TP 10 : Couplage maximum

Le but de ce TP est de mettre en œuvre l'algorithme de recherche de couplage maximum dans un graphe biparti vu en cours, reposant sur le principe de la recherche de chemins augmentants.

Une structure de graphe permettant de représenter des graphes bipartis est fournie, ainsi qu'une fonction permettant de visualiser ces graphes sous forme d'image.

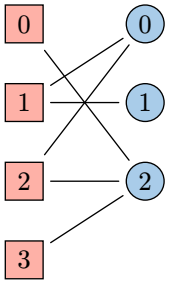
I. Prise en main

Un graphe non-orienté est *biparti* si on peut partitionner les sommets en deux ensembles disjoints S_1 et S_2 tels que les arêtes ne lient que des sommets d'un ensemble à un autre.

On va ainsi représenter les deux ensembles de sommets S_1 et S_2 par des intervalles d'entiers, respectivement $[\![0; n_1 - 1]\!]$ et $[\![0; n_2 - 1]\!]$, où n_1 et n_2 sont les tailles respectives de S_1 et S_2. Une arête de ce graphe sera toujours représentée par une paire (s_1, s_2), où s_1 est un sommet de S_1 et s_2 un sommet de S_2.

Les arêtes de ce graphes peuvent alors être représentées par une matrice d'adjacence de taille $n_1 \times n_2$, dans laquelle la case d'indice $(s_1, s_2) \in [\![0; n_1 - 1]\!] \times [\![0; n_2 - 1]\!]$ dénote la présence ou non d'une arête entre les sommets s_1 et s_2.

Le graphe ci-contre peut être représenté par les ensembles $S_1 = [\![0; 1; 2; 3]\!]$ et $S_2 = [\![0; 1; 2]\!]$, et l'ensemble des arêtes est $(0, 2), (1, 0), (1, 1), (2, 0), (2, 2)$ et $(3, 2)$. La matrice d'adjacence associée est ainsi :



$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

On fera attention au fait que, en fonction du contexte, un même indice peut faire référence à deux sommets différents (dans l'exemple ci-dessus, il y a deux sommets 0, 1 et 2).

On fournit les deux types ci-dessous :

```
type arete = int * int
type graphe_biparti = bool array array
```

Le premier type permet de représenter une arête comme un couple d'entiers. On fera attention au fait que le premier entier est toujours l'indice du sommet de S_1, et le second est toujours l'indice du sommet de S_2.

Le type `graphe_biparti` permet de représenter un graphe biparti comme expliqué précédemment. On fournit alors une fonction `ajouter_arete : graphe_biparti → arete → unit`, pour ajouter une arête dans le graphe. Par exemple, pour ajouter dans le graphe `g` une arête du sommet 3 de S_1 vers le sommet 0 de de S_2, on l'utilise de la façon suivante :

```
ajouter_arete g (3, 0)
```

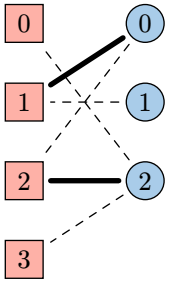
On fournit également une fonction `tester_arete : graphe_biparti → arete → bool` permettant de tester l'existence d'une arête dans le graphe, et qui s'utilise de façon similaire.

Afin de créer des graphes aléatoires pour tester les fonctions de ce TP, on donne une fonction `graphe_aleatoire : int → int → float → graphe_biparti` telle que `graphe_aleatoire n1 n2 p` créé un graphe à `n1` sommets dans S_1, `n2` sommets dans S_2, et tel que chaque arête dans le graphe a une probabilité `p` d'être présente. En pratique, `p` $= 0.2$ génère des graphes intéressants.

II. Couplage

Un couplage d'un graphe biparti est un sous-ensemble d'arêtes du graphe tel que chaque sommet du graphe n'est connecté qu'à au plus une arête de ce sous ensemble. Notre but ici est bien sûr de trouver un couplage de taille maximum.

Pour représenter un couplage d'un graphe biparti dont les ensembles de sommets sont S_1 et S_2, on va utiliser un tableau de taille S_2, qui va associer à chaque sommet $i_2 \in S_2$ l'éventuel sommet de S_1 auquel il est relié dans le couplage. On utilise ainsi le type suivant :




```
type couplage = int option array
```

Si un sommet `i2` $\in S_2$ est relié à un sommet `i1` $\in S_1$, on trouvera `Some i1` dans la case d'indice `i2` du couplage. S'il n'est pas présent dans le couplage, on trouvera `None`. Ainsi, le couplage représenté ci-contre par les arêtes épaisses est représenté par `[| Some 1; None; Some 2 |]`.

On remarquera qu'un sommet de S_2 est libre ssi la case associée dans le `couplage` est à `None`.

