ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

BÁO CÁO ĐỒ ÁN

MÔN HỌC: PHÂN TÍCH VÀ THIẾT KẾ THUẬT TOÁN

ĐỀ TÀI

GRAPH COLORING – BÀI TOÁN TÔ MÀU ĐỒ THỊ

Sinh viên thực hiện: Đoàn Duy Ân – 19521172
Võ Phạm Duy Đức – 19521383
Trịnh Công Danh – 19521326

Lớp: CS112.L22.KHCL

Thành phố Hồ Chí Minh, ngày 25 tháng 8 năm 2021

MỤC LỤC

CHƯƠNG 1: GIỚI THIỆU

1. Giới thiệu về bài toán:

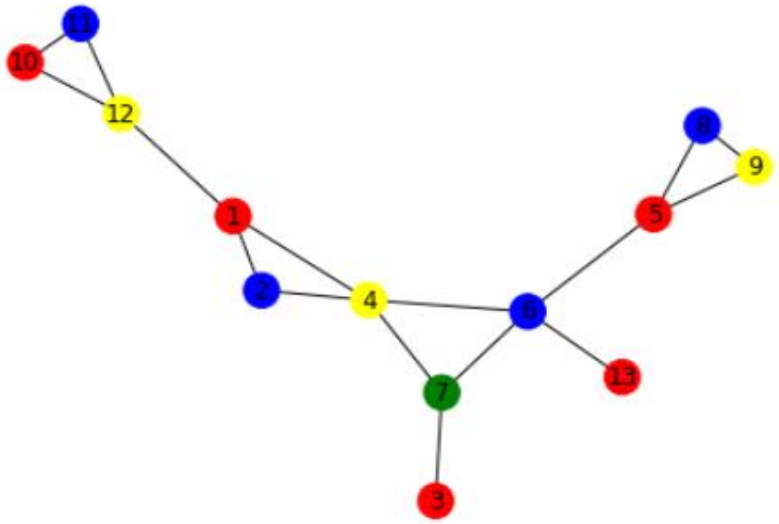
- Giới thiệu sơ lược:
    - Bài toán tô màu đồ thị là gán màu cho các phần tử nhất định của đồ thị theo những ràng buộc nhất định. Tô màu đỉnh là vấn đề tô màu đồ thị phổ biến nhất. Bài toán là, cho trước m màu, hãy tìm cách tô màu các đỉnh của đồ thị sao cho không có hai đỉnh liền kề nào được tô cùng màu. Các bài toán tô màu đồ thị khác như Tô màu cạnh (Không có đỉnh nào được nối bởi hai cạnh cùng màu) và Tô màu khuôn mặt (Tô màu bản đồ địa lý) có thể được chuyển thành tô màu đỉnh. Số màu: Số màu nhỏ nhất cần thiết để tô màu một đồ thị G được.
- Phát biểu bài toán
    - Cho 1 đồ thị vô hướng G có N đỉnh, việc cần làm là gán K màu cho mỗi đỉnh của đồ thị, sao cho hai đỉnh kề nhau không được trùng màu, và số màu được sử dụng là ít nhất. Chúng ta có thể hiểu như sau:
    - Hãy để v là bất kỳ đỉnh nào trong đồ thị và u_1, u_2,…..,u_n là những đỉnh liền kề của v:

      Color(v) != Color(u_1) Λ Color(v) != Color(u_2) Λ…. Λ Color(v) != Color(u_n)
    - Input: Dữ liệu đầu vào cho thuật toán sẽ là 1 đồ thị vô hướng G gồm có số đỉnh là V, tập cạnh và số màu giới hạn
        - Số đỉnh: Kiểu dữ liệu là số nguyên (int) cho biết số đỉnh của đồ thị G
        - Tập cạnh: Kiểu list, mỗi phần tử trong list là 1 mảng có 2 phần tử, đó là liên kết giữa 2 đỉnh hay còn gọi là 1 cạnh của đồ thị
        - Số màu giới hạn: Chỉ có thể tô màu cho mỗi đỉnh trong giới hạn màu này

      1 ví dụ về một input như sau:
        - Số đỉnh: 13
        - Tập cạnh: [[1, 2] , [1, 4], [1, 12], [2 , 4], [3, 7], [4, 7], [4, 6], [5, 6], [5, 9], [5, 8], [6 , 7], [6, 13], [8, 9], [10, 12], [10, 11], [11, 12]]
        - Số màu giới hạn: 4

- Output: Dữ liệu đầu ra của thuật toán sẽ là 1 mảng một chiều gồm V phần tử tương ứng với V đỉnh, mỗi phần tử là một số nguyên k ($1 \leq k \leq m$) thể hiện màu của mỗi đỉnh. Ví dụ: chỉ số 0 của mảng sẽ là đỉnh đầu tiên và giá trị tại chỉ số 0 của mảng sẽ là màu của đỉnh đó

  1 ví dụ về một output như sau:
  - Colors = [1, 2, 1, 3, 1, 2, 4, 2, 3, 1, 2, 3, 1]
  - Hình ảnh đồ thị sau khi được tô màu. Giá trị 1 sẽ được tô màu đỏ, giá trị 2 sẽ được tô màu xanh dương, giá trị 3 sẽ được tô màu vàng, giá trị 4 sẽ được tô màu xanh lá.



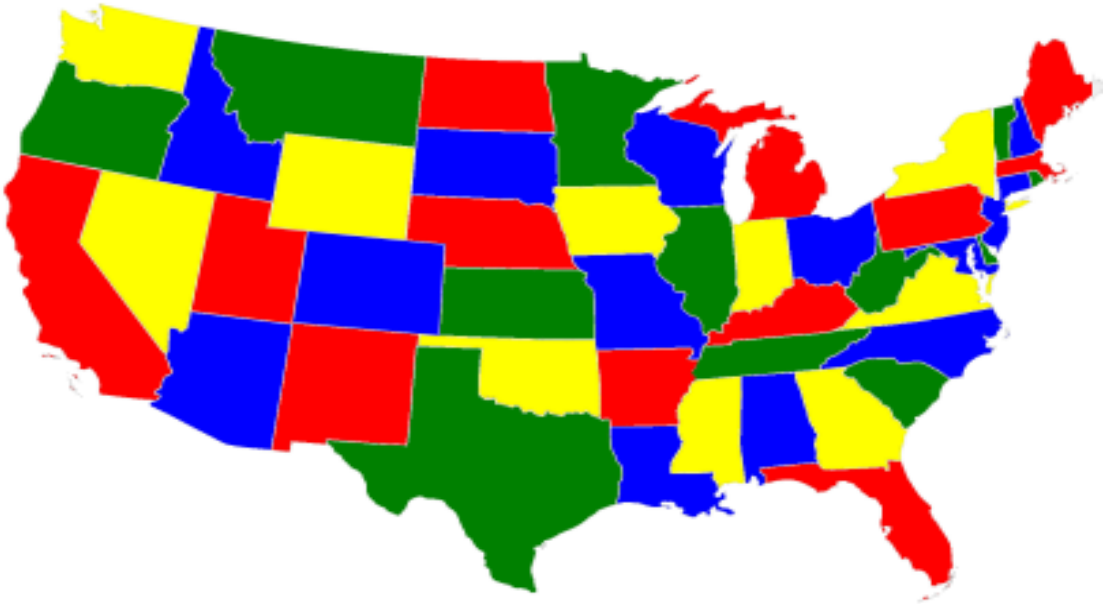
Hình ảnh minh họa cho một đồ thị đã được tô màu

