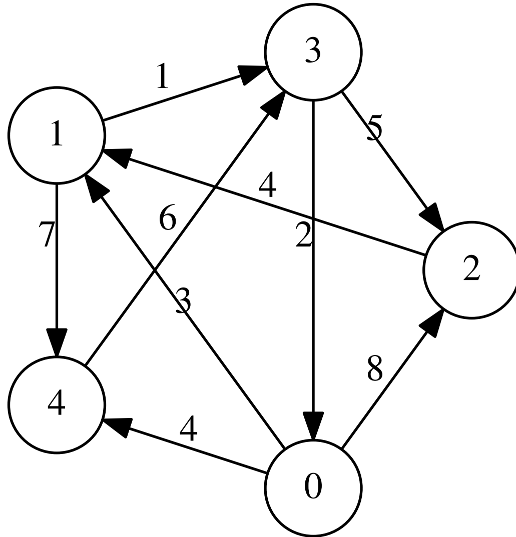
TP 10 : Algorithmes : Dijsktra, Floyd-Warshall

—



1 Outils mathématiques

Pour pouvoir appliquer l'algorithme de Floyd directement, nous aurons besoin de travailler dans $\mathbb{R} \cup \{\infty\}$

1. Définir un type correspondant à $\bar{\mathbb{R}} = \mathbb{R} \cup \{\infty\}$
2. Définir les opérateurs suivants pour manipuler ce type : calculer le minimum de deux valeurs, leur somme, et les comparer.

2 Floyd-Warshall

1. Définir le type des matrices d'adjacence et écrire la matrice d'adjacence correspondant à l'exemple ci-dessus.
2. Écrire une fonction qui permet de définir une matrice carrée vide de taille $n \times n$
3. Écrire une fonction qui permet de recopier une matrice dans une autre.
4. Écrire l'aglorithme de Floyd-Warshall. On fera attention à ne pas faire n'importe quoi avec les matrices et les pointeurs : il faudra a priori recopier des matrices plusieurs fois.

3 Dijkstra

1. Définir le type des listes d'adjacence et écrire un exemple de graphe (avec des poids positifs biensûr).
2. Définir une file de priorité : dans un premier temps, utiliser une liste que l'on reparcourera à chaque fois (Pas efficace).
3. Écrire l'algorithme de Dijkstra. On pourra le voir comme un parcours de graphe. . .

4. (Bonus) On pourra utiliser des tas binaires (cf TP 02), ou des tas binomiaux (cf "Lettre de Caml numéro 5", de Laurent Cheno)
5. Écrire des fonctions qui permettent de transformer une matrice d'adjacence en listes d'adjacences, et vice-versa.

4 Floyd-Warshall et Automates

On peut utiliser une variante de l'algorithme de Floyd-Warshall pour construire, à partir d'un automate, une expression régulière correspondant au même langage :

Étant donné un automate à n états (numérotés de 0 à $n-1$), on va construire une matrice $n \times n$ contenant des expressions régulières. L'élément en position (i, j) correspondra à la fin de l'algorithme à l'expression régulière décrivant tous les mots m tels que $\delta(i, m) = j$. Il suffira que considérer tous les couples (i, f) avec i un état initial et f un état final.

Initialisation : pour tout couple (i, j), on initialisera à $D_{i,j}^{(0)} = E_{i,j}$ si $i \neq j$ et $D_{i,j}^{(0)} = \epsilon + E_{i,i}$ sinon. De quelle forme sont les $E_{i,j}$ au départ ?

Comme dans l'algorithme de Floyd-Warshall, on va faire n itérations :

$$D_{i,j}^{(k+1)} = D_{i,j}^{(k)} + D_{i,k}^{(k)}.(D_{k,k}^{(k)})^*.D_{k,j}^{(k)}$$

1. Définir un type correspondant aux expressions rationnelles
2. On pourra aussi définir une fonction `let rec simplifier e = e` permettant de simplifier une expression rationnelle, à compléter au fur et à mesure.
3. Définir des expressions rationnelles simpels, et un automate d'exemple simple.
4. Ecrire l'algorithme présenté ci-dessus :

- Prendre en entrée un automate
- Construire (et initialiser) une matrice contenant des expressions rationnelles
- Appliquer la variante de l'algorithme de Floyd sur cette matrice
- Déterminer une expression rationnelle correspondant au langage reconnu par cet automate.

