

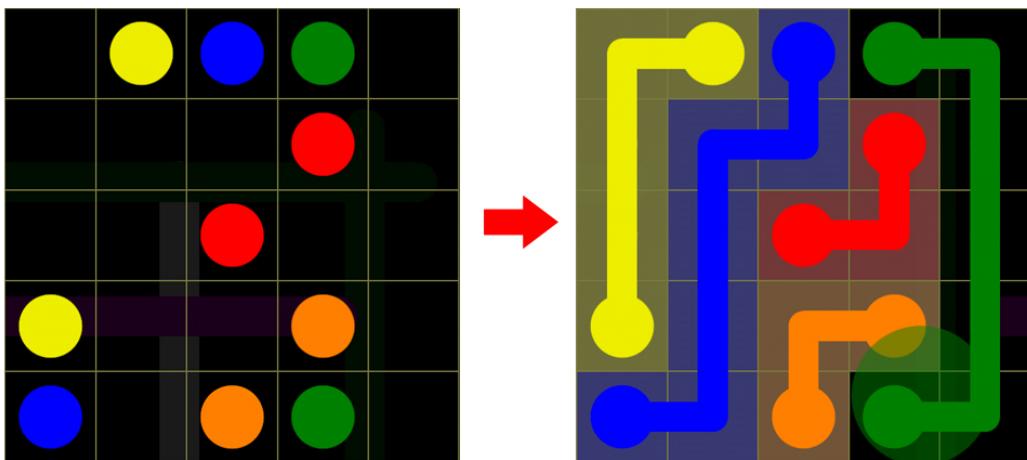
# 資料結構 Final Project 期末報告

電機四 A 110501529 鄭凱方

電機四 A 110501531 戴仕庭

## 一、題目介紹

Flow Free 是一款備受推崇的經典益智解謎遊戲，以其簡潔的設計和深刻的邏輯挑戰吸引了眾多玩家。遊戲的核心目標是在棋盤上，將相同顏色的起點和終點以線條相連，形成完整且不交錯的“管道”路徑。同時，所有路徑必須填滿棋盤，達成無縫覆蓋。在基本規則之外，遊戲更進一步加入了一項策略性極高的挑戰—在相鄰的四個色塊中，同一顏色的數量不得超過兩個，這一規則對玩家的規劃能力和邏輯思維提出了更高要求。



在 Flow Free 的 Final Project 中，採用雙人合作模式，以充分發揮團隊協作與個人創意。專案主要分為兩個核心部分：出題設計與解題實現。出題設計的重點在於如何根據使用者提供的棋盤尺寸，設計出既合理又具有挑戰性的題目。不僅需要確保題目符合遊戲規則，還可以通過設置額外限制（例如，相同顏色連線的最短距離）來提升遊戲的趣味性與難度。解題實現則聚焦於開發高效的演算法，以最短的時間正確解出給定尺寸的任意題目。這需要探索並實現先進的搜尋與優化技術，確保演算法的準確性和性能，特別是在面對複雜題目時能快速產生最佳解。

## 二、組員分工

- A. 戴仕庭: FLOW FREE 遊戲出題
- B. 鄭凱方: FLOW FREE 遊戲解題

### 三、實作方法&實際結果

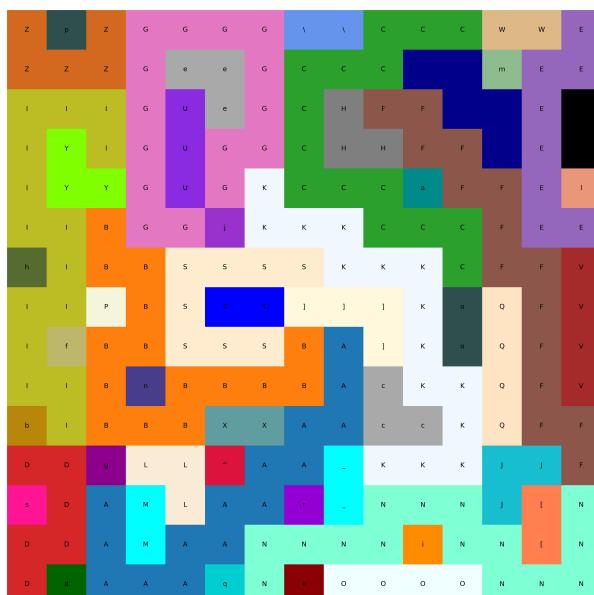
#### A. FLOW FREE 遊戲出題:

