Fonctionnalité et Modularité

Cours 8

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-fm

Plan

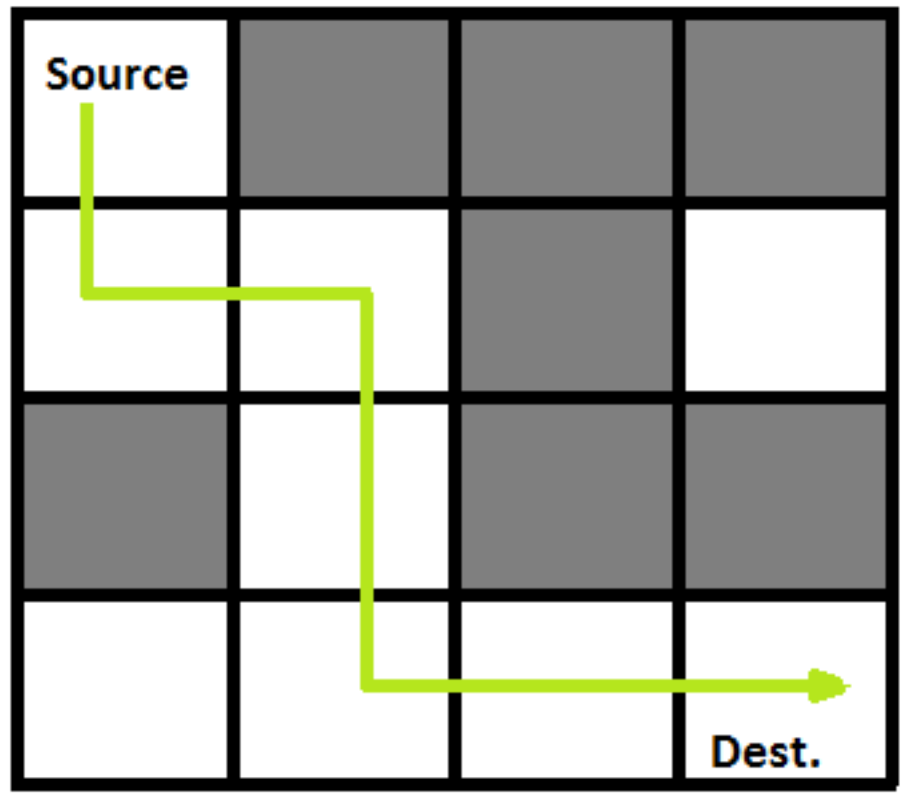
- graphes
- représentations
- parcours en profondeur d'abord
- sortie de labyrinthe
- parcours en largeur d'abord
- plus court chemin

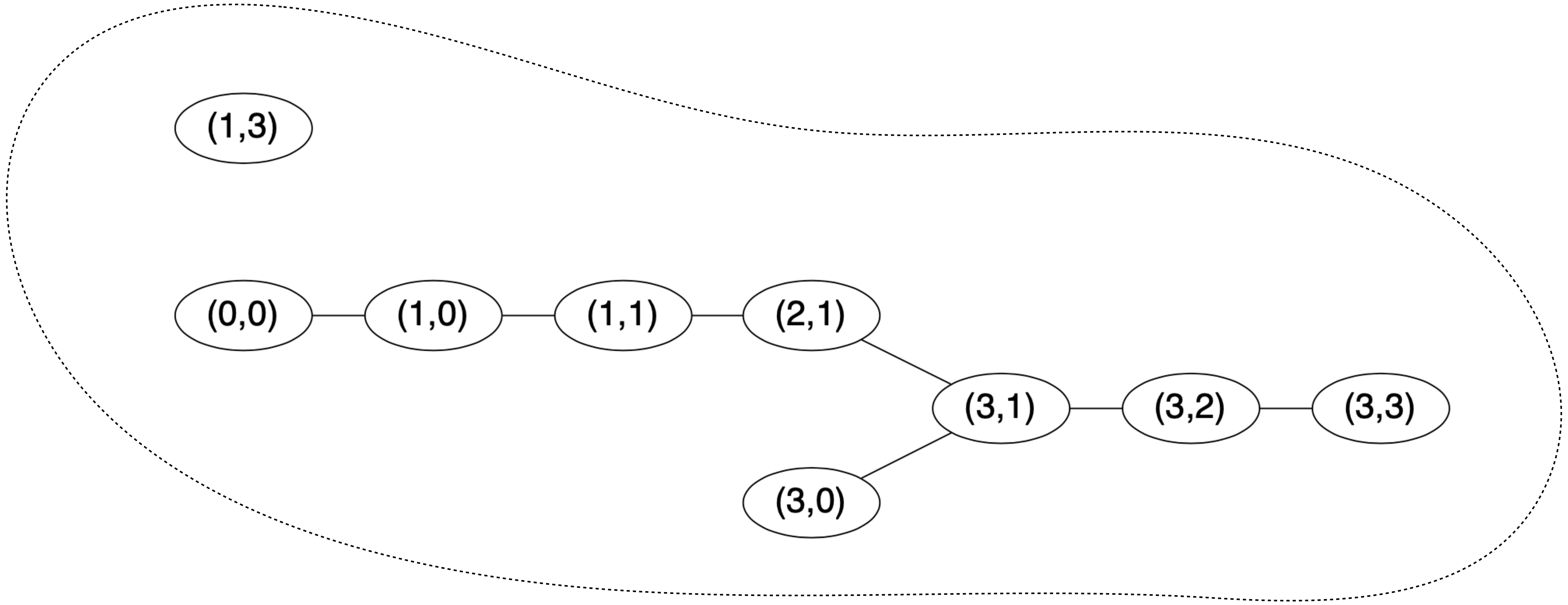
télécharger Ocaml en `http://www.ocaml.org`

