Option Informatique

Devoir Surveillé *n*°3

Durée : 4h. Le 31/01/2024.

LES CALCULATRICES SONT INTERDITES

Exercice 1 : Mots et langages

Dans cet exercice, on s'intéresse à des transformations sur des langages définis sur un alphabet à deux lettres $\Sigma = \{a, b\}$. On identifiera systématiquement une expression régulière avec le langage qu'elle dénote.

Soient L et K deux langages sur Σ. On définit le langage noté $L \triangleright K$ par :

$$L \triangleright K = \Big\{ uvw \ \mid \ (u,v,w) \in \Sigma^3,\ uw \in K,\ v \in L \Big\}.$$

1. Donner une description simple du langage $a^* \triangleright b^*$. Est-il rationnel ?

2. Donner une description simple du langage $(ba)^* \triangleright (ab)^*$. Est-il rationnel ?

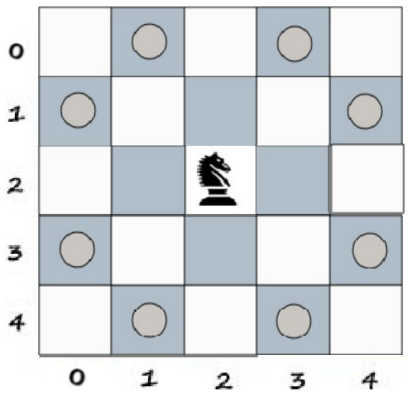
3. Donner une description simple du langage $a^* \triangleright \Big\{ a^p b a^q \ \mid \ (p,q) \in \mathbb{N}^2,\ p \leqslant q \Big\}$. Est-il rationnel ?

Exercice 2 : Chemin eulérien d'un cavalier sur l'échiquier

Dans cet exercice on considère un échiquier $n \times n$ qui sera codé par une matrice $n \times n$, avec $n \geqslant 3$. On s'intéresse au problème des chemins eulériens d'un cavalier sur un tel échiquier, qui sera explicité ci-dessous.

Pour comprendre ce problème il n'y a pas besoin de savoir jouer aux échecs. On doit juste connaître le fonctionnement du Cavalier qui a un déplacement particulier en forme de L : ce L peut être un L renversé verticalement ou un L renversé horizontalement voire un L renversé verticalement et horizontalement. Illustrons simplement les différentes règles du déplacement du Cavalier sur l'échiquier. Si ce Cavalier se trouve en ligne i et en colonne j, il y aura au maximum 8 destinations possibles qui seront :

- ligne i+2, colonne j+1
- ligne i+1, colonne j+2
- ligne i–1, colonne j+2
- ligne i–2, colonne j+1
- ligne i–2, colonne j–1
- ligne i–1, colonne j–2
- ligne i+1, colonne j–2
- ligne i+2, colonne j–1



sachant que toutes les cases précédentes qui sortiraient de l'échiquier sont à exclure.

On appelle chemin eulérien du cavalier une liste de cases de l'échiquier telle que :

- la première case peut être n'importe quelle case de l'échiquier ;
- chaque case est obtenue à partir de la précédente par un déplacement du cavalier ;
- chaque case de l'échiquier apparaît une et une seule fois dans la liste de cases.

4. Justifier que, pour $n = 3$, il n'existe aucun chemin eulérien du cavalier.

Cependant, il existe en général des solutions (beaucoup). Voici une solution possible pour un échiquier 5×5 :

	0	1	2	3	4
0	7	12	17	22	5
1	18	23	6	11	16
2	13	8	25	4	21
3	24	19	2	15	10
4	1	14	9	20	3

Dans la suite, on définit le type `echiquier` comme suit :

```
type echiquier = {taille : int; contenu : int option array array};;
```

Le champ `taille` correspond au format de l'échiquier : une taille n indiquera que l'échiquier considéré est de format $n \times n$.
Le champ `contenu` correspond à la matrice qui code l'échiquier : les cases non parcourues par le cavalier ont la valeur `None`, les cases déjà parcourues ont la valeur `Some k` avec `k` l'instant entre 1 et n^2 auquel le cavalier s'est rendu sur la case.

On souhaite déterminer un chemin eulérien du cavalier, s'il en existe, en procédant par backtracking. On commence par écrires quelques fonctions auxiliaires.

