TP 19 : Algorithmes de plus court chemin : Dijkstra

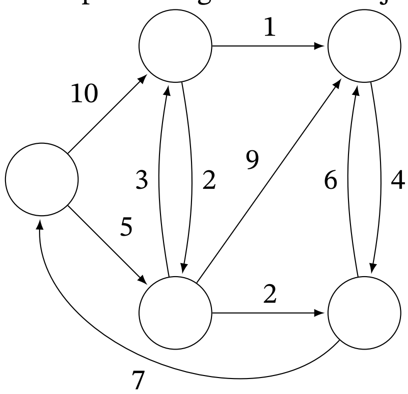
On est maintenant au second TP d'algorithme de plus court chemin dans les graphes. Ici, on travaille sur l'algorithme de Dijkstra : je vous fournis d'abord une implémentation simpliste des files de priorités, puis on met en place l'algorithme de Dijkstra et on travaille autour, et enfin on implémente efficacement les files de priorités pour accélérer l'algorithme de Dijkstra.

Exercice 1 – Mise en place du TP

▶ **Question 1** Extraire l'archive de travail dans un nouveau dossier.

Cette archive contient les fichiers avec lesquels on va travailler :

— Vous pourrez travailler dans le fichier `tp19.ml`. Il contient deux exemples de graphes : le premier correspond au graphe du cours en exemple de l'algorithme de Dijkstra.



— Le fichier `USA- ... .gr` est un fichier contenant toutes les arêtes d'un graphe représentant le réseau routier de la ville de New York. Il contient 264346 sommets et 733846 arêtes. D'autres graphes dans le même format de fichier sont disponibles ici, notamment des graphes plus gros : `http://www.diag.uniroma1.it/challenge9/download.shtml`, notamment le plus gros risque d'être limitant si vos fonctions sont inefficaces avec 23947347 sommets et 58333344 arcs.

— Le fichier `generer_graphe.ml` qui fournit un module pour lire des graphes depuis des fichiers `.gr` et en tirer leur liste d'adjacence. Un exemple d'utilisation est dans le fichier principal.

— Le fichier `fileprio.ml` propose un module implémentant les files de priorité. Les fonctions importantes sont, avec `'s` le type des éléments de la file et `'p` le type de leurs priorités :
  — `Fileprio.create : unit -> Fileprio.t` crée une nouvelle file de priorité vide,
  — `Fileprio.is_empty : Fileprio.t -> bool` vérifiant si une file de priorité est vide.
  — `Fileprio.add : 's -> 'p -> Fileprio.t -> unit` tel que l'appel `Fileprio.add x d f` ajoute l'élément `x` avec la priorité `d` dans la file de priorité `f`. On suppose que `x` n'y est pas encore.
  — `Fileprio.decrease : 's -> 'p -> Fileprio.t -> unit` diminuant la priorité d'un élément de la file déjà présent.

— Le fichier `fileprio2.ml` qui commence l'implémentation des files de priorité par les tas, à remplir dans l'Exercice 3.

À la fin du fichier `tp19.ml` se trouve des tests de cette structure de donnée. Normalement, les occurrences du module `Fileprio` sont soulignées en rouge, signe qu'OCaml ne connaît pas ce module. Il faut en effet compiler ce module avant qu'un autre fichier puisse l'utiliser.

▶ **Question 2** Vérifier que les deux fichiers `fileprio.ml` et `generer_graphe.ml` sont dans votre dossier de travail.

▶ **Question 3** Lancer la commandes `ocamlc fileprio.ml`. Cela produit des fichiers *bytecode* d'extensions `.cmo` et `.cmi` rendant accessible le module `Fileprio` dans tout fichier `.ml` se trouvant dans le même dossier : grosso modo, ce sont des fichiers binaires contenant des versions précompilées des fonctions du module.

Lancez la commande `Ctrl+s` sur votre fichier `tp19.ml` et les lignes rouges auront disparus ! Vous pouvez maintenant utiliser le module `Fileprio`. Si vous voulez éviter d'utiliser la syntaxe `Fileprio.(...)` pour utiliser les fonctions de ce module, la déclaration `open Fileprio` permet d'utiliser directement les fonctions (à l'instar de `from ... import *` en Python). Par exemple :

OCaml

```ocaml
(* ici, is_empty n'existe pas, mais on peut écrire *)
let _ = assert (Fileprio.is_empty (Fileprio.create ()))

(* on "ouvre" maintenant le module *)
open Fileprio

(* ici, le module est ouvert et on n'a plus besoin d'utiliser la syntaxe [Fileprio.(...)] *)
let _ = assert (is_empty (create ()))

(* attention, on ne peut pas fermer Fileprio dans la suite du fichier ! *)
```

Attention, la compilation `ocamlc fileprio.ml` ne permet que de faire comprendre au serveur de langage d'OCaml (celui qui s'occupe de souligner les erreurs en direct pendant l'édition du fichier dans Codium). Pour utiliser ce module dans Utop, il faut évaluer l'expression suivante :

Utop — OCaml

```
utop # let p = Fileprio.create ();;
Line 1:
Error: Reference to undefined global `Fileprio'

utop # #load "fileprio.cmo";; (* <----- ici *)

utop # let p = Fileprio.create ();;
val p : ('_weak1, '_weak2) Fileprio.t = {Fileprio.file = []; taille = 0}
```

qui permet alors d'utiliser le module.

▶ **Question 4** Faire de même pour le module `Generer_graphe`. De même, la ligne rouge sous l'occurence de ce module dans `tp19.ml` devrait disparaître.

Remarque 1

Choisir d'ouvrir le module est une question de goût et de style. Pour des modules aussi simples, vous pouvez le faire, mais je vous le déconseille pour des plus gros modules dont les noms de fonctions risquent de se confondre avec les vôtres / entre plusieurs modules. Par exemple, si on écrivait `open Stack` puis immédiatement `open Queue`, la fonction `create` viens du module `Queue` et pas `Stack` ...Alors qu'entre les deux `open`, `create` viendrais du module `Queue`. Ce problème arrivera dès l'exercice 3.

▶ **Question 5** Déterminer la complexité de chaque opération du module `Fileprio`.

Exercice 2 – Implémentation de l'algorithme de Dijkstra

Pour rappel, voici l'algorithme de Dijkstra implémenté en pseudocode.

Pseudo-code

```
Fonction DIJKSTRA(𝒢, s)
    dist ← [∞; ∞; … ; ∞]                    ▷ tableau de taille n = |S|
    dist[s] ← 0
    à-visiter = FILEPRIOVIDE()
    INSÉRER(à-visiter, s, 0)
    Tant que ¬ESTVIDE(à-visiter) , faire
        (x, d_x) ← RETIREMIN(à-visiter)
        Pour y voisin de x , faire
            d ← d_x + p(x, y)
            Si d < dist[y] alors
                Si dist[y] < ∞ alors
                    DIMINUERPRIORITÉ(à-visiter, y, d)
                Sinon
                    INSÉRER(à-visiter, y, d)
                dist[y] ← d
    retourne dist
```

▶ **Question 1** Implémenter cet algorithme en une fonction `dijkstra : (int * float) list array -> int -> float array`. Les ∞ du pseudocode seront représentés par la valeur `infinity`, et si le sommet y n'est pas accessible depuis le sommet initial, le tableau en sortie contiendra `infinity`.

▶ **Question 2** Déterminer précisément sa complexité temporelle et spatiale.

▶ **Question 3** Adapter cette fonction en une fonction `dijkstra_point_a_point : (int * float) list array -> int -> int -> float` qui prend un graphe et deux sommets en entrée et calcule le poids du plus court chemin entre les deux.

▶ **Question 4** Adapter votre fonction en `dijkstra_parents : (int * float) list array -> int -> int option array` où l'appel `dijkstra_parents g s` renvoie le tableau `par` des parents des sommets dans l'arbre de parcours, comme suis :

— `par.(i) = None` signifie que le sommet `i` n'est pas accessible depuis `y`,
— `par.(s) = Some s` (le père du sommet initial est lui-même)
— `par.(i) = Some j` sinon, avec `j` si le plus court chemin construit par l'algorithme termine par l'arc $j \to i$.

▶ **Question 5** En déduire une fonction calculant l'arbre de parcours de l'algorithme de Dijkstra. On pourra utiliser le type `type arbre = N of int * arbre list`.

Exercice 3 – La structure de tas

On va maintenant implémenter les tas min pour rendre plus efficace les opérations sur les files de priorités dans l'algorithme de Dijkstra. On va adapter quelque peu les tas du cours pour ce dont on a besoin ici :

OCaml

```ocaml
type t = {
  mutable taille : int;
  priorites : float array;
  position : int array;
  cle : int array;
}
```

Dans ce type :

— `taille` désigne la taille actuelle du tas,
— La capacité de la file est fixée à l'initialisation et est la taille des tableaux `priorites, position, cle`.

Selon le même principe que pour les piles implémentées par des tableaux, une partie de ces tableaux contiendra des valeurs quelconques.

— `priorites` correspond à la notion de tas vue en cours : la tranche `priorites[0:taille]` représente un arbre binaire complet à `taille` nœuds respectant la propriété de tas min.
— Pour le tableau `position`, on a deux cas :
  — si `x` est dans la file de priorité, `position.(x)` est la position de la priorité de `x` dans le tableau `priorite`,
  — sinon, `position.(x) = -1`
— pour `0 <= i < taille`, `cle.(i)` est le sommet dont la priorité est stockée dans `position.(i)`.

Pour un sommet `x`, sa priorité est donc stockée en `priorite.(position.(x))` et on a `cle.(position.(x)) = x`.

On implémentera ces tas dans un nouveau module, dans le fichier `fileprio2.ml`. Le temps d'implémenter les fonctions de ce module, on pourra les tester en utilisant des assertions et en les évaluant directement dans Utop : après l'avoir fait, un petit coup de `ocamlc fileprio2.ml` et il sera disponible pour l'utiliser dans `tp19.ml` !

▶ **Question 1** Implémenter la fonction `create : int -> t` tel que `create n` crée une file de priorité vide de capacité maximale `n`. Implémenter également `is_empty : t -> bool`.

Remarque 2

Il est possible de réserver l'utilisation de certaines fonctions d'un module `Mod` à l'intérieur de `mod.ml` : pour cela, il faut écrire un fichier `mod.mli` contenant l'interface voulue du module et toute fonction définie dans `mod.ml` mais pas dans `mod.mli` ne sera pas accessibles à l'extérieur du module.

Commençons par le commencement : si une simple fonction `swap` habituelle permettait de permuter deux éléments dans le tableau représentant nos tas dans le cours, ici il faut faire attention à garder à jour les trois tableaux en même temps.

▶ **Question 2** Écrire la fonction `echanger : t -> int -> int -> unit` tel que `full_swap f x y` échange les positions dans le tas des sommets `x` et `y`. On maintiendra tous les invariants cités précédemment, sauf la propriété de tas min.

▶ **Question 3** Deux trois fonctions qui seront utiles plus tard : `fils_gauche : int -> int`, `fils_droit : int -> int`, `parent : int -> int` qui à un nœud du tas renvoie son fils gauche / fils droit / père.

Pour insérer dans le tas un sommet, on peut l'ajouter au prochain emplacement disponible dans le tas (situé en `t.taille`). Alors, éventuellement la propriété de tas min n'est plus respectée et la priorité du sommet ajoutée est inférieure à celle de son père dans le tas : il faut alors faire "remonter" le sommet et sa priorité dans le tas jusqu'à ce que la propriété soit respectée (éventuelllement, on fait remontre le sommet jusqu'à la racine). Cette opération de remontée est appelée une percolation, vers le haut.

▶ **Question 4** En déduire une fonction `percoler_haut : t -> int -> unit` tel que `percoler_haut f i` percole vers le haut le sommet `f.cle.(i)` dans le tas.

Cela suffit à implémenter `add`. Pour `decrease`, on remarque que puisque l'on connaît l'emplacement dans le tas du sommet `x` grâce au tableau `t.position`, il suffit d'y diminuer sa priorité puis éventuellement de le faire remonter si besoin, toujours avec `percoler_haut`.

▶ **Question 5** En déduire la fonction `add : t -> int -> float -> unit` tel que `add f x p` ajoute le sommet `x` au tas avec la priorité `p` et `decrease : t -> int -> float -> unit` tel que `decrease f x p` diminue la priorité de `x` dans `f` à `p`. Cette fois-ci, ces fonctions préserve tous les invariants, notamment celui de tas min.

Pour `pop`, l'élément à retirer du tas est celui en position `0`. Pour le retirer, on peut l'échanger avec l'élément d'indice `t.taille - 1` (en dernier dans le tas) : la propriété de tas min n'est plus vérifiée puisque la nouvelle racine a éventuellement une priorité supérieure à l'un ou ses deux fils : il faut alors faire percoler ce sommet vers le bas jusqu'à ce que la propriété soit respectée.

▶ **Question 6** Écrire la fonction `percoler_bas : t -> int -> unit` percolant vers le bas le sommet `x` dans le tas `t`. Je vous conseille de commencer par un dessin pour déterminer les opérations à faire.

▶ **Question 7** En déduire la fonction `pop : t -> int * float` qui extraie le sommet de plus petite priorité du tas et le renvoie avec sa priorité.

▶ **Question 8** Implémenter l'algorithme de Dijkstra avec cette nouvelle implémentation des files de priorités. On vérifiera qu'elle donne les mêmes résultats que l'ancienne implémentation, (environ un facteur 10 chez moi sur le gros graphe, facteur qui serait bien pire sur des plus gros graphes).

Exercice 4 – Implémentation des files de priorités

On peut également implémenter les tas avec la structure de donnée d'arbre binaire de recherche, dont le parcours infixe est trié selon la priorité croissante.

▶ **Question 1** Implémenter les files de priorités par les ABR. On ne cherchera pas à l'équilibrer.

▶ **Question 2** Dans les ABR simples (sans équilibrage), quelle est la complexité au pire de l'algorithme de Dijkstra ?

▶ **Question 3** Et avec des ABR équilibrés (comme les arbres bicolores), quelle serait la complexité ?