On trouvera sur le dépôt les documents à télécharger pour ce TP. On y trouve :

- un fichier `TP4.c` à compléter pendant le TP ;
- un fichier `utilitaire.c` (et son fichier d'entête `utilitaire.h`) contenant la définition de certains types de données et de fonctions à utiliser pendant le TP. Ces fichiers ne devront pas être modifiés ;
- un fichier `makefile` permettant d'effectuer la compilation du code source et l'exécution du programme. On pourra utiliser les commandes suivantes dans la console :
  - `make build` pour compiler et créer l'exécutable `TP4` ;
  - `make run` pour exécuter le code ;
  - `make all` pour faire les deux l'un après l'autre.

1 Préliminaires

1. Lire la description des types de données `liste` et `graphe` dans le fichier `utilitaire.h`.
2. Lire les descriptions des fonctions utilitaires dans le fichier `utilitaire.c`.
3. Écrire une fonction `int* creer_tab(int n, int val)` qui crée un tableau contenant n fois la valeur `val`.

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 :

- $n = |X|$ est égal à `G.n`, $p = |Y|$ est égal à `G.p`, $X = [\![0, n-1]\!]$ et $Y = [\![n, n+p-1]\!]$;
- pour $s \in S$, `G.adj[s]` est un pointeur vers 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 la fonction `main` 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.

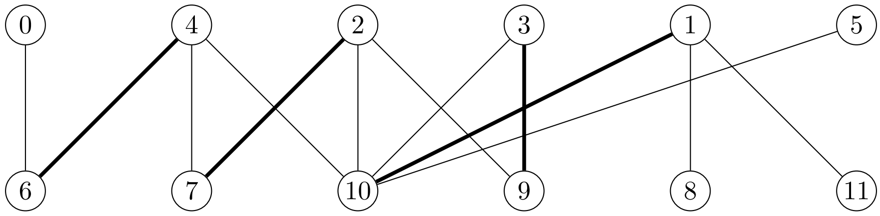

FIGURE 1 – Un graphe biparti G_1 et un couplage C_1 (en gras).

4. Avec l'affichage des listes d'adjacence dans la fonction `main`, vérifier que le graphe G_1 correspond bien au graphe représenté à la figure 1. Après vérification, on pourra commenter le code d'affichage.
5. Dans la fonction `main`, créer un tableau statique correspondant au couplage C_1.
6. Écrire une fonction `int cardinal_couplage(graphe G, int* C)` qui calcule le cardinal d'un couplage C dans un graphe G.

2 Couplage de cardinal maximum

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

7. Écrire une fonction `liste* chemin_alternant(graphe G, int* C, bool* vus, int x)` qui prend en argument un graphe biparti $G = (X \sqcup Y, A)$, un couplage C de G, un tableau de booléens `vus` et un sommet $x \in X$ et renvoie un chemin alternant pour C commençant par x et terminant par un sommet

libre de Y. La fonction ne devra pas explorer les sommets déjà vus (dans le tableau `vus`), et marquer comme `vus` les sommets explorés. La fonction renverra `NULL` s'il n'existe pas de tel chemin.

Indication : les arêtes de Y vers X d'un tel chemin sont directement déterminées par C.

8. Écrire une fonction `void augmenter(int* C, liste* sigma)` 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.
9. En déduire une fonction `int* couplage_maximum(graphe G)` qui calcule et renvoie un couplage de cardinal maximum de G.
10. Vérifier qu'un couplage maximum pour le graphe `G2` est de cardinal 79 et qu'un couplage maximum pour le graphe `G3` est de cardinal 9034.

3 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 $\mathcal{O}(|S||E|)$ 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.

11. Écrire une fonction `int* bfs_alternant(graphe G, int* C, int* dist)` qui prend en argument un graphe biparti $G = (S = X \sqcup Y, A)$, un couplage C et un tableau `dist` de taille $|S|$ et :
    - modifie le tableau `dist` pour que pour $s \in S$, `dist[s]` contienne 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` ;
    - renvoie un tableau `ordre_bfs` de taille $|S|$ qui contient les sommets de S par ordre croissant de `dist`. S'il existe des sommets de S non accessibles par un chemin alternant depuis un sommet libre de X, 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.
12. Écrire une fonction `liste* dfs_alternant(graphe G, int* C, int* dist, bool* vus, int y)` qui prend en argument un graphe biparti $G = (S = X \sqcup Y, A)$, un couplage C, un tableau `dist` tel que modifié par la fonction précédente, un tableau de booléens `vus` et un sommet $y \in Y$ et renvoie un **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 `NULL` s'il n'existe pas de tel chemin.

    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.