Graphes

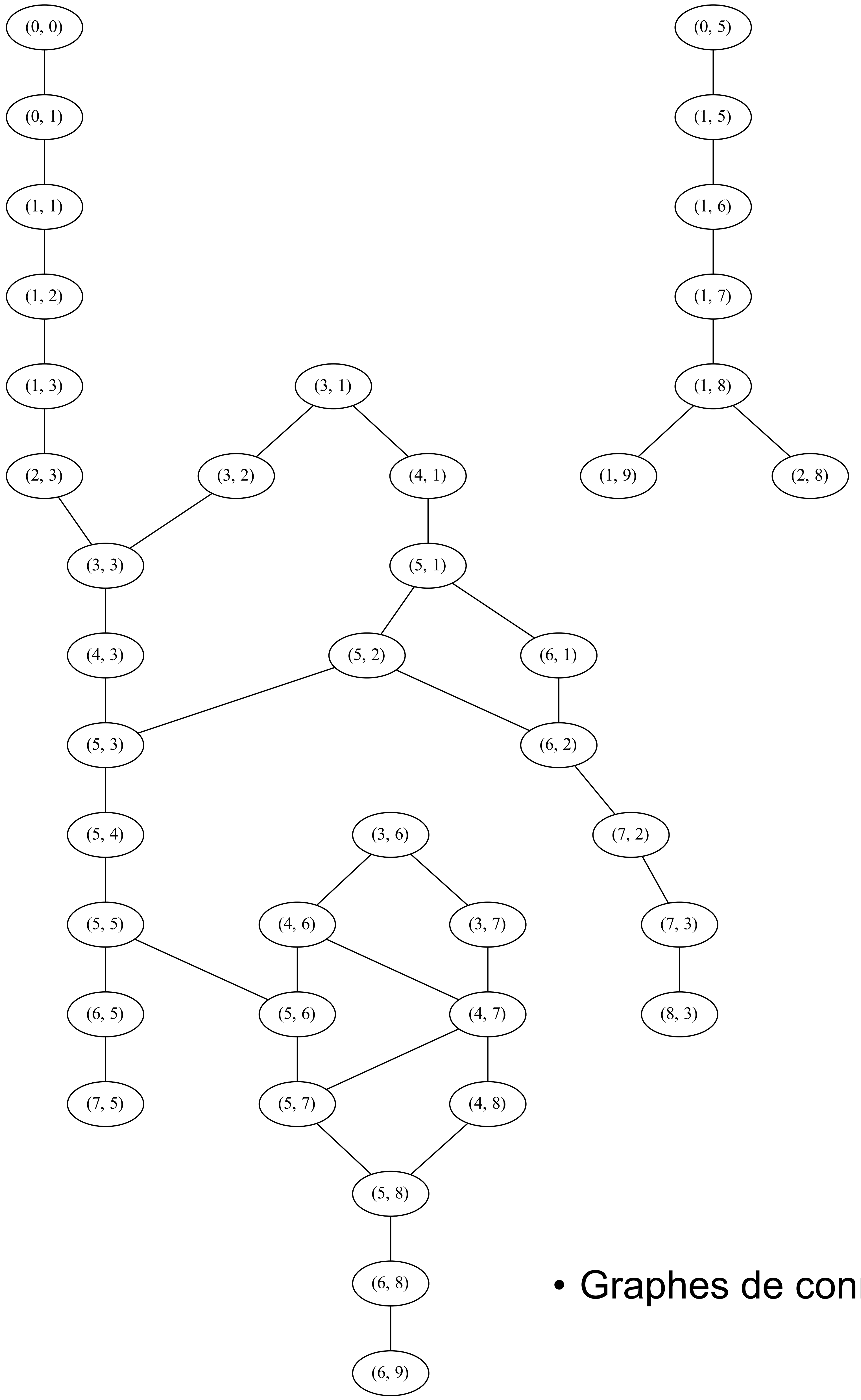
- Graphes de connexion des cases d'un labyrinthe

```python
maze = [[0, 1, 1, 1],
        [0, 0, 1, 0],
        [1, 0, 1, 1],
        [0, 0, 0, 0]]
```





Graphes




```
 .  .  *  *  *  .  *  *  *  *
 *  .  .  .  *  .  .  .  .  .
 *  *  *  .  *  *  *  *  .  *
 *  .  .  .  *  *  .  .  *  *
 *  .  *  .  *  *  .  .  .  *
 *  .  .  .  .  .  .  .  .  *
 *  .  .  *  *  .  *  *  .  .
 .  *  .  .  *  .  *  *  *  *
 .  .  *  .  *  *  *  *  *  *
 .  .  *  *  .  .  *  *  .  .
-----------------------------
```

```
m1 = [[0, 0, 1, 1, 1, 0, 1, 1, 1, 1],
      [1, 0, 0, 0, 1, 0, 0, 0, 0, 0],
      [1, 1, 1, 0, 1, 1, 1, 1, 0, 1],
      [1, 0, 0, 0, 1, 1, 0, 0, 1, 1],
      [1, 0, 1, 0, 1, 1, 0, 0, 0, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
      [1, 0, 0, 1, 1, 0, 1, 1, 0, 0],
      [0, 1, 0, 0, 1, 0, 1, 1, 1, 1],
      [0, 0, 1, 0, 1, 1, 1, 1, 1, 1],
      [0, 0, 1, 1, 0, 0, 1, 1, 0, 0]]
```

