

Fonctionnalité et Modularité

Cours 7

Jean-Jacques Lévy

`jean-jacques.levy@inria.fr`

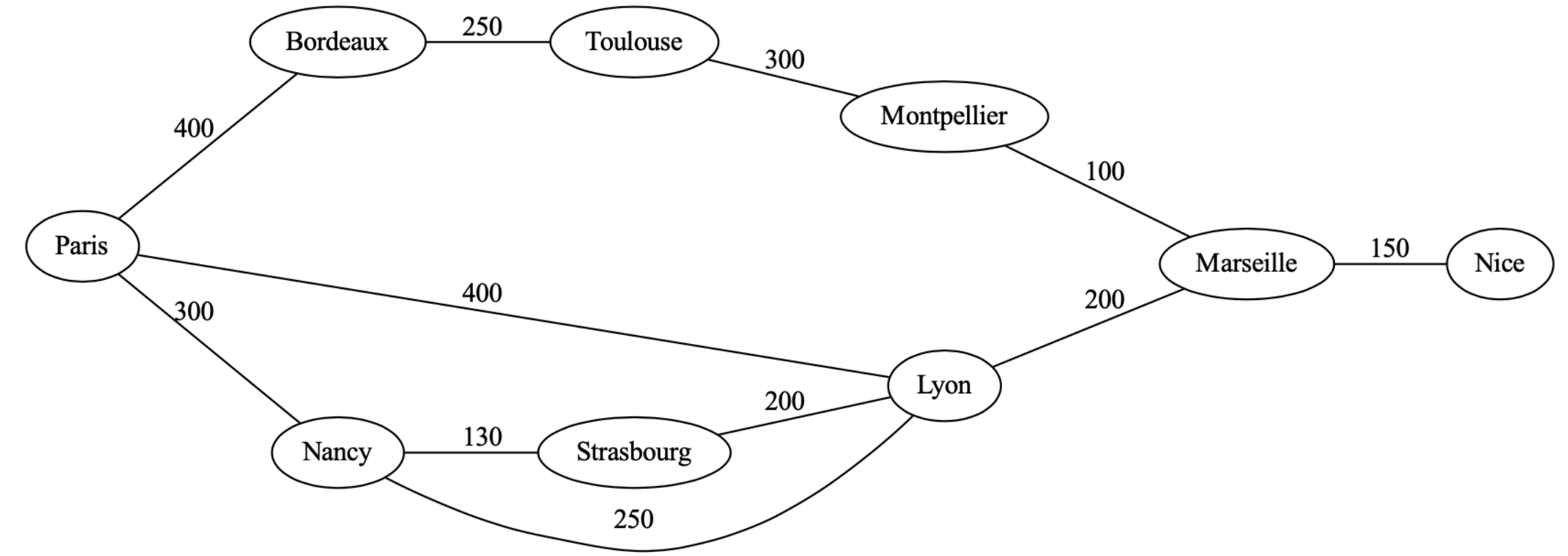
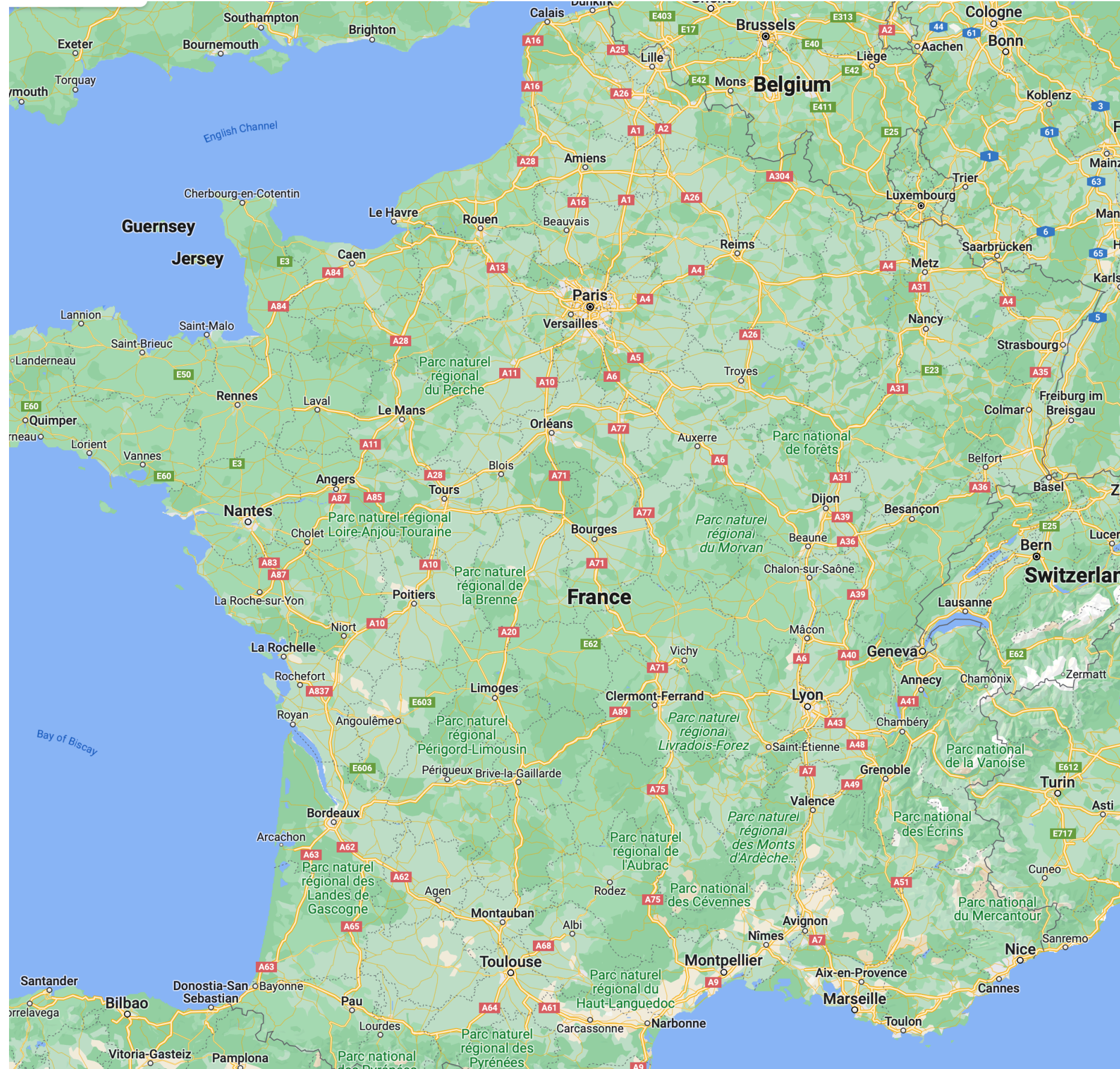
`http://jeanjacqueslevy.net/prog-fm`

