Modélisation, Graphes et Algorithmes Projet : calcul d'un couplage dans un graphe biparti

- Joachim Larrouy : joachim.larrouy@etu.univ-orleans.fr
- Nathan Rissot : nathan.rissot@etu.univ-orleans.fr

Utilisation

Programme a été développé avec la version 3.13 de python. Le code utilisant des annotations de type, une version 3.5 ou ultérieure est requis.

Le programme attend en arguement un chemin vers le fichier contenant la représentation de l'échiquier mutilé, et un nom pour le fichier de sortie en cas de succés de pavage.

```
py ./main.py <fichier_entrée.txt> <nom_fichier_sortie>
```

Partie Théorique

Question 1

Expliquez pourquoi le graphe représentant l'échiquier mutilé est un graphe biparti.

Soit le graphe $G = (V, E)$ representant l'échiquier tel qu'indiqué dans l'énoncé. Soit deux ensembles N et B tel que $V = N \biguplus B$, chaque sommet representant une case noire (resp. blanche) est placé dans l'ensemble N (resp. B).

Deux sommets appartiennent à une arête de E ssi les cases de l'échiquiers qu'ils representententent sont adjacentes, comme chaque case de l'échiquier (mutilé ou non) ne peut être adjacente qu'a des cases de l'autre couleur, les sommets du graphe ne peuvent eux aussi n'êtres adjacent qu'a des sommets représentant des cases de l'autre couleur.

(Bonus: le graphe G est 2-colorable, ce qui est évident sachant qu'il est bipartite, une bipartition étant équivalent à une 2-coloration.)

Donc:

$$\nexists (v_1, v_2) \in E \text{ tq } v_1, v_2 \in B \text{ et } \nexists (v_1, v_2) \in E \text{ tq } v_1, v_2 \in N$$

la séparation de l'ensembles V entre N et B selon la couleur de la case représentée est donc bien une bipartion du graphe.

Question 2

Montrez que ce graphe contient au plus 2n arêtes, où n est le nombre de cases de l'échiquier mutilé.

On note $n_{i,j}$ le sommet du graphe représentant la case de l'échiquier aux coordonnées i, j et soit m le nombre d'arêtes de G ($m = |E|$).

Nous cherchons a reconstruire le graphe d'un échiquier mutilé en ajoutant un à un les sommets représentant une case à notre graphe. Pour cela on choisis de les ajouters dans l'ordre suivant :

Parmis les cases qui ne sont pas encore représenté possédant le plus petit indice i, nous choisissons celle possdédant le plus petit indice j. (ie. la case la plus en haut a gauche n'ayant pas encore été ajoutée.)

Cas de base : Prenons un échiquier contenant une seule case, ce graphe respecte bien la propriétée $m = 0 \leq 2n = 2$

Récurrence : Prenons un échiquier mutilé de taille $N - 1$, auquel on veut rajouter une case $n_{i',j'}$. Par hypothèse de récurrence, il possède la propriété : $(m - 2) \leq 2(n - 1)$.

Rajoutons cette case. De par notre méthode de construction : cette nouvelle case peut avoir pour voisins $n_{i'-1,j'}$ ou $n_{i',j'-1}$. On ajoute donc au maximum 2 arrêtes si ces deux voisins existent. ce qui nous donne donc $(m - 2) + 2 \leq 2(n - 1 + 1)$. On a bien : $m \leq 2n$.

note : S'il possède moins de deux voisins, la propriété reste vrai : $a + b \leq c + 2$ est vrai si : $a \leq c$ et $b \leq 2$.

Question 3

Expliquez en quoi un couplage parfait, dans le graphe représentant l'échiquier mutilé, est utile pour le paver des dominos.

Chaque arête de E represente une paire de cases adjacentes de l'échiquier. Un couplage est un sous ensemble de E, c'est a dire une collection de paires de cases adjacentes, ne se chevauchant pas les unes les autres.

Un couplage parfait est donc une collection de paire de cases adjacentes telles que toutes les cases soient incluses, sans qu'il y-ait de de chevauchement.

Une collection de paire de cases adjacentes telles que toute les cases soient couvertes sans chevauchement est un pavage avec des dominos.

Implémentation

Nous avons choisi de representer les sommets sous forme de chaine de caractère décrivant leur position de l'échiquier `"<couleur:B|N>-<x>:<y>"`, les arêtes (et arcs) comme des tuple(sommet, sommet). Les Graphes sont representé par une classe `Graph` ayant comme attribut les ensembles B, N de sommets, et l'ensemble E des arêtes

> Remarque: l'ensemble des sommet V est aussi conservé et est calculé comme $N \cup B$.

Nous avons choisi d'utiliser des `set` pour leur transparence avec la formalisation et la simplicité d'écriture des opérations.

Question 5

Écrire une fonction `construire_niveaux` *qui attend en paramètre un graphe G_M et qui construit le graphe des niveaux ainsi que la valeur k.*

- **Entrée** : le graphe G_M.
- **Sortie** : H le graphe des niveaux, k le nombre de niveau de H.

```
fonction construire_niveaux(GM, k) {
    soit freeN (resp. freeB) l'ensemble des sommets noirs (resp.
blancs) libres, c-a-d N (resp B) privé des sommets captifs, donc
appartenant à des arcs retourné B → N au lieu de N → B:
    freeN = N \ {y | (x,y) ∈ E et y ∈ N}
    freeB = B \ {x | (x,y) ∈ E et x ∈ B}

    itération actuelle ← freeN
    listeNiveaux[0] ← freeN

    si itération actuelle est vide, alors {
        retourner le graphe vide et 0
    }

    k ← 0
    tant que l'on a pas atteint un sommet de freeB {
        pour chaque sommet n de l'itération actuelle {
            pour chaque sommet voisin de n {
                ajouter le sommet à l'itération suivante
                ajouter le sommet dans H.N ou H.B selon sa couleur
                ajouter l'arc entre n et son voisin dans H.E
            }
        }
        k++
        listeNiveaux[k] ← itération suivante
        si on a trouvé un sommet de freeB dans l'itération suivante,
alors {
            on retourne le graphe H(HN, HB, HE, listeNiveaux) et k
        } sinon {
            si itération suivante est vide, alors {
                cas d'échec, le graphe n'est pas pavable,
                retourner le graphe H(HN, HB, HE, listeNiveaux) et k
            }
            itération actuelle ← itération suivante
            itération suivante ← ∅
        }
    }
}
```

Question 7 - chemins augmentants

Écrire une fonction `chemins_augmentants` *qui se charge de calculer l'ensemble P de chemins utilisés par l'algorithme de* `HopcroftKarp`

- **Entrée** : H^T le graphe transposé de H.
- **Sortie** : une liste de chemins augmentants.

```
fonction chemins_augmentants(HT) {
    ajouter les sommets <START> avant les sommet du niveau k et <END>
aprés les sommets du niveau 0
    ajouter les arcs entre <START> et les sommets du niveau k et entre
<END> et les sommets du niveau k
    sommet autorisé ← HT.V

    chemins trouvés ← ∅

    tant qu'il existe des chemins trouvable {
        trouver un chemin de <START> à <END> avec la recherche en
profondeur en ne passant que par des sommets autorisés

        sommet autorisé ← sommet autorisé \ sommets du chemin

        chemins trouvés ← chemins trouvés U chemin
    }
    retourner les chemins trouvés

}
```