- **2. Lịch sử:**

- Vấn đề này lần đầu tiên được đề cập vào năm 1852 bởi Francis Guthrie khi ông thử tô màu bản đồ nước Anh và ông nhận ra rằng chỉ cần bốn màu khác nhau là đủ. Ông đã đem vấn đề này hỏi người anh trai là Fredrick, lúc đó đang là sinh viên của trường Đại học Học viện London (UCL). Fredrick đã đưa vấn đề này hỏi thầy của mình là nhà toán học Augustus De Morgan nhưng người thầy cũng chưa biết rõ vấn đề này.
- Người đầu tiên giới thiệu vấn đề ra trước công chúng là nhà toán học Arthur Cayley vào năm 1878 tại Hội Toán học London, ông đã chỉ ra người đề cập vấn đề là De Morgan. Vào tháng 10/1852, giáo sư De Morgan ở trường Đại học Luân Đôn viết thư cho đồng nghiệp của mình là ông William Hamilton để bàn về bài toán: "Mọi

bản đồ đều có thể tô bằng 4 màu sao cho hai nước nằm kề nhau phải được tô bằng hai màu khác nhau".

- Người đầu tiên chứng minh định lý này là Alfred Kempe vào năm 1879. Năm 1880, có thêm một cách chứng minh khác của Peter Guthrie Tait. Nhưng đến năm 1890 Percy Heawood đã chỉ ra sai lầm trong cách chứng minh của Kempe, và đến năm 1891 Julius Petersen chỉ ra sai lầm trong cách chứng minh của Tait.
- Trong việc chỉ ra sai lầm của Kempe, Heawood còn chứng minh rằng tất cả các Đồ thị phẳng phải sử dụng năm màu khác nhau, và làm cơ sở phát triển cho lời giải sau này
- Trong những năm 1960 và 1970, nhà toán học người Đức là Heinrich Heesch đã phát triển phương pháp sử dụng máy vi tính cho việc chứng minh vấn đề.
- Năm 1976, cuối cùng thì định lý cũng được chứng minh bởi Kenneth Appel và Wolfgang Haken tại trường Đại học Illinois với sự trợ giúp của máy vi tính (trong khoảng 1000 giờ máy). Nhà khoa học John A. Koch cũng góp phần cải tiến thuật toán để giải quyết trọn vẹn bài toán 4 màu.
- Tô màu đồ thị đã được nghiên cứu như một thuật toán từ đầu những năm 1970, bài toán về số màu sắc là một trong 21 bài toán NP đầy đủ của Karp từ năm 1972, cùng thời gian đó thì các thuật toán được giải trong thời gian cấp số mũ khác nhau cũng được phát triển dựa trên giải thuật quay lui.



Hình 1: Bản đồ nước Mĩ được tô theo định lý 4 màu

3. Phát biểu liên quan thực tế:

- Trong 1 học kì tại trường UIT, tổng số môn học khi mở ra để cho sinh viên đăng kí học phần lên tới khoảng n môn học được đánh dấu từ 1 đến n, khi đến cuối kì thì nhà trường sẽ sắp xếp lịch thi nhưng chúng ta chắc chắn rằng không bao giờ mà chúng ta bị trùng lịch thi. Vì vậy chúng ta thấy là người ta sẽ vận dụng bài toán tô màu đồ thị vào. Mỗi 1 đỉnh là 1 môn học. Nếu có 1 sinh viên cùng học cả 2 môn thì 2 đỉnh đại diện cho 2 môn này sẽ được nối lại bằng 1 cạnh và sẽ không được tô màu trùng nhau tức là 2 môn này sẽ không bị trùng lịch.
- Input: Số lượng môn n mở trong 1 học kì, số lượng màu m thể hiện số ca thi trường có thể tổ chức, một dãy số chỉ sự liền kề giữa môn i và j. Nếu như cặp [i,j] không tồn tại trong dãy nghĩa là chúng không liền kề.
- Output: Tổng số màu có thể tô ít nhất tương ứng với số ca thi ít nhất mà trường có thể tổ chức trong học kì đó, nếu không có cách giải nào phù hợp thì xuất ra “No solution”

- Chú ý: Tất cả các môn học đều được tổ chức thi tập trung.
- Ví dụ:

Input	Output
n = 100 m = 6 arr = [[1,2],[2,3],[3,4],[4,1],[4,2],[3,1]]	4
n = 50 m=4 arr = [[1,2],[2,3],[3,4],[4,5],[5,1],[5,2],[4,2],[4,1],[5,3]]	No solution

4. Ứng dụng:

- Bài toán tô màu đồ thị có số lượng ứng dụng rất lớn:
  1) ***Lập lịch trình hoặc lịch thi:***
  2) ***Chỉ định tần số vô tuyến điện thoại di động:*** Khi tần số được chỉ định cho các tháp, các tần số được gán cho tất cả các tháp ở cùng một vị trí phải khác nhau. Làm thế nào để gán tần số với ràng buộc này? Số lượng tần số tối thiểu cần thiết là bao nhiêu? Bài toán này cũng là một ví dụ của bài toán tô màu đồ thị trong đó mọi tháp đại diện cho một đỉnh và một cạnh giữa hai tháp biểu thị rằng chúng nằm trong phạm vi của nhau.
  3) ***Sudoku:*** Sudoku cũng là một biến thể của bài toán tô màu Đồ thị trong đó mỗi ô biểu thị một đỉnh. Có một cạnh giữa hai đỉnh nếu chúng ở cùng hàng hoặc cùng cột hoặc cùng khối.
  4) ***Phân bổ thanh ghi:*** Trong tối ưu hóa trình biên dịch, cấp phát thanh ghi là quá trình gán một số lượng lớn các biến chương trình mục tiêu vào một số lượng nhỏ các thanh ghi CPU. Bài toán này cũng là một bài toán tô màu đồ thị.
  5) ***Đồ thị Bipartite:*** Chúng ta có thể kiểm tra xem một đồ thị có phải là Bipartite hay không bằng cách tô màu cho đồ thị bằng cách sử dụng hai

màu. Nếu một đồ thị nhất định có thể có 2 màu, thì nó là Bipartite, ngược lại thì không.

6) ***Tô màu bản đồ:*** Bản đồ địa lý của các quốc gia hoặc tiểu bang không có hai thành phố liền kề không thể được gán cùng một màu. Bốn màu là đủ để tô màu cho bất kỳ bản đồ nào.

CHƯƠNG 2: CÁC PHƯƠNG PHÁP ĐỂ GIẢI BÀI TOÁN

I/ Greedy Coloring (Tô màu tham lam):

1. Lịch sử phát triển của phương pháp:

- Trong lý thuyết đồ thị và trí tuệ nhân tạo, thuật toán tô màu tham lam (*Greedy Coloring*) là một trong những phương pháp tô màu cho đồ thị áp dụng giải thuật tham lam (*Greedy algorithm*). Giải thuật tham lam (*Greedy algorithm*) là một thuật toán giải quyết bài toán bằng cách chọn phương án tốt nhất hiện có mà không cần quan tâm về kết quả trong tương lai mà nó sẽ mang lại. Nói cách khác, nó sẽ lựa chọn tốt nhất một cách cục bộ nhằm mục đích tạo ra kết quả tốt nhất trên toàn cục.
- Các thuật toán tham lam hoạt động hiệu quả cho các bài toán mà trong đó, ở mỗi bước đi, sẽ có một lựa chọn tối ưu cho bài toán đó, và sau bước cuối cùng, thuật toán đưa ra giải pháp tối ưu hoàn chỉnh cho bài toán.

