TP n°31 - Graphes - Parcours en largeur (BFS) - Plus court chemin

Dans ce TP, on implémente un algorithme de parcours en largeur d'un graphe (*Breadth-First Search* BFS an anglais) de manière itérative, puis, on montre l'utilisation de ce parcours pour l'algorithme de calcul de plus court chemin de Dijkstra. On propose ensuite une implémentation d'un autre algorithme de calcul de plus court chemin, l'algorithme de Floyd-Warshall et un petit devoir maison.

Pour réaliser ce TP, plusieurs bibliothèque C sont mises à votre disposition, à chaque fois sous la forme d'un fichier *header* `xxxx.h` représentant l'interface abstraite, et d'un fichier C `xxxx.c` d'implémentation concrète :

`mygraph :` bibliothèque de manipulation de graphes pondérés orientés statiques. Les sommets sont étiquetés par des entiers et les poids sont des flottants double précision `double`. On propose deux implémentations :

- par listes d'adjacence dans le fichier `mygraph-lst.c` : nous utiliserons cette implémentation concrète pour l'implémentation du parcours BFS (ex1) et de l'algorithme de Dijkstra (ex2, ex3 et ex5).
- par matrice d'adjacence dans le fichier `mygraph-mat.c` : nous utiliserons cette implémentation concrète uniquement pour l'implémentation de l'algorithme de Floyd-Warshall.

Quelque soit l'implémentation concrète choisie, l'interface abstraite `mygraph.h` est la même et contient les primitives classiques implémentées lors du précédent TP. La bibliothèque utilise la bibliothèque `mylist`, également fournie.

`mystack :` bibliothèque de manipulation de piles d'entiers, implémentées par maillons chaînés

`myqueue :` bibliothèque de manipulation de files d'entiers, implémentée par ring buffer.

`mypqueue :` bibliothèque de manipulation de files de priorité (clés de type `double`, valeurs de type `entier`), implémentées concrètement avec des tas

Toutes les primitives mises à dispositions par ces bibliothèques, notamment leurs noms et leur prototypes, sont décrites dans les fichiers *header* associés. Pour les plus curieux, vous pourrez prendre le temps de regarder les implémentations concrètes dans les fichiers C correspondant, notamment pour réviser l'implémentation des files de priorités avec une structure de données de tas.

Tous les tests seront écrits dans le fichier `test_a_completer.c` qui contiendra la fonction principale `main` qui est mis à votre disposition. Ce fichier contient déjà les fonctions permettant de créer les graphes exemples cités dans ce TP pour vous faire gagner un peu de temps. Vous renommerez ce fichier `NOM_test.c`

L'usage de `valgrind` est recommandé pour valider l'absence de fuites mémoires et d'accès mémoire illégaux.

Tous les fichiers de code devront être déposés sur Moodle pour mercredi prochain.

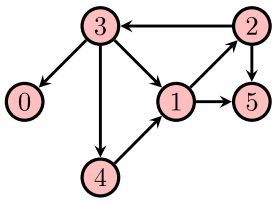

Figure 1 – Graphe 1

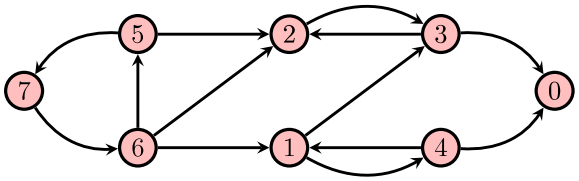

FIGURE 2 – Graphe 2

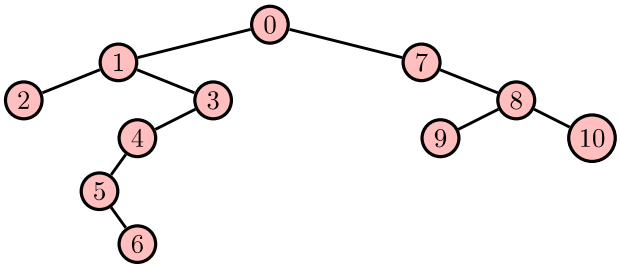

FIGURE 3 – Graphe 3 Arbre

Exercice 1 (Parcours en largeur).