5. Écrire une fonction OCaml `cases_initiales : int → (int*int) list` telle que `cases_initiales n` donne la liste de tous les couples (i, j) de positions d'un échiquier $n \times n$, c'est-à-dire la liste de tous les couples (i, j) avec $0 \leqslant i, j \leqslant n - 1$. L'ordre importe peu.

6. Écrire une fonction OCaml `cases_accessibles : int*int → echiquier → (int*int) list` telle que `cases_accessibles (i,j) e` donne la liste des cases accessibles sur l'échiquier `e` à partir de la case `(i,j)`.

   Attention : on prendra bien garde décarter toutes les cases qui sortiraient de l'échiquier, mais aussi toutes les cases par lesquelles est déjà passé le cavalier, c'est-à-dire à qui ne sont pas affectées la valeur `None`.

On peut alors déterminer un chemin eulérien du cavalier comme suit :

— Au départ, l'échiquier est Vide (toutes les cases de son contenu sont à `None`) et la liste des cases accessibles est la liste des cases initiales.

— Étant donnée un entier k et la liste des cases accessibles à l'instant k, on a plusieurs cas :

- Si $k = n^2 + 1$, on a obtenu un chemin eulérien !
- Si $k \leqslant n^2$ mais que la liste est vide, alors il n'existe pas de chemin eulérien qui commence par les k déplacements précédents.
- Sinon, on prend la première case accessible, on lui affecte la valeur `Some k`, et on essaie d'obtenir récursivement un chemin eulérien. Si c'est possible, c'est gagné. Sinon, on remet la valeur `None` à la case en question et on retire la case fautive de la liste des cases accessibles.

7. Écrire une fonction OCaml `trouver_chemin_eulerien : int → int*int list` telle que `trouver_chemin_eulerien n` retourne la liste des positions successives du cavalier sur un échiquier $n \times n$ permettant d'obtenir un chemin eulérien. S'il n'existe pas de tel chemin, elle provoquera une exception.

Par exemple `trouver_chemin_eulerien 5` ne doit pas provoquer d'exception, puisqu'il existe au moins une solution, donnée en haut de la page précédente. Si c'est celle-ci qui est trouvée, la fonction devra renvoyer la liste `[(4,0);(3,2);(4,4);(2,3);(0,4);...;(3,0);(2;2)]`.

8. Modifier votre fonction pour obtenir une fonction `nombre_chemins_euleriens : int → int` qui donne le nombre de chemins eulériens.

On appelle cycle eulérien du cavalier un chemin eulérien du cavalier pour lequel la première case du chemin est accessible par le cavalier à partir de la dernière case du chemin.

9. Modifier la fonction `trouver_chemin_eulerien : int → int*int list` pour obtenir une fonction `trouver_cycle_eulerien : int → int*int list` qui retourne la liste des positions successives du cavalier sur un échiquier $n \times n$ permettant d'obtenir un cycle eulérien, ou provoque une erreur s'il en existe pas.

10. Démontrer que, si n est impair, il n'existe pas de cycle eulérien du cavalier.

    Indication : s'intéresser aux couleurs des cases par lesquelles passe le cavalier.

Mini-problème : Chemins arc-en ciel

On désignera par $[\![n]\!]$ l'ensemble des entiers de 0 à $n-1$: $[\![n]\!] = \{0, \dots, n-1\}$.

L'objectif est de déterminer s'il existe, entre deux villes données, un chemin passant par exactement k villes intermédiaires distinctes, dans un plan contenant au total n villes reliées par m routes. L'algorithme d'exploration naturel s'exécute en temps $O(n^k m)$. Le but est d'obtenir un algorithme qui s'exécute en un temps $O(f(k)n(n+m))$, qui croit polynomialement en la taille $(n+m)$ du problème quelle que soit la valeur de k demandée.

A. Création et manipulation de plans

Un plan p est déefini par : un ensemble de n villes numérotées de 0 à $n-1$ et un ensemble de m routes (toutes 'a double-sens) reliant chacune deux villes ensemble. On dira que deux villes $x, y \in [\![n]\!]$ sont voisines lorsqu'elles sont reliées par une route, ce que l'on notera par $x \sim y$. On appellera chemin de longueur k toute suite de villes $v_0, \dots, v_k$ telle que $v_0 \sim v_1 \sim \cdots \sim v_k$. On représentera les villes d'un plan par des ronds contenant leur numéro et les routes par des traits reliant les villes voisines (voir Figure 1).