- Graphes de connexion des cases d’un labyrinthe

Graphes (représentation 3)

- Représentation par tableau de listes d'adjacence (connexions seules et en paramétrant par les sommets)
- Ajout des fonctions d'itération sur sommets et arrêtes

```
module type GRAPH = sig
  type t
  type vertex
  val order : t -> int
  val make : int -> t
  val add_edge : t -> vertex -> vertex -> unit
  val succ : t -> vertex -> vertex list
  val iter_vertex : (vertex -> unit) -> t -> unit
  val fold_vertex : (vertex -> 'a -> 'a) -> t -> 'a -> 'a
  val iter_succ : (vertex -> unit) -> t -> vertex -> unit
  val fold_succ : (vertex -> 'a -> 'a) -> t -> vertex -> 'a -> 'a
end ;;
```

```
module type VERTEX = sig
  type t
  val ord : t -> int
  val lab : int -> t
end ;;
```

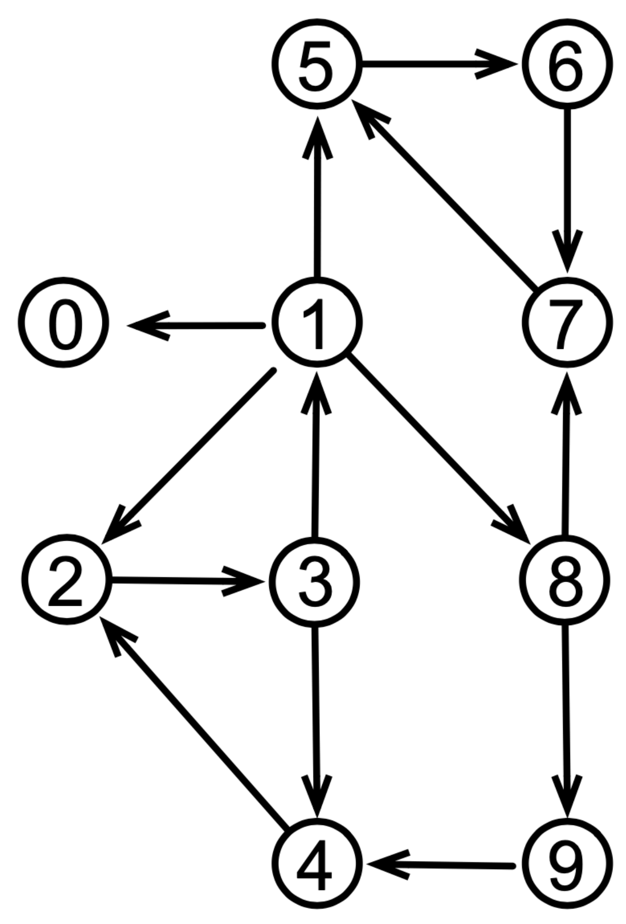
```
module Graph (V: VERTEX): (GRAPH with type vertex = V.t) =  struct
 type vertex = V.t
 type t = vertex list array
 let make n = Array.make n [ ]
 let order g = Array.length g
 let add_edge g x y = g.(V.ord x) <- y :: g.(V.ord x)
 let succ g x = g.(V.ord x)
 let vertices g = List.init (order g) (fun i -> V.lab i)
 let iter_vertex f g = List.iter f (vertices g)
 let fold_vertex f g v0 = List.fold_left (Fun.flip f) v0 (vertices g)
 let iter_succ f g x = List.iter f (succ g x)
 let fold_succ f g x v0 = List.fold_left (Fun.flip f) v0 (succ g x)
end ;;
```

Simplification des modules de `Ocamlgraph`

Graphes (représentation 3)

- Représentation par tableau de listes d'adjacence (connexions seules et en abstrayant les sommets)

```
module G = Graph (Vint) ;;

let add_edges g  =
  List.iter (fun (i, j) ->
      G.add_edge g (Vint.lab i)
                   (Vint.lab j) );;

let g = G.make 10 ;;

add_edges g [1,0; 1,2; 1,5; 1,8; 2,3;
    3,1; 3,4; 4,2; 5,6; 6,7;
    7,5; 8,7; 8,9; 9,4];;
```