Plan

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

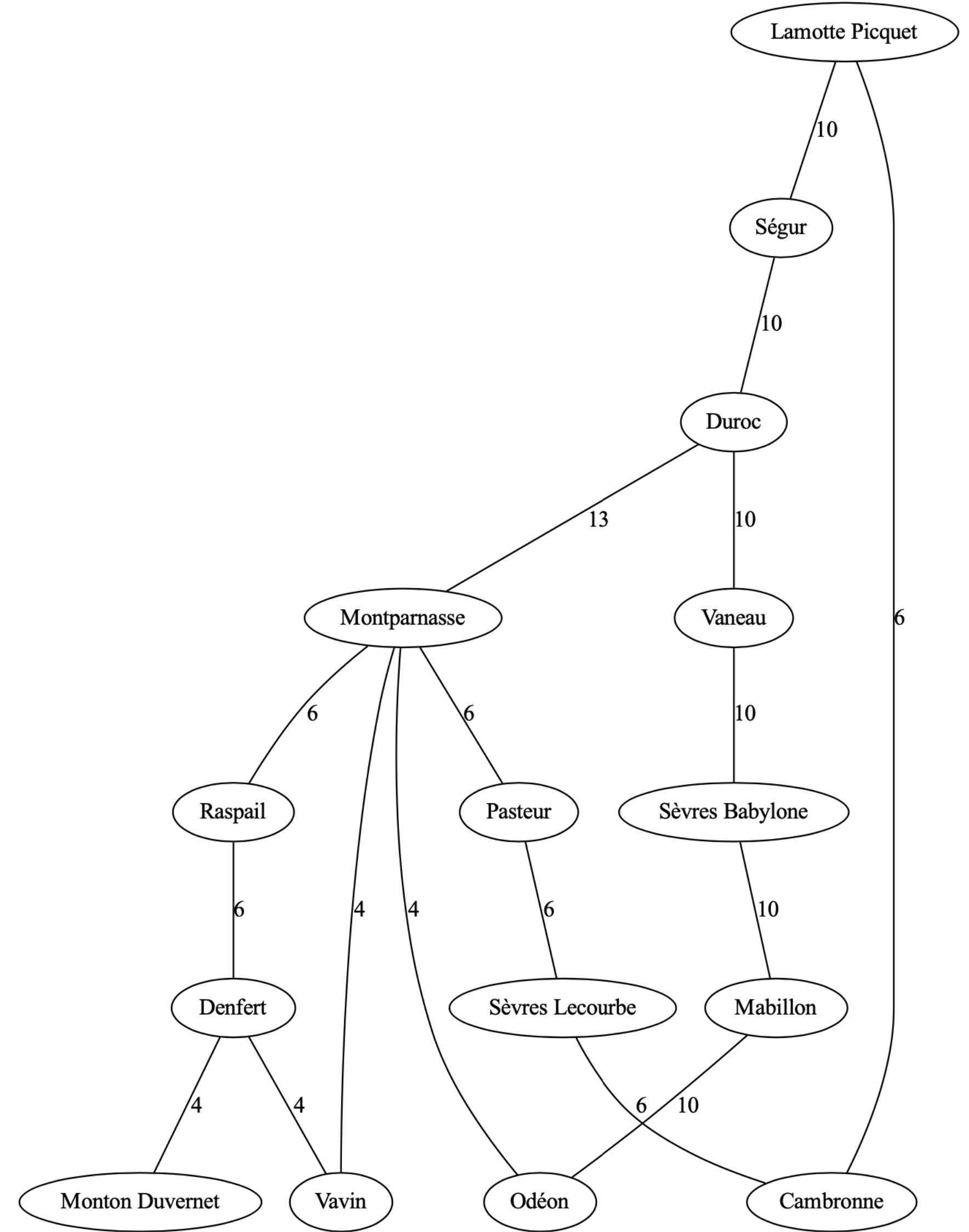