5 Source

On pourra retrouver des versions Caml Light des algorithmes de Dijkstra et Floyd-Warshall dans `Une boîte d'outils pour la programmation en Caml` de Laurent Cheno.

```
type poids = Val of float | Inf;;

let mini a b =
  match a, b with
  | Val va, Val vb -> Val (min va vb)
  | Inf, _ -> b
  | _, Inf -> a
;;

let plus a b =
  match a, b with
  | Val va, Val vb -> Val (va +. vb)
  | _, _ -> Inf
;;

let inferieur a b = (mini a b) = a ;;

type matrice_adjacence = poids array array;;

let graphe_mat:matrice_adjacence = [|   (* Matrice d'adjacence *)
   [|Inf   ; Val 3.; Val 8.   ; Inf   ; Val 4.   |];
   [|Inf   ; Inf   ; Inf      ; Val 1.; Val 7.   |];
   [|Inf   ; Val 4.; Inf      ; Inf   ; Inf      |];
   [|Val 2.; Inf   ; Val 5.   ; Inf   ; Inf      |];
   [|Inf   ; Inf   ; Inf      ; Val 6.; Inf      |];
 |]
;; (* Convention : on pourrait avoir des 0 sur la diagonale... *)

let matrice_vide n =
  Array.create_matrix n n (Val 0.)
;;

let recopier m_source m_cible =
  let n = Array.length m_source in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
        m_cible.(i).(j) <- m_source.(i).(j)
    done
  done
;;

let floyd (graphe:matrice_adjacence) =
  let n = Array.length graphe in
  let d0 = matrice_vide n in
  let d1 = matrice_vide n in
  recopier graphe d0;
  for k = 0 to n-1 do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        d1.(i).(j) <- mini d0.(i).(j) (plus d0.(i).(k) d0.(k).(j))
      done
    done;
    recopier d1 d0 (* On pourrait s'en passer *)
  done;
  (d1:matrice_adjacence)
;;

floyd graphe_mat;;

type noeud = int
type liste_adjacence = (noeud * poids) list array

let graphe_liste:liste_adjacence = [|    (* Listes d'adjacence *)
    [(1, Val 3.); (2, Val 8.); (4, Val 4.)];
    [(3, Val 1.); (4, Val 7.)];
    [(1, Val 4.)];
    [(0, Val 2.); (2, Val 5.)];
    [(3, Val 6.)]
  |]
;;

let rec file_prio_ajoute file element =
  match file, element with
  | [], e -> [e]
  | x::t, e -> if fst x = fst e
               then (fst x, mini (snd x) (snd e))::t
               else x::(file_prio_ajoute t e)
;;

let rec file_prio_maj file liste_elements dist =
  match liste_elements with
  | [] -> file
  | x::t -> file_prio_maj (file_prio_ajoute file (fst x, plus (snd x) dist)) t
dist
;;

let file_prio_getmin file =
  let rec getmin_aux file reste noeud_min poids_min =
    match file with
    | x::t -> if inferieur (snd x) poids_min
              then if poids_min = Inf
                   then getmin_aux t reste (fst x) (snd x)
                   else getmin_aux t ((noeud_min, poids_min)::reste) (fst x) (snd
x)
              else getmin_aux t (x::reste) noeud_min poids_min
    | [] -> (reste, noeud_min, poids_min)
  in getmin_aux file [] (-1) Inf
;;

let dijkstra (graphe:liste_adjacence) source =
  let n = Array.length graphe in
  let vus = Array.create n false in
  let plus_court = Array.create n Inf in
  let rec dijkstra_aux vus atraiter =
    match atraiter with
    | [] -> ()
    | _ -> let (reste_atraiter, noeud_min, poids_min) = file_prio_getmin atraiter
in
           if vus.(noeud_min) then dijkstra_aux vus reste_atraiter else
             begin
               vus.(noeud_min) <- true;
               plus_court.(noeud_min) <- poids_min;
               dijkstra_aux vus (file_prio_maj reste_atraiter graphe.(noeud_min)
poids_min)
             end
  in
  dijkstra_aux vus [(source, Val 0.)];
  plus_court
