Rapport Algorithmique des graphes

Perrot Killiann & Espada Mora Matthieu & Nalet Antoine

Contents

Chapter 1

L'Étude des Coloriages de Graphes : Entre Culture et Utilité

Introduction

Les coloriages de graphes constituent un domaine d'étude fascinant, alliant la richesse culturelle des mathématiques à leur utilité pratique dans divers domaines. Dans cette exploration, nous plongerons dans la partie culturelle en examinant pourquoi les coloriages de graphes ont captivé l'attention des mathématiciens et quels défis stimulants ils soulèvent.

Partie culturelle

Les coloriages de graphes trouvent leurs racines dans la théorie des graphes, une branche des mathématiques. Leur attrait réside dans la manière dont ils permettent de visualiser et de comprendre les relations entre les entités représentées par les nœuds d'un graphe. Depuis les premières explorations de ce concept par Francis Guthrie au XIXe siècle avec le problème des quatre couleurs, les mathématiciens ont été intrigués par la question fondamentale : "Quel est le nombre minimum de couleurs nécessaires pour colorier un graphe de telle sorte que deux nœuds voisins aient des couleurs différentes ?" Cette question en apparence simple a donné naissance à une pléthore de résultats profonds, de conjectures célèbres, et d'applications dans divers domaines.

Partie utilitaire

Au-delà de leur intérêt académique, les coloriages de graphes trouvent des applications pratiques dans plusieurs domaines. En informatique, par exemple, la coloration de graphes est cruciale pour l'ordonnancement des tâches et la gestion des ressources. Dans les réseaux de télécommunications, les fréquences assignées aux canaux peuvent être modélisées par des coloriages de graphes pour éviter les interférences. Dans le domaine biologique, les coloriages de graphes sont utilisés pour représenter des interactions moléculaires complexes.

Conclusion

En conclusion, l'étude des coloriages de graphes offre une perspective fascinante qui transcende les frontières entre la culture mathématique pure et les applications concrètes. De la résolution du problème des quatre couleurs à l'application dans des domaines aussi divers que l'informatique et la biologie, les coloriages de graphes continuent d'enrichir notre compréhension des relations entre les entités et de nous offrir des outils précieux pour résoudre des problèmes du monde réel.

Chapter 2

Les algorithmes utilisés

Algorithme Glouton

Structures de données utilisées:

- `adj` : Liste d'adjacence
- `result` : Liste des couleurs des sommets
- `available` : Liste des couleurs disponibles

Primitives utilisées :

- `addEdge` : Utilisée pour ajouter une arête entre deux sommets si l'utilisateur veut créer son graphe à la main

Algorithm 1: Greedy Coloring

Input:

- `adj`: Adjacency list representing the graph
- `V`: Number of vertices

Output:

- `result`: List of colors of the vertices

```
1  Function addEdge(adj, V, W):
2      adj[V].append(W);
3      adj[W].append(V);
4      return ajd;
5  Function greedyColoring(adj, V):
6      result[V] ← -1, -1, ..., -1 ;                          // Initialize the list of colors to -1
7      result[0] ← 0 ;                                        // Color the first vertex with the first color
8      available[V] ← False, False, ..., False ;              // Initialize the list of available colors to False
9      for u from 1 to V − 1 do
10         for i in ajd[u] do
11             if result[i] ≠-1 then
12                 available[result[i]] ← True;
13     end
14         cr ← 0;
15         while cr < v do
16             if available[cr] == false then
17                 break;
18     end
19         result[u] ← cr;
20 end
21         for i in ajd[u] do
22             if result[i] ≠ -1 then
23                 available[result[i]] ← False;
24 end
25     return result;
```

Note:-

La complexité dépend du nombre de sommets et des arêtes dans le graphe. Si V est le nombre de sommets et E le nombre d'arêtes dans le graphe , la complexité serait de l'ordre de $O(V \times E)$.

Algorithme Welsh-Powell

Structures de données utilisées:

- `G` : Dictionnaire représentant le graphe
- `Ma` : Matrice d'adjacence
- `sommets` : Liste des sommets du graphe
- `degres` : Liste des degrés des sommets du graphe
- `result` : Liste représentant les couleurs attribuées aux sommets
- `couleurs` : Liste des couleurs disponibles

Primitives utilisées :

- `MAdjacence` : Retourne une matrice d'adjacence Ma du graphe G ou Ma[i][j] = 1 si il existe une arête entre les sommets i et j, 0 sinon
- `Sommets` : Retourne une liste des sommets du graphe G
- `Voisinage` : Retourne un ensemble des voisins du sommet donné en paramètre

Algorithm 2: Welsh-Powell Coloring

Input:

- `G`: Adjacency list representing the graph

Output:

- `result`: List of colors of the vertices

```
1  Function MAdjacence(G):
2      n ← length(G);
3      Ma ← matrix of size n × n initialize with 0, 0, ..., 0 ;
4      for i from 0 to n − 1 do
5          for j from 0 to n − 1 do
6              if i == j then
7                  continue;
8              end
9              if j in G[i] then
10                 Ma[i][j] ← 1; // If there is an edge between vertex i and j, set the
                       corresponding entry to 1
11             end
12         end
13 end
14     return Ma;
15 Function Sommets(G):
16     return list of vertices in G;
17 Function Voisinage(G, sommet):
18     return neighbors of sommet in G;
```

Algorithm 3: Welsh-Powell Coloring (continued)

Input:

- `G`: Adjacency list representing the graph

Output:

- `result`: List of colors of the vertices

```
1  Function WelshPowell(G):
2      couleurs ← 0,1,2,3,4,5 ;
3      Ma ← MAdjacence(G);
4      sommets ← Sommets(G);
5      degres, result ← emptyArray, emptyArray ;
6      nb = 0; // Counter for assigning colors
7      for sommet in sommets do
8          degres.append(len(Voisinage(G,sommet)));
9          result.append(0);
10 end
11     degres, sommets ← zip(*sorted(zip(degres, sommets)));// Sort vertices based on their degrees in
       descending order
12     for i from 0 to len(degres)-1 do
       // Assign colors to vertices
13         if result[i] == 0 then
14             nb ← nb + 1;
15             result[i] ← couleurs[nb];
16     end
17             for j from i+1 to len(degres)-1 do
18                 if result[j] == 0 and Ma[sommets[i]][sommets[j]] == 0 then
19                     for k from 0 to len(degres) -1 do
20                         passe ← 0;
21                         if Ma[sommets[j]][sommets[k]] == 1 and result[k] == result[i]
                           then
22                             passe ← 1;
23                             break;
24                     end
25                 end
26                 if passe == 0 then
27                     i                 result[j] ← result[i];
28                 end
29             end
30     end
31 end
   // Create a dictionary with the vertices as keys and their corresponding colors as values
32     d, i ← emptySet, 0;
33     for sommet in sommets do
34         d[sommet] ← result[i];
35         i ← i + 1;
36 end
37     return d;
```

Note:-

La complexité dépend du nombre de sommets dans le graphe. Si V est le nombre de sommets, la complexité serait de l'ordre de $O(V^2)$.

Algorithme de coloration par backtracking

Structures de données utilisées:

- `self.V` : Le nombre de sommets dans le graphe
- `self.graph` :Une matrice d'adjacence représentant le graphe.

Primitives utilisées :

- `init` : Initialisation de la classe Graph avec le nombre de sommets (vertices). Création d'une matrice d'adjacence self.graph initialisée à zéro.
- `isSafe` : Vérifie si la couleur c peut être attribuée au sommet v sans violer les contraintes de couleur adjacentes. Parcourt les voisins du sommet v et vérifie si la couleur c est déjà attribuée à l'un d'entre eux.
- `addEdge` : Ajoute une arête entre les sommets u et v en modifiant la matrice d'adjacence.
- `graphColourUtil` : Fonction récursive principale pour résoudre le problème de coloration du graphe. Attribue les couleurs aux sommets de manière récursive en utilisant la recherche de profondeur.
- `graphColouring` : Initialise la couleur avec 1. Utilise une boucle while pour incrémenter le nombre de couleurs (m) jusqu'à ce que la coloration soit possible. Retourne la liste des couleurs attribuées aux sommets du graphe.

Algorithm 4: Backtracking Coloring

Input:

- `G`: An graph

Output:

- `result`: List of colors of the vertices

```
function init(self, vertices):
    self.V ← vertices;
    self.graph ← 0, 0, ..., 0 matrix of size vertices × vertices;
// A utility function to check if the current color assignment is safe for vertex v
function isSafe(self, v, colour, c):
    for i from 0 to self.V-1 do
        if self.graph[v][i] == 1 and colour[i] == c then
            return False;
    return True;
function addEdge(self, u, v):
    self.graph[u][v] ← 1;
    self.graph[v][u] ← 1;
// A recursive utility function to solve m coloring problem
function graphColourUtil(self, m, colour, v):
    if v == self.V then
        return True;
    for c from 1 to m do
        if self.isSafe(v, colour, c) then
            colour[v] ← c;
            if self.graphColourUtil(m, colour, v+1) then
                return True;
            colour[v] ← 0;
def graphColouring(self):
    m ← 1;
    colour ← 0, 0, ..., 0 list of size self.V;
    while self.graphColourUtil(m, colour, 0) == False do
        m ← m + 1;
        colour ← 0, 0, ..., 0 list of size self.V;
    return colour;
```

Note:-

La complexité de l'algorithme de coloration de graphe backtracking est exponentielle dans le pire des cas. Plus précisément, la complexité de cet algorithme est O(m^V), où m est le nombre de couleurs et V est le nombre de sommets dans le graphe.