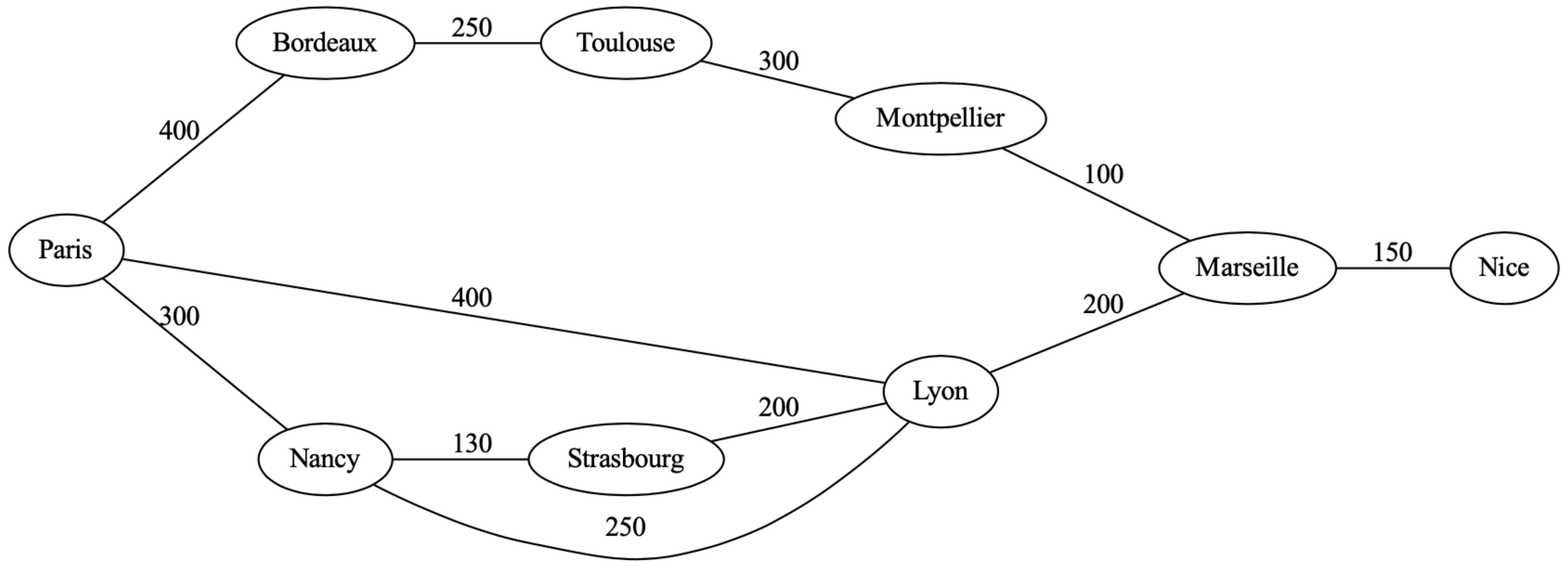

```
module Vint : VERTEX = struct
  type t = int
  let ord x = x
  let lab i = i
end  ;;
```

Graphes (représentation 3)

- Représentation par tableau de listes d'adjacence (connexions seules et en abstrayant les sommets)

```
module G = Graph (Villes) ;;

let add_edges g  =
  List.iter (fun (v1, v2) ->
      G.add_edge g (Villes.lab (pos v1 villes))
                   (Villes.lab (pos v2 villes)) );;

let g = G.make (Array.length villes) ;;

add_edges g ["Paris","Bordeaux"; "Paris","Nancy"; "Paris","Lyon";
 "Bordeaux","Toulouse"; "Toulouse","Montpellier"; "Montpellier","Marseille";
 "Nancy","Strasbourg"; "Nancy","Lyon"; "Strasbourg","Lyon";
 "Lyon","Marseille"; "Marseille","Nice"] ;;
```

```
let villes = [| "Paris"; "Bordeaux"; "Toulouse";
 "Montpellier"; "Marseille"; "Nancy";
 "Strasbourg"; "Lyon"; "Nice" |] ;;

let pos x l = let ch = Array.find_index (fun y -> y = x) l in
  match ch with Some i -> i
  | _ -> failwith "" ;;
```

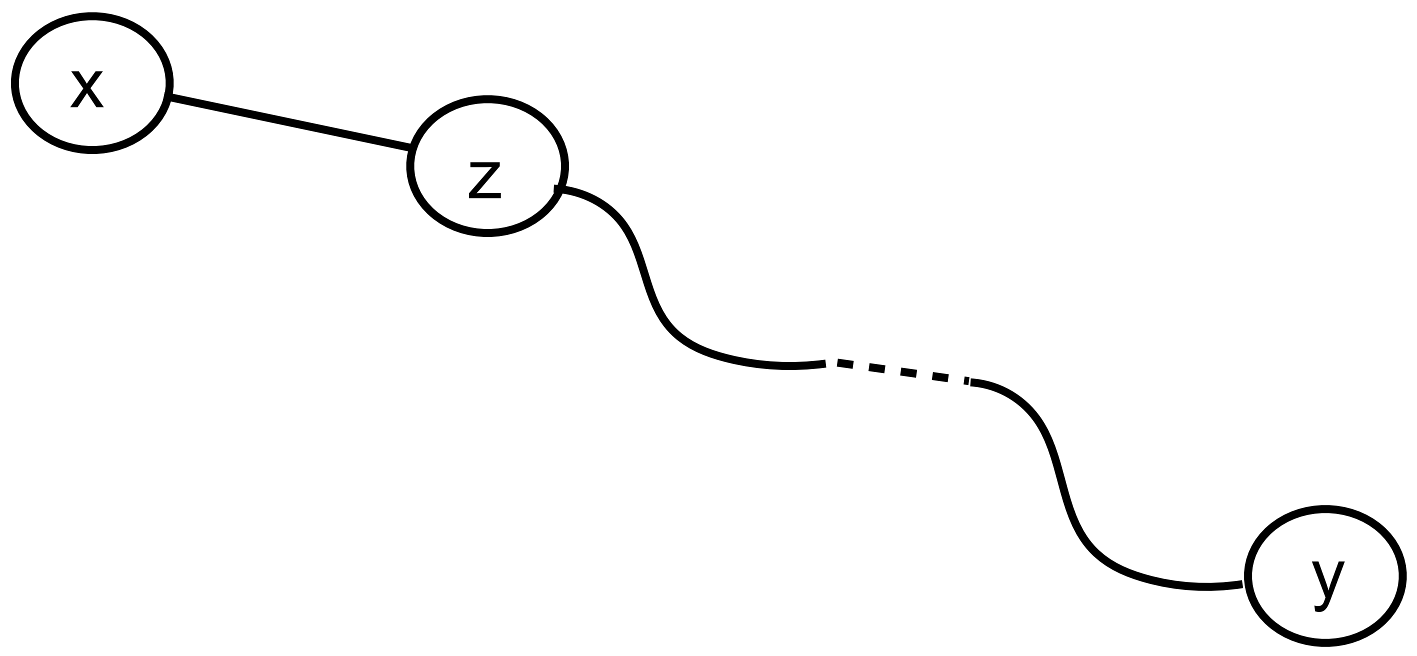



```
module Villes : VERTEX = struct
  type t = string
  let lab i = villes.(i)
  let ord x = pos x villes
end ;;
```

