尋演算法而言，Fitness 越小，代表解越好。

結果和最佳盤面搜尋演算法的實驗類似，過低和過高的突變率都可能導致搜尋的性能變差，根據對實驗結果的觀察，最終我們選擇了以 0.3 做為路徑搜尋演算法在後續實驗中的突變率。

2. TSPF 完整實驗

在本節中，我們使用了表 4~6 的參數設定，對每個測試問題進行了 10 次的完整搜尋，並記錄與分析 10 次結果的數據，其完整結果如表 7 和表 8。

表 7：ToS 盤面搜尋結果

filename	max	avg	min	max std	avg std	avg run time
board01	0.775	0.775	0.775	0	0	0.283
board02	0.956	0.946	0.903	0.020	0.015	22.257
board03	0.925	0.901	0.869	0.018	0.012	22.809
board04	0.901	0.890	0.878	0.009	0.005	28.012
board05	0.885	0.840	0.816	0.023	0.006	27.582
board06	0.850	0.823	0.807	0.014	0.002	32.465
board07	0.799	0.788	0.764	0.013	0	17.813
board08	0.929	0.927	0.923	0.003	0.001	26.322
board09	0.743	0.724	0.693	0.014	0.010	20.301
board10	0.889	0.844	0.820	0.028	0.003	30.224
board11	0.959	0.952	0.939	0.006	0.001	25.238
board12	0.857	0.825	0.799	0.017	0	25.563
board13	0.825	0.795	0.768	0.016	0.002	29.211
board14	0.878	0.869	0.860	0.005	0	28.612
board15	0.771	0.756	0.714	0.003	0.003	28.594

表 7 展示了對於每個測試問題，在 10 次的完整實驗中盤面搜尋階段的統計結果。各個欄位的具體說明如下：

- max：10 次實驗中所獲得的最佳解的最大值。
- avg：10 次實驗中所獲得的最佳解的平均值。
- min：10 次實驗中所獲得的最佳解的最小值。
- max std：10 次實驗最佳解的標準差。
- avg std：10 次實驗最終族群的平均標準差。
- avg run time：10 次實驗的平均運行時間。

首先觀察 max std，我們可以大致將其分為三組來觀察：

1. board01、08、11、14、15：

在這組測試問題中，它們取得了較低的 max std，這代表它們在找到最佳解的能力上具有足夠的穩定性。我們的觀察是，對於普遍的問題而言，最佳盤面演算法都能有不錯的搜尋效果。

2. board02、03、04、07、12：

在這組測試問題中，它們取得了較低的 max std，我們的觀察是，它們都具有單一、大量、且聚集的同屬性符石，這使得盤面在 combo 拆分上的精準度要求很高，也導致了最佳解的搜尋較不穩定。

尤其比較 board11 和 board12 便可發現，同樣屬於 combo 拆分的問題，兩者的差異就在於 board11 為五種符石各自聚集，而 board12 則屬於單一種符石於盤面中心大量聚集，而兩者在搜尋結果上便有明顯的差距。

3. board05、06、09、10、13：

在這組測試問題中，它們同樣取得了較低的 max std，而與上組問題的差別在於，它們屬於符石零散程度較高的盤面，例如零散程度最高的 board10 在所有問題中取得了最大的 max std。

這類問題同樣對於符石位置擾動具有較高的精準度要求，這也使其較難找到最佳解。

接著，讓我們觀察 avg std，可以發現除了 board02、03、和 09 外，其他測試問題在實驗最後都取得了不錯的收斂。

而對於這三組問題，我們的觀察是，從 avg run time 來看，它們都具有較短的運行時間，這可能表示它們提前觸發了演化停滯的終止條件，過早陷入了局部解、或因為盤面的複雜度而使其難以找到更好的解，才導致族群在收斂以前，就提前終止了演化。

表 8：ToS 路徑搜尋結果

filename	max	avg	min	min std	find	avg run time
board01	0	0	0	0	10	0.632
board02	0	0	0	0	10	71.115
board03	0	0	0	0	10	179.456
board04	0.015	0.006	0	0.008	6	284.588
board05	0.044	0.007	0	0.014	7	367.589
board06	0.074	0.012	0	0.023	6	352.894
board07	0.044	0.012	0	0.017	6	437.343
board08	0.030	0.007	0	0.010	6	353.773
board09	0.015	0.001	0	0.005	9	37.946
board10	0.074	0.024	0	0.024	3	484.216
board11	0.030	0.004	0	0.010	8	164.981
board12	0.074	0.009	0	0.023	8	159.887
board13	0.059	0.019	0	0.023	4	327.695
board14	0.030	0.021	0	0.012	2	376.561
board15	0.044	0.012	0	0.015	5	413.204