2. Ý tưởng của thuật toán:

- Thuật toán tô màu tham lam xem xét đồ thị G(V) với tập hợp các đỉnh V = $[v_1, \dots, v_n]$ và tập các đỉnh kề A_{v_j}. Đầu tiên ta xét các đỉnh theo thứ tự và gán cho mỗi đỉnh một màu riêng theo nguyên tắc: các đỉnh không kề với đỉnh đang xét (không có cạnh nối trực tiếp) thì được phép tô cùng một màu, cấm tô màu đó cho các đỉnh có cạnh kề với đỉnh đang xét. Thuật toán lặp lại cho đến khi tất cả các đỉnh được tô màu.

3. Mã giả:

Funtion Greedy_Coloring
Input:
Một đồ thị vô hướng G với các đỉnh V(G) = $\{v_1, v_2, \dots, v_n\}$
Danh sách các màu sẽ là {1, 2, 3, … , n}
Output:
Danh sách các đỉnh đã được tô màu

set c(v_j) ← 0 ∀ 1 ≤ j ≤ n
set c(v_1) ← 1
for v_2 to v_n **do**
// Thực hiện tô màu cho đỉnh hiện tại bằng màu được đánh số thấp nhất và màu đó chưa được tô ở tập các đỉnh kề với nó A_{v_j}
// Nếu nếu tất cả các màu được sử dụng trước đó đều được tô ở A_{v_j} thì sẽ tô một màu mới cho v_j
C(v_j) ← min(k ϵ ℕ | k > 0 and c(u) ∀ u ϵ A_{v_j})
end for

4. Phân tích độ phức tạp bằng các phương pháp toán học:

- Đồ thị G sẽ có n đỉnh, mỗi đỉnh sẽ có E quan hệ với các đỉnh còn lại. Với mỗi đỉnh sẽ có m lần thực hiện tìm màu cho đỉnh đó. Dựa vào mã giả, ta tìm độ phức tạp thuật toán bằng toán học như sau:
  - Đầu tiên gán các giá trị màu cho các đỉnh là 0 thì thời gian thực hiện cho thao tác này sẽ là n lần
  - Gán màu 1 cho đỉnh đầu tiên sẽ có thời gian thực hiện là 1 lần
  - Thực hiện lặp từ đỉnh thứ 2 đến đỉnh thứ n sẽ có n – 1 lần thực hiện, mỗi lần lặp ta sẽ thực hiện các bước sau
    - Thực hiện xác định màu của những đỉnh kề với đỉnh hiện tại, mỗi đỉnh sẽ có E quan hệ với đỉnh còn lại nên thời gian cho lần thực hiện này là E lần
    - Sau khi xác định màu của những đỉnh kề với đỉnh hiện tại, tiếp theo sẽ tô màu cho đỉnh đó, thực hiện vòng lặp đến số lượng m màu nào đó nên sẽ có m lần thực hiện

- Thời gian chạy của thuật toán sẽ là

$$T(n) = n + 1 + \sum_{i=2}^{n} (E_i + m_i)$$

→ Độ phức tạp trung bình của thuật toán sẽ là: O($n + 1 + \sum_{i=2}^{n}(E_i + m_i)$)

- Ta có E sẽ có giá trị trong đoạn [0, n - 1], m sẽ có giá trị trong đoạn [1, n] vì số màu để tô không thể vượt quá số đỉnh nên trong trường hợp xấu nhất, mỗi

đỉnh sẽ đều có quan hệ với n – 1 đỉnh còn lại nên mỗi lần lặp sẽ thực hiện n – 1 lần xác định màu của đỉnh liền kề, số màu dùng để tô sẽ tăng dần qua mỗi lần lặp nên tổng số lần thực hiện để tìm màu thích hợp cho đỉnh sẽ là tổng từ màu 2 đến màu n (vì màu 1 đã được tô trước đó)

$$T(n) = n + 1 + (n-1) * (n-1) + 2 + \cdots + n$$

$$T(n) = n^2 - 2n + 1 + 1 + 2 + \cdots + 2n$$

$$T(n) = n^2 + 4 + \cdots + n - 1$$

→ Độ phức tạp trong trường hợp xấu nhất của thuật toán là: O(n^2)

6. Mã nguồn cài đặt:

```
def addEdge(Edges, V):
  G = [[] for i in range(V)]
  # Tạo 1 cạnh giữa 2 đỉnh
  for i in Edges:
    G[i[0] - 1].append(i[1] - 1)
    G[i[1] - 1].append(i[0] - 1)
  return G
def Greedy_Coloring(G, V, m):
    color = [-1] * V # color là mảng chứa màu của mỗi đỉnh
    available = [False] * V # Mảng available thể hiện các màu của các
                            # đỉnh liền kề với đỉnh đang xét
    color[0] = 1 # Tô màu số 0 cho đỉnh 0
    for i in range(1, V):
        for u in G[i]:
            if color[u] != -1:
                available[color[u] - 1] = True # Nếu đỉnh liền kề đã được đánh số màu thì màu đó
                                               # sẽ không được dùng cho đỉnh hiện tại
        count = 1 # Biến count biểu thị số thứ tự màu

        # Tìm màu có sẵn đầu tiên cho đỉnh đang xét
        while count <= m:
            if available[count - 1] == False:
                break
            count += 1
        # Nếu số màu cần gán cho các đỉnh mà nhiều hơn số màu giới hạn thì sẽ
        # trả về False (Không thực hiện tô được cho đồ thị)
        if count > m:
            return False
        # Gán màu cho đỉnh
        color[i] = count

        # Cập nhật lại các giá trị của available về False
        # để thực hiện vòng lặp tiếp theo
        for u in G[i]:
            if color[u] != -1:
                available[color[u]] = False
    return color
```

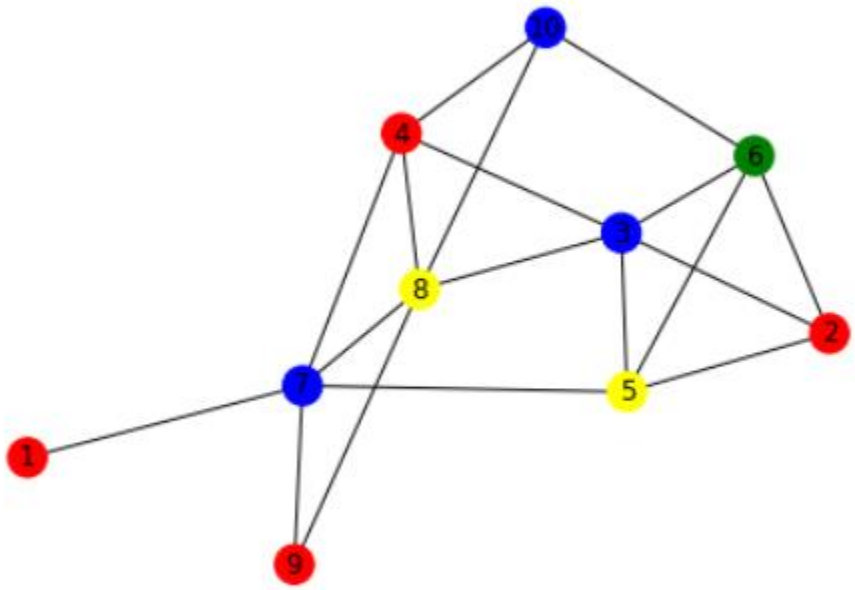
7. Cách thức phát sinh input/output đã dùng để kiểm tra tính đúng đắn của quá trình cài đặt:

- Cách 1: Hàm phát sinh input tự thực hiện
  - Hàm **generate_edge** sẽ trả về 1 tập các liên kết ngẫu nhiên của các đỉnh (hay goi là các cạnh). Hàm sẽ giới hạn số cạnh sao cho một đỉnh chỉ sinh màu trong giới hạn m màu và loại nếu phát sinh các cạnh trùng nhau, ví dụ: [2, 1] được phát sinh sau khi đã có [1, 2] thì sẽ loại bỏ giá trị phát sinh [2, 1]

```
[ ]  # Hàm phát sinh liên kết ngẫu nhiên giữa các đỉnh
     import random
     def generate_edge(V, m):
       max_edges = (V * (V - 1)) // 2 # Tìm số cạnh của đa giác V đỉnh
       edges = [] # Mảng edges chứa các cạnh
       if max_edges != 0:
         n_edges = random.randint(int(max_edges * 0.25), int(max_edges * 0.5)) # Random số cạnh
         i = 1
         while i <= n_edges:
           a = random.randint(1, V)
           b = random.randint(1, V)
           while a == b:
             a = random.randint(1, V)
           # Giới hạn 1 đỉnh không liên kết quá m đỉnh
           # a
           while True:
             sum = 0
             for k in edges:
               sum += k.count(a)
             if sum > m:
               a = random.randint(1, V)
             else:
               break
           # b
           while True:
             sum = 0
             for k in edges:
               sum += k.count(b)
             if sum > m:
               b = random.randint(1, V)
             else:
               break
           temp = [a, b]
           #print("temp: ", temp)
           if temp not in edges and temp[::-1] not in edges:
             edges.append(temp)
             i += 1
       return edges, n_edges
```

  - Với những input phát sinh từ hàm **generate_edge**, thực hiên cho ra output với mã nguồn cài đặt ở trên để kiểm tra tính đúng đắn của input sinh ra từ hàm này. Nhóm sẽ sử dụng thư viện Networks để vẽ đồ thị, với input được phát sinh từ hàm **generate_edge** như sau:

- Số đỉnh: 10
- Số cạnh: 18
- Tập cạnh: [[3, 5], [3, 2], [5, 2], [7, 1], [10, 6], [5, 6], [10, 4], [3, 8], [5, 7], [8, 4], [2, 6], [10, 8], [8, 7], [4, 7], [6, 3], [8, 9], [9, 7], [3, 4]]
- Kết quả sau khi chạy mã nguồn: [1, 1, 2, 1, 3, 4, 2, 3, 1, 2]



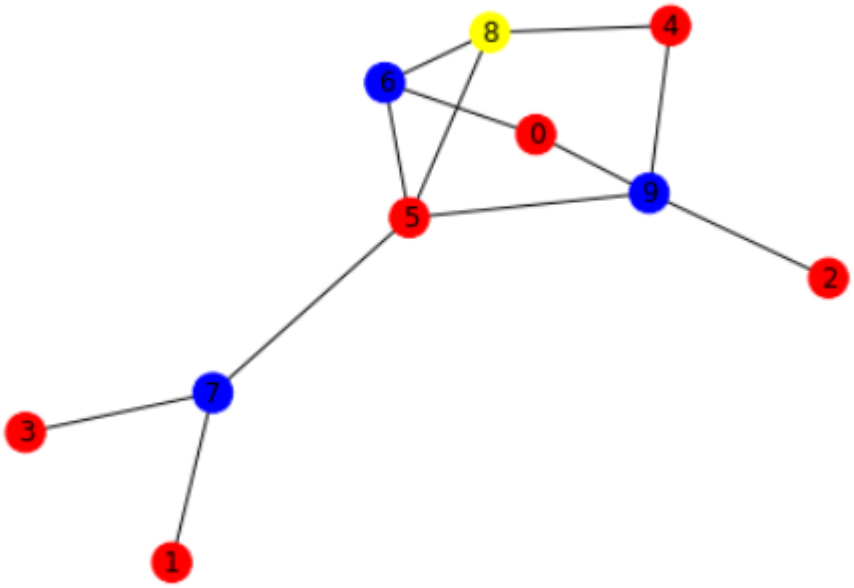
Ảnh biểu diễn output với input được phát sinh từ hàm ***generate_edge***

- Việc phát sinh input bằng cách tự thực hiện này nếu cho số cạnh lớn (khoảng vài chục nghìn cạnh) thì sẽ thực hiện rất lâu nên nhóm có cách thức phát sinh input khác là sử dụng thư viện có sẵn Networks

- Cách 2: Hàm phát sinh input sử dụng thư viện Networks

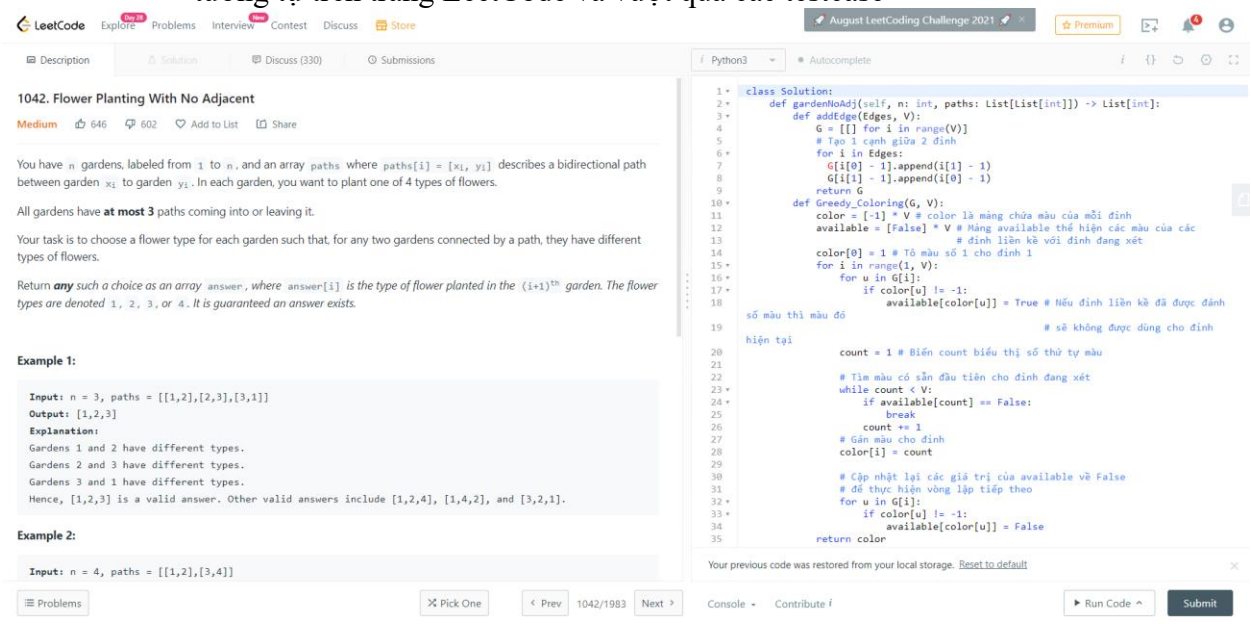
```
[ ]  from networkx.generators.random_graphs import fast_gnp_random_graph #generate a random graph with networkx
     def generate_edge_by_networkx(n, e):
       G = fast_gnp_random_graph(n, e)
       return G, G.edges
```