Chemin dans un graphe

- chemin de x à y (*le fil d'ariane*)

```
let path g x y  =
  let n = G.order g in
  let visited = Array.make n false in
  let rec path1 x y =
    if not visited.(V.ord x) then begin
      visited.(V.ord x) <- true;
      if x = y then Some [x] else
          G.fold_succ (fun z -> fun r ->
            if r = None then match path1 z y with
              | None -> None
              | Some trail -> Some (x :: trail)
            else r)
            g x None
      end else None in
    path1 x y;;
```



(on peut optimiser en ne prenant que la première solution dans les successeurs de x)

```
match path g x1 x2 with None ->  ()
  | Some tr ->
      List.iter (fun x -> Printf.printf "%d " (V.ord x)) tr ;;
```

Chemin dans un graphe

- sortie de labyrinthe)

```
let a = [| [|0; 0; 1; 1; 1; 0; 1; 1; 1; 1|];
        [|1; 0; 0; 0; 1; 0; 0; 0; 0; 0|];
        [|1; 1; 1; 0; 1; 1; 1; 1; 0; 1|];
        [|1; 0; 0; 0; 1; 1; 0; 0; 1; 1|];
        [|1; 0; 1; 0; 1; 1; 0; 0; 0; 1|];
        [|1; 0; 0; 0; 0; 0; 0; 0; 0; 1|];
        [|1; 0; 0; 1; 1; 0; 1; 1; 0; 0|];
        [|0; 1; 0; 0; 1; 0; 1; 1; 1; 1|];
        [|0; 0; 1; 0; 1; 1; 1; 1; 1; 1|];
        [|0; 0; 1; 1; 0; 0; 1; 1; 0; 0|] |] ;;
```

```
let next a i j =
    let m = Array.length a and n = Array.length a.(0) in
    List.fold_left (fun r -> fun (i',j') ->
        if (0 <= i' && i' < m) && (0 <= j' && j' < n) && a.(i').(j') = 0 then
            (i',j') :: r else r) [ ] [i-1,j; i,j-1; i,j+1; i+1,j] ;;
```

```
let vE a i j = let m = Array.length a in
  let ord i j = m * i + j in
  V.lab (ord i j) ;;

let vD a x = let m = Array.length a in
  let k = V.ord x in
  (k / m) , (k mod m) ;;
```

```
let mk_graph_maze a =
  let m = Array.length a in
  let n = Array.length a.(0) in
  let g = G.make (m*n) in
  for i = 0 to m-1 do
    for j = 0 to n-1 do
      List.iter (fun (i', j') ->
        G.add_edge g (vE a i j) (vE a i' j'))
      (next a i j)
    done
  done; g ;;
```

```
let maze = mk_graph_maze a ;;

let Some ch = path maze (vE a 0 0) (vE a 6 9) in
List.iter (fun x ->
  let (i, j) = vD a x in
  Printf.printf "(%d, %d) -> " i j) ch ;;
```

- Représentation compacte de la matrice de connexion

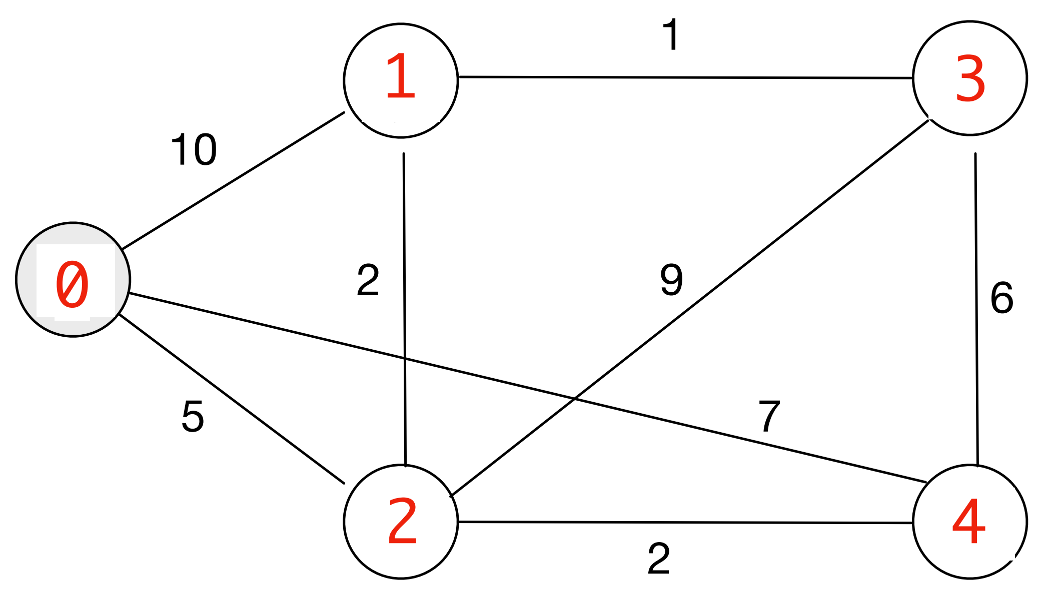
Graphes (représentation 4)