1. 使用者輸入: 棋盤的尺寸(長&寬)、單一顏色的最短長度
2. 程式輸出: \_game.txt、\_game.jpg、\_solution.txt、\_solution.jpg、\_info.txt
3. C 語言程式: Flow\_Free\_Generator.c、Generator\_IO.h(產生 txt 檔案)
4. 自動化程式: color\_auto\_process.py(txt 轉 jpg)、auto\_process.bat
5. 資料結構使用: 2D Array and Linked List with dynamic memory allocation
6. 主程式運行流程:

主要為七個步驟生成圖片，為了確保每次圖片都有足夠的隨機性，所以用了大量的 random()，也導致程式運行時間難以估計。當棋盤越大或是最短長度限制越長，收斂的時間就會越長。以下程式介紹以功能介紹和各階段結果輸出為範例來說明。因為製作過程中輸出為一個字符代表一個顏色，但是由於電腦能輸出的字符有限，因此超出一定限制的 ASCII CODE 會導致程式錯誤，所以本程式仍有一定的棋盤尺寸限制。

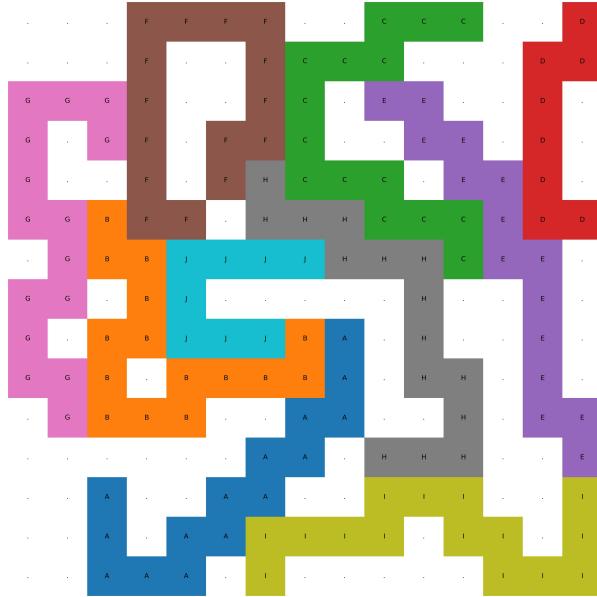
##### a. build\_path()

先隨機從棋盤中挑選一個起點，在規則下(相同顏色不可相鄰)盡可能延長此顏色的管道，並將整個棋盤都佈滿顏色。此部分主要是為了在短時間先構建一個初始解，讓後續更容易優化調整。