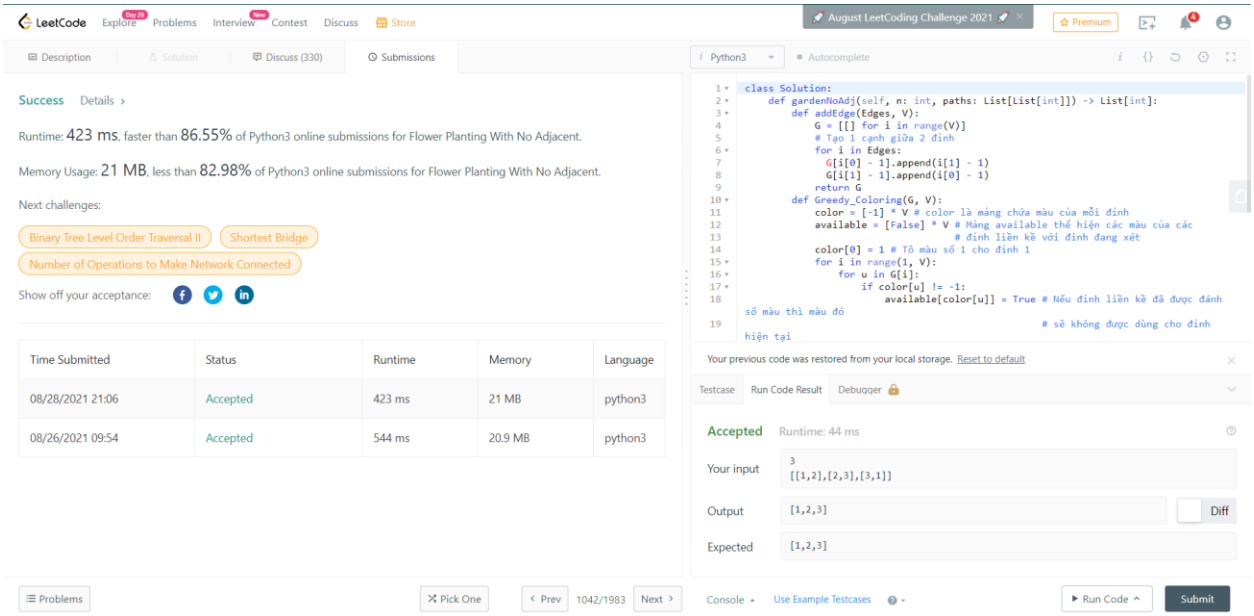
- Tương tự với hàm phát sinh input tự thực hiện ở trên, thực hiện cho ra output với input phát sinh từ hàm **generate_edge_by_networkx**
  - Số đỉnh: 10
  - Số cạnh: 12
  - Tập cạnh: [(0, 6), (0, 9), (1, 7), (2, 9), (3, 7), (4, 8), (4, 9), (5, 6), (5, 7), (5, 8), (5, 9), (6, 8)]
  - Kết quả sau khi chạy mã nguồn: [1, 1, 1, 1, 1, 1, 2, 2, 3, 2]



Ảnh biểu diễn output với input được phát sinh từ hàm ***generate_edge_by_networkx***

- Ngoài ra bài toán cũng đã được kiểm tra tính đúng đắn thông qua một bài toán tương tự trên trang LeetCode và vượt qua các testcase





- Link Leetcode: https://leetcode.com/problems/flower-planting-with-no-adjacent/

8. Phân tích độ phức tạp cài đặt bằng thực nghiệm:

Để xác định độ phức tạp của thuật toán ta tính thời gian chạy của nó bằng hàm số bậc nhất T(n) = a.f(n) + b với f(n) là các hàm log(n), n, n^2, nlog(n), n^3, $\sqrt{n}$ (Nhóm không xét $2^n, n!$ vì sau khi xét độ phức tạp bằng toán học cho kết quả n^2 thì nếu xét 2 hàm này chắc chắn sẽ cho MSE lớn hơn hàm n^2 mà mục đích của thực nghiệm là tìm độ phức tạp có MSE nhỏ nhất). Sử dụng thuật toán hồi quy tuyến tính để tính ra 2 thông số coef_ và intercept_ tương ứng là 2 thông số a và b. Sau đó thay vào biểu thức T(n) để tìm thời gian chạy, từ đó tìm các MSE. MSE có giá trị nhỏ nhất thì độ phức tạp sẽ thuộc vào lớp đó

n	sum_E	sum_m	time	n + 1 + sum_E + sum_m	O(n + 1 + sum_E + sum_m)		O(log(n)		O(n)		O(n^2)		O(nlog(n))		O(n^3)		O(sqrt(n))	
5	5	7	0	18	-0.034404919	0.001184	-1.19041	1.42E+00	-3.67E-01	1.35E-01	-0.03181	1.01E-03	-0.30195	0.091174	0.121583	0.014782	-0.75297	0.566961
7	14	14	0	36	-0.034400148	0.001183	-1.0081	1.02E+00	-3.66E-01	1.34E-01	-0.0318	1.01E-03	-0.30142	0.090852	0.121583	0.014782	-0.72583	0.526825
10	28	18	0	57	-0.034394581	0.001183	-0.81483	6.64E-01	-3.63E-01	1.32E-01	-0.0318	1.01E-03	-0.30052	0.090311	0.121583	0.014782	-0.69161	0.478322
15	78	36	0	130	-0.034375232	0.001182	-0.59513	3.54E-01	-3.59E-01	1.29E-01	-0.03179	1.01E-03	-0.29884	0.089304	0.121583	0.014782	-0.64452	0.415412
22	169	62	0	254	-0.034342364	0.001179	-0.38761	1.50E-01	-3.53E-01	1.25E-01	-0.03176	1.01E-03	-0.29622	0.087746	0.121583	0.014782	-0.59037	0.348538
33	377	115	0	526	-0.034270267	0.001174	-0.16791	2.82E-02	-3.44E-01	1.18E-01	-0.03169	1.00E-03	-0.29169	0.085084	0.121583	0.014782	-0.52054	0.270957
49	917	227	0	1194	-0.034093205	0.001162	0.046283	2.14E-03	-3.30E-01	1.09E-01	-0.03155	9.96E-04	-0.28449	0.080936	0.121584	0.014783	-0.43736	0.191287
73	1914	435	0.001	2423	-0.033767442	0.001209	0.262283	6.83E-02	-3.10E-01	9.66E-02	-0.03124	1.04E-03	-0.27278	0.074958	0.121587	0.014541	-0.33508	0.112947
109	4284	822	0.001	5216	-0.033027122	0.001158	0.479502	2.29E-01	-2.79E-01	7.86E-02	-0.03053	9.94E-04	-0.25385	0.064947	0.121598	0.014544	-0.20945	0.044289
163	9867	1736	0.002	11767	-0.031290694	0.001108	0.697541	4.84E-01	-2.34E-01	5.55E-02	-0.02896	9.58E-04	-0.22336	0.050789	0.121634	0.014312	-0.0553	0.003283
244	22347	3273	0.005	25865	-0.027553836	0.00106	0.91613	8.30E-01	-1.65E-01	2.89E-02	-0.02542	9.25E-04	-0.17452	0.032228	0.121755	0.013632	0.133732	0.016572
336	50380	6905	0.014	57622	-0.019136232	0.001098	1.089489	1.16E+00	-8.73E-02	1.03E-02	-0.01969	1.14E-03	-0.11591	0.016877	0.122034	0.011671	0.313253	0.089552
549	112520	14238	0.029	127308	-0.000665054	0.00088	1.355528	1.76E+00	9.29E-02	4.08E-03	0.000537	8.10E-04	0.028273	5.28E-07	0.123549	0.00894	0.651151	0.387072
823	253190	29951	0.074	283965	0.040858915	0.001098	1.574897	2.25E+00	3.25E-01	6.28E-02	0.040883	1.10E-03	0.225314	0.022896	0.128207	0.002938	0.999435	0.85643
1234	570968	60868	0.148	633071	0.133393987	0.000213	1.794376	2.71E+00	6.72E-01	2.75E-01	0.131616	2.68E-04	0.536782	0.151152	0.143912	1.67E-05	1.4261	1.633541
1851	1283958	129119	0.319	1414929	0.340635587	0.000468	2.014075	2.87E+00	1.19E+00	7.66E-01	0.335898	2.86E-04	1.028264	0.503056	0.196944	0.014898	1.949128	2.657317
2776	2886113	269486	0.729	3158376	0.80275882	0.00544	2.233676	2.26E+00	1.98E+00	1.56E+00	0.795234	4.39E-03	1.800926	1.149025	0.375791	0.124757	2.589389	3.461046
4164	6497849	571480	1.6	7073494	1.84051161	0.057846	2.453374	7.28E-01	3.15E+00	2.40E+00	1.829039	5.25E-02	3.014115	1.99972	0.979535	0.384976	3.373859	3.146576
6246	14628616	1218753	3.799	15853616	4.167796789	0.136011	2.673073	1.27E+00	4.91E+00	1.24E+00	4.155099	1.27E-01	4.91458	1.24452	3.017173	0.611254	4.334635	0.286905
9369	32916223	2587384	9.598	35512977	9.378765325	0.048064	2.892772	4.50E+01	7.55E+00	4.18E+00	9.388736	4.38E-02	7.886303	2.929905	9.894199	0.087734	5.511341	16.70078
MSE						0.013195		3.26081		0.5819		0.012101		0.442774		0.070385		1.609731

