Dijkstra

1 File de priorité

Pour implémenter l'algorithme de Dijkstra, nous allons utiliser la structure de file de priorité enrichie de l'opération DecreasePrio vue en cours. Pour simplifier, nous allons nous limiter au cas où :

— la capacité N de la file est fixée à la création ;
— les clés sont des entiers positifs entre 0 et $N-1$;
— les priorités sont des flottants.

Pour représenter cette structure, on utilisera le type suivant :

```
type t =
  {mutable last : int;
   priorities : float array;
   keys : int array;
   mapping : int array}
```

— Les trois tableaux `priorities`, `keys` et `mapping` sont de même taille N : la capacité de la file, fixée à la création.
— La partie « active » du tas est contenue dans la partie des tableaux `keys` et `priorities` située entre les indices `0` et `last` (inclus).
— On utilise, comme d'habitude, un tas binaire sous la forme d'un arbre binaire complet gauche stocké implicitement dans un (ou ici deux) tableaux, avec la racine à l'indice zéro.
— Le tableau `mapping` vérifie l'invariant suivant :
  — si la clé `i` n'est pas présente dans le tas, alors `mapping.(i) = -1` ;
  — sinon, `keys.(mapping.(i)) = i`, et la priorité correspondante est dans `priorities.(mapping.(i))`. Autrement dit, le tableau `mapping` permet de retrouver l'emplacement d'une clé dans le tas (ce qui est nécessaire lorsque l'on souhaite diminuer la priorité associée).

Pour une fois, nous allons définir un module pour regrouper les différentes fonctions agissant sur une file de priorité. La syntaxe est la suivante :

```
module PrioQ :
sig
  type t
  val get_min : t -> (int * float)
  ...
  val mem : t -> int -> bool
end = struct
  type t = {mutable last : int; ... }

  let get_min q = ...
  ...
  let mem q x = ...
end
```

Après cette définition, les types et fonctions déclarées dans la signature sont disponibles, en les préfixant avec le nom du module (`PrioQ.t` pour le type, `PrioQ.get_min`...). On peut définir des fonctions auxiliaires supplémentaires dans la partie `struct`, mais ces fonctions seront « privées » (inaccessibles depuis l'extérieur).

Remarque

⇨ Si vous ne vous souvenez pas très bien du fonctionnement de la structure de tas (enrichie ou non), n'hésitez pas à consulter le cours. La seule différence est qu'on utilise ici deux tableaux `keys` et `priorities` au lieu d'un seul tableau contenant des couples (clé, priorité). Il est cependant toujours profitable d'essayer de retrouver les algorithmes par vous-mêmes.

1. Redonner les formules permettant de retrouver les indices du fils gauche, du fils droit et du père de i dans un arbre binaire complet gauche implicite.
2. Écrire une fonction `full_swap` telle que `full_swap q i j` échange les positions dans le tas des clés `q.keys.(i)` et `q.keys.(j)`, en maintenant les invariants (il faudra donc aussi agir sur les tableaux `q.priorities` et `q.mapping`). On pourra supposer sans le vérifier que `i` et `j` sont des indices valides (compris entre 0 et `q.last`).

```
full_swap : t -> int -> int -> unit
```

3. Écrire la fonction `sift_up` effectuant la percolation vers le haut d'une clé du tas.

```
sift_up : t -> int -> unit
```

   À nouveau (et il en ira de même pour toutes les questions suivantes), on prendra garde à maintenir tous les invariants de la structure.

4. Écrire la fonction `insert` réalisant l'insertion d'une clé dans la file (avec une priorité associée). On pourra supposer sans le vérifier que la clé n'est pas déjà présente dans la file.

```
insert : t -> (int * float) -> unit
```

5. Écrire la fonction `sift_down` effectuant la percolation vers le bas d'une clé.

```
sift_down : t -> int -> unit
```

6. Écrire la fonction `extract_min` réalisant l'extraction du minimum.

```
extract_min : t -> int * float
```

7. Écrire la fonction `decrease_priority` qui diminue la priorité associée à une clé. On vérifiera que la clé est présente et que la nouvelle priorité est bien inférieure à l'ancienne (et lèvera une exception sinon).

```
decrease_priority : t -> int * float -> unit
```

8. Que faudrait-il modifier si l'on souhaitait pouvoir *augmenter* la priorité d'une clé existante ?
9. Déterminer la complexité des opérations `insert`, `extract_min`, `mem` et `decrease_prio`.

2 Algorithme de Dijkstra

Dans cette partie, on considère des graphes *a priori* orientés et pondérés par des flottants positifs ou nuls, représentés sous forme de tableaux de listes d'adjacence :

```
type weighted_graph = (int * float) list array
```

1. Écrire sous forme de pseudo-code l'algorithme de Dijkstra, puis comparer avec le cours.
2. Écrire une fonction `dijktra` prenant en entrée un graphe à n sommets (numérotés $0, \dots, n-1$) et un indice x_0 de sommet, et renvoyant un tableau `dist` de taille n telle que `dist.(j)` soit le poids d'un plus court chemin de x_0 à j.

   Si j n'est pas accessible depuis x_0, alors `dist.(j)` vaudra `infinity`.

```
dijkstra : weighted_graph -> int -> float array
```

3. Modifier la fonction `dijkstra` en une fonction `dijkstra_tree` qui renvoie également un tableau `tree` codant l'arbre de parcours associé. Autrement dit, on devra avoir (en notant `x0` le sommet initial passé en argument à la fonction) :
   — `tree.(i) = None` si `i` n'est pas accessible depuis `x0` ;
   — `tree.(x0) = Some x0` ;
   — `tree.(i) = Some j` si le plus court chemin trouvé par l'algorithme pour aller de `x0` à `i` se termine par l'arc $j \to i$.

```
dijkstra_tree : weighted_graph -> int -> (float array * int option array)
```

4. Écrire une fonction `reconstruct_path` prenant en entrée un tableau codant un arbre de parcours comme ci dessus et un indice de sommet `goal` et renvoyant un plus court chemin de `x0` à `goal` sous forme d'une liste de sommets.

— La fonction ne prend pas `x0` en entrée puisque ce n'est pas nécessaire : c'est le seul sommet qui est son propre père.
— On lèvera une exception s'il n'existe pas de chemin.

```
reconstruct_path : int option array -> int -> int list
```

3 Calcul d'itinéraire pour misanthropes

3.1 Graphe des communes de France

Les deux fichiers `communes.csv` et `adjacence.csv` contiennent des informations sur les communes françaises :
— `communes.csv` contient, pour chaque commune, un identifiant entier unique, le code INSEE (identifiant alphanumérique unique), le nom, le département et la population ;
— `adjacence.csv` contient la liste des communes immédiatement adjacentes (l'identifiant utilisé est l'identifiant entier unique du fichier `communes.csv`). Chaque paire de communes adjacentes n'est présente qu'une seule fois (s'il y a une ligne pour x, y, la ligne y, x n'est pas présente).

1. Déterminer la structure des fichiers. On pourra utiliser les commandes `head` et `tail` en ligne de commande (par défaut, elles affichent respectivement les dix premières et dix dernières lignes d'un fichier).

On définit le type suivant pour représenter une commune :

```
type commune =
  {id : int;
   insee : string;
   nom : string;
   pop : int;
   dep : string}
```

`insee` et `dep` sont de type `string` (et pas `int`) à cause des communes corses (départements 2A et 2B).

2. Créer à partir du fichier `communes.csv` un tableau `tab_communes` contenant à l'indice i la commune dont l'`id` vaut i.

```
lire_communes : string -> communes array
```

Pour lire une ligne, on pourra utiliser :
— `input_line` pour récupérer la ligne (sans le `\n` final) sous forme d'une chaîne de caractères ;
— `Scanf.sscanf` pour en extraire les données. Pour lire une chaîne de caractères jusqu'à la première occurrence d'un certain caractère, le code de format est `%s@;` (pour lire jusqu'au premier `;`, ce qui sera le cas ici). Le code suivant, par exemple, lit une chaîne de caractère et deux entiers, séparés par des virgules, et renvoie le couple constitué de la chaîne de caractères et de la somme des deux entiers :

```
Scanf.sscanf s "%s@,%d,%d" (fun s x y -> (s, x + y))
```

3. Créer à partir du fichier `adjacences.csv` le graphe (non orienté et non pondéré) défini par ces adjacences.

```
lire_graphe : int -> string -> int list array
```

L'argument entier est le nombre de communes, qui correspond par exemple à la taille du tableau de communes créé par le code fourni.

On suppose à présent que l'on dispose de deux variables globales `tab_communes` et `g_adj` correspondant respectivement au résultat de l'appel à la fonction `lire_communes` et `lire_graphe`.

3.2 Saute canton

Le jeu *Saute canton* consiste à partir d'une commune aléatoire et à passer de commune adjacente en commune adjacente en essayant d'arriver le plus rapidement possible à une commune d'au moins 50000 habitants.

1. Écrire une fonction `saute_canton` qui renvoie un chemin de longueur minimale reliant la commune passée en argument à une commune (quelconque) d'au moins 50000 habitants. Le chemin sera donné sous forme d'une liste d'identifiants de communes.

```
saute_canton : int -> int list
```

— On arrêtera le parcours dès que possible, en utilisant par exemple une exception (d'autres solutions sont possibles).
— Il n'existe pas toujours de chemin gagnant (certaines composantes connexes ne contiennent pas de communes de plus de 50000 habitants). On pourra renvoyer un chemin vide dans ce cas.

2. Déterminer la (ou une des) commune la plus « perdue » de France suivant le critère de saute canton (celle pour laquelle le chemin minimal vers une « grande » commune est le plus long possible).

3.3 Le saute canton du misanthrope

On s'intéresse désormais au cas d'un voyageur misanthrope : il souhaite voyager d'une commune A à une commune B (toutes deux fixées), mais tient absolument à rencontrer le moins de personnes possible en route. Autrement dit, il cherche un chemin minimisant la somme des populations des communes traversées.

1. Expliquer comment construire un graphe permettant de résoudre ce problème à l'aide de l'algorithme de Dijkstra.
2. Quel chemin conseillez-vous au misanthrope pour relier Villeurbanne à La Mulatière ? Montrouge à Aubervilliers ?

   Les chemins les plus courts sont respectivement Villeurbanne - Lyon - La Mulatière et Montrouge - Paris - Aubervilliers, mais notre ami misanthrope est prêt à de très longs détours !