- Représentation avec des arcs valués (par des entiers)

```
module type GRAPHV = sig
  type t
  type vertex
  type weight = int
  type edge = vertex * weight
  val order : t -> int
  val make : int -> t
  val add_edge : t -> vertex -> edge -> unit
  val succ : t -> vertex -> edge list
  val iter_vertex : (vertex -> unit) -> t -> unit
  val fold_vertex : (vertex -> 'a -> 'a) -> t -> 'a -> 'a
  val iter_succ : (edge -> unit) -> t -> vertex -> unit
  val fold_succ : (edge -> 'a -> 'a) -> t -> vertex -> 'a -> 'a
end ;;
```

```
module GraphV (V: VERTEX): (GRAPHV with type vertex = V.t) =  struct
  type vertex = V.t
  type weight = int
  type edge = vertex * weight
  type t = edge list array
  let make n = Array.make n [ ]
  let order g = Array.length g
  let add_edge g x e = g.(V.ord x) <- e :: g.(V.ord x)
  let succ g x = g.(V.ord x)
  let vertices g = List.init (order g) (fun i -> V.lab i)
  let iter_vertex f g = List.iter f (vertices g)
  let fold_vertex f g v0 = List.fold_left (Fun.flip f) v0 (vertices g)
  let iter_succ f g x = List.iter f (succ g x)
  let fold_succ f g x v0 = List.fold_left (Fun.flip f) v0 (succ g x)
 end ;;
```

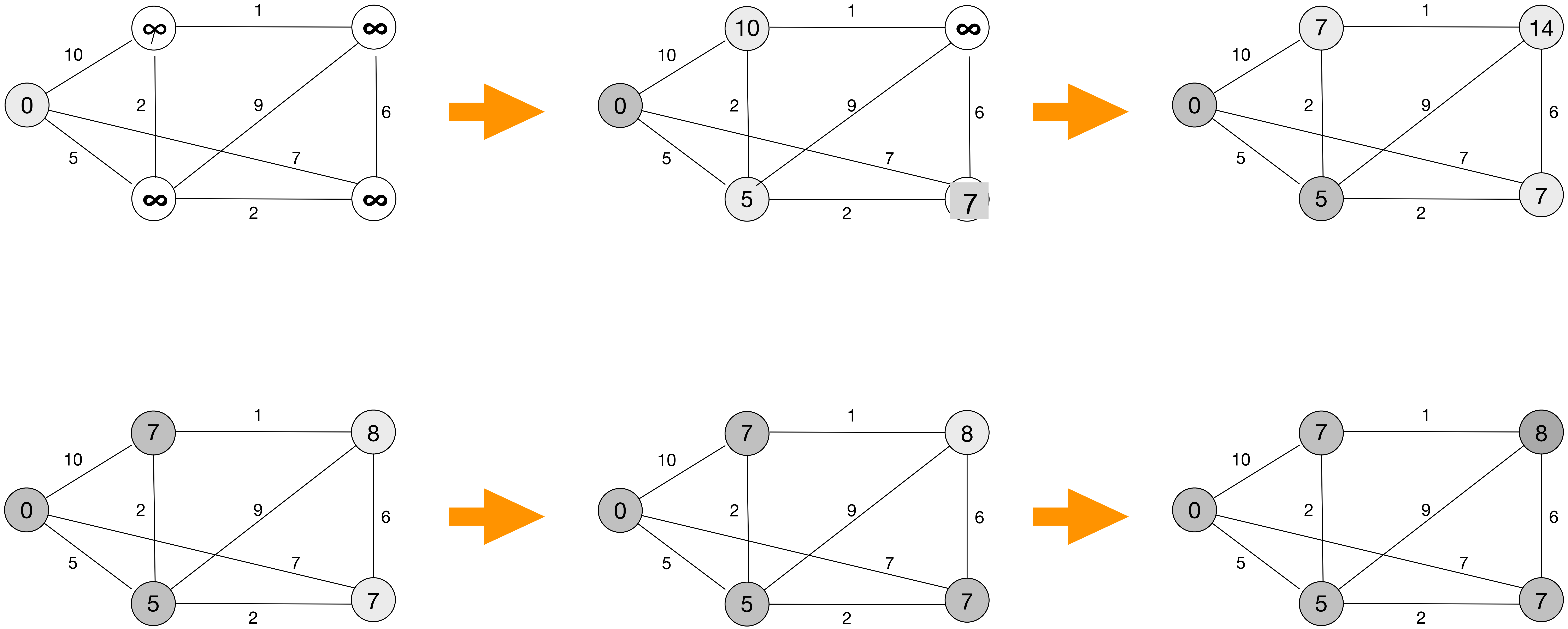
```
module G = GraphV (Vint) ;;

let g = G.make 5;;

let add_edges g  =
  List.iter (fun (i, j, w) ->
      G.add_edge g (Vint.lab i)
                   ((Vint.lab j), w) );;

add_edges g [0,1,10; 0,2,5; 0,4,7;
             1,0,10; 1,2,2; 1,3,1;
             2,0,5; 2,1,2; 2,3,9; 2,4,2;
             3,1,1; 3,2,9; 3,4,6;
             4,0,7; 4,2,2; 4,3,6] ;;
```