La fonction ci-dessous sera implémentée dans un fichier C nommé `NOM_bfs.c`

1. Écrire une fonction `stack *bfs(wgraph *g, int source)` qui implémente le parcours en largeur sur un graphe à partir d'un sommet d'étiquette `source` à l'aide d'une file simple. La fonction renvoie une pile dans laquelle on a empilé les étiquettes des sommets traités au fur et à mesure.
2. Renommer le script de compilation `compile-a-adapter.sh` fourni en `compile.sh` et l'adapter pour compiler pour le moment uniquement les fichiers nécessaires à l'exécution de votre code et effectuer l'édition de liens.
3. En créant une fonction `test_bfs` dans votre fichier `NOM_test.c`, appliquer l'algorithme au graphe 1 en prenant le sommet d'étiquette 1 comme source, puis le sommet d'étiquette 3.
4. En complétant la fonction `test_bfs`, appliquer l'algorithme au graphe 2 en prenant le sommet d'étiquette 1 comme source. Quelle propriété du parcours BFS est illustrée par cet exemple ?
5. Appliquer l'algorithme au graphe 3 arbre suivant en prenant le sommet d'étiquette 0 comme source : A quel genre de parcours d'arbre correspond le parcours en largeur ?
6. L'utilisation d'une file est-elle indispensable dans l'algorithme BFS ?

On considère maintenant des graphes pondérés. **La pondération associée à un arc** $(u, v) \in E$ **sera toujours positive ou nulle**, et sera vue comme une distance entre les deux sommets u et v. Elle sera

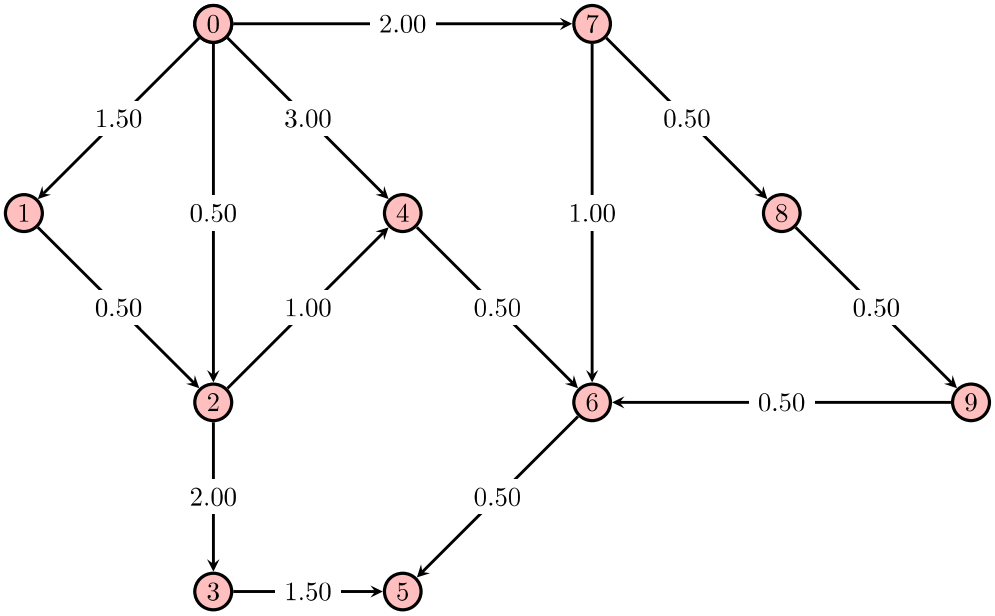

FIGURE 4 – Graphe pondéré 4

donc notée $d(u, v)$ au lieu de $w(u, v)$.

