

RAPPORT PROJET

MODÉLISATION-GRAPHES-ALGORITHMES

Mohamed Haady TIEMTORE
Rabah TOUBAL

Projet : calcul d'un couplage dans un graphe biparti

Question 1 :

Étant donné la définition du voisinage dans ce problème (les cases en diagonale ne sont pas prises en compte), on modélise chaque case par un sommet, coloré en blanc ou noir comme dans un damier. En construisant le graphe associé à l'échiquier mutilé, on relie deux sommets si leurs cases sont adjacentes orthogonalement. Or deux cases adjacentes ont toujours des couleurs opposées : un sommet blanc ne peut donc être adjacent qu'à un sommet noir, et réciproquement. Ainsi, aucune arête ne relie deux sommets d'une même couleur.

D'après la définition d'un graphe biparti, ce graphe est bien biparti.

Question 2 :

Dans l'échiquier (sous la forme d'un graphe $G = (V, E)$), on peut remarquer que

=> 1. Chaque case a au plus 4 voisins donc :

$$- \deg(v) \leq 4$$

=> 2. La somme des degrés de ce graphe (échiquier mutilé) :

$$- \text{Somme des } \deg(v) \leq 4n \text{ (avec } n \text{ le nombre de sommets)}$$

=> 3. On sait que Somme des $\deg(v)$ (pour tout v élément de V) = $2|E|$.

On en conclut que:

$$\Rightarrow 2|E| \leq 4n$$

$$\Rightarrow |E| \leq 2n.$$

Question 3 :

On peut voir un domino comme une arête reliant deux sommets, un noir et un blanc. L'objectif étant de pavier toutes les cases de l'échiquier (modélisé comme un graphe biparti, comme démontré précédemment), cela revient à couvrir deux cases voisines (deux sommets) avec un domino.

Parvenir à couvrir tous les sommets avec les dominos revient donc à chercher un couplage parfait pour ce graphe.

Question 4 :

Pour réaliser la fonction **construire_GM(G, M)**, nous codons d'abord une **fonction utilitaire**, **bipartition(G)**, qui à partir d'un graphe biparti G, nous retourne les 2 partitions (N et B) de ce graphe. Nous utilisons dans **bipartition(G)** un parcours en largeur pour colorier les sommets. Puis vient la fonction **construire_GM(G, M)**.

Question 5 :

Dans la fonction **construire_niveaux**, on manipule des sommets spécifiques aux différentes partitions N et B du graphe GM. On utilise leur **sommets libres**, c'est-à-dire, **les sommets libres de la partition N et les sommets libres de la partition B, du graphe GM**.

Ainsi, nous avons codé une fonction utilitaire : **sommets_libres(M, N, B)**, qui prend en paramètre un **couplage M** et les **deux (2) partitions** du graphe concerné **N et B** (obtenus avec la fonction bipartition), et nous retourne les **sommets libres de chacune des deux partitions**.

Notre fonction **construire_niveaux**, prend donc en paramètre, en plus de graphe GM, les ensembles **libres_N**, et **libres_B**, qui sont les sommets libres des deux partitions du graphe GM. Ces deux ensembles définissent le point de départ et le point d'arrivée cibles du parcours en largeur BFS pour déterminer k.

Nous avons fait le choix de **retourner en plus du graphe à niveau H**, un dictionnaire **niveau**, qui, **à chaque sommet, lui associe son niveau dans H**. Ce dictionnaire **niveau**, nous servira plus tard lors du calcul des chemins augmentants.

Question 6 :

Notre fonction **renverser(H)**, calcule l'inverse H_T du graphe H, en **inversant le sens de tous les arcs**. Elle est utilisée pour remonter les **chemins augmentants** du niveau k (**libres_B**) au niveau 0 (**libres_N**), avec un parcours en profondeur.

Question 7 :

Concernant cette question, nous avons fait le choix de séparer la structure de notre code :

- Une première fonction **dfs_augmentant**, avec en **paramètre** :
 - un **sommet u**.
 - le dictionnaire des niveaux {sommet: niveau}.
 - le **graphe inversé H_T** (niveau i+1 -> niveau i).
 - **chemin** : La liste des sommets visités jusqu'au sommet u (chemin partiel).
 - La liste **chemins** où les sets d'arêtes des chemins augmentants complets sont stockés.
 - **N** : L'ensemble des sommets (de type set), de la partition N (pour l'orientation N->B).
 - **bloques** : L'ensemble des sommets (de type set) déjà utilisés dans un chemin augmentant trouvé dans cette phase BFS (pour garantir le caractère disjoint des chemins).

- et retournant un **bool: True** si un **chemin augmentant** a été trouvé depuis **u**, **False** sinon.

Pour la fonction principale **chemins_augmentants**, nous avons en paramètre :

Le graphe inversé H_T, niveau : Le dictionnaire des niveaux, **libres_B** : Les sommets libres dans la partition B (points de départ du DFS), **N** : L'ensemble des sommets de la partition N.

