Algorithme de Dijkstra — corrigé

Jean-Baptiste Rouquier

1 graphes

Question 1.1. Un graphe orienté $(V, \mathcal{R})$ est un ensemble V et une relation $\mathcal{R}$ sur V, telle que pour aucun x de V on n'ait $x\mathcal{R}x$. Il est non orienté si $\mathcal{R}$ est symétrique, et on considère alors que $\mathcal{R}$ contient la paire $\{x, y\}$ plutôt que les deux couples (x, y) et (y, x).

2 L'algorithme de Dijkstra

Question 2.1. On stocke les statuts des sommets dans un tableau dont la case i contient le statut du sommet i.

Question 2.2. Avec une structure de données fonctionnelle, on aurait un type différent :

```
change_group : groups -> sommet -> status -> groups
```

Question 2.3. S'il existait un chemin de s à v, noté p, plus court que $c(v)$, il passerait (par définition de c) par un sommet non examiné autre que v. Considérons le premier sommet u de p non examiné, on a u différent de v et u dans `current`. Le sous-chemin de p allant de s à u est strictement plus court que p, lui-même plus court que $c(v)$. Contradiction avec le choix de v minimisant $c(v)$.

Question 2.4. On utilise une liste de couples (sommet, longueur). Il suffit de parcourir la liste pour trouver l'élément minimal. La fonction proposée est récursive terminale. Une fois l'élément trouvé, on renvoie une nouvelle liste ne contenant plus cet élément.

En utilisant le TP sur les tas (Heapsort), on a une structure de données plus efficace, qui permet d'écrire `take_minimum` et `add` avec une complexité $O(\log n)$ où n est le nombre d'éléments de la structure.

Question 2.5. Pour mettre à jour `current` il suffit de lui ajouter les voisins de v inconnus. Pour mettre à jour c, il n'y a à considérer que les chemins passant par v. De plus le plus court chemin de s à un sommet examiné est connu donc il n'y a à considérer que les chemins dont l'avant dernier sommet est v. Pour tous les voisins u de v, on compare $c(u)$ et $c(v) + 1$, on stocke le minimum dans $c(u)$.

Question 2.6. On s'arrête lorsque le sommet v choisi est t (on a alors trouvé le plus court chemin), ou bien lorsque `current` est vide, ce qui signifie qu'il n'existe pas de chemin de s à t.

Question 2.9. Le nombre de sommets examinés augment d'un à chaque itération donc l'algorithme termine. On a montré que $c(v)$ est la longueur du plus court cheminde s à v pour tout sommet v examiné, en particulier $c(t)$ est la valeur cherchée lorsque l'algorithme s'arrête.

Question 2.10. L'algorithme marche sans modification sur les graphes contenant des cycles.

Question 2.11 (bonus). On stocke dans un tableau le sommet qui a permis d'arriver à v par le plus court chemin. Il suffit alors de partir de t pour reconstituer la solution du problème.

Question 2.12 (bonus). Il n'y a pas besoin d'un algorithme aussi sophistiqué pour calculer le plus court chemin, il est utile lorsque chaque arc a un poids et qu'il faut calculer le chemin de poids minimum.

Les poids des sommets peuvent être stockés dans la représentation matricielle du graphe : au lieu d'un `bool array array` dont la case (i, j) contient **true** si et seulement si il y a un arc de i à j, on utilise un `int array array` dont la case (i, j) contient le poids de l'arc (i, j). Il n'y a qu'un endroit à modifier : au lieu de $c(v) + 1$ on calcule $c(v) + w(vu)$.

Question 2.13 (bonus). L'algorithme ne marche pas s'il y a des poids négatifs : le résultat de la question 2.3 est alors faux. On peut construire un contre-exemple montrant que l'algorithme donne alors un résultat erroné.

Pire, s'il existe un cycle de poids négatif, chaque parcours de ce cycle fait diminuer le poids et il n'y alors pas de circuit de poids minimum (il suffit de faire un tour de plus pour diminuer le poids).