	Mean square errors
$\log(n)$ * 0.37557723 -2.06247389	**3.26081**
n * 0.00084582 - 0.37149097	**0.5819**
n^2 * 1.07322199e-07 - 0.03180962	**0.012101**
n * $\log(n)$ * 6.62480215e-05 - 0.30271935	**0.442774**
n^3 * 1.18831373e-11 + 0.12158275	**0.070385**
$\sqrt{n}$ * 0.06624866 - 0.90110482	**1.609731**
***(n + 1 + sum_E + sum_m)* * 2.65062966e-07 - 0.03440969**	**0.013195**

Nhận xét:

- Qua lần phân tích thực nghiệm cho độ phức tạp của thuật toán thuộc vào lớp **O(n^2).** Kết quả này bằng với kết quả khi phân tích độ phức tạp bằng thực nghiệm. Vậy độ phức tạp của bài toán tô màu đồ thị áp dụng giải thuật tham lam cho kết quả như bảng dưới đây

	Tốt nhất (n = 1)	Trung bình	Xấu nhất
Độ phức tạp	2	$n + 1 + \sum_{i=2}^{n}(E_i + m_i)$	n^2

- Ưu, nhược điểm của thuật toán: Với thuật toán trên việc cài đặt dễ dàng hơn nhưng hiệu quả không cao, thuật toán chỉ đáp ứng được yêu cầu bài toán đặt ra, nhưng không phải với số màu ít nhất có thể tô được

II/ Backtracking (Quay lui):

1. Lịch sử phát triển của phương pháp:

- Trong lý thuyết đồ thị, ý tưởng đầu tiên để giải 1 bài liên quan tới đồ thị luôn bắt đầu với thuật toán backtracking, sau đó chúng từ backtracking chúng ta mới phát triển những thuật toán tốt hơn

2. Ý tưởng của thuật toán:

- Thuật toán tô màu backtracking xét đồ thị G(V) với tập hợp các đỉnh V = [v_1, … , v_n] và tập các đỉnh kề nó. Thuật toán sẽ kiểm tra đỉnh hiện tại với màu hiện tại chúng ta muốn tô xem màu hiện tại có vi phạm hay không (trùng màu với các đỉnh liền kề), nếu không vi phạm, chúng ta sẽ tô màu cho đỉnh đó và gọi backtracking cho đỉnh tiếp theo. Với mỗi lần tô màu khác nhau, màu được chọn luôn được kiểm tra vi phạm, nếu màu hiện tại vi phạm thì sẽ qua màu tiếp theo trong dãy màu. Thuật toán lặp lại cho tới khi đã tô được hết màu cho tất cả đỉnh.

3. Mã giả:

Funtion Backtracking

Input:

Một đồ thị vô hướng G với các đỉnh V(G) = $\{v_1, v_2, \dots, v_n\}$

Danh sách các màu sẽ là {1, 2, 3, … , n}

Output:

Danh sách các đỉnh đã được tô màu

if current < n return colors //output

Endif

for $i\ in\ range$(1, n):

Kiểm tra vi phạm với màu hiện tại so với vị trí đỉnh hiện tại

Nếu không vị phạm, chúng ta tô màu đấy cho đỉnh đang xét, và gọi Backtracking cho đỉnh tiếp theo

Nếu có vi phạm, chúng ta đánh dấu màu tại đỉnh này là 0 (không có màu) và xét tiếp màu tiếp theo

end for

return False

4. Phân tích độ phức tạp bằng các phương pháp toán học:

- Do ý tưởng phát triển backtracking gần rất giống với phương pháp bruteforce, đó là thử tất cả các màu cho tới khi chúng ta đã tô được hết màu cho biểu đồ. Ta xét công thức độ phức tạp dựa trên ba biến, **n** là số đỉnh của đồ thị, **E** là số màu cần tô, và **s** là số liên kết ít nhất của đỉnh trong đồ thị
-

$$(n, E, s) = \begin{cases} E & (n = 1) \\ E * T(n-1, E, s) & (n > 1, s = 0) \\ (E - s) * T(n-1, E, s) & (n > 1, s > 0) \end{cases}$$

- Trường hợp 1: T(n, E, s) = E * T(n – 1, E, s) (n > 1, s = 0)

$$T(n, E, s) = E * T(n-1, E, s)$$
$$T(n, E, s) = E * E * T(n-2, E, s)$$
$$T(n, E, s) = E * E * E * T(n-3, E, s)$$
$$T(n, E, s) = E^{n-1} * T(1, E, s)$$
$$T(n, E, s) = E^{n-1} * E = E^n \leq E^n$$

