TP 4 : Graphes

06 novembre

1 Couplage dans un graphe biparti

Nous allons ici implémenter l'algorithme de recherche d'un couplage de cardinal maximum dans un graphe biparti vu en classe.

On utilisera les notations suivantes :

- $\mathrm{G} = (\mathrm{X} \sqcup \mathrm{Y}, \mathrm{E})$ désigne un graphe biparti - on note $V = X \sqcup \mathrm{Y}$ son ensemble de sommets;
- $X = \{0, \dots, n_1 - 1\}, Y = \{n_1, \dots, n-1\}$ - toutes les arêtes relient donc un sommet d'indice strictement inférieur à n_1 à un sommet d'indice supérieur ou égal à n_1;
- on note $p = |E|$ le nombre d'arêtes du graphe;
- on note $A \oplus B = (A \cup B) \backslash (A \cap B)$ la différence symétrique de deux ensembles A et B - on rappelle que cette opération est associative et commutative.

On représente un graphe biparti par le type suivant :

```
type bipartite = {
n1 : int;
adj : int list array;
}
```

Un couplage est représenté par :

```
type matching = int option array
```

Pour un couplage m, on aura :

- `m.(i)= None` si le sommet `i` est libre;
- `m.(i)= Some j` et `m.(j)= Some i` si les sommets `i` et `j` sont appariés.

Un chemin $x_0, \dots, x_k$ sera simplement représenté par la liste $[x_0; \dots; x_k]$:

```
type path = int list
```

Remarque 1. *Mathématiquement, on verra parfois un chemin de longueur k comme une liste de k arêtes, parfois comme une liste de $k+1$ sommets. En revanche, il sera toujours représenté informatiquement par une liste de $k+1$ sommets.*

▷ **Question 1.** Écrire une fonction `print_matching : matching -> unit` qui prend en entrée un couplage et affiche les couples de sommets appariés : un couple par ligne. ◁

▷ **Question 2.** Écrire une fonction `correct_matching : bipartite -> matching -> bool` qui teste si un tableau correspond à un couplage correctement formé. ◁

▷ **Question 3.** Écrire une fonction `cardinal_matching : matching -> int` qui renvoie le nombre d'arêtes présentes dans le couplage passé en argument. ◁

▷ **Question 4.** Écrire une fonction `is_augmenting : matching -> path -> bool` prenant en entrée un couplage `m` et un chemin et indiquant si le chemin est augmentant. On supposera sans le vérifier que le chemin est élémentaire et que les entiers sont bien entre 0 et la taille du tableau `m`.
◁

Remarque 2. *Ces fonctions ne sont pas nécessaires pour la suite mais peut aider à identifier d'éventuelles erreurs dans le code.*

▷ **Question 5.** Écrire une fonction `delta : matching -> path -> unit` prenant en entrée un couplage `m` et un chemin `p` supposé augmentant pour `m` et effectuant l'opération $m \leftarrow m \oplus p$, où $\oplus$ représente la différence symétrique.
◁

Remarque 3. *Mathématiquement, on voit ici p et M comme des ensembles d'arêtes.*

▷ **Question 6.** Écrire une fonction `orient : bipartite -> matching -> int list array` prenant en entrée un graphe biparti `g` et un couplage `m` et renvoyant un graphe orienté G_m (sous forme d'un int list array) tel que :

- les sommets de G_m sont les mêmes que ceux de `g`;
- G_m contient exactement un arc orienté pour chaque arête $\{x, y\}$ (avec $x \in X$ et $y \in Y$) de `g` :
  - si $\{x, y\} \in m$, l'arc de G_m est (y, x);
  - si $\{x, y\} \notin m$, l'arc de G_m est (x, y).