Définition 1 (Distance entre deux sommets)

Soit u et v deux sommets d'un graphe $g = (V, E)$. S'il existe un chemin $u \to^* v$ entre u et v de la forme :

$$u = x_0 \to x_1 \to \ldots \to x_{p-1} \to x_p = v$$

alors la distance entre u et v est définie par :

$$\text{dist}\,(u, v) = \sum_{i=0}^{p-1} d(x_i, x_{i+1}) \geq 0$$

S'il n'existe pas de chemin entre u et v, cette distance est considérée comme étant infinie :

$$\text{dist}\,(u, v) = +\infty$$

Enfin, on considérera que la relation d'adjacence est réflexive et qu'un sommet est toujours à distance 0 de lui-même :

$$\text{dist}\,(u, u) = 0$$

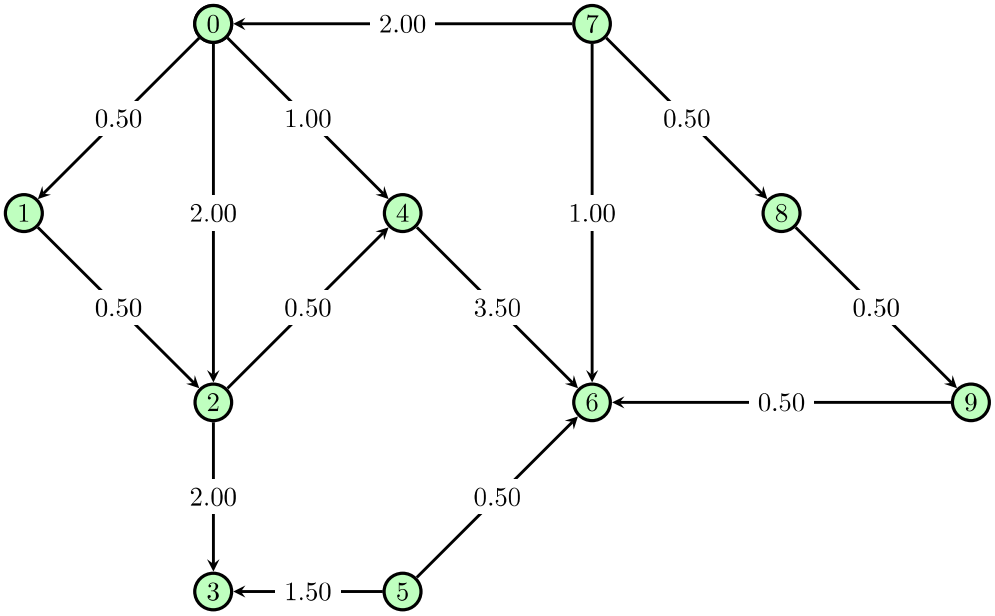

FIGURE 5 – Graphe pondéré 5

Exercice 2 (Calcul du plus court chemin par Dijkstra).

La fonction ci-dessous sera implémentée dans un fichier C nommé `NOM_dijkstra.c`

1. Écrire une fonction `double *dijkstra(wgraph *g, int source)` qui implémente l'algorithme de Dijkstra vu en cours. L'algorithme alloue, remplit et renvoie un tableau de taille n_v contenant la distance du chemin le plus court entre la source et chacun des sommets du graphe. Si un sommet d'étiquette v n'est pas atteignable depuis la source, `dist[v]` contiendra la valeur `DBL_MAX`.
2. Adapter le script `compile.sh` pour intégrer ce fichier lors de la compilation séparée.
3. En créant une nouvelle fonction de test `test_dijkstra` dans le fichier principal `NOM_test.c`, tester votre algorithme sur le graphe pondéré 4 vu en cours et sur le graphe 5. Pour valider les distances les plus courtes renvoyées par l'algorithme, vous afficherez le tableau des distances obtenu avec la fonction fournie `print_dist_array`.
4. Commenter votre code ligne à ligne en évaluant la complexité dans le pire des cas de votre algorithme. Puis donner la complexité globale de l'algorithme de Dijkstra telle que vous l'avez implémenté.
5. Quelle est la complexité spatiale de l'algorithme implémenté ?
6. Pourquoi avons-nous privilégiez l'implémentation concrète de graphe par listes d'adjacence, plutôt que celle par matrice d'adjacence ?