- Vậy độ phức tạp của thuật toán ở trường hợp 1 này là $O(E^n)$

- Trường hợp 2: T(n, E, s) = (E – s) * T(n – 1, E, s) (n > 1, s > 0)

$$T(n, E, s) = (E - s) * T(n-1, E, s)$$
$$T(n, E, s) = (E - s) * (E - s) * T(n-2, E, s)$$
$$T(n, E, s) = (E - s) * (E - s) * (E - s) * T(n-3, E, s)$$
$$T(n, E, s) = (E - s)^{n-1} * T(1, E, s)$$

$$T(n, E, s) = (E - s)^{n-1} * E \leq E^n$$

➢ Vậy độ phức tạp của thuật toán ở trường hợp 2 này là $\boldsymbol{O(E^n)}$

6. Mã nguồn cài đặt:

```
def PathsToGraph(paths, n):
        graph = defaultdict(list) #build a bidrectional graph
        for u, v in paths:
            graph[u].append(v)
            graph[v].append(u)
        return graph

    def isSafe(current, graph, color, colors):
        for i in graph[current]:
            if color == colors[i-1]:
                return False
        return True

    def backtrack(current, graph, colors, n):
        if current > n:
            return True

        for color in range(1, 6):
            if isSafe(current, graph, color, colors):
                colors[current - 1] = color
            if backtrack(current + 1, graph, colors, n):
                return colors

        colors[current - 1] = 0
        return False
```

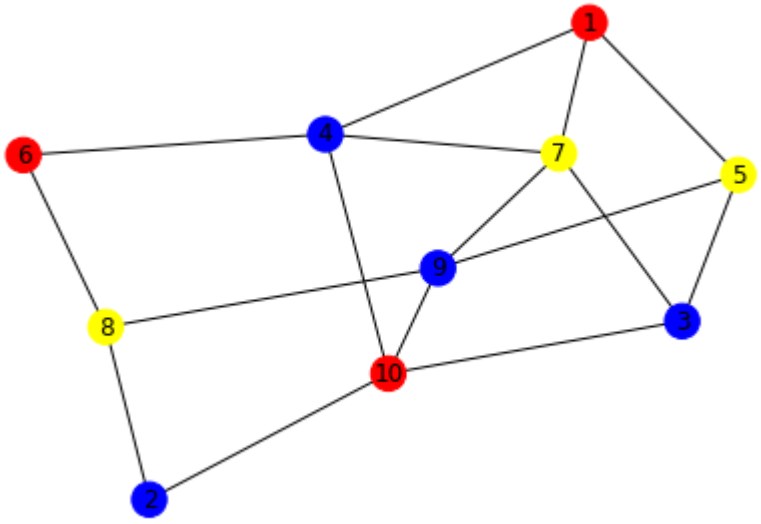
7. Cách thức phát sinh input/output đã dùng để kiểm tra tính đúng đắn của quá trình cài đặt:

Cách 1: Hàm phát sinh input tự thực hiện

- Hàm **generate_edge** sẽ trả về 1 tập các liên kết ngẫu nhiên của các đỉnh (hay gọi là các cạnh). Hàm sẽ giới hạn số cạnh sao cho một đỉnh chỉ sinh ra giới hạn là m màu và loại những trường hợp liên kết trùng với nhau. Ví dụ: phát sinh ra liên kết [1, 2] sau trước đó đã có [2, 1] thì chúng ta sẽ loại bỏ giá trị [1, 2]

```
import random
def generate_edge(V, m):
  max_edges = (V * (V - 1)) // 2 # Tìm số cạnh của đa giác V đỉnh
  edges = [] # Mảng edges chứa các cạnh
  if max_edges != 0:
    n_edges = random.randint(int(max_edges * 0.25), int(max_edges * 0.5))
                                    i = 1
    while i <= n_edges:
      a = random.randint(1, V)
      b = random.randint(1, V)
      while a == b:
        a = random.randint(1, V)
      # Giới hạn 1 đỉnh không liên kết quá m đỉnh
      while True:
        sum = 0
        for k in edges:
          sum += k.count(a)
        if sum > m:
          a = random.randint(1, V)
        else:
          break
      while True:
        sum = 0
        for k in edges:
          sum += k.count(b)
        if sum > m:
          b = random.randint(1, V)
        else:
          break
      temp = [a, b]
      if temp not in edges and temp[::-1] not in edges:
        edges.append(temp)
        i += 1
  return edges, n edges
```

- Với những input được phát sinh từ hàm generate_edges, thực hiện tạo ra output bằng thuật toán đã cài đặt để kiểm tra tính đúng đắn của input sinh ra từ hàm này, nhóm sẽ sử dụng thư viện networkx để vẽ đồ thị với input là tập các liên kết hàm **generate_edges** sinh ra và list **colors** mà thuật toán trả về. Các input gồm:
  - Số đỉnh: 10
  - Số cạnh: 16
  - Tập cạnh: [[3, 10], [8, 9], [5, 1], [6, 4], [6, 8], [5, 9], [4, 7], [3, 5], [9, 7], [2, 10], [7, 1], [7, 3], [2, 8], [9, 10], [1, 4], [4, 10]]
  - Kết quả sau khi chạy mã nguồn: [1, 2, 2, 2, 3, 1, 3, 3, 2, 1]



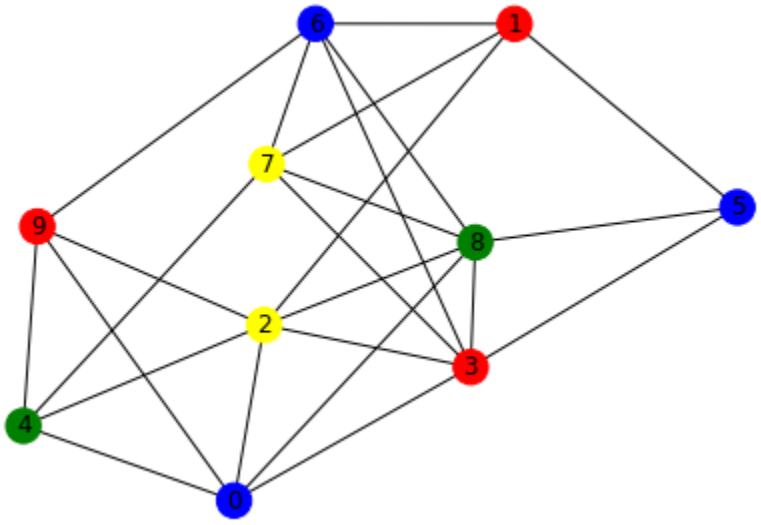
*Ảnh biểu diễn output với input được phát sinh từ hàm **generate_edge***

Cách 2: Hàm phát sinh input từ thư viện networkx

- Việc phát sinh input từ hàm **generate_edge** nếu cho số cạnh quá lớn (khoảng vài chục nghìn cạnh) thì sẽ thực hiện rất lâu, nên nhóm còn cách 2 là sử dụng thư viện networkx để tạo 1 đồ thị ngẫu nhiên với n đỉnh, và 1 tỉ lệ sinh ra cạnh giữa 2 đỉnh cho trước (ở đây là 37.5%)

```
from networkx.generators.random_graphs import fast_gnp_random_graph #ge
nerate a random graph with networkx
def generate_edge_by_networks(n, e):
  e = e #tỉ lệ tạo ra cạnh giữa 2 đỉnh
  G = fast_gnp_random_graph(n, e)
  return G, G.edges
```

- Hàm **generate_edge_by_networkx** nhận vào n đỉnh, sau đó trả về 1 đồ thị G format của networkx, với tập liên kết của đồ thị. Tương tự với hàm phát sinh **generate_edges** bên trên, thực hiện cho ra output với input phát sinh từ hàm của thư viện networkx.
  - Số đỉnh: 10
  - Số cạnh: 24
  - Tập cạnh: [(0, 2), (0, 3), (0, 4), (0, 8), (0, 9), (1, 2), (1, 5), (1, 6), (1, 7), (2, 3), (2, 4), (2, 8), (2, 9), (3, 5), (3, 6), (3, 7), (3, 8), (4, 7), (4, 9), (5, 8), (6, 7), (6, 8), (6, 9), (7, 8)]
  - Kết quả sau khi chạy mã nguồn: [2, 1, 3, 1, 4, 2, 2, 3, 4, 1]