Chapter 3

La composition de notre projet

Notre projet est composé de 6 blocks principaux.

Le premier block (graph tools)

Ce bloc de code regroupe les fonctions permettant de créer, de modifier et de visualiser un graphe. Tous nos tests sont effectués sur les structures graph_matrice_adjacence et graph_liste_adjacence, à l'exception du sudoku. Nous vous recommandons ainsi de modifier la structure graph_matrice_adjacence si vous souhaitez tester les algorithmes sur un graphe différent. De plus, la structure graph_matrice_adjacence définie par défaut est un graphe en couronne, offrant ainsi une mise à l'épreuve complète des algorithmes et de leur réussite ou échec.

Les trois prochains blocks 2-4 (coloring tools)

Ces blocs renferment les algorithmes de coloration de graphe. De plus, ils contiennent du code commenté permettant de tester les algorithmes. Lorsque vous souhaitez effectuer des tests, vous pouvez décommenter le code et l'exécuter. Cependant, veillez à le recommenter par la suite afin d'éviter tout conflit avec d'autres algorithmes et une surcharge de la mémoire. N'oubliez surtout pas que si vous souhaitez changer le graphe, il vous suffit de modifier la structure graph_matrice_adjacence.

Le cinquième block (modify_graph_dynamically)

Ce bloc de code contient des fonctions permettant de modifier dynamiquement le graphe, c'est-à-dire de le modifier de manière aléatoire. Ainsi, il peut ajouter un nœud, une arête, ne rien faire, retirer un nœud, une arête, ajouter des voisins à un nœud, ou retirer des voisins à un nœud.
Le commentaire associé est :
""" observe_graph_evolution(graph_liste_adjacence, 4) """
Vous pouvez modifier le chiffre 4 par le nombre de modifications que vous souhaitez apporter au graphe.

Le sixième block (sudoku)

Ce bloc de code contient des fonctions permettant de résoudre un sudoku. Un sudoku de base est déjà fourni, mais si vous souhaitez effectuer rapidement des tests, veuillez le modifier (sudoku9) et décommenter la partie nécessaire. L'affichage montre le sudoku non rempli, puis le sudoku rempli. Vous pouvez également entrer votre propre sudoku en décommentant la première partie et en saisissant votre sudoku ligne par ligne, en utilisant des 0 pour les valeurs inexistantes. Le temps d'exécution est intéressant à observer ; vous pouvez utiliser la fonction 'time' avant de l'exécuter dans le terminal.

Note:-

N'oubliez surtout pas que si vous voulez changer le graphe il vous suffit de changer graph_matrice_adjacence. N'oubliez pas d'installer toutes les bibliothèques pour que le projet fonctionne, c'est a dire "math", "networkx", "matplotlib", "random", "numpy".

Chapter 4

Conclusion

En conclusion, notre projet explore le passionnant domaine de l'algorithmique des graphes, mettant en lumière divers aspects allant de la théorie pure à des applications pratiques. L'étude des coloriages de graphes nous a plongés dans un monde riche en culture mathématique, avec des défis intrigants tels que le problème des quatre couleurs et des conjectures célèbres.

D'un point de vue utilitaire, les algorithmes de coloration de graphes présentés trouvent des applications dans des domaines variés tels que l'informatique, les réseaux de télécommunications, et la biologie. La capacité à modéliser et résoudre des problèmes complexes à l'aide de ces algorithmes offre des perspectives intéressantes pour résoudre des défis du monde réel.

En examinant les différentes approches, de l'algorithme glouton à la coloration par backtracking, nous avons pu apprécier la diversité des méthodes pour aborder le problème de la coloration de graphes. Chaque algorithme a ses avantages et inconvénients, et le choix dépend souvent du contexte spécifique et des contraintes du problème.

En outre, notre projet propose des outils flexibles pour créer, modifier et visualiser des graphes, ainsi que des fonctions pour résoudre des sudokus. Ces fonctionnalités offrent une expérience complète pour explorer et expérimenter avec les concepts abordés.

En résumé, notre projet offre une plongée approfondie dans l'algorithmique des graphes, illustrant la synergie entre la culture mathématique et les applications pratiques. Nous espérons que ce rapport et le code associé serviront de ressources utiles pour ceux qui souhaitent approfondir leur compréhension de ce domaine passionnant.