Dans ce TP, on va travailler sur des graphes pondérés (par des poids positifs), représentés en C par des listes d'adjacence. On se donne les deux structures suivantes :

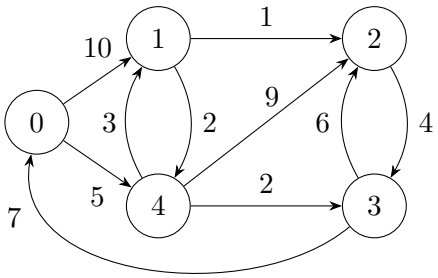
```c
struct arc {
  int sommet; // sommet d'arrivée
  float poids;
  struct arc* next;
};
typedef struct arc arc;

struct graphe {
  int nb_sommets;
  arc** voisins;
};
typedef struct graphe graphe;
```

Si un graphe $G = (S, A, \omega)$ possède n sommets, on supposera que $S = [\![0, n-1]\!]$. Un tel graphe sera représenté par une variable `G` de type `graphe` telle que :

- `G.nb_sommets` contient la valeur n ;
- `G.voisins` pointe vers le début d'un tableau de taille n tel que, pour $s \in [\![0, n-1]\!]$, `G.voisins[s]` pointe vers le début d'une liste chaînée des voisins de s :
  - si `G.voisins[s]` contient `NULL`, alors s n'a pas de voisin ;
  - sinon :
    - `G.voisins[s]->sommet` contient un sommet t et `G.voisins[s]->poids` contient une valeur ω tels que $s \xrightarrow{\omega} t$ est un arc de G ;
    - `G.voisins[s]->next` pointe vers le prochain maillon de la liste chaînée, qui contient le reste des voisins de s (ou contient `NULL` s'il n'y a pas d'autre voisin).

Un fichier `tp_graphe.c` est disponible sur la page web du cours, et fournit la structure ci-dessus ainsi qu'une implémentation du graphe suivant (dans la fonction `main`) :



1 Fonctions utilitaires

1. Écrire une fonction `graphe init_graphe(int n);` créant un graphe à n sommets n'ayant aucun arc.
2. Écrire une fonction `float poids(graphe G, int s, int t);` renvoyant le poids de l'arc $s \to t$ dans le graphe G (on renverra -1 si l'arc n'existe pas).
3. Écrire une fonction `void ajouter_arc(graphe G, int s, float w, int t);` rajoutant (s'il n'existe pas) l'arc $s \xrightarrow{\omega} t$ au graphe G.
4. Écrire une fonction `void free_graphe(graphe G);` libérant toute la mémoire allouée sur le tas par le graphe G.
5. Tester vos fonctions sur le graphe fourni.

2 Parcours en largeur, calcul des distances

Dans cette partie, on ignore les poids des arcs, et on considère que le graphe est simplement un graphe orienté. Dans ce cas, on peut calculer les distances et les plus courts chemins vers n'importe quel sommet $t \in S$ depuis un sommet source s à l'aide d'un parcours en largeur, dont on rappelle ici le pseudo-code :

Algorithme 1 : Parcours en largeur, calcul des distances

Données : Un graphe G donné par listes d'adjacences,
un sommet de départ s
`a_traiter` $\leftarrow \{s\}$; `// implémentée par une file`
`deja_vu` $\leftarrow$ [`Faux`, ..., `Faux`] ;
`dist` $\leftarrow [-1, \ldots, -1]$;
`pred` $\leftarrow [-1, \ldots, -1]$;
`deja_vu`$[s] \leftarrow$ `Vrai` ;
`dist`$[s] \leftarrow 0$;
`pred`$[s] \leftarrow s$;
tant que *`a_traiter` est non vide* **faire**
  $u \leftarrow$ sortir le prochain élément de `a_traiter` ;
  pour *tout voisin v de u tel que* `deja_vu`$[v]$ *est* `Faux` **faire**
    `a_traiter` $\leftarrow$ `a_traiter` $\cup \{v\}$;
    `deja_vu`$[v] \leftarrow$ `Vrai` ;
    `dist`$[v] \leftarrow$ `dist`$[u] + 1$;
    `pred`$[v] \leftarrow u$;
retourner *`dist`, `pred`* ;