Exercice 3 (Amélioration : renvoie explicite des plus courts chemin vers chaque sommet).

La fonction `dijkstra` implémentée à l'exercice précédent ne renvoie que le tableau des distances les plus courtes vers chaque sommet, mais ne donne pas le détail du chemin le plus court permettant d'aller de la source jusqu'à un autre sommet.
Il est possible de reconstruire ces chemins les plus courts en mettant à jour, au fur et à mesure de l'algorithme, un tableau `previous_vertex` de taille n_v de sorte que `previous_vertex[v]` contienne toujours, à la fin de chaque tour de boucle, l'étiquette du prédécesseur de v **situé sur le chemin le plus court depuis la source**, parmi tous les chemins les plus courts découverts à ce stade de l'algorithme.
On recompose ensuite, pour chaque sommet destination v, le chemin le plus de court de la source jusqu'à v - s'il existe - en remontant le chemin à partir de v et jusqu'à la source grâce à ce tableau. On empile les sommets rencontrés lors de cette remontée dans une pile, ce qui permettra finalement de reconstituer l'intégralité du chemin, dans le bon ordre si on lit du haut vers le fond de pile.

1. Pour un sommet v, comment pourra-t-on s'apercevoir qu'il n'existe pas de chemin reliant la source à v (et donc pas de plus court chemin !) ?
2. Copier la fonction `dijkstra` en une nouvelle fonction `dijkstra_with_paths` et compléter cette nouvelle fonction pour qu'elle mette à jour un tableau `previous_vertex` comme indiqué ci-dessous. Faites afficher et valider le tableau obtenu sur les tests de l'exercice précédent.
3. Implémenter une fonction

```
stack *create_path(int nv, int ustart, int uend, int *previous_vertex)
```

   qui reconstitue le plus court chemin entre la source d'étiquette `ustart` et le sommet d'étiquette `uend`.
4. Tester sur les graphes 4 et 5 en prenant comme source le sommet 0 et en vérifiant si tous les chemins trouvés, lorsqu'ils existent, sont bien les plus courts. Une fonction permettant de calculer tous les chemins à la fois depuis une source est mise à disposition.

Exercice 4 (Application-DM : mini-Mappy).

On fournit deux fichiers de données :

`cities.txt` : un nombre de villes, suivi de la liste des villes, l'ordre d'énumération de ces villes permettant d'étiqueter chaque ville avec un numéro. La première ville du fichier est étiquetée par 0, la seconde par 1...

`distances.txt` : un fichier indiquant des liaisons autoroutières existantes entre deux villes et la distance associée en km.

Créer un code `mini-mappy` utilisable en ligne de commande permettant d'obtenir le trajet le plus court entre deux villes. L'utilisateur donne en arguments d'entrée, sur la ligne de commande, le nom de sa ville de départ et le nom d'une ville d'arrivée, et le code renvoie le chemin le plus court entre ceux deux villes, en indiquant toutes les étapes intermédiaires et la distance totale.
Pour vous faire gagner du temps, on met également à disposition :

- une bibliothèque de table de hachage `myhashtbl` avec des clés de type chaînes de caractères et des valeurs associées entières, qui utilise des listes de couple (`char *`, `int`) implémentées par une bibliothèque `myhashtbl-lst` ;
- un script de compilation `compile-mini-mappy.sh`
- le code `mini-mappy-a-completer.c` à télécharger, renommer `NOM_mini-mappy.c` et à compléter