TP n 16 - Graphes : Plus courts chemins

Notions abordées

- Représentation de graphes orientés pondérés
- Listes chaînées (encore)
- Plus courts chemins par l'algorithme de Dijkstra
- File de priorité.

Avant de commencer, télécharger sur cahier de prépa l'archive `materiel _TP16.zip` et la décompresser. Il peut aussi être utile d'avoir à disposition les fichiers codés au TP précédent.

Exercice 1 Algorithme de Dijkstra en C

Dans cet exercice on s'intéresse à l'implantation de l'algorithme de Dijkstra en C, basé sur une implémentation naïve de file de priorité qui devra être améliorée par la suite. Dans l'algorithme de Dijkstra, comme dans les algorithmes de parcours en général, on est amené à souvent parcourir les successeurs d'un sommet donné, et pour que cette opération soit efficace, on choisit une représentation par listes de successeurs, c'est-à-dire par un tableau de liste (chaînées) de successeurs. Cependant il faut un peu modifier la structure de graphe vue au TP précédent pour tenir compte de la pondération des arcs.

Question 1
Prendre connaissance de la structure de liste chaînée pour des entiers munis d'une pondération flottante proposée dans les fichiers `liste_ch_int_float.h` et `liste_ch_int_float.c`. Compléter la fonction `main` de ce dernier afin de tester les fonctions.

Question 2
Une implémentation naïve de file de priorité consiste à mettre les éléments dans une liste chaînée. Ainsi les fichiers `file_prio.h` et `file_prio.c` proposent une implémentation **naïve** de la structure de file de priorité basée sur la structure de liste chaînée pour des entiers munis d'une pondération flottante utilisée ci-avant. Prendre connaissance de cette implémentation et noter les complexités des opérations. Les comparer avec celles que l'on pourrait espérer dans une bonne implémentation de file de priorité (avec des tas binaires par exemples).

Question 3
Définir une fonction `dijkstra` prenant en argument un graphe et un sommet s, ainsi que l'adresse de deux pointeurs de types `float*` et `int*`. Cette fonction devra

- créer le tableau (persistant) des distances de s à chaque sommet i du graphe et enregistrer ce tableau dans le pointeur de type `float*` passé par référence. On utilisera `-1` pour représenter l'absence de chemin de s à i.
- créer le tableau donnant les prédécesseurs possibles pour la fabrication de plus court chemin de s à chaque sommet i du graphe et enregistrer ce tableau dans le pointeur de type `int*` passé

par référence. On utilisera à nouveau `-1` pour représenter l'absence de prédécesseurs possible en l'absence de chemin de s à i.

Question 4
Définir une fonction `construit_chemin` prenant en argument le tableau des prédécesseurs fourni par la fonction précédente et un sommet i et calculant un chemin optimal de la source à i (sous la forme d'une liste chaînée par exemple).

Question 5 $\ast$
Proposer une meilleure implémentation de file de priorité. On peut reprendre l'implémentation de tas par tableau développée dans le DM, la traduire en C et y ajouter une fonction de diminution de priorité d'un élément déjà présent.

Exercice 2 Algorithme de Dijkstra en OCaml

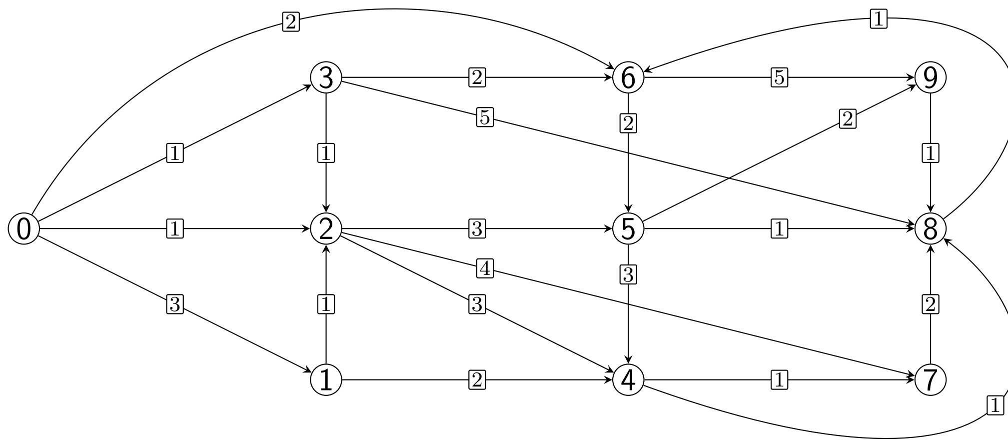
Dans cet exercice on s'intéresse à l'implantation de l'algorithme de Dijkstra en OCaml. On définit le type suivant, permettant la représentation d'un graphe orienté et pondéré :

```
1   type graph = (int * float) list array
```

Vous trouverez sur cahier de prépa un fichier `bheap.ml` contenant une implantation de tas binaire qui n'est pas sans rappeler ce que vous avez du implanter pour l'algorithme de Huffman. Cependant l'interface est ici plus riche, et permet notamment la diminution de la priorité d'un élément déjà dans la structure, ce qui en fait une implantation du type abstrait FILE DE PRIORITÉ. Plus précisément, voici la liste des fonctions disponibles :

- `cree_tas_vide : unit -> tas` création d'un tas vide.
- `is_empty : tas -> bool` teste si un tas est vide.
- `mem_tas : tas -> int -> bool` teste si un entier est déjà présent dans le tas.
- `elem_prio_min : tas -> int * float` retourne (*sans l'enlever*) l'élément du tas de priorité minimale, ainsi que sa priorité (qui est donc . . . minimale).
- `insere_tas : tas -> int -> float -> unit` permet l'ajout d'un élément muni d'une priorité dans le tas.
- `decrease_prio : tas -> int -> float -> unit` permet de décrément la priorité d'un élément *se trouvant déjà dans le tas* : `(decrease_prio t x f)` change la priorité de `x` à `f` dans le tas `t`.
- `supprime_min : tas -> unit` supprime l'élément de priorité minimale dans le tas retourné par `elem_prio_min`.

Le module permet le stockage d'entiers auxquels sont attachés une priorité flottante (sous la forme d'un couple de type `int * float`).
On fournit de plus un graphe `g_ex` représenté ci-dessous. Ce graphe *devra* être utilisé pour tester l'algorithme de Dijkstra.



Question 1
Implanter `dijkstra : graph -> int -> float option array * int option array` prenant en argument un graphe et un sommet s et calculant :

- le tableau des distances s à chaque sommet i du graphe (on utilisera `None` pour représenter l'absence de chemin de s à i)
- un tableau donnant les prédécesseurs possibles pour la fabrication de plus court chemin de s à chaque sommet i du graphe (on utilisera `None` pour représenter l'absence de prédécesseurs possible)

À titre d'exemple, sur le graphe `g_ex` vous devriez obtenir les tableaux :

- `[|Some 0.; Some 3.; Some 1.; Some 1.; Some 4.; Some 4.; Some 2.; Some 5.; Some 5.; Some 6.|]`
- `[|Some 0; Some 0; Some 0; Some 0; Some 2; Some 2; Some 0; Some 2; Some 5; Some 5|]`

Question 2
Implanter une fonction `construit_chemin : int option array -> int -> int list` prenant en argument le tableau des prédécesseurs fourni par la fonction précédente et un sommet i et calculant un chemin optimal de la source à i.