Une implémentation d'une structure de file est fournie sur la page web du cours : télécharger les fichiers `file.c` et `file.h` (dans le même dossier que le fichier de votre TP), et rajouter l'instruction suivante au début de votre fichier.

```
#include "file.h"
```

Cette implémentation fournit un type `file` ainsi que les fonctions suivantes (attention : on travaille avec des **pointeurs** vers des files) :

file.h

```
file* creer_file(void);
int file_taille(file* f);
bool file_est_vide(file* f);
void file_push(file* f, int x);
int file_peek(file* f);
int file_pop(file* f);
void free_file(file* f);
```

On rappelle une limitation du langage C : il n'existe pas de type couple, on ne peut donc pas renvoyer deux valeurs à la fin d'une fonction. Ainsi, il est usuel en C d'avoir une fonction prenant des arguments supplémentaires qui sont des pointeurs, et d'écrire les résultats de la fonction dans ces pointeurs.

6. Écrire une fonction `void distances(graphe G, int s, float* dist, int* pred);` prenant en entrée un graphe G et un sommet de départ s, et implémentant l'algorithme de calcul des distances. On supposera que les pointeurs `dist` et `pred` pointent vers le début de tableaux de taille `G.nb_sommets` dont les cases sont déjà initialisées par des -1, et on modifiera ces tableaux par effets de bords.
7. Écrire une fonction `void chemin(int s, int t, int* pred);` prenant en arguments deux sommets s et t d'un graphe, et un tableau `pred` rempli par la fonction précédente, et affichant le plus court chemin de s à t.

Plus court chemin de 0 à 3 dans le graphe fourni

```
3 <- 4 <- 0
```

3 Algorithme de Dijkstra

On rappelle que l'algorithme de Dijkstra travaille sur un graphe **pondéré** avec des poids positifs, représenté par listes d'adjacence, et calcule les distances et plus courts chemins vers tous les sommets accessibles depuis un sommet s donné. Son pseudo-code est le suivant :

Algorithme 2 : Algorithme de Dijkstra

Données : Un graphe pondéré $G = (S, A, \omega)$ donné par listes d'adjacence avec $\omega(A) \subset \mathbb{R}_+$, un sommet s
Résultat : Les distances $\delta(s,t)$ pour tout $t \in S$

```
dist ← [+∞, ··· , +∞] ;
pred ← [−1, ··· , −1] ;
deja_vu ← [Faux, ..., Faux] ;
dist[s] ← 0 ;
pred[s] ← s ;
F ← {s} ;
tant que F ≠ ∅ faire
    u ← Retirer de F un sommet v vérifiant dist[v] minimal ;
    pour tout voisin v de u faire
        si dist[v] > dist[u] + ω(u,v) alors
            dist[v] ← dist[u] + ω(u,v) ;
            pred[v] = u ;
            si v ∉ F et deja_vu[v] est Faux alors
                Ajouter v à F ;
    deja_vu[u] ← Vrai ;
retourner dist, pred;
```

8. Appliquer à la main l'algorithme de Dijkstra sur le graphe fourni depuis le sommet source 0.

On pourrait utiliser un tableau pour représenter F, et le parcourir avec un boucle à chaque fois qu'on veut extraire un sommet v vérifiant `dist`$[v]$ minimal, mais ce ne serait pas très efficace. Un meilleur choix serait d'utiliser une structure de file de priorité-min, en utilisant `dist`$[v]$ comme priorité lorsqu'on rajoute le sommet v.
Mais il reste alors un problème : si `dist`$[v]$ diminue, comment mettre à jour la priorité de v dans F ? On adopte alors la stratégie suivante : ignorer le test $v \notin F$. Un même sommet v sera donc potentiellement présent plusieurs fois dans F avec des priorités différentes, ce que l'on compense de la manière suivante : ne pas traiter u dans la boucle `while` s'il a déjà été vu.
On fournit une structure de priorité-min (implémentée par un tas) dans les fichiers `tasmin.c` et `tasmin.h` :

tasmin.h

```c
tasmin* creer_tasmin(void);
int tasmin_taille(tasmin* t);
bool tasmin_est_vide(tasmin* t);
void tasmin_push(tasmin* t, int x, float p); // O(log n) ; x : sommet, p : priorité
int tasmin_peek(tasmin* t); // O(1)
int tasmin_pop(tasmin* t); // O(log n)
void free_tasmin(tasmin* t);
```

9. Écrire une fonction `void dijkstra(graphe G, int s, float* dist, int* pred);` implémentant cet algorithme. La valeur $+\infty$ n'existant pas en C, on supposera que les cases de `dist` ont été initialisées par une valeur suffisamment grande (par exemple : 1000).
On pourra tester cette fonction sur le graphe fourni (à l'aide de la fonction `chemin`) :

Plus court chemin de 0 à 2 dans le graphe fourni

```
distance 0--2 : 9.000000
2 <- 1 <- 4 <- 0
```

4 Algorithme $A^\star$

L'algorithme $A^\star$ est une variante de l'algorithme de Dijkstra se concentrant sur un sommet destination t particulier, en utilisant une **heuristique** qui estime la distance entre un sommet v et ce sommet t. Dans cette partie, nos fonctions prendront un paramètre supplémentaire `pos` qui sera un tableau dont les indices sont les sommets du graphe et les valeurs associées sont les coordonnées des sommets (afin de pouvoir calculer les distances). Le pseudo-code de $A^\star$ est rappelé ci-dessous :

Algorithme 3 : Algorithme $A^\star$

Données : Un graphe pondéré $G = (S, A, \omega)$ donné par listes d'adjacence avec $\omega(A) \subset \mathbb{R}_+$, deux sommets s et t, une heuristique h
Résultat : La distance $\delta(s,t)$
`dist` $\leftarrow [+\infty, \cdots, +\infty]$;
`pred` $\leftarrow [-1, \cdots, -1]$;
`deja_vu` $\leftarrow [Faux, \ldots, Faux]$;
`dist`$[s] \leftarrow 0$;
`pred`$[s] \leftarrow s$;
$F \leftarrow \{s\}$;
tant que $F \neq \emptyset$ **faire**
  $u \leftarrow$ Retirer de F un sommet v vérifiant $d_s[v] + h(v,t)$ minimal ;
  si $u = t$ **alors**
    retourner $d_s[t]$;
    pour *tout voisin* v *de* u **faire**
      si $dist[v] > dist[u] + \omega(u,v)$ **alors**
        `dist`$[v] \leftarrow$ `dist`$[u] + \omega(u,v)$;
        `pred`$[v] = u$;
        si $v \notin F$ *et* `deja_vu`$[v]$ *est Faux* **alors**
          Ajouter v à F ;
    `deja_vu`$[u] \leftarrow Vrai$;
retourner $+\infty$ `// pas de chemin entre` s `et` t

On fournit le type suivant pour représenter des coordonnées en C :

```
struct coord{
  float x;
  float y;
};
typedef struct coord coord;
```

10. Écrire une fonction `float h(coord* pos, int v,int t);` prenant en arguments un tableau `pos` comme décrit précédemment et deux sommets v et t, et renvoyant la distance euclidienne entre v et t.
11. Écrire une fonction `float astar(graphe G, int s, int t, coord* pos, float* dist, int* pred);` implémentant l'algorithme précédent et renvoyant la plus courte distance de s à t dans G.
12. Tester votre fonction avec le graphe fourni et le tableau des positions suivant :

```
coord pos[5] = {{.x=0,.y=0},{.x=1,.y=2}, {.x=-3,.y=-1}, {.x=4,.y=-1}, {.x=2,.y=2}};
```

Plus court chemin de 0 à 2 dans le graphe fourni

```
distance 0--2 : 9.000000
2 <- 1 <- 4 <- 0
```

13. Si l'heuristique surestime la distance réelle de v à t, l'algorithme $A^\star$ peut renvoyer un chemin non optimal. Observer ce phénomène en modifiant les positions des sommets.