- Ajout des fonctions d’itération sur sommets et arrêtes

Plus court chemin

- calculer le chemin le plus court pour aller d'un sommet à un autre sommet `[Dijkstra, 59]`



Plus court chemin

- calculer le chemin le plus court pour aller d'un sommet à un autre `[Dijkstra, 59]`

  - on démarre avec l'ensemble frontière `{x}`

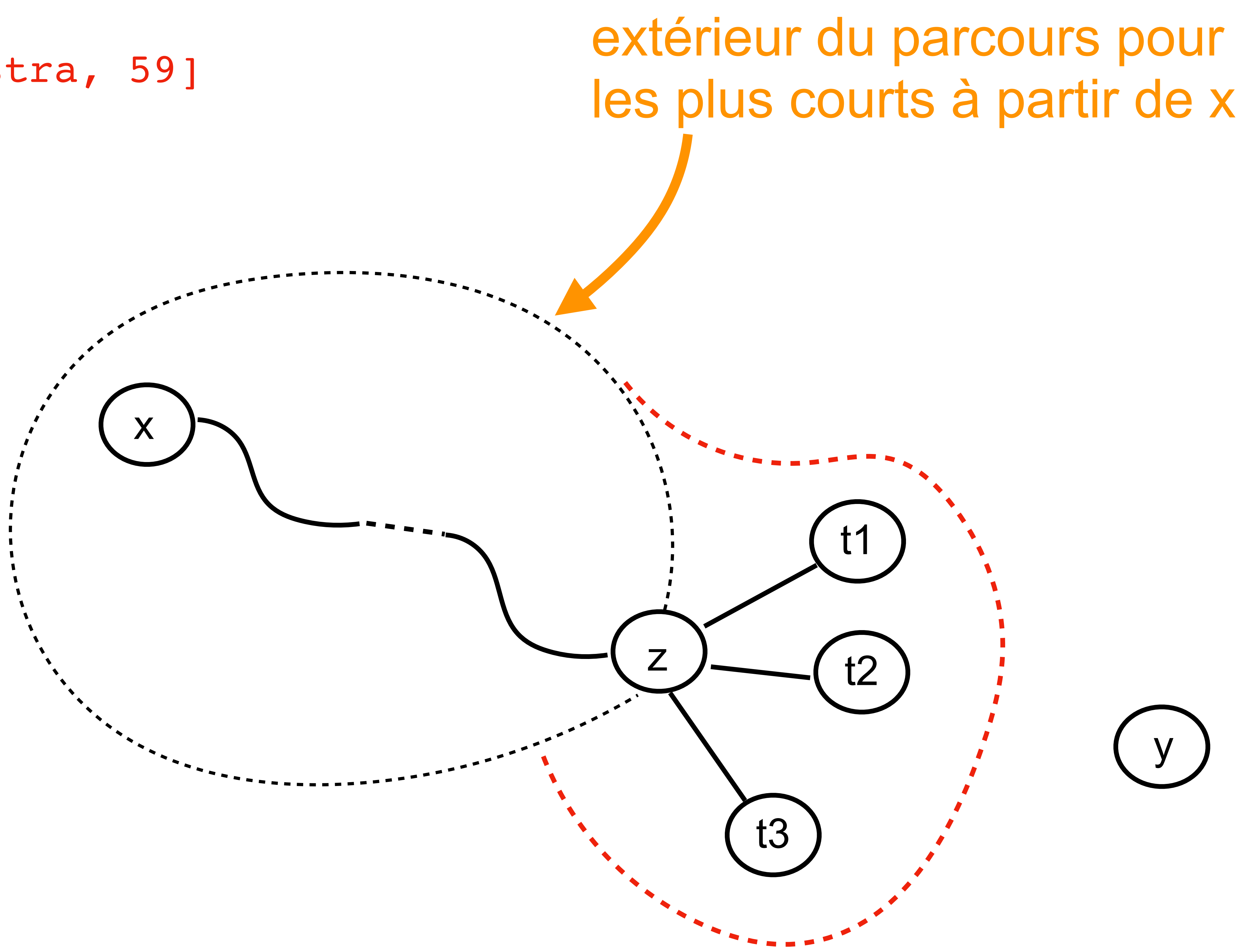
  - tant que `y` n'est pas dans la frontière :

    - soit `d` la distance min à partir de `x` à `z` dans la frontière
    - retirer `z` de la frontière
    - pour tous les successeurs `z'` à distance `dzz'` de `z`

      rajouter `z'` à la frontière (s'il n'y est pas déjà)