13. En déduire une fonction `int* hopcroft_karp(graphe G)` qui calcule un couplage de cardinal maximum dans un graphe biparti selon l'algorithme de Hopcroft-Karp.
14. Vérifier la correction de l'algorithme sur les graphe `G1` et `G2`.
15. Comparer les performances temporelles entre les fonctions `couplage_maximum` et `hopcroft_karp` sur les graphes `G2` et `G3`.
16. [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.
17. [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 $\sqrt{|S|}$.
18. [À faire après le TP] En déduire la complexité temporelle de l'algorithme de Hopcroft-Karp.

4 Couplage de cardinal maximum et de poids minimum

Adapté d'un sujet de J.-B. Bianquis.

Si on considère $G = (S = X \sqcup Y, A, f)$ un graphe non orienté pondéré biparti, on peut étendre la recherche de couplage maximum en cherchant un couplage de poids minimum parmi ceux de cardinal maximum, le poids $f(C)$ d'un couplage C étant la somme des poids de ses arêtes.

Si C est un couplage de G et σ est un chemin augmentant pour C, alors on définit le **coût** de σ par :

$$f_C(\sigma) = \sum_{a \in \sigma \setminus C} f(a) - \sum_{a \in \sigma \cap C} f(a)$$

On propose d'implémenter l'algorithme suivant :

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 σ un chemin augmentant pour C de coût minimal.
    $C \leftarrow C \Delta \sigma$.
  Renvoyer C

19. [À faire après le TP] Montrer que si σ est un chemin augmentant pour C, alors $f(C\Delta\sigma) = f(C) + f_C(\sigma)$.
20. [À faire après le TP] Montrer que l'algorithme précédent renvoie un couplage de poids minimum parmi les couplages de cardinal maximum.

La recherche d'un chemin augmentant de coût minimal se ramène à une recherche de plus court chemin dans un graphe orienté particulier. Si C est un couplage de G, on pose $G_C = (S', A', g)$ le graphe orienté pondéré défini par :

- $S' = S \cup \{s, t\}$;
- l'ensemble A' contient les arêtes suivantes :
  * les (s, x) pour $x \in X$ sommet libre pour C, avec un poids $g(s, x) = 0$;
  * les (y, t) pour $y \in Y$ sommet libre pour C, avec un poids $g(y, t) = 0$;
  * les (x, y) avec $x \in X$, $y \in Y$ et $\{x, y\} \in A \setminus C$, avec un poids $g(x, y) = f(x, y)$;
  * les (y, x) avec $x \in X$, $y \in Y$ et $\{x, y\} \in C$, avec un poids $g(y, x) = -f(x, y)$.

21. [À faire après le TP] Montrer que $(s, s_1, \ldots, s_k, t)$ est un chemin de poids minimal dans G_C si et seulement si $(s_1, \ldots, s_k)$ est un chemin augmentant pour C dans G de coût minimal.

On représente un graphe non orienté biparti pondéré comme la donnée d'un objet de type `graphe` et d'une matrice d'adjacence pondéré de type `double**` (les poids sont des flottants). Un graphe orienté pondéré sera représenté par une matrice de type `double**` et un entier correspondant à sa taille.

22. Écrire une fonction `double poids_couplage(graphe G, double** M, int* C)` qui détermine le poids d'un couplage C dans un graphe G.
23. Écrire une fonction `int** floyd_warshall(double** M, int n)` qui prend en argument une matrice et un entier correspondant à la description d'un graphe orienté pondéré $G = (S, A, f)$ et renvoie une matrice `pred` de taille $n \times n$ telle que pour $s, t \in S^2$, `pred[s][t]` contient le prédécesseur de t dans un plus court chemin de s à t s'il en existe un, et -1 sinon. Par convention, on posera `pred[s][s]` égal à -1. La fonction `floyd_warshall` pourra modifier la matrice `M` directement si nécessaire.
24. Écrire une fonction `liste* plus_court_chemin(double** M, int n, int s, int t)` qui prend en argument une matrice et un entier correspondant à la description d'un graphe orienté pondéré $G = (S, A, f)$ et deux sommets $s, t \in S^2$ et renvoie une liste contenant les sommets intermédiaires d'un plus court chemin de s à t (c'est-à-dire le chemin de s à t sans ses extrémités s et t). La fonction renverra `NULL` s'il n'existe pas de tel chemin.
25. Écrire une fonction `double** construire_GC(graphe G, double** M, int* C)` qui construit une matrice d'adjacence pondéré correspondant au graphe G_C. Le choix de la numérotation des sommets s et t ajoutés est laissé libre.
26. En déduire une fonction `int* couplage_maximum_poids_minimum(graphe G, double** M)` qui renvoie un couplage de poids minimum parmi ceux de cardinal maximum dans le graphe G.
27. Vérifier que le poids du couplage renvoyé pour G_1 et M_1 est 18,8 et que celui pour G_2 et M_2 est 2890,47.
28. Déterminer la complexité temporelle de la fonction précédente.