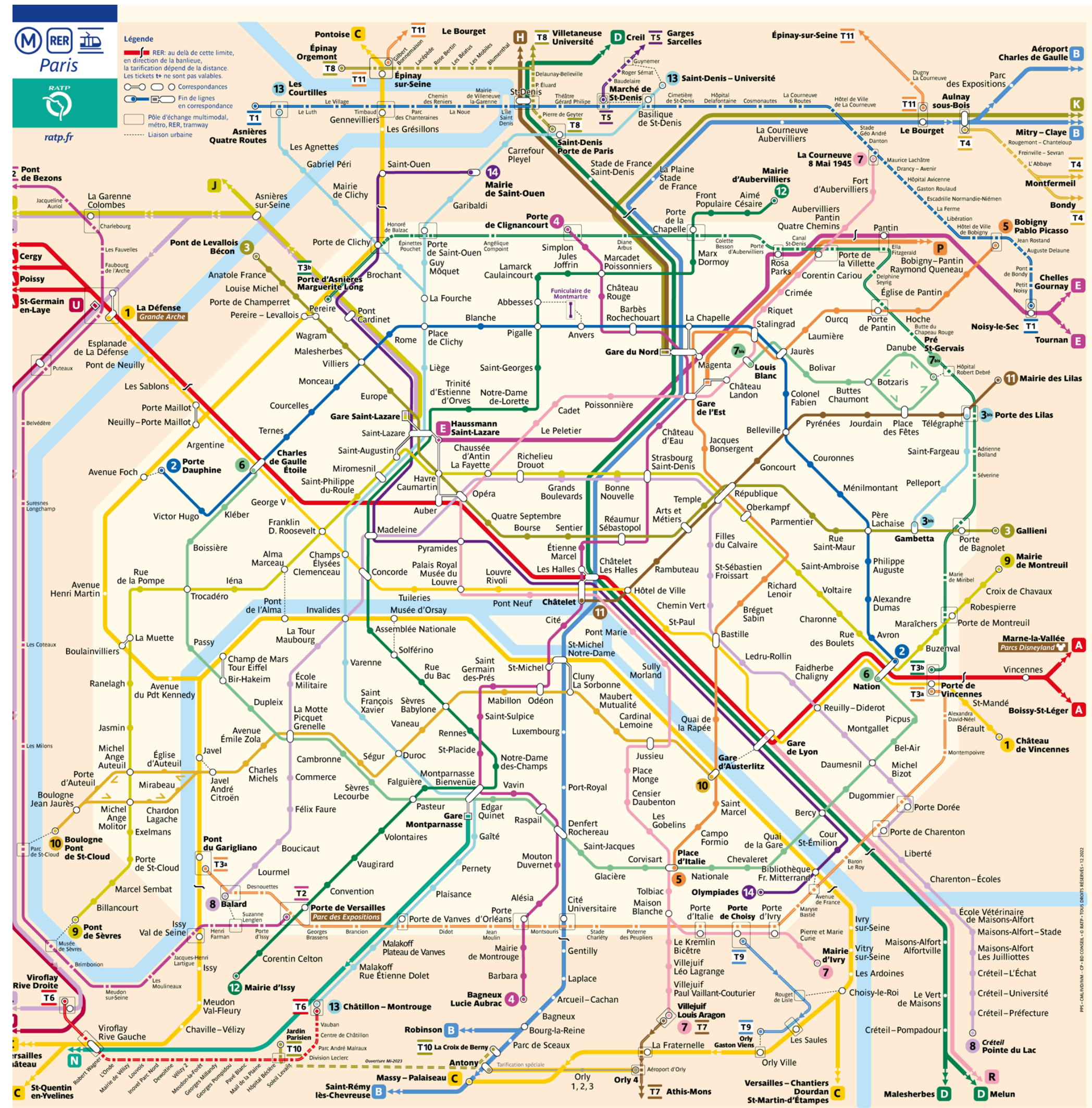
télécharger Ocaml en <http://www.ocaml.org>

