1 Découverte du type `option`

Le type `'a option` est un type déjà existant en OCaml. Il permet la possibilité de créer un objet qui vaut « Rien » ou « Quelque chose » et est une alternative toujours correctement typée à une valeur par défaut renvoyée lorsqu'on n'a pas trouvé un objet (par exemple, renvoyer -1 lorsqu'aucun indice d'un tableau ne vérifie une condition).

Le type est créé de la manière suivante (que vous n'avez pas besoin de recopier car, comme dit ci-dessus, le type existe déjà) :

```
type 'a option =
    | None
    | Some of 'a
```

Un filtrage sur un objet de type `'a option` se fait très simplement comme :

```
let f opt = match opt with
    | None -> ...
    | Some x -> ...
```

1. Écrire une fonction `premiere_occ (tab: 'a array) (x: 'a) : int option` qui prend en argument un tableau et un élément x et renvoie `Some i` où i est l'indice de la première occurrence de x dans le tableau, s'il en existe une, et `None` sinon.
2. Écrire une fonction `liste_inf (lst: 'a list) (x: 'a) (y: 'a) : 'a list option` qui prend en argument une liste et deux éléments x et y et renvoie :
   - `None` s'il existe un élément plus grand que y ;
   - `Some occ` où `occ` est la liste des éléments de `lst` plus petit que x sinon.

   On demande de ne faire qu'un seul parcours de la liste `lst` et de ne pas utiliser d'autre fonction.

2 Préliminaires

On trouvera sur le site `https://github.com/nathaniel-carre/MP-LLG` un fichier `utilitaire.ml` à télécharger (dans le dossier `TP/TP1-Couplages`). Pour ouvrir ce fichier depuis un autre fichier `.ml`, on peut écrire, en entête, la commande suivante :

```
#use "Chemin/du/fichier/utilitaire.ml";;
```

en remplaçant bien sûr le chemin du fichier de manière adéquate (attention, il faut penser à transformer les `\` des chemins Windows en `/`, le cas échéant).

3. Lire les descriptions du type de données et des fonctions utilitaires dans le fichier `utilitaire.ml`.

On représente un graphe biparti non orienté non pondéré par un objet de type `graphe` de telle sorte que si $G = (S = X \sqcup Y, A)$ est un graphe non orienté non pondéré représenté par un objet `g` de type `graphe`, alors :

- $n_X = |X|$ est égal à `g.nx`, $n_Y = |Y|$ est égal à `g.ny`, $X = [\![0, n_X - 1]\!]$ et $Y = [\![n_X, n_X + n_Y - 1]\!]$;
- pour $s \in S$, `g.adj.(s)` est une liste chaînée contenant les voisins de s.

Un couplage C dans un graphe $G = (S, A)$ est représenté par un tableau d'entiers `c` de taille $|S|$ tel que pour $s \in S$, `c.(s)` est égal à $t \in S$ si $\{s, t\} \in C$, et `c.(s)` est égal à -1 si s est un sommet libre pour C.

Le graphe `g1` créé dans le fichier `utilitaire.ml` est représenté figure 1. On y représente également un couplage C_1. On pourra utiliser le graphe et le couplage pour tester les fonctions demandées.

4. Créer un tableau correspondant au couplage C_1.
5. Écrire une fonction `cardinal_couplage (g: graphe) (c: int array) : int` qui calcule le cardinal d'un couplage C dans un graphe G.

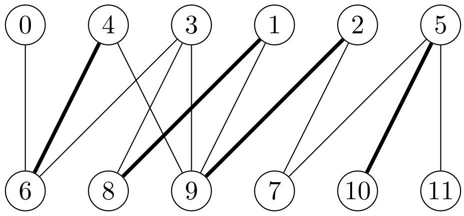

Figure 1 – Un graphe biparti G_1 et un couplage C_1 (en gras).

3 Couplage de cardinal maximum

On représente un chemin sous la forme d'une liste de ses sommets.

6. Écrire une fonction `graphe_augmentation (g: graphe) (c: int array) : int list array` qui prend qui prend en argument un graphe biparti $G = (X \sqcup Y, A)$, un couplage C de G et renvoie le graphe d'augmentation G_C, représenté par tableau de listes d'adjacence. On représentera le sommet source s par l'entier $|X| + |Y|$ et le sommet puits t par $|X| + |Y| + 1$.
7. Écrire une fonction `arborescence (gc: int list array) (s: int) : int array` qui prend en argument un graphe orienté $G_C = (S, A)$ sous forme de tableau de listes d'adjacence et un sommet $s \in S$ et renvoie l'arborescence d'un parcours en profondeur de G_C depuis s sous forme d'un tableau d'entiers `arbo` tels que :
   - `arbo.(s)` vaut -1 ;
   - `arbo.(u)` vaut le parent de $u \in S \setminus \{s\}$ dans l'arborescence du parcours depuis s, s'il existe un chemin de s à u ;
   - `arbo.(u)` vaut -2 s'il n'existe pas de chemin de s à u.
8. En déduire une fonction `chemin (gc: int list array) (s: int) (t: int) : int list option` qui prend en argument un graphe orienté $G_C = (S, A)$ sous forme de tableau de listes d'adjacence et deux sommets $s, t \in S$ et renvoie :
   - `None` s'il n'existe pas de chemin de s à t dans G_C ;
   - `Some lst` où `lst` est la liste des sommets intermédiaires dans un chemin de s à t dans G_C, sinon.