表 8 展示了對於每個測試問題，在 10 次的完整實驗中路徑搜尋階段的統計結果。各個欄位的具體說明如下：

- max：10 次實驗中所獲得的最佳解的最大值。
- avg：10 次實驗中所獲得的最佳解的平均值。
- min：10 次實驗中所獲得的最佳解的最小值。
- min std：10 次實驗最佳解的標準差。
- find：10 次的實驗中，找到到達最佳盤面的路徑的次數。
- avg run time：10 次實驗的平均運行時間。

其中，Fitness 為 0 時，表示此路徑可以到達最佳盤面。

首先觀察 board09，可以發現雖然 board09 對於盤面搜尋是個較為困難的問題，但對於路徑搜尋而言，它的結果相當不錯，這也側面反應了在 TSPF 中，盤面搜尋與路徑搜尋兩個問題間具有一定的獨立性。

接著，觀察 board10、13，這兩個盤面都取得了較差的 min std，我們的觀察是，這些盤面都屬於路徑較為複雜的問題，較高的搜尋難度導致了較低的 min std。

另一方面，雖然 board06 和 12 也具有較差的 min std，但這兩個盤面在十次實驗中卻分別成功找到了 6 次和 8 次路徑，屬於成功率較高的盤面。我們的觀察是，這可能是因為它們在特定幾次演化時陷入一個很差的局部解中無法跳脫，從而導致了整體 min std 的降低。

另外，board14 雖然在 min std 上有不錯的表現，但在 avg 和 find 上的數值相當差，我們的觀察是 board14 在路徑上很容易陷入一個特定的局部解中，使其難以找到可以到達最佳盤面的路徑。

VII.結論

在本研究中，我們提出並實現了基於基因演算法的《神魔之塔》（ToS）路徑搜尋演算法（TSPF），並進行了一系列實驗來評估其性能和穩健性。以下是我們的主要發現和結論：

1. 最佳盤面搜尋演算法

突變率對於演算法的性能有顯著影響。實驗結果表明，突變率為 0.5 時，能夠在大多數測試盤面中取得最佳的搜尋效果。

對於大多數的測試盤面而言，演算法都能穩定找到不錯的解，而即便是在搜尋結果相對較差的問題中，演算法也能找到不錯的解盤面，這包含了一些對於盤面擾動精準度要求較高的問題。

2. 路徑搜尋演算法

突變率的選擇同樣對於演算法的性能有顯著的影響。實驗結果表明，當突變率為 0.3 時，演算法能夠在多數的測試盤面中找到較好的解。

在路徑搜尋中，演算法能夠大致有效地應對大多數測試盤面的路徑複雜度，在十次的完整運行實驗中，演算法找到路徑的總體比率約落在 0.66 左右。

然而，對於一些路徑特別複雜的問題，路徑搜尋的成效就較差了。

總的來說，本研究設計並證明了 TFPS 的可行性，尤其在最佳盤面搜尋上取得了相當不錯的成果。

在未來的研究方向中，對於盤面搜尋，可以加入更多的盤面限制，例如特定的排列圖形組合，以及，可以以縮短搜尋時間與樣本數為目標，將問題進一步修改為在有限時間內找到局部較佳解，而非如本文中的全局最佳解。

而在路徑搜尋方面，可以考慮進一步最佳化路徑搜尋的 Fitness 計算方式，將路徑長度也納入考量，並且同樣地，也需要對搜尋時間進行最佳化，以提升整體的搜尋效率。

VIII. 參考文獻

[1] J. H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, Michigan Press., 1975.

[2] Bhasin, H., & Singla, N. (2012). Genetic based Algorithm for N - Puzzle Problem. International Journal of Computer Applications, 51, 44-50.

[3] Paredis, Jan. (1993). Genetic State-Space Search for Constrained Optimization Problems.. Proceedings of the 13th International Joint Conference on Artificial Intelligence. 967-973.

[4] Godefroid, Patrice & Khurshid, Sarfraz. (2002). Exploring Very Large State Spaces Using Genetic Algorithms.. International Journal on Software Tools for Technology Transfer - STTT. 6. 266-280. 10.1007/s10009-004-0141-1.

[5] Zedan, Ruqaya & Ban, Sha & Alkallak, Isra & Sulaiman, Mawadah. (2009). Genetic Algorithm to Solve Sliding Tile 8-Puzzle Problem.