Nous représenterons tout plan p par la donnée de son nombre de villes, de sont nombre de routes, et d'une représentation par listes d'adjacence du graphe non orienté correspondant.

```
type plan = {villes : int; mutable routes : int; graphe : int list array};;
```

La figure 1 donne un exemple de plan :

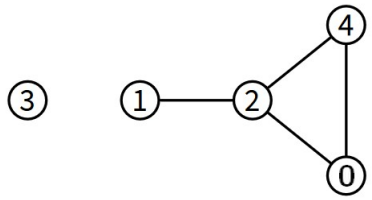
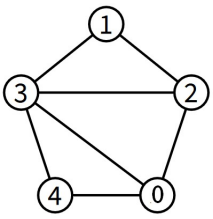


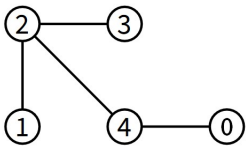
Figure 1 – Un plan à 5 villes et 4 routes

Une représentation possible est {`villes = 5; routes = 4; graphe = [|[2;4];[2];[0;4];[];[0;2]|]`}.

11. Représenter en OCaml les deux plans suivants :



Plan 1



Plan 2

12. Écrire une fonction `sont_voisines : plan → int → int → bool` qui indique si deux villes sont ou pas des villes voisines.

13. Écrire une fonction `ajoute_route : plan → int → int → unit` qui modifie le plan donné en argument pour ajouter une route entre les villes x et y si elle n'était pas déjà présente et ne fait rien sinon. (Attention, le graphe est non orienté.)

14. Écrire une fonction `toutes_les_routes : plan → (int*int) list` qui retourne une liste de toutes les routes, c'est-à-dire des couples (x, y) de villes voisines, chaque route devant apparaître une seule fois (ainsi, si le couple (x, y) apparaît, le couple (y, x) ne doit pas apparaitre). Par exemple, pour le plan de la figure 1, on pourra retourner `[(0,4); (0,5); (1,2); (2,4)]`.

15. Quelle est la complexité en temps de votre fonction `toutes_les_routes` ?

B. Recherche de chemins arc-en-ciel

Étant données deux villes distinctes $s, t \in [\![n]\!]$, nous recherchons un chemin de s à t passant par exactement k villes intermédiaires toutes distinctes. L'objectif de cette partie et de la suivante est de construire une fonction qui va détecter en temps linéaire en $n(n+m)$, l'existence d'un tel chemin avec une probabilité indépendante de la taille du plan $n+m$.

Le principe de l'algorithme est d'attribuer à chaque ville $i \in [\![n]\!] \setminus \{s, t\}$ une couleur aléatoire codée par un entier aléatoire uniforme `couleur.(i)` $\in \{1, \ldots, k\}$ stocké dans un tableau couleur de taille n. Les villes s et t reçoivent respectivement les couleurs spéciales 0 et $k+1$, *i.e.* `couleur.(s) = 0` et `couleur.(t) = k+1`.

L'objectif de cette partie est d'écrire une fonction qui détermine s'il existe un chemin de longueur $k+1$ allant de s à t dont la $j^{\text{ième}}$ ville a reçu la couleur j. Dans l'exemple de la figure 2, le chemin $6 \sim 7 \sim 8 \sim 3 \sim 4$ de longueur $4 = k+1$ qui relie $s = 6$ à $t = 4$ vérifie cette propriété pour $k = 3$.

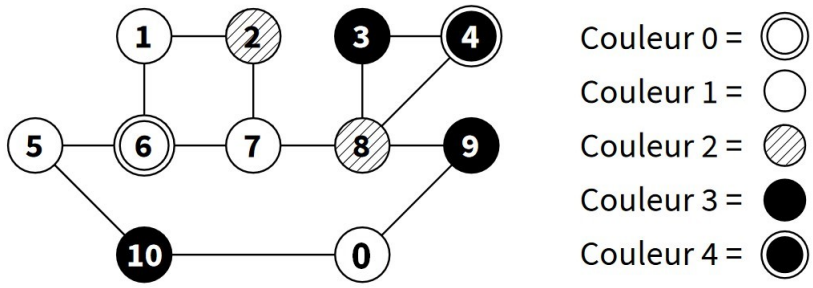


Figure 2 – Exemple de plan colorié pour $k = 3$, $s = 6$, $t = 4$.