Graphes



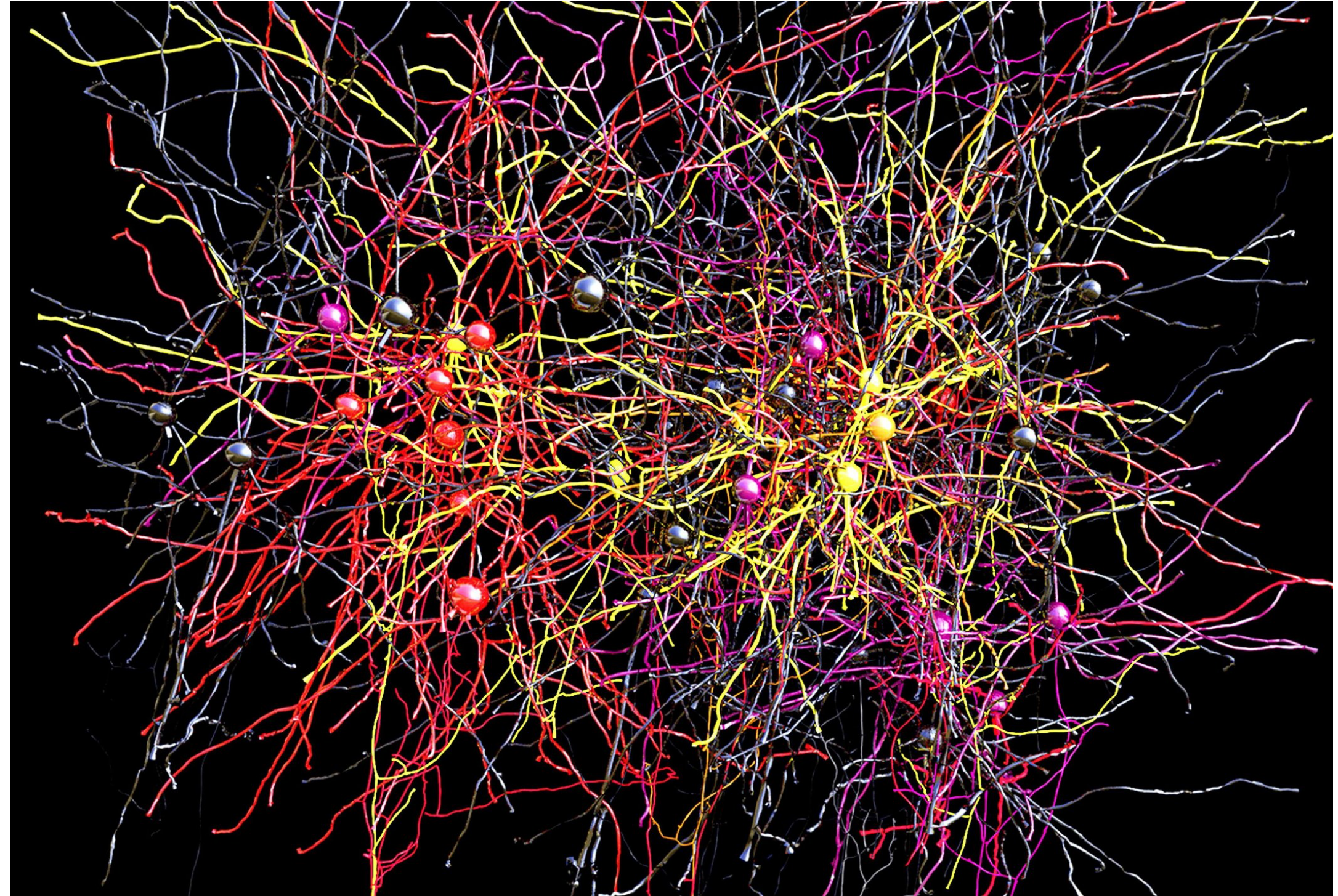
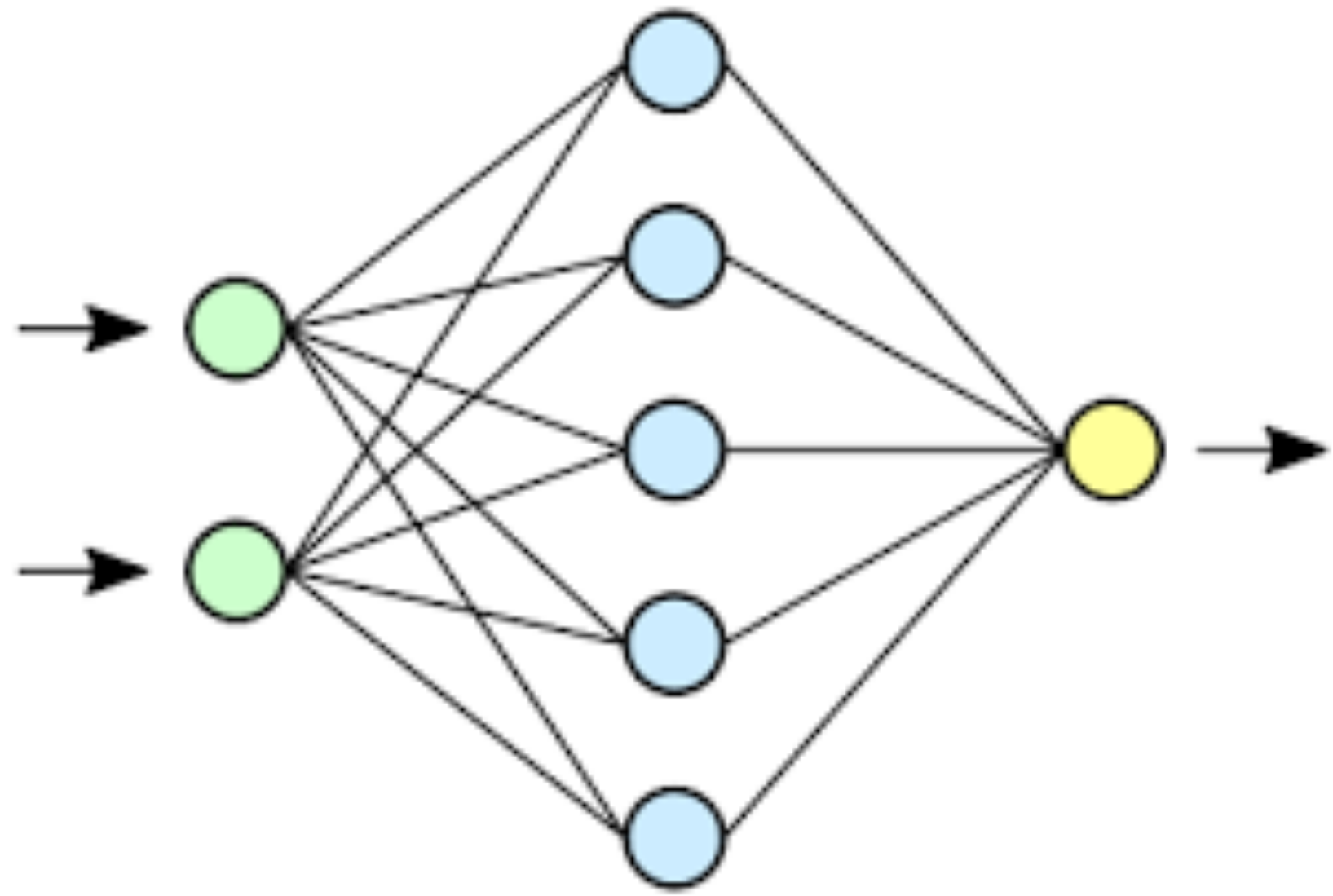
- carte routière et graphe des connexions entre villes avec la distance en km

Graphes