   Attention, on distinguera bien `None` (pas de chemin) et `Some []` (un chemin sans sommet intermédiaire, correspondant à l'arête (s, t)).
9. En déduire une fonction `chemin_augmentant (g: graphe) (c: int array) : int list option` qui prend en argument un graphe biparti $G = (X \sqcup Y, A)$ et un couplage C de G et renvoie `None` s'il n'existe pas de chemin augmentant pour C dans G, et `Some sigma` où σ est un chemin augmentant sinon.
10. Écrire une fonction `augmenter (c: int array) (sigma: int list) : unit` qui prend en argument un couplage C et un chemin σ supposé augmentant pour C et modifie C en $C \Delta \sigma$.

    Indication : la liste `sigma` sera supposée de longueur paire.
11. En déduire une fonction `couplage_maximum (g: graphe) : int array` qui calcule et renvoie un couplage de cardinal maximum de G.
12. Vérifier qu'un couplage maximum pour le graphe `g2` est de cardinal 73 et qu'un couplage maximum pour le graphe `g3` est de cardinal 9060.

4 Algorithme de Hopcroft-Karp

L'algorithme usuel de recherche de couplage maximum dans un graphe biparti effectue autant de parcours de graphe que le cardinal du couplage, ce qui résulte en une complexité en ‰$(|S||E|)$ dans le cas général. L'algorithme de Hopcroft-Karp améliore cette complexité en trouvant plusieurs chemins augmentants d'un coup pour augmenter le couplage en cours de construction. Il se résume de cette manière.

Entrée : Graphe $G = (X \sqcup Y, A)$ biparti non orienté
Début algorithme
  $C \leftarrow \varnothing$
  Tant que il existe un chemin augmentant pour C **Faire**
    Trouver $E = \{\sigma_1, \sigma_2, \dots, \sigma_k\}$ un ensemble maximal de plus courts chemins augmentants pour C, disjoints deux à deux.
    Augmenter C avec les chemins de E.
  Renvoyer C

L'ensemble E est maximal au sens où tout autre chemin augmentant aurait au moins un sommet en commun avec l'un des σ_i. La difficulté de l'algorithme consiste à déterminer un tel ensemble E en complexité linéaire en la taille du graphe. L'idée pour ce faire est la suivante :

- on effectue un parcours en **largeur** alternant depuis les sommets libres de X, pour déterminer les distances des sommets de G à un sommet libre de X dans des chemins alternants ;
- dans l'ordre croissant des distances précédentes, on effectue des parcours en **profondeur** depuis les sommets libres de Y, pour trouver des plus courts chemins alternants jusqu'aux sommets libres de X.

13. Écrire une fonction `bfs_alternant (g: graphe) (c: int array) : int array * int array` qui prend en argument un graphe biparti $G = (S = X \sqcup Y, A)$ et un couplage C et renvoie un couple `(ordre_bfs, dist)` qui sont deux tableaux d'entiers de taille $|S|$ tels que :
    - pour $s \in S$, `dist.(s)` contient la longueur minimale d'un chemin alternant d'un sommet libre de X à s. En particulier, si $x \in X$ est un sommet libre, alors `dist.(x)` doit valoir 0. Par convention, s'il n'existe pas de tel chemin, on posera `dist.(s) = -1` ;
    - `ordre_bfs` contient les sommets de S accessibles par un chemin alternant depuis un sommet libre de X, par ordre croissant de `dist`. S'il existe des sommets de S non accessibles par un tel chemin, on complètera le tableau `ordre_bfs` par des valeurs -1.

    Indication : on pourra utiliser le tableau `ordre_bfs` en guise de file, en gardant en mémoire l'indice du prochain élément à sortir de la file et l'indice de la prochaine case libre du tableau.
14. Écrire une fonction

    ```
    dfs_alternant (g: graphe) (c: int array) (dist: int array)
    (vus: bool array) (y: int) : int list option
    ```

    qui prend en argument un graphe biparti $G = (S = X \sqcup Y, A)$, un couplage C, un tableau `dist` tel que renvoyé par la fonction précédente, un tableau de booléens `vus` et un sommet $y \in Y$ et renvoie une option de **plus court** chemin alternant pour C commençant par y et terminant par un sommet libre de X. La fonction ne devra pas explorer les sommets déjà vus (dans le tableau `vus`), et marquer comme `vus` les sommets explorés. La fonction renverra `None` s'il n'existe pas de tel chemin et `Some sigma` s'il existe un tel chemin `sigma`.

    Indication : pour garantir qu'il s'agit d'un plus court chemin, on n'explorera que les voisins dont la distance vaut 1 de moins que y.
15. En déduire une fonction `hopcroft_karp (g: graphe)` qui calcule un couplage de cardinal maximum dans un graphe biparti selon l'algorithme de Hopcroft-Karp.
16. Vérifier la correction de l'algorithme sur les graphe `g1`, `g2` et `g3`.
17. Comparer les performances temporelles entre les fonctions `couplage_maximum` et `hopcroft_karp` sur les graphes `g2` et `g3`.
18. [À faire après le TP] Montrer qu'après chaque passage dans la boucle **Tant que** de l'algorithme de Hopcroft-Karp, la longueur minimale d'un chemin augmentant pour C augmente d'au moins 1.
19. [À faire après le TP] Soit C^* un couplage de cardinal maximum. Montrer qu'après $\sqrt{|S|}$ passages dans la boucle **Tant que**, $|C^*| - |C|$ vaut au plus $\sqrt{|S|}$.
20. [À faire après le TP] En déduire la complexité temporelle de l'algorithme de Hopcroft-Karp.