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 G non pondéré, orienté ou non, par un objet de type `graphe` de telle sorte que si $G = (S, A)$ est représenté par un objet `G` de type `graphe`, alors :
- $n = |S|$ est égal à `G.n` et $S = [\![0, n-1]\!]$;
- 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 $G = (S, A)$ non orienté 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.

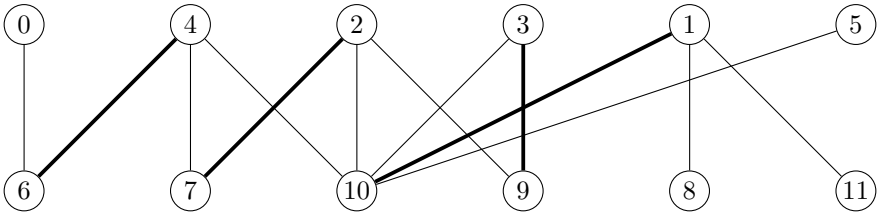

FIGURE 1 – Un graphe biparti G_1 et un couplage C_1 (en gras).

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.
7. Écrire une fonction `int* accessibles(graphe G, int s)` qui prend en argument un graphe $G = (S, A)$ et un sommet source s et renvoie un tableau de prédécesseurs `pred` de taille $n = |S|$ tel que pour $t \in S$, `pred[t]` est égal à :
   - -1 si $t = s$;
   - -2 si t n'est pas accessible par un chemin depuis s ;
   - un sommet $u \in S$ qui est le prédécesseur de t dans un chemin de s à t s'il existe un tel chemin.

   Indication : on pourra écrire une fonction récursive auxiliaire qui prend en argument le graphe G, le tableau `pred` en cours de construction, un sommet t et un sommet u qui est un prédécesseur de t.

PALIER 1

2 Couplage de cardinal maximum

On cherche à calculer un couplage de cardinal maximum dans un graphe biparti. Un graphe biparti $G = (X \sqcup Y, A)$ sera représenté par un objet de type `graphe` tel que les indices des sommets de X sont inférieurs aux indices des sommets de Y. Lorsqu'on travaillera sur un graphe biparti, on donnera en argument un entier `nX` correspondant au cardinal de X.

8. Écrire une fonction `int calcul_nX(graphe G)` qui prend en argument un graphe $G = (X \sqcup Y, A)$ supposé biparti et renvoie un entier correspondant à une valeur possible de $|X|$.
9. Écrire une fonction `graphe graphe_augmentation(graphe G, int nX, int* C)` qui prend en argument un graphe biparti $G = (S = X \sqcup Y, A)$, le cardinal $|X|$ et un couplage C et calcule le graphe d'augmentation correspondant. On supposera que le sommet source s rajouté sera d'indice $n = |S|$ et que le sommet puits t rajouté sera d'indice $n + 1$.

On représente un chemin par une liste chaînée de ses sommets.

10. En utilisant les fonctions `graphe_augmentation` et `accessibles`, et en prenant garde à la gestion de la mémoire, écrire une fonction `liste* chemin_augmentant(graphe G, int nX, int* C)` qui calcule un chemin augmentant pour C. La fonction renverra le pointeur `NULL` s'il n'existe pas de tel chemin.
11. 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.
12. En déduire une fonction `int* couplage_maximum(graphe G, int nX)` qui calcule et renvoie un couplage de cardinal maximum de G.
13. 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 4527.

Palier 2

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 $E = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ 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.

14. Écrire une fonction `int* bfs_alternant(graphe G, int nX, int* C, int* dist)` qui prend en argument un graphe biparti $G = (S = X \sqcup Y, A)$, la valeur $|X|$, 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.
15. 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.
16. En déduire une fonction `int* hopcroft_karp(graphe G, int nX)` qui calcule un couplage de cardinal maximum dans un graphe biparti selon l'algorithme de Hopcroft-Karp.
17. Vérifier la correction de l'algorithme sur les graphe `G1` et `G2`.
18. Comparer les performances temporelles entre les fonctions `couplage_maximum` et `hopcroft_karp` sur les graphes `G2` et `G3`.

Palier 3

19. 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.
20. Soit C^* un couplage de cardinal maximum. Montrer qu'après $\sqrt{|S|}$ passages dans la boucle **Tant que**, $|C^*| - |C|$ vaut au plus $\sqrt{|S|}$.
21. En déduire la complexité temporelle de l'algorithme de Hopcroft-Karp.