On donne une fonction `image` qui permet de visualiser un couplage dans un graphe.

Question 1. Écrire une fonction `taille_couplage : couplage → int` qui renvoie la taille d'un couplage.

Question 2. Le type utilisé permet d'assurer qu'un sommet de S_2 ne peut être lié qu'à au plus un sommet de S_1. Mais ce n'est pas la seule condition à vérifier pour qu'un couplage soit valide. Écrire une fonction `est_couplage : couplage → graphe_biparti → bool` qui vérifie qu'un couplage est valide pour un graphe donné.

II. A. Algorithme naïf

À l'aide des deux fonctions, précédentes, on peut écrire un algorithme naïf **particulièrement inefficace** pour trouver la taille maximum d'un couplage. On va énumérer tous les couplage possibles, et parmi ceux qui sont valides, on trouve celui de taille maximum. On énumère les couplages dans l'ordre lexicographique à partir du couplage `[| None; None; ...; None |]`. Le suivant est `[| Some 0; None; ...; None |]`, et le suivant est `[| Some 1; None; ...; None |]`. Quand tous les sommets ont été parcourus pour la première case, on passe à `[| None; Some 0; None; ...; None |]` et ensuite `[| Some 0; Some 0; None; ...; None |]`, etc.

Question 3. Écrire une fonction `suivant : couplage → int → bool` telle que `suivant couplage n1` modifie le `couplage` de taille `n2` en le couplage suivant selon l'ordre lexicographique. Le nombre de sommet de S_1 est donné par la variable `n1`. La fonction renvoie `false` si on a atteint le plus grand élément de l'ordre lexicographique, et `true` sinon.

Question 4. En déduire une fonction `taille_couplage_maximum_naif : graphe_biparti → int` qui renvoie la taille du plus grand couplage d'un graphe. Pour cela, on énumère tous les couplages avec la fonction `suivant` tant que celle-ci renvoie `true`, et on calcule la taille du plus grand d'entre eux qui est bien un couplage valide.

Question 5. Avant d'exécuter cette fonction, donner la complexité de la fonction `taille_couplage_maximum_naif`, et donner une taille de graphe raisonnable pour tester cette fonction.

Cet algorithme est évidemment très mauvais, mais on s'en servira pour s'assurer que l'algorithme efficace de la partie suivante donne le même résultat pour des graphes de petite taille.

II. B. Recherche de chemins augmentants

Passons maintenant à un algorithme efficace : celui vu en cours qui fonctionne par recherche de chemins augmentants. Pour représenter un chemin augmentant, on propose le type suivant :

```
type chemin = arete list
```

Si on reprend le couplage d'exemple précédent, le chemin $3 - - - 2 \mathrel{\text{━━}} 2 - - - 0 \mathrel{\text{━━}} 1 - - - 1$, qui est bien un chemin augmentant, est alors représenté par la liste d'arête : `[(3, 2) ; (2, 0) ; (1, 1)]`. On remarque que seules les arêtes hors du couplage sont représentées dans ce chemin.

Question 6. Écrire une fonction récursive `difference_symetrique : couplage → chemin → unit` qui applique l'opération de différence symétrique à un couplage pour un chemin donné. (Indication : il n'y a besoin que de parcourir les arêtes du chemin *hors* du couplage, et de les marquer comme *dans* le couplage).

Question 7. Compléter la fonction `chemin_augmentant : graphe_biparti → couplage → int → chemin` qui doit renvoyer un chemin augmentant dans le graphe pour le couplage donné à partir du sommet de S_1 donné, en appliquant l'algorithme vu en cours. Le squelette de cette fonction est déjà donné. On peut constater qu'on commence par déclarer un tableau de booléen, pour marquer chaque sommet de S_2 comme visité ou non. La fonction auxiliaire récursive `visiter : sommet_1 → chemin`, à compléter, doit effectuer un parcours en profondeur depuis un sommet x de S_1 à la recherche d'un sommet libre de S_2. Si aucun sommet libre de S_2 n'a été trouvé, on renvoie le chemin augmentant vide `[]`, sinon, on renvoie le chemin (nécessairement non-vide) permettant d'atteindre ce sommet depuis x.

Question 8. Écrire une fonction `couplage_maximum : graphe_biparti → couplage` qui applique l'algorithme des chemins augmentants. Il s'agit ici de partir du couplage vide, et pour chaque sommet de S_1, trouver un chemin augmentant, et l'appliquer au couplage avec l'opération de `difference_symetrique`. On renvoie le couplage final. On s'assurera qu'à chaque étape, y compris à la fin, le couplage est bien valide, avec l'instruction `assert (est_couplage couplage);`. On pourra également dessiner le graphe obtenu à chaque étape, avec la fonction `image`, en sauvegardant le graphe dans un fichier différent à chaque fois.