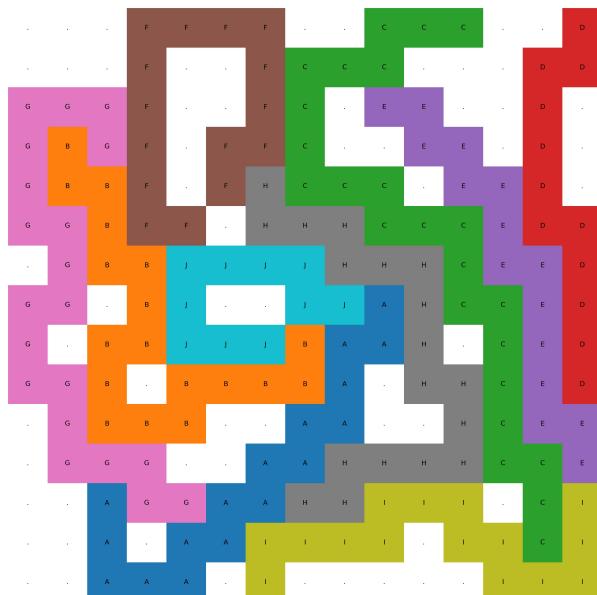
b. `delete_short_path()`

刪除棋盤上比最短長度限制還要短的顏色管道。由於前一步不會去管長度的問題，所以此部分是為了清空不成功的顏色布局，讓後續繼續根據長度限制去優化。



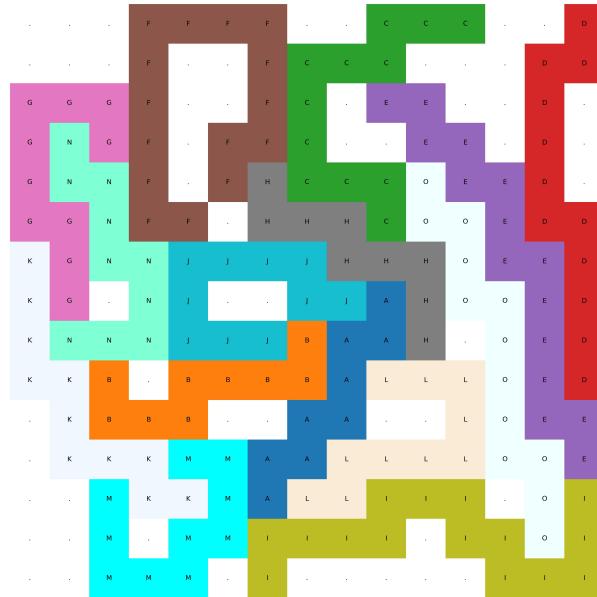
c. `extend_the_head_of_path()`

根據既有的顏色通道去優化，讓其盡可能延伸其起點端，此步驟是為了拉長原本的顏色，盡可能多把零碎空間去除，此步驟不會去延伸終點端是因為考量，終點端的線段若可以延長是由於原先早於該顏色的色塊佔據，但若延伸此部分，則容易導致空白的空間更加零碎，會增加進一步優化的難度。



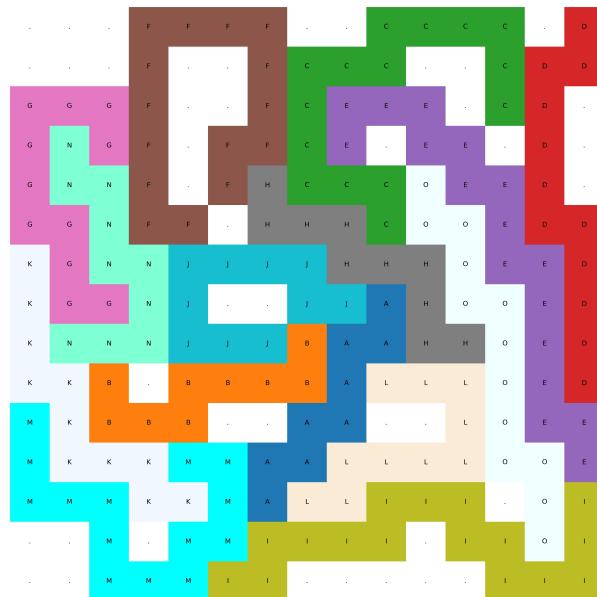
d. seperate\_long\_path()

此步驟會去將原先足夠長的顏色通道，根據其周遭的空白空間去切分為多段，讓其延伸到原先通道附近的空白空間。此部分主要是對長度做犧牲，由於越長的顏色通道對於生成題目的限制越大，因此這樣的調整可以讓題目生成速度提高。



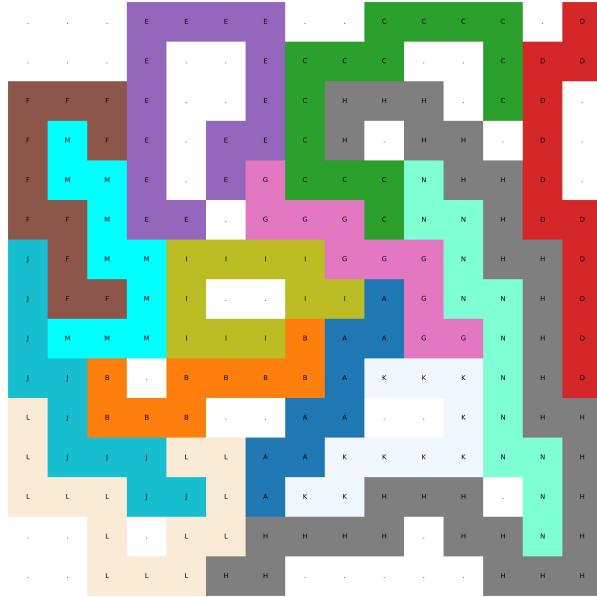
e. extend\_two\_ends\_of\_path()