Ảnh biểu diễn output với input được phát sinh từ hàm ***generate_edge_by_networkx***

- Để đảm bảo tính đúng đắn của input/output, hình ảnh đồ thị sau khi tô màu bởi thuật toán trên cho ra đồ thì đã được tô màu mà không có 2 đỉnh liền kề nào có màu giống nhau, dó đó có thể đảm bảo được tính đúng đắn.
- Ngoài ra, thuật toán cũng đã được thử trên Leetcode với 1 bài tượng tự với bài toán mà nhóm đang phân tích, kết quả là vượt qua hết tất cả testcase

8. Phân tích độ phức tạp bằng thực nghiệm:

n	colors	times	Olog(n)		O(n)		O(n^2)		O(nlog(n))	O(2^n)		O(n^3)		O(sqrt(n))		O(m^n))	
300	73	0.04772	-2.11363	4.671439	-1.33256	1.905174	-0.53397	0.338359	-1.21955	1.605966	2.340595	5.257277	0.056167	7.13E-05	-1.75161	3.237581	0.784806	0.543295
330	79	0.0628	-1.66502	2.985378	-1.15085	1.472945	-0.47886	0.293392	-1.06267	1.266687	2.340595	5.188351	0.069823	4.93E-05	-1.45433	2.301672	1.515736	2.111024
363	85	0.07994	-1.21642	1.680544	-0.95097	1.06277	-0.41217	0.242177	-0.88767	0.936276	2.340595	5.110562	0.087999	6.49E-05	-1.14253	1.494444	3.052427	8.83568
399	90	0.10443	-0.77135	0.766989	-0.73291	0.701146	-0.33219	0.190635	-0.69412	0.637682	2.340595	5.000435	0.111972	5.69E-05	-0.81817	0.851182	1.847744	3.039145
438	97	0.14565	-0.3324	0.228535	-0.49669	0.412601	-0.23701	0.146426	-0.4816	0.393443	2.340595	4.817785	0.143306	5.50E-06	-0.48288	0.39505	1.575578	2.044694
481	104	0.18063	0.108381	0.00522	-0.23624	0.173779	-0.12178	0.091453	-0.24417	0.180457	2.340595	4.66545	0.184955	1.87E-05	-0.13008	0.096543	1.847744	2.779271
529	115	0.24826	0.556098	0.094764	0.054499	0.037543	0.019578	0.052295	0.024373	0.050125	2.340595	4.377867	0.241111	5.11E-05	0.245578	7.20E-06	3.800114	12.61566
581	121	0.33272	0.997419	0.441825	0.369465	0.00135	0.18788	0.020979	0.3191	0.000186	2.340595	4.031563	0.314588	0.000329	0.633777	0.090635	2.544429	4.891655
639	135	0.46607	1.445289	0.95887	0.720773	0.064873	0.394204	0.005165	0.652077	0.034599	2.340595	3.513845	0.413595	0.002754	1.046791	0.337237	2.567897	4.417678
702	144	0.53822	1.887865	1.821543	1.102365	0.31826	0.640542	0.01047	1.018374	0.230548	2.340595	3.248557	0.543523	2.81E-05	1.474689	0.876974	1.847744	1.714854
772	156	0.69141	2.335253	2.702219	1.526357	0.697137	0.941398	0.062494	1.430496	0.546248	2.340595	2.719812	0.717948	0.000704	1.928177	1.529592	1.847744	1.337109
849	171	0.94657	2.782751	3.37156	1.992749	1.09449	1.305343	0.128718	1.889499	0.889116	2.340595	1.943306	0.949992	1.17E-05	2.403858	2.123688	0.028955	0.842018
933	185	1.28488	3.226819	3.771128	2.501539	1.48026	1.741808	0.208783	2.396391	1.235456	2.340595	1.114535	1.255909	0.000839	2.898784	2.604685	3.373352	4.361714
1026	199	1.65957	3.67405	4.058129	3.064843	1.974792	2.273034	0.376339	2.964412	1.702612	2.340595	0.463795	1.665231	3.21E-05	3.421399	3.104041	3.496098	3.372836
1128	216	2.20784	4.120154	3.656943	3.68266	2.175094	2.913666	0.49819	3.594899	1.923932	2.340595	0.017624	2.207986	2.13E-08	3.968028	3.098263	1.847744	0.129669
1240	232	2.87165	4.565725	2.869888	4.361047	2.218304	3.686991	0.664781	4.295414	2.027103	2.340595	0.282019	2.928252	0.003204	4.540485	2.785011	1.847744	1.048382
1364	249	3.84515	5.014331	1.366985	5.112119	1.60521	4.628501	0.613639	5.080107	1.525118	2.340595	2.263685	3.892568	0.002248	5.144877	1.689289	3.817043	0.00079
1500	270	5.31163	5.461683	0.022516	5.935875	0.389682	5.764229	0.204846	5.950762	0.408489	2.340595	8.827048	5.171945	0.019512	5.776952	0.216525	1.847744	11.9985
1650	293	6.86667	5.910289	0.914664	6.844429	0.000495	7.141959	0.075784	6.922099	0.003072	2.340595	20.48535	6.878924	0.00015	6.441694	0.180605	3.574541	10.83811
1815	319	9.02949	6.358896	7.132072	7.843839	1.405768	8.809013	0.04861	8.00275	1.054195	2.340595	44.74131	9.150912	0.014743	7.138881	3.574402	3.881872	26.49797
1996	349	12.2312	6.806324	29.42928	8.940161	10.83094	10.82033	1.990562	9.201533	9.178881	2.340595	97.82406	12.1658	0.004278	7.86813	19.03638	2.205441	100.5159
				3.473833		1.429648		0.29829		1.230009		10.75687		0.002341		2.363038		9.711234

	MSE
$\mathbf{log(n)}$ *** 3.26251017 - 28.96023592**	3.473833
$\boldsymbol{n}$ *** 0.00605703 -3.14966914**	1.429648
$\boldsymbol{n^2}$*** 2.91583121e-06 -0.79639108**	0.29829
n * $\mathbf{log(n)}$*** 0.0005368 -2.54471919**	1.230009
$\boldsymbol{2^n * 0. + 2.34059524}$	10.75687
$\boldsymbol{n^3}$ *** 1.52801032e-09 + 0.0149106**	0.002341
$\boldsymbol{\sqrt{n}}$ *** 0.35164863 -7.84234094**	2.363038
$\boldsymbol{m^n}$ *** 2.33215411e-19 + 1.84774449**	9.711234

Nhận xét:

- Qua 2 lần phân tích thực nghiêm đều cho ra độ phức tạp của thuật toán nằm khoảng ở lớp $O(n^3)$, khác so với độ phức tạp chứng minh bằng phương pháp toán học là $O(m^n)$ (m là số màu, n là số đỉnh). Do với thuật toán backtracking, số vòng lặp là không thể không đếm được, nên độ phức tạp với trường hợp xấu nhất cho backtracking là $O(m^n)$. Nhưng theo thực tế, thuật toán chạy trung bình là $O(n^3)$.
- Do là 1 thuật toán backtracking, nên số vòng lặp gọi có thể lớn cho số đỉnh và số màu lớn hơn, nên thuật toán cần rất nhiều bộ nhớ để chạy.