;;

dijkstra graphe_liste 1;;

(* Avec des automates ... réciproque du Théorème de Kleene *)

type expr = Vide | Epsilon | Id of string | Etoile of expr | Plus of expr * expr
| Concat of expr * expr;;

(* val simplifier : expr -> expr = <fun> *)
let rec simplifier e =
  match e with
  | Concat (a, b) ->
     let sa = simplifier a and sb = simplifier b in
     (match sa, sb with
      | Vide, _ | _, Vide -> Vide
      | Epsilon, _ -> sb
      | _, Epsilon -> sa
      | _, _ -> Concat (sa, sb)
     )
  | Etoile a ->
```

```ocaml
      let sa = simplifier a in
      (match sa with
       | Vide -> Vide
       | Epsilon -> Epsilon
       | Etoile (b) -> Etoile b
       | Plus (Epsilon, c) -> Etoile c
       | _ -> Etoile sa
      )
  | Plus (a, b) ->
      let sa = simplifier a and sb = simplifier b in
      (match sa, sb with
       | Vide, _ -> sb
       | _, Vide -> sa
       | Etoile et, Epsilon -> sa
       | Epsilon, Etoile et -> sb
       | Epsilon, Epsilon -> Epsilon
       | _, Epsilon -> Plus (sb, Epsilon)
       | _, _ -> Plus (sa, sb)
      )
  | _ -> e
;;

let e1 = Etoile (Concat (Id "a", Epsilon));;
let e2 = Plus (Etoile (Concat (Id "a_1", Etoile (Concat (Id "a_2", Id "b_3")))),
Etoile (Concat (Id "b_4", Id "a_5")));;

type transition = (int * string * int);;
type automate = int * int list * transition list;;

(* val nb_etats : transition list -> int = <fun> *)
let nb_etats (transitions: transition list )=
  let rec nb_aux tr n =
    match tr with
    | [] -> n + 1
    | (s1, _, s2)::tl -> nb_aux tl (max n (max s1 s2))
  in
  nb_aux transitions 0
;;

(* val recopier : 'a array array -> 'a array array -> unit = <fun> *)
let recopier m_source m_cible =
  let n = Array.length m_source in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      m_cible.(i).(j) <- m_source.(i).(j)
    done
  done
;;

(* val floyd_automate : expr array array -> expr array array = <fun> *)
let floyd_automate matrice =
  let n = Array.length matrice in
  let d0 = Array.make_matrix n n Vide in
  let d1 = Array.make_matrix n n Vide in
  recopier matrice d0;
  for k = 0 to n-1 do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        d1.(i).(j) <- simplifier (Plus (d0.(i).(j), Concat (d0.(i).(k), Concat
(Etoile (d0.(k).(k)), d0.(k).(j)))))
      done
    done;
    recopier d1 d0
  done;
  d1
;;

(* val automate_vers_expression : automate -> expr = <fun> *)
let automate_vers_expression (automate: automate) =
  let initial, final, transitions = automate in
  let n = nb_etats transitions in
  let d = Array.make_matrix n n Vide in
  let rec init trans mat =
    match trans with
    | (source, lettre, cible)::tl ->
      let maj = if source = cible
                then Plus (Epsilon, Id lettre)
                else Id lettre in
      begin
        mat.(source).(cible) <- simplifier (Plus(mat.(source).(cible), maj));
        init tl mat
      end
    | [] -> ()
  in
  let rec make_expr initial finaux matrice expression =
    match finaux with
    | [] -> expression
    | i::tl -> make_expr initial tl matrice (Plus (expression, matrice.(initial).
(i)))
  in
  init transitions d;
  let m = floyd_automate d in
  simplifier (make_expr initial final m Vide)
;;

let (a1: automate) = (0, [2], [(0, "a", 1); (1, "b", 2); (1, "c", 1)]);;

automate_vers_expression a1;;

(*
#   - : expr = Concat (Id u"a", Concat (Etoile (Id u"c"), Id u"b"))
*)
```