在做完上述優化後，會把既有線段的兩端同時延長，將空白空間盡可能刪除。此步驟主要是考量到如果留下越多空白空間到最後一步，通常會需要更多時間才能收斂出結果。



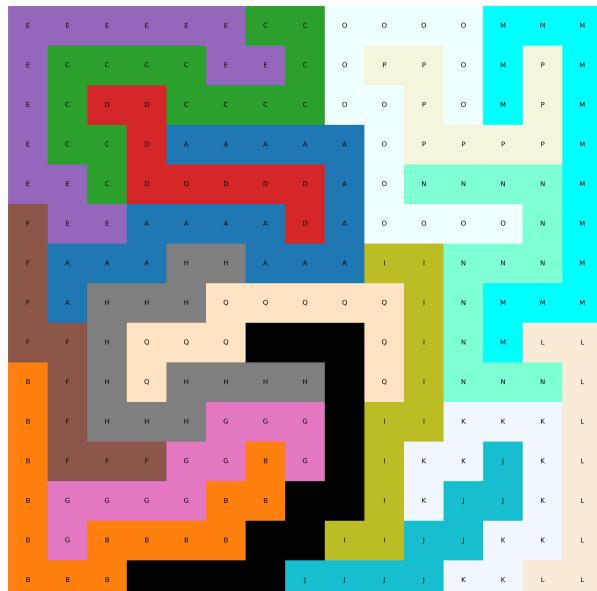
f. `connect_two_paths()`

此步驟會將既有顏色的通道，如果滿足不相鄰的限制，且其兩顏色的其中一端有相鄰，將之相連。此步驟之目的在於減少顏色數量，由於在最後一步執行過程中容易產生大量顏色而出錯。



g. `fill_empty_cells()`

此為整個程式的最後一步，因此會使用較為激進的作法，通常也是造成程式運行時間較長的原因。此步驟會以空白空間為單位去優化，將空白空間附近的隨機一個顏色通道刪除，然後以原本空白空間為起點，重新產生顏色並盡可能延伸。而如果新的線段不夠長，則會被刪除，留下更多空白空間，來讓下一次的優化有更多的完整空間可以利用。



## B. FLOW FREE 遊戲解題

1. 使用者輸入: 於程式中設定題目名稱，如 11x13\_game.txt
2. 程式輸出: solution.txt
3. C 語言程式: FlowFree.c、Engine.c、Engine.h、Interface.c、Interface.h、AC3.c、AC3.h、Astar.c、Astar.h
4. 資料結構使用: 2D array(Graph), Queue, Linked List and Tree with dynamic memory allocation
5. 主程式運行流程:

Flow Free 解題是一個 NP Hard 問題，目前有許多人嘗試以不同的演算法挑戰這個題目，例如以透過 Boolean Satisfiability 抑或是 Dijkstra's 演算法進行解題。我們閱讀過許多相關文獻後，決定以 AC-3 和 Astar 演算法兩者搭配進行解題，透過 AC-3 快速化簡題目，找出初始解減少 Astar 搜尋路徑需要花費的時間，再透過 Astar 進行路徑搜尋得到最終解。我們將 Flow Free 解題大致拆分成 4 個部分，依序分別為 a. 讀取題目並建立 graph 儲存題目、b. AC-3 演算法化簡題目、c. Astar 演算法得到 Flow Free 題目對應的解、d. 輸出解答並存入 txt。

### a. 讀取題目並建立 graph 儲存題目

#### i. ReadMap()

將 txt 讀入程式中，透過 2D Array 建成的 Graph 在程式中建立題目模型“game”，並透過 Linked List “colorLines”儲存每一個顏色的基本資料，包括起點、終點以及目前連線情況。

### b. AC-3 演算法化簡題目

在讀取完題目後，我們先透過 AC-3 演算法透過 IsRevise() 中多個條件檢查題目中不同節點的關係，透過條件刪除無解的情況，化簡題目。

#### i. InitializeConstraintQueue()

我們將任兩相鄰節點的關係透過 Queue 儲存，此函式會將 Queue 初始化，並將所有相鄰節點關係 push 進 Queue 中。若 Queue 不為空則目前仍有相鄰節點的關係需透過 IsRevise() 檢查，反之則已檢查過所有相鄰節點之間的關係。