Question 8 :

Pour cette fonction finale HopcroftKarp, nous combinons juste l'appel aux autres fonctions, en itérant simplement les étapes. D'abord, on **calcule les partitions N et B du graphe biparti en entrée**, puis on initialise le couplage M (de type set).

Ensuite dans une boucle while, on itère les étapes suivantes :

- **Construction de GM.**
- **Récupération des sommets libres des partitions N et B**, via la fonction **sommets_libres**.
- **Construction du graphe à niveau** avec GM, et les sommets libres des partitions N et B. On récupère ainsi le graphe à niveau H , et le dictionnaire de niveau.
- **On passe le graphe à niveau H à la fonction renverser** pour récupérer le **graphe inversé H_reversed**.
- On récupère dans une **liste P**, les chemins augmentants grâce à la fonction dédiée, en lui passant en paramètre, le **graphe inversé H_reversed**, le **dictionnaire des niveaux**, le **sommets libres de la partition B**, puis la **partition N** (partition d'arrivée).
- Si la liste P est vide, on s'arrête, c'est la fin de l'algorithme
- Sinon, on parcourt la liste P, contenant des set de chemins. Ainsi, on met à jour le couplage M en faisant la différence symétrique de chacun de ces sets avec le set M.

Comment exécuter le code ?

Pour tester nos échiquiers mis à disposition : Pour tester un fichier mutilé pavable, lancer le fichier **run_tests.py**. On retrouvera l'échiquier mutilé pavé avec les dominos, exporté dans le répertoire **results** , fichier **dominos_{nomFichier}.txt** .

Pour tester votre propre échiquier : vous pouvez placer votre fichier dans le répertoire **fichiers_test**, et en faire l'appel dans la fonction **run_all_tests** situé dans **tests/run_tests.py**.

Pour un échiquier que vous savez pavable, veuillez respecter ce pattern :

assert main("fichiers_test/nomFichier.txt") == True

Pour un échiquier que vous savez non pavable, veuillez respecter ce pattern :

assert main("fichiers_test/nomFichier.txt") == False

Choix d'implémentation, difficultés rencontrées

Pour notre **implémentation de l'algorithme de Hopcroft Karp**, nous avons choisi d'utiliser le langage python. Ce choix s'est imposé pour plusieurs raisons.

- Python est simple d'utilisation et bien adapté au travail en groupe, notamment pour la lecture et la compréhension du code.
- Disponibilité en Python de nombreuses méthodes utiles pour la manipulation, la gestion de notre structure de données (graphe).

En termes de **difficultés rencontrées**, le challenge était plutôt au niveau du code de la **fonction construire_niveaux** et de la **fonction utilitaire dfs_augmentant**.

Concernant la **fonction construire_niveaux**, la difficulté n'était pas tant le BFS à réaliser, mais plutôt la **détermination de k et le filtrage du graphe H**.

Solutions proposées :

- **Détermination de k** : Utilisation d'une variable **k = float('inf')**. Dès qu'un sommet libre v élément de libres B est atteint, la valeur de **niveau[v]** est capturée comme une borne supérieure pour k. Le BFS continue jusqu'à ce que la file soit vide ou que tous les sommets jusqu'au niveau k aient été explorés.
- **Filtrage de H** : Le graphe H est construit en deux étapes pour garantir la propriété "à niveaux" :
 - **Filtrage des sommets** : Seuls les sommets avec un niveau $\leq k$ sont conservés
 - **Filtrage des arcs** : garantit qu'un arc (u, v) n'est conservé dans H que s'il est **progressif** (c'est à dire qu'il respecte la progression stricte des niveaux).

Pour ce qui est de la fonction utilitaire **dfs_augmentant**, un **DFS standard**, remontant le graphe inversé H_T , aurait trouvé un chemin, mais **n'aurait pas empêché les DFS suivants de réutiliser les sommets ou les arêtes de ce chemin**. Cela aurait conduit à une augmentation non valide du couplage (sommets avec plusieurs arêtes couplées). Il fallait donc optimiser et adapter un DFS classique.

Solutions proposées : On va utiliser un système de **blocage permanent** des sommets au sein d'une phase de Hopcroft-Karp.

- Un ensemble **bloques** est initialisé dans **chemins_augmentants** et transmis à chaque appel récursif de **dfs_augmentant**.
- Avant d'explorer un voisin v dans H_T , on vérifie son état. **Si le sommet est déjà bloqué** (utilisé par un chemin trouvé précédemment), il est **ignoré**.
- Le succès du DFS (l'atteinte du niveau 0) déclenche un marquage : l'ensemble **full_path** (tous les sommets du chemin trouvé) est ajouté à l'ensemble **bloques**. Ce blocage est maintenu pour toutes les autres recherches DFS de la phase en cours. L'ensemble P est **M-disjoint**. Chaque sommet contribue au plus à un seul chemin dans la phase actuelle.