      mettre à jour `xdmin.(z') <- min (xdmin.(z')) (d + dzz')`

  - le résultat est `dmin.(y)` en se souvenant du chemin qui y a mené

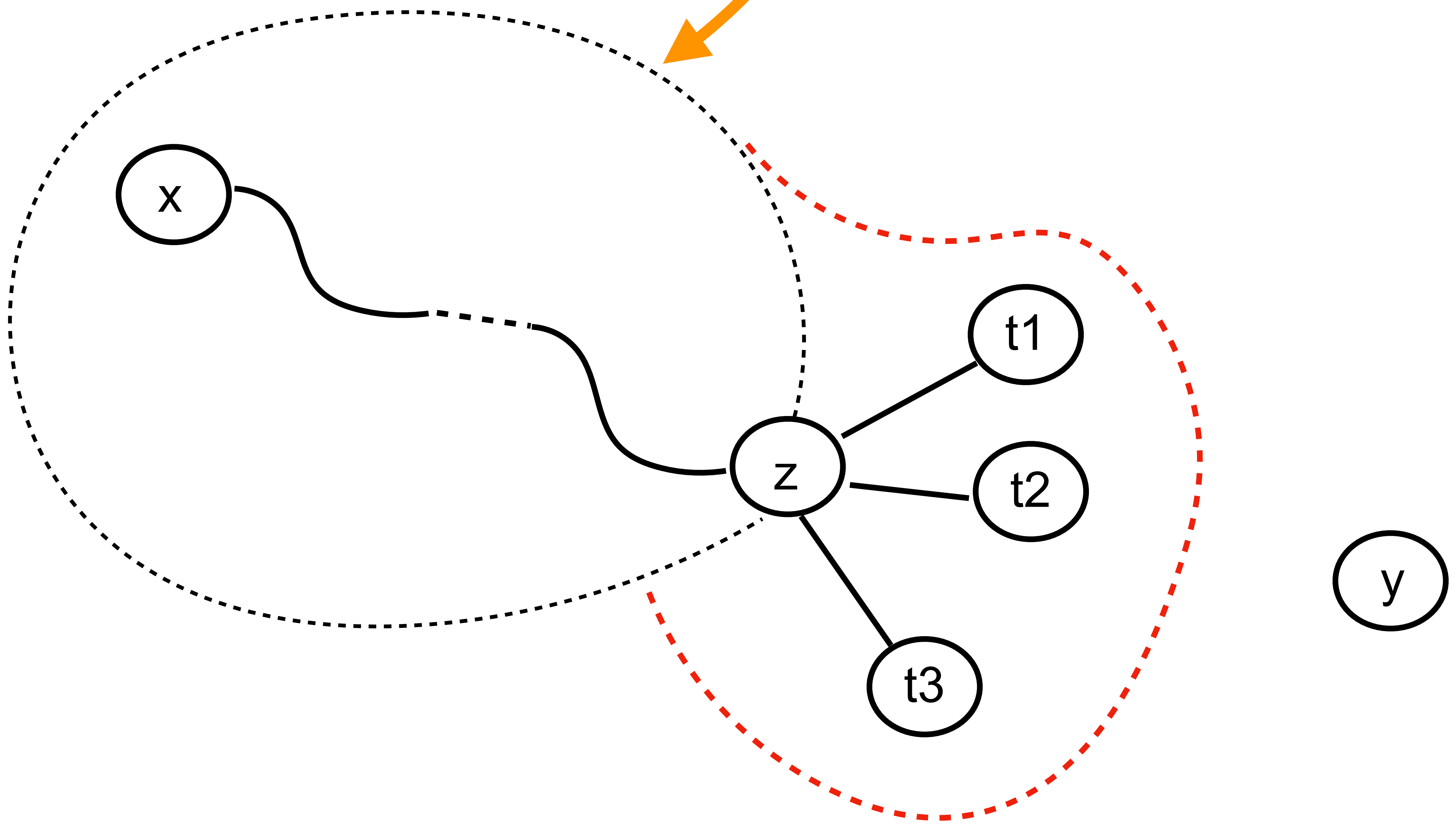


Plus court chemin

- calculer le chemin le plus court pour aller d'un sommet à un autre [Dijkstra, 59]

```
let shortest_path g x y =
  let n = G.order g in
  let xdist = Array.make n max_int in
  let xpath = Array.make n [ ] in
  let solved = Array.make n false in
  xdist.(V.ord x) <- 0;
  let rec loop fringe =
    let (d, z) = xdmin xdist solved !fringe in
    solved.(V.ord z) <- true ;
    if d = max_int then [ ] else
    if z = y then List.rev (y :: xpath.(V.ord y)) else begin
      G.iter_succ (fun (z', dzz') ->
        if not solved.(V.ord z') then
          if xdist.(V.ord z') = max_int then
             fringe := z' :: !fringe ;
          if xdist.(V.ord z') > d + dzz' then begin
            xdist.(V.ord z') <- d + dzz';
            xpath.(V.ord z') <- z :: xpath.(V.ord z)
          end) g z ;
      loop fringe end in
  loop (ref [x]) ;;
```

extérieur du parcours pour les plus courts à partir de x




```
let x = V.lab 0 and y = V.lab 3 in
let Some ch = shortest_path g x y in
List.iter (fun x -> Printf.printf "%d " (V.ord x)) ch ;;
```

```
let xdmin xdist solved =
  List.fold_left
    (fun (d, r) x -> let i = V.ord x in
      if not solved.(i) && d > xdist.(i)
        then (xdist.(i), x) else (d, r))
    (max_int, V.lab 0) ;;
```

on peut rendre cette fonction plus efficace avec une file de priorité

Plus court chemin

- l’algorithme précédent ne marche que si les distances sont positives
- avec des distances quelconques : algorithme Ford-Fulkerson
- pour trouver les distances minimales entre tous les sommets: programmation dynamique

Algorithmes sur les graphes

- graphes orientés ou non-orientés
- arbres de recouvrement
- plus courts chemins dans un graphe
- arbres de recouvrements minimaux
- flux dans un graphe
- tri topologique
- composantes (fortement) connexes
- bi-connexité et points d’articulation
- mariages stables
- cycles, chemins hamiltoniens
- problème du représentant de commerce
- . . .

Conclusion

VU:

- graphes
- représentations
- parcours en profondeur d’abord
- sortie de labyrinthe
- parcours en largeur d’abord
- plus court chemin

TODO list

- objets
- parallélisme
- autres langages fonctionnels