#### ii. IsRevise()

我們透過 IsRevise() 檢查相鄰節點的關係是否有因為條件改變，IsRevise() 在檢查相鄰節點 node1 和 node2 之間的關係時只能更改 node1 的狀態，因此，若 node1 因為檢查條件改變狀態，則需透過 findRevisedArc() 重新檢查被修改的節點，其鄰近節點與該節點的關係，因此需將上述相鄰節點關係如 node2 和 node1 再 push 進 Queue 中，檢查 node2 狀態是否需要更改。

在 IsRevise() 中有以下條件：

- 若 node1 為單色，則不需檢查
- 如果 node2 已確定為單色，且 node1 為題目矩形圖形的四個角，則 node1 與 node2 的顏色相同
- 若 node2 為題目矩形圖形的四個角，其為單色，則 node1 與 node2 的顏色相同
- 若 node1 非矩形的四個角且四周皆為單色，則根據四周的格子決定 node1 的顏色
- 若 node2 只剩唯一相鄰格子 node1 未確定(非單色)，則根據 node2 四周的結果決定 node1 「與 node2 的顏色相同」或「與 node2 的顏色不相同」
- 若 node1 只剩唯一相鄰格子非單色，則 node1 中的可能性為周圍單色格子的聯集
- 若 node1 為矩形的四個角，且其對角為單色，則 node1 不會是該顏色

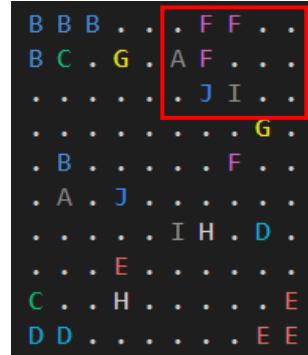
iii. findRevisedArc()

如上述，當 node1 因為檢查條件改變狀態，則需透過 findRevisedArc() 重新檢查被修改的節點，其鄰近節點與該節點的關係，因此需透過此函式將上述相鄰節點關係如 node2 和 node1 再 push 進 Queue 中，再次透過 IsRevised() 檢查 node2 狀態是否亦需要更改。

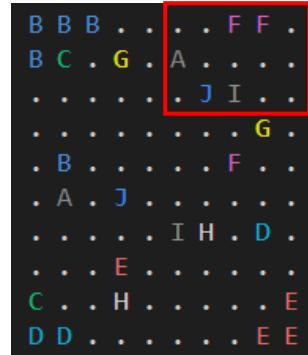
c. Astar 演算法得到 Flow Free 題目對應的解

當 AC-3 將題目進行化簡後，若仍無法得到唯一解，則我們要透過 Astar 演算法進行路徑搜尋找出各個顏色從起點到終點的路徑。我們透過 tree 將合法的路徑和可能的路徑選項記錄下來，在每次搜尋新路徑時，會檢查當前節點四周相鄰的節點是否為合法路徑，並選擇透過 Heuristic function 計算最小 cost 的節點繼續搜尋，其他合法相鄰節點則儲存在目前節點的其他子節點中。當一路徑為非法路徑，則透過回溯尋找該節點的父節點所保

留的其他合法可能路徑繼續進行搜尋，而非法路徑將被遺棄且不會再被搜尋，如下圖所示。



透過 IsPathLegal() 發現當前路徑為非法



尋找父節點所保留的其他合法可能路徑繼續進行搜尋

在 Astar 演算法中最為關鍵的函式有兩個，分別是 Heuristic() 和 IsPathLegal()，前者計算從目前節點走到新節點的 cost，後者檢查新增的節點是否合法，若合法則加入 tree，否則捨棄。

### i. Heuristic()

此函式用於計算從目前節點走到新節點的 cost，條件如以下所示：

- mahattanDistance: 計算目前新節點與終點的曼哈頓距離
- obstaclePenalty: 若新節點與終點間的已確認節點愈多(單色節點)，則每多一已確認節點則 obstaclePenalty 加 5，減少路徑搜尋至死路的情況
- edgeDiscount: 若目前節點和新節點皆位於題目範圍的邊界(四個邊)，則 edgeDiscount 為 3，反之則為 1，有利於讓路徑維持於邊界行走，提升合法路徑搜尋效率