On rappelle que la fonction `Random.int : int → int` est telle que `Random.int k` retourne un entier aléatoire uniforme entre 0 et $k-1$ (i.e. telle que tout entier entre 0 et $l-1$ peut être retourné avec la même probabilité $\frac{1}{k}$. Pour obtenir un entier aléatoire entre 1 et k, il convient donc d'ajouter 1.

16. Écrire une fonction `coloriage_aleatoire : int array → int → int → int → unit` telle que, si `couleur` est un tableau de taille n, `k` est un entier, et `s` et `t` $\in [\![n]\!]$ sont deux villes, alors `coloriage_aleatoire p couleur k s t` ne retourne rien, mais a pour effet de bord de remplir le tableau couleur avec : une couleur aléatoire uniforme dans $\{1,\dots,k\}$ choisie indépendamment pour chaque ville $i \in [\![n]\!] \setminus \{s,t\}$, et les couleurs 0 et $k+1$ pour s et t respectivement.

Nous cherchons maintenant à écrire une fonction qui calcule l'ensemble des villes de couleur c voisines d'un ensemble de villes donnée. Dans l'exemple de la figure 2, l'ensemble des villes de couleur 2 voisines des villes $\{1,5,7\}$ est $\{2,8\}$.

17. Écrire une fonction `voisines_de_couleur : plan → int array → int → int → int list` telle que `voisines_de_couleur p couleur c i` retourne la liste sans redondance des villes de couleur `c` voisines de la ville `i` pour le plan `p` coloriée par le tableau `couleur`.

18. Écrire une fonction `existe_chemin_arc_en_ciel : plan → int array → int → int → int → bool` telle que `existe_chemin_arc_en_ciel p couleur k s t` retourne `true` si et seulement si il existe dans le plan `p`, un chemin $s \sim v_1 \sim \ldots \sim v_k \sim t$ de longueur $k+1$, tel que `couleur.(`v_j`)`$= j$ pour tout $j \in \{1,\dots,k\}$. On procèdera par backtracking.
Quelle est la complexité de votre fonction dans le pire cas en fonction de k, n et m ?

C. Recherche de chemin passant par exactement k villes intermédiaires

Si les couleurs des villes sont choisies aléatoirement et uniformément dans $\{1,\dots,k\}$, la probabilité que j soit la couleur de la $j^{\text{ième}}$ ville d'une suite fixée de k villes vaut $\frac{1}{k}$ indépendamment pour tout j.

Ainsi, étant données deux villes distinctes s et $t \in [\![n]\!]$, s'il existe dans le plan p un chemin de s à t passant par exactement k villes intermédiaires toutes distinctes (et en admettant que le coloriage *couleur* est choisi aléatoirement par la fonction `coloriage_aleatoire`), la fonction `existe_chemin_arc_en_ciel` retourne `true` avec probabilité au moins k^{-k} ; et retourne toujours `false` sinon.

Ainsi, si un tel chemin existe, la probabilité qu'une parmi k^k exécutions indépendantes de `existe_chemin_arc_en_ciel` réponde `true` est supérieure ou égale à $1-\left(1-k^{-k}\right)^{k^k} \geqslant 1-\frac{1}{\mathrm{e}}$ qui est de l'ordre de $\frac{2}{3}$, donc très supérieure à 0.

(On admet toutes les propriétés mathématiques données ci-dessus.)

19. Écrire une fonction `existe_chemin_simple : plan → int → int → int → bool` telle que `existe_chemin_simple p k s t` retourne `true` avec probabilité au moins $1-1/\mathrm{e}$ s'il existe un chemin de s à t passant par exactement k villes intermédiaires toutes distinctes dans le plan p.

20. Quelle est la complexité de la fonction précédente en fonction de k, n et m dans le pire cas ?
Exprimez-la sous la forme $O(f(k)g(n,m))$ pour f et g bien choisies.

21. Modifier votre fonction pour renvoyer un tel chemin s'il est détecté avec succès.

22. Écrire une fonction sans argument permettant de faire 1000 tests sur le plan de la figure 2 (on a $k=3$, $s=6$, $t=4$, $k^k=27$) pour vérifier empiriquement la probabilité supérieure à $1-1/e$.

Fin.