Le code source proposé utilise un certain nombre de subtilités de caml, n'hésitez pas à me poser des questions sur les points que vous n'auriez pas compris. Il implémente la recherche de v minimisant $c(v)$ par des listes (au lieu d'un tas, non exigible) et calcule le plus court chemin (et non celui de poids minimum comme demandé dans une question bonus).

Voici l'algorithme complet :

```
foreach v ∈ V do c(v) := +∞
current := {s}
c(s) := 0
examined :=∅
while current ≠ ∅ do
    v := élément de current minimisant c(v)
    examined ∪= {v}
    current ⧵= {v}
    foreach u ∈ Γ⁺(v) do
        if not u ∈ examined then current ∪= {u}
        if u ∈ current et c(v) + coût(v, u) < c(u) then
            c(u) := c(v) + coût(v, u)
            précédent(u) := v
```

Algorithme 1: algorithme de Dijkstra : calcul du chemin de poids minimum

```
type sommet = int
type graph_matrix = bool array array
type graph_list = sommet list array

let example1_list = [|[1;2]; [3;4]; [3]; [5]; []; [6]; []; |]
let example2_list = [|[1]; [2;4]; [0;3]; [5]; [3]; [6]; [] |]
let example1_matrix =
  [|[|false; true ; true ; false; false; false; false|];
    [|false; false; false; true ; true ; false; false|];
    [|false; false; false; true ; false; false; false|];
    [|false; false; false; false; false; true ; false|];
    [|false; false; false; false; false; false; false|];
    [|false; false; false; false; false; false; true |];
    [|false; false; false; false; false; false; false|];  |]
let example2_matrix =
  [|[|false; true ; false; false; false; false; false|];
    [|false; false; true ; false; true ; false; false|];
    [|true ; false; false; true ; false; false; false|];
    [|false; false; false; false; false; true ; false|];
    [|false; false; false; true ; false; false; false|];
    [|false; false; false; false; false; false; true |];
    [|false; false; false; false; false; false; false|];  |]

type graph = graph_list
type status = Examined | Current | Unknown
type groups = status array
let initialize graph = Array.make (Array.length graph) Unknown
let get_group = Array.get
let change_group = Array.set

type longueur = int
type current = (sommet * longueur) list
let empty = []
let take_minimum current =
  let rec tm_tail_rec ((sommet, longueur_min) as accu) = function
    | (sommet', longueur) :: tail ->
        tm_tail_rec
          (if longueur < longueur_min then sommet',longueur else accu)
          tail
    | [] -> accu in
  match current with
    | [] -> raise Not_found
    | head :: tail ->
        let sommet, _ = tm_tail_rec head tail in
        sommet, List.remove_assoc sommet current
let add sommet longueur current = (sommet,longueur) :: current


exception Found of int option

let dijkstra graph source target =
  let groups = initialize graph in
  let current = ref empty in
  let c = Array.make (Array.length graph) None in
  c.(source) <- Some 0;
  let update_c v u =
    let succ_cv = match c.(v) with Some l -> succ l | None -> failwith "bug1" in
    match c.(u) with
      | None -> c.(u) <- Some succ_cv
      | Some cu -> c.(u) <- Some (min cu succ_cv) in
  let mise_a_jour sommet =
    change_group groups sommet Examined;
    List.iter
      (fun voisin ->
         match get_group groups voisin with
           | Examined -> ()
           | Current -> update_c sommet voisin
           | Unknown ->
               change_group groups voisin Current;
               update_c sommet voisin;
               current := add voisin c.(voisin) !current)
      graph.(sommet) in
  mise_a_jour source;
  try
    while true do
      let v,new_current = take_minimum !current in
      if v = target
      then raise (Found c.(v))
      else (
        current := new_current;
        mise_a_jour v;)
    done; failwith "bug2"
  with
    | Found (Some longueur) -> longueur
    | Not_found -> failwith "il n'y a pas de chemin de s à t";;
```