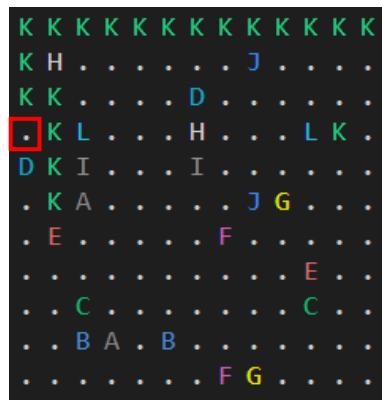
公式為： $\text{cost} = (\text{mahattanDistance} + \text{obstaclePenalty}) / \text{edgeDiscount}$

### ii. IsPathLegal()

此函式目的為檢查新增的節點是否合法，透過 6 個條件檢查合法性，

若合法則加入 tree，否則捨棄。

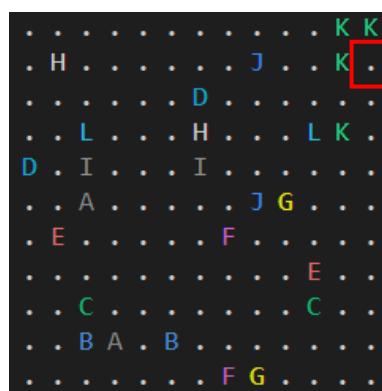
- 如果新增節點的(左、左上、上)、(左、左下、下)、(右、右上、上)、(右、右下、下)與其顏色相同則不新增節點
- 如果新增節點的四周有兩個以上的同色節點，且其中之一並非終點，則不新增節點
- 如果新增節點導致該節點周圍的未確定節點被孤立(周圍節點顏色確定且相鄰節點沒有兩點同色且該色不為目前搜尋路徑顏色)，則不新增節點



- 如果新增節點與其路徑中產生非法未確認節點，則不新增節點



- 若新增節點後該節點周圍的節點無法合法拓展，則不新增節點

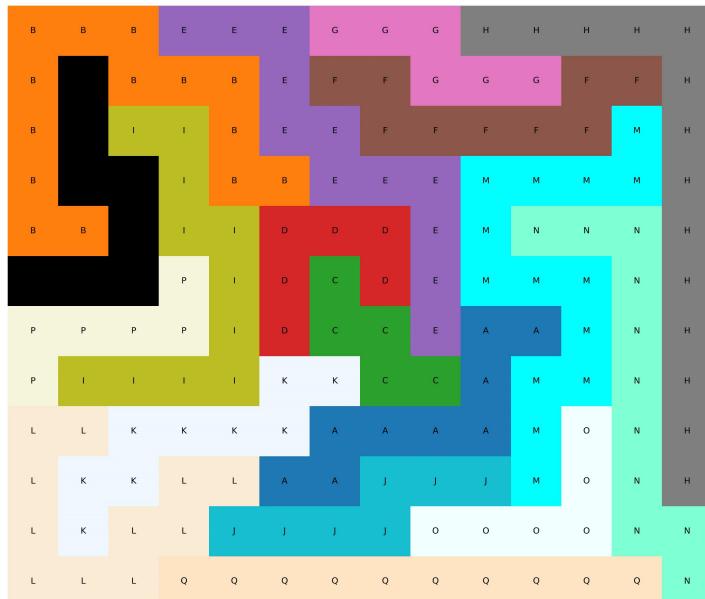


➤ 如果新增節點使其他顏色無法連線，則不新增節點



d. 輸出解答並存入 txt

在 Astar 演算法找出所有顏色的唯一合法路徑後，將最終結果輸出成 txt 檔，並透過 color\_auto\_process.py 將結果圖像化展示，如下圖所示。



實測結果：

題目大小	運行時間(sec)
5x5	0.017
6x6	0.017
7x7	0.022
10x10	0.062
11x13	8.924(單色路徑長)
12x14	0.136

運行時間將因題目難度有所不同，若題目顏色愈少，單色路徑愈長，題目愈大，則所需運行時間將越長。