Cette orientation est celle présentée dans le cours - on ne rajoute en revanche pas de sommet source ni de sommet puit.
◁

On définit l'exception suivante

```
exception Found of int
```

▷ **Question 7.** Écrire une fonction
`explore : bipartite -> matching -> int array-> int -> int-> unit` telle que
`explore g m peres p x` fait un parcours en profondeur récursif dans le graphe orienté `g` depuis le sommet `x` $\in X$ de père (dans l'arborescence de parcours) `p`, l'arborescence du parcours est stockée dans le tableau `peres` et le parcours s'arrête dès lors que l'on accède à un sommet de $y \in Y$ libre et lève l'exception `Found y` le cas échéant. Si aucun tel sommet n'est accessible depuis `x` alors la fonction ne fait rien. ◁

▷ **Question 8.** Écrire une fonction `reconstruct_path : int -> int array-> int list` telle que `reconstruct_path x peres` reconstruit le chemin entre `x` et la racine dans l'arborescence obtenue par un parcours en profondeur stockée dans le tableau `peres` passé en argument.
◁

▷ **Question 9.** Écrire une fonction `chemin_augmentant : bipartite->matching-> int list option` prenant en entrée un graphe biparti `g` et un couplage `m` et renvoyant :

- `None` si `m` n'admet pas de chemin augmentant;
- `Some p`, avec `p` un chemin augmentant, s'il en existe un.

◁

▷ **Question 10.** Écrire une fonction `get_maximum_matching : bipartite -> matching` prenant en entrée un graphe biparti `g` et renvoyant un couplage `m` de `g` de cardinalité maximale.
◁

c'est du cours

▷ **Question 11.** Déterminer la complexité de la fonction précédente. ◁

▷ **Question 12.** Une série de fichiers nommés `graph_n1_n2_c.txt` sont fournis : chacun décrit un graphe biparti $G = (X \sqcup Y, E)$ avec $|X| = n_1, |Y| = n_2$ et c le cardinal maximal d'un couplage de `g`. Le format de fichier est très simple, mais il est inutile de s'y intéresser : une fonction `read_bipartite` est fournie. Elle prend en entrée un nom de fichier (ou plutôt un chemin d'accès) et renvoie un bipartite.

Écrire un programme, que l'on compilera et exécutera en ligne de commande, qui accepte en argument un nom de fichier et affiche le cardinal du couplage renvoyé par `get_maximum_matching` sur le graphe correspondant. Vérifier que vous obtenez bien les valeurs attendues. On rappelle que pour lire un argument de l'exécutable on utilise `Sys.argv.(1)`. ◁

2 Labyrinthe et union find

Consignes : Le sujet fournit un code compagnon `laby.c` contenant certaines des fonctions décrites dans l'énoncé. Il est à compléter avec les fonctions demandées. On utilisera la commande de compilation:

```
gcc -g -Wall -fsanitize=address -o laby laby.c
```

On implémente une structure union-find naïve pour gérer une partition de $[\![0, n-1]\!]$ à l'aide d'un tableau de taille n contenant en case i la classe de l'élément i, définie comme étant le plus petit numéro qui est dans la même classe que i. Par exemple le tableau $[0, 0, 2, 0]$ représente la partition $\{\{0, 1, 3\}, \{2\}\}$ de $[\![0, 3]\!]$.

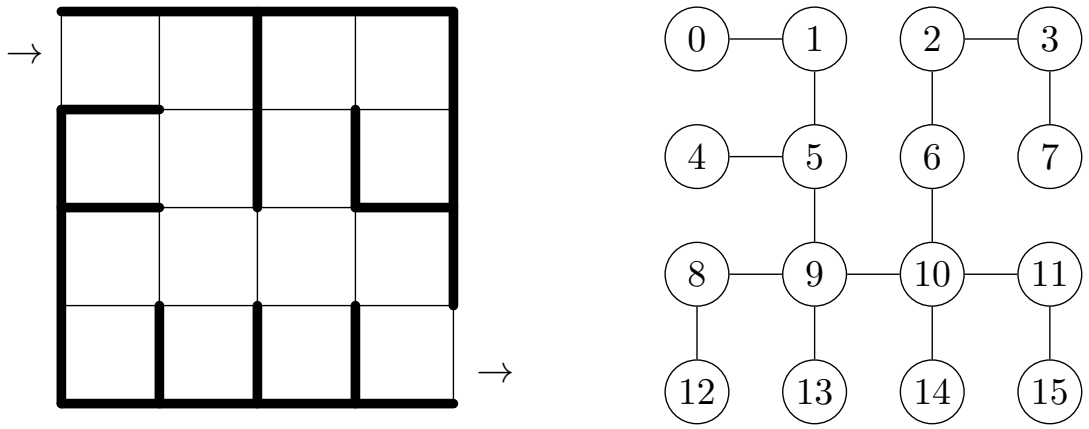
Le code compagnon fournit la structure suivante permettant de manipuler une telle structure union-find :

```
struct uf {
int taille; //nombre d'éléments dans la partition
int* classes; //tableau des classes à taille cases
};

typedef struct uf uf;
```

1. Ecrire une fonction `uf* initialiser_partition(int n)` créant une partition de $[\![0, n-1]\!]$ dans laquelle chaque élément est seul dans sa classe puis une fonction `void liberer_partition(uf* p)` qui libère toute la mémoire que cette fonction d'initialisation a alloué sur le tas pour un objet de type `uf*`.
2. Ecrire une fonction `int trouver_classe(uf* partition, int elem)` renvoyant la classe d'un élément.
3. Ecrire une fonction `void fusionner_classes(uf* partition, int i, int j)` qui modifie partition de sorte à fusionner les classes des éléments `i` et `j`. On la testera sur un exemple pertinent.
4. Quelles sont les complexités des fonctions d'initialisation, de recherche et d'union dans ce contexte?

On cherche à présent à construire des labyrinthes sur des grilles carrées dans lesquels il est garanti qu'il existe un unique chemin simple entre le carré en haut à gauche et le carré en bas à droite (on ne se déplace pas en diagonale). Voici un exemple d'un labyrinthe de côté 4 (les traits en gras représentent les murs) et la représentation qui en sera faite par un graphe et explicitée ci-dessous :



Ainsi, un labyrinthe de côté n est encodé par un graphe non orienté à n^2 sommets correspondant à une numérotation de haut en bas et de gauche à droite des cases de la grille. On peut passer de la case u à la case v dans le labyrinthe si et seulement si il y a une arête entre u et v dans le graphe correspondant. Les murs externes ne sont pas encodés : il y en a par défaut partout sauf en haut à gauche et en bas à droite.

La structure utilisée pour représenter le graphe encodant un labyrinthe est donnée dans le code compagnon :

```
struct graphe {
int cote; //taille du côté du labyrinthe
int nb_sommets; //nombre de sommets correspondant = cote*cote
bool** aretes; //matrice d'adjacence de taille nb_sommets*nb_sommets
};

typedef struct graphe graphe;
```

Le code fournit également une fonction de création d'un (pointeur sur un) labyrinthe (de type `graphe*`) ne contenant aucune arête (donc avec des murs partout) et de libération d'un tel objet.

5. Ecrire une fonction `void coordonnees(int s, int n, int* i, int* j)` stockant les coordonnées (i, j) du sommet `s` dans `i` et `j` respectivement en sachant que `s` est le numéro d'une case dans une grille de côté `n`. Par exemple, les coordonnées du sommet 13 dans une grille de côté 4 sont $(3, 1)$.

La fonction `sont_cote_a_cote` fournie par le code compagnon renvoie un booléen indiquant si deux sommets sont côte à côte dans une grille donnée. Par exemple, dans une grille de côté 4,8 et 4 sont côte à côte mais pas 3 et 4 .

6. Déterminer la complexité temporelle pire cas de `sont_cote_a_cote`.

On propose à présent de créer un labyrinthe de côté n aléatoirement à l'aide de la procédure suivante:

- Initialiser un labyrinthe `L` de côté n (donc à n^2 sommets) avec des murs partout
- Initialiser une partition P des sommets
- Pour chacune des n^4 arêtes possibles (u, v), prises dans un ordre aléatoire, si u et v sont côte à côte et que u et v ne sont pas dans la même classe d'après la partition P, casser le mur entre u et v dans `L`.

- Renvoyer `L`.

7. Compléter la fonction `graphe* creer_laby(int n)` dans le code compagnon en suivant cette stratégie. Le code contient déjà une implémentation d'une fonction mélanger qui permet le tirage aléatoire d'arêtes.
8. Créer un labyrinthe de côté 3. Le dessiner. On pourra utiliser la fonction `afficher_matrice`.
9. Quelle est la complexité de `creer_laby` ?
10. Donner une structure efficace pour la structure union-find, est ce que son utilisation permet d'améliorer la complexité de `creer_laby`.

3 Un exercice d'oral CCINP

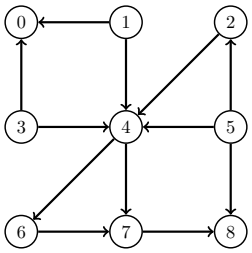
Exercice 5. L'exercice suivant est à traiter dans le langage C.

Dans cet exercice, tous les graphes seront orientés. On représente un graphe orienté $G = (S, A)$, avec $S = \{0, \dots, n-1\}$, en C par la structure suivante :

```
struct graph_s {
  int n;
  int degre[100];
  int voisins[100][10];
};
```

L'entier `n` correspond au nombre de sommets $|S|$ du graphe. On suppose que $n \leq 100$. Pour $0 \leq s < n$, la case `degre[s]` contient le degré sortant $d^+(s)$, c'est-à-dire le nombre de successeurs, appelés ici *voisins*, de s. On suppose que ce degré est toujours inférieur à 10. Pour $0 \leq s < n$, la case `voisins[s]` est un tableau contenant, aux indices $0 \leq i < d^+(s)$, les voisins du sommet s. Il s'agit donc d'une représentation par listes d'adjacences où les listes sont représentées par des tableaux en C.

Un programme en C vous est fourni dans lequel le graphe suivant est représenté par la variable `g_exemple`.



Pour $s \in S$ on note $\mathcal{A}(s)$ l'ensemble des sommets accessibles à partir de s. Pour $s \in S$, le maximum des degrés d'un sommet accessible à partir de s est noté $d^*(s) = \max\{d^+(s') \mid s' \in \mathcal{A}(s)\}$. Par exemple, pour le graphe ci-dessus, $\mathcal{A}(2) = \{2, 4, 6, 7, 8\}$ et $d^*(2) = 2$ car $d^+(4) = 2$. Dans cet exercice, on cherche à calculer $d^*(s)$ pour chaque sommet $s \in S$.

On représente un sous-ensemble de sommets $S' \subseteq S$ par un tableau de booléens de taille n, contenant `true` à la case d'indice s' si $s' \in \mathcal{A}(s)$ et `false` sinon.

1. Écrire une fonction `int degre_max(graph* g, bool* partie)` qui calcule le degré maximal d'un sommet $s' \in S'$ dans un graphe $G = (S, A)$ pour une partie $S' \subseteq S$ représentée par S, c'est-à-dire qui calcule $\max\{d^+(s') \mid s' \in S'\}$.
2. Écrire une fonction `bool* accessibles(graph* g, int s)` qui prend en paramètre un graphe et un sommet s et qui renvoie un (pointeur sur un) tableau de booléens de taille n représentant $\mathcal{A}(s)$. Une fonction `nb_accessibles` qui utilise votre fonction et un test pour l'exemple ci-dessus vous sont donnés dans le fichier à compléter.
3. Écrire une fonction `int degre_etoile(graph* g, int s)` qui calcule $d^*(s)$ pour un graphe et un sommet passé en paramètre. Quelle est la complexité de votre approche ?
4. Linéariser le graphe donné en exemple ci-dessus, c'est-à-dire représenter ses sommets sur une même ligne dans l'ordre donné par un tri topologique, tous les arcs allant de gauche à droite.
5. Dans cette question, on suppose que le graphe $G = (S, A)$ est acyclique. Décrire un algorithme permettant de calculer tous les $d^*(s)$ pour $s \in S$ en $O(|S| + |A|)$.
6. On ne suppose plus le graphe acyclique. Décrire un algorithme permettant de calculer tous les $d^*(s)$ pour $s \in S$ en $O(|S| + |A|)$.