TD XLII

AUTOUR DE 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.

▶ **Question 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.

▶ **Question 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
```

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

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

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

▶ **Question 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
```

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

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

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

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

▶ **Question 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
```

▶ **Question 8** Que faudrait-il modifier si l'on souhaitait pouvoir *augmenter* la priorité d'une clé existante ?

▶ **Question 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
```

▶ **Question 10** Écrire sous forme de pseudo-code l'algorithme de Dijkstra, puis comparer avec le cours.

▶ **Question 11** 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.

Remarque

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

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

▶ **Question 12** 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)
```

▶ **Question 13** 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.

Remarques

- 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).

▸ **Question 14** 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}
```

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

▸ **Question 15** 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))
  ```

▸ **Question 16** 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
```

Remarque
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 50 000 habitants.

▸ **Question 17** 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 50 000 habitants. Le chemin sera donné sous forme d'une liste d'identifiants de communes.

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

Remarques

- 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 50 000 habitants). On pourra renvoyer un chemin vide dans ce cas.

► **Question 18** 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.

► **Question 19** Expliquer comment construire un graphe permettant de résoudre ce problème à l'aide de l'algorithme de Dijkstra.

► **Question 20** Quel chemin conseillez-vous au misanthrope pour relier Villeurbanne à La Mulatière ? Montrouge à Aubervilliers ?

Remarque

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 !

Solutions

▶ **Question 1** Les enfants sont en $2i+1$ et $2i+2$ (sous réserve d'existence), le parent en $\lfloor (i-1)/2 \rfloor$.

▶ **Question 2** On suppose qu'une fonction `swap` est préalablement définie (c'est le cas dans le fichier fourni).

```
let full_swap q i j =
  swap q.keys i j;
  swap q.priorities i j;
  swap q.mapping q.keys.(i) q.keys.(j)
```

Pour améliorer la lisibilité, on définit :

```
let left i = 2 * i + 1
let right i = 2 * i + 2
let parent i = (i - 1) / 2
```

▶ **Question 3**

```
let rec sift_up q i =
    let j = parent i in
    if i > 0 && q.priorities.(i) < q.priorities.(j) then begin
      full_swap q i j;
      sift_up q j
    end
```

▶ **Question 4**

```
let insert q (x, prio) =
    if length q = capacity q then failwith "insert"
    else begin
      let l = q.last + 1 in
      q.keys.(l) <- x;
      q.priorities.(l) <- prio;
      q.mapping.(x) <- l;
      q.last <- l;
      sift_up q q.last
    end
```

▶ **Question 5**

```
let rec sift_down q i =
    let prio = q.priorities in
    let smallest = ref i in
    if left i <= q.last && prio.(left i) < prio.(i) then
      smallest := left i;
    if right i <= q.last && prio.(right i) < prio.(!smallest) then
      smallest := right i;
    if !smallest <> i then begin
      full_swap q i !smallest;
      sift_down q !smallest
    end
```

▶ **Question 6** Ici, il ne suffit pas de faire un `full_swap` : il faut aussi penser à marquer la clé extraite comme absente.

```
let extract_min q =
    if q.last < 0 then
      failwith "extract_min"
    else
      begin
        let key = q.keys.(0) in
        let prio = q.priorities.(0) in
        full_swap q 0 q.last;
        q.mapping.(key) <- -1;
        q.last <- q.last - 1;
        sift_down q 0;
        key, prio
      end
```

▶ **Question 7**

```
let decrease_priority q (x, prio) =
    let i = q.mapping.(x) in
    assert (mem q x && prio <= q.priorities.(i));
    q.priorities.(i) <- prio;
    sift_up q i
```

▶ **Question 8** Si l'on veut augmenter une priorité, il faut faire percoler la clé associée vers le bas. Ce n'est pas un problème, on pourrait facilement définir une fonction `update_priority` qui effectuerait l'une ou l'autre percolation suivant les cas.

▶ **Question 9** La fonction `mem` est, de manière évidente, en temps constant. Les trois autres fonctions effectuent une percolation (vers le haut ou vers le bas) et quelques opérations en temps constant. Les percolations sont clairement en temps $O(h)$, et dans le cas d'un arbre complet gauche on a $h = O(\log n)$. Par conséquent, les complexités sont en $O(\log n)$.

▶ **Question 10** C'est dans le cours...

▶ **Question 11** C'est une traduction vraiment directe du pseudo-code. On a juste remplacé le test $\texttt{dist}[k] \neq \infty$ par un `PrioQ.mem`, mais cela n'a pas d'importance.

```
let dijkstra g i =
  let n = Array.length g in
  let dist = Array.make n infinity in
  let queue = PrioQ.make_empty n in
  PrioQ.insert queue (i, 0.);
  dist.(i) <- 0.;
  while PrioQ.length queue <> 0 do
    let (j, d) = PrioQ.extract_min queue in
    let update (k, x) =
      let new_d = d +. x in
      if new_d < dist.(k) then begin
          dist.(k) <- new_d;
          if PrioQ.mem queue k then PrioQ.decrease_priority queue (k, new_d)
          else PrioQ.insert queue (k, new_d)
        end in
    List.iter update g.(j)
  done;
  dist
```

▶ **Question 12** À nouveau, c'est essentiellement du cours.

```
let dijkstra_tree g i =
  let n = Array.length g in
  let dist = Array.make n infinity in
  let p = Array.make n None in
  let queue = PrioQ.make_empty n in
  PrioQ.insert queue (i, 0.);
  dist.(i) <- 0.;
  p.(i) <- Some i;
  while PrioQ.length queue <> 0 do
    let (j, d) = PrioQ.extract_min queue in
    let update (k, x) =
      let new_d = d +. x in
      if new_d < dist.(k) then begin
          dist.(k) <- new_d;
          p.(k) <- Some j;
          if PrioQ.mem queue k then PrioQ.decrease_priority queue (k, new_d)
          else PrioQ.insert queue (k, new_d)
        end in
    List.iter update g.(j)
  done;
  dist, p
```

▶ **Question 13** C'est typiquement une question qui ne pose aucun problème sur machine mais sur laquelle il est facile de faire une erreur sur papier (chemin dans le mauvais ordre, extrémité manquante ou présente deux fois). Il faut s'entraîner à détecter ce type de problème en faisant les tests « de tête ».

```
let reconstruct_path p goal =
  let rec aux current =
    match p.(current) with
    | None -> failwith "no path"
    | Some i when i = current -> [current]
    | Some i -> current :: aux i in
  List.rev (aux goal)
```

▶ **Question 14** Pour le fichier `communes.csv`, quelques commandes permettent d'obtenir l'ordre des colonnes et le délimiteur, et indiquent que les `id` correspondent aux numéros de ligne (en commençant à zéro) :

```
$ head --lines 4 communes.csv
id;insee;nom;departement;population
0;01001;L'Abergement-Clémenciat;1;784
1;01002;L'Abergement-de-Varey;1;221
2;01004;Ambérieu-en-Bugey;1;13835
$ tail --lines 3 communes.csv
35843;2B364;Zuani;2B;35
35844;2B365;San-Gavino-di-Fiumorbo;2B;174
35845;2B366;Chisa;2B;100
$ wc --lines communes.csv
35847 communes.csv
```

▶ **Question 15** Plusieurs solutions sont possibles (faire une passe pour compter le nombre de lignes, créer un tableau ayant la bonne taille et le remplir dans un deuxième temps. . .). Ici, on a choisi d'utiliser une liste et de tirer parti du fait que les communes apparaissent dans l'ordre de leur `id` dans le fichier.

```
let lire_communes nom_fichier =
  let ic = open_in nom_fichier in
  let liste = ref [] in
  try
    (* On saute la première ligne (entête). *)
    let _ = input_line ic in
    while true do
      let s = input_line ic in
      let f id insee nom dep pop =
        {id; insee; nom; dep; pop} in
      let c = Scanf.sscanf s "%d;%s@;%s@;%s@;%d" f in
      liste := c :: !liste
    done;
    assert false
  with
  | End_of_file -> close_in ic; Array.of_list (List.rev !liste)
```

Remarque

On écrirait normalement `{id = id; insee = insee; ...}`, mais dans le cas où les noms de variables correspondent exactement aux noms des champs de l'enregistrement OCaml accepte la version plus courte. Ce n'est pas à retenir.

▶ **Question 16** Il faut simplement penser à ajouter l'arête aux deux listes d'adjacence, le graphe n'étant pas orienté.

```
let lire_graphe nb_communes fichier_adjacence =
  let ic = open_in fichier_adjacence in
  let g = Array.make nb_communes [] in
  try
    let _ = input_line ic in
    while true do
      let s = input_line ic in
      let x, y = Scanf.sscanf s "%d;%d" (fun x y -> (x, y)) in
      g.(x) <- y :: g.(x);
      g.(y) <- x :: g.(y)
    done;
    assert false
  with
  | End_of_file ->
    close_in ic;
    g
```

▶ **Question 17** On pourrait utiliser Dijkstra avec un graphe non pondéré (en mettant un poids unitaire à toutes les arêtes), mais c'est assez nettement moins efficace qu'un simple parcours en largeur (il y a un facteur $\log |V|$, et $|V|$ est de l'ordre de 200 000 ici). On fait donc un parcours en largeur, avec une exception pour s'arrêter dès qu'on trouve une commune acceptable.

```
exception Gagne of int

let saute_canton g init tab_communes =
  let f = Queue.create () in
  let n = Array.length g in
  let vus = Array.make n false in
  let arbre = Array.make n None in
  arbre.(init) <- Some init;
  Queue.add init f;
  vus.(init) <- true;
  try
    while not (Queue.is_empty f) do
      let i = Queue.pop f in
      if tab_communes.(i).pop >= 50_000 then raise (Gagne i);
      let rec ajoute j =
        if not vus.(j) then begin
          vus.(j) <- true;
          Queue.add j f;
          arbre.(j) <- Some i
        end in
      List.iter ajoute g.(i)
    done;
    []
  with
  | Gagne i -> reconstruct_path arbre i
```

► Question 18

```
let commune_perdue g tab_communes =
  let dmax = ref (-1) in
  let chemin_max = ref [] in
  let n = Array.length tab_communes in
  for i = 0 to n - 1 do
    let chemin = saute_canton g i tab_communes in
    if List.length chemin > !dmax then begin
      dmax := List.length chemin;
      chemin_max := chemin
    end
  done;
  affiche tab_communes !chemin_max
```

On obtient un chemin de longueur 32 :

```
# commune_perdue g_adj tab_communes;;
Auderville (50) : 267
Jobourg (50) : 501
Omonville-la-Petite (50) : 137
Digulleville (50) : 291
Omonville-la-Rogue (50) : 524
...
Saint-Manvieu-Norrey (14) : 1729
Carpiquet (14) : 2374
Caen (14) : 108954
```

Remarque

Ce résultat n'est malheureusement pas à jour, la commune de Cherbourg-Octeville (par laquelle passe ce chemin) ayant récemment fusionné avec ses voisines pour devenir Cherbourg-en-Cotentin et dépasser ainsi la barre des 50 000 habitants. L'histoire ne dit pas comment les habitants d'Auderville ont réagi à la perte de leur titre de gloire.

▶ **Question 19** On construit un graphe orienté pondéré de la manière suivante : l'arc (x, y) a pour poids la population de la commune y. Le problème se ramène alors clairement à une recherche de chemin de poids minimal dans ce graphe, qui vérifie les conditions d'application de l'algorithme de Dijksta (poids positifs).

▶ **Question 20** Faisons les choses bien, en créant un dictionnaire ayant pour clés les couples (nom, département) et pour valeurs les communes correspondantes (l'utilisation du département permet de s'affranchir des homonymes) :

```
let cree_dictionnaire_communes () =
  let dict = Hashtbl.create 30_000 in
  let ajoute c =
    if Hashtbl.mem dict (c.nom, c.dep) then failwith "couples non uniques ?";
    Hashtbl.add dict (c.nom, c.dep) c in
  Array.iter ajoute tab_communes;
  dict

let dict_communes = cree_dictionnaire_communes ()
```

On crée ensuite le graphe pondéré décrit à la question suivante :

```
let cree_graphe_pondere () =
  let n = Array.length tab_communes in
  let g = Array.make n [] in
  let traite_arc i j =
    g.(i) <- (j, float tab_communes.(j).pop) :: g.(i) in
  for i = 0 to n - 1 do
    List.iter (traite_arc i) g_adj.(i)
  done;
  g

let g_pond = cree_graphe_pondere ()
```

La fonction est alors immédiate à écrire :

```
let misanthrope init but =
  let c_init = Hashtbl.find dict_communes init in
  let c_but = Hashtbl.find dict_communes but in
  let _, arbre = dijkstra_tree g_pond c_init.id in
  reconstruct_path arbre c_but.id
```

On obtient :

```
# affiche (misanthrope ("Villeurbanne", "69") ("La Mulatière", "69"));;
Villeurbanne (69) : 145150
Rillieux-la-Pape (69) : 29952
Cailloux-sur-Fontaines (69) : 2480
Mionnay (1) : 2077
Civrieux (1) : 1362
Parcieux (1) : 1095
Quincieux (69) : 3002
Lucenay (69) : 1752
Marcy (69) : 634
Charnay (69) : 1076
Bagnols (69) : 695
Le Breuil (69) : 454
Bully (69) : 2069
Savigny (69) : 1938
Chevinay (69) : 543
Courzieu (69) : 1161
Yzeron (69) : 1036
Thurins (69) : 2917
Soucieu-en-Jarrest (69) : 3769
Chaponost (69) : 7978
Sainte-Foy-lès-Lyon (69) : 21742
La Mulatière (69) : 6480
```

Pour Montrouge-Aubervilliers, on va se contenter de la longueur :

```
# List.length (misanthrope ("Montrouge", "92") ("Aubervilliers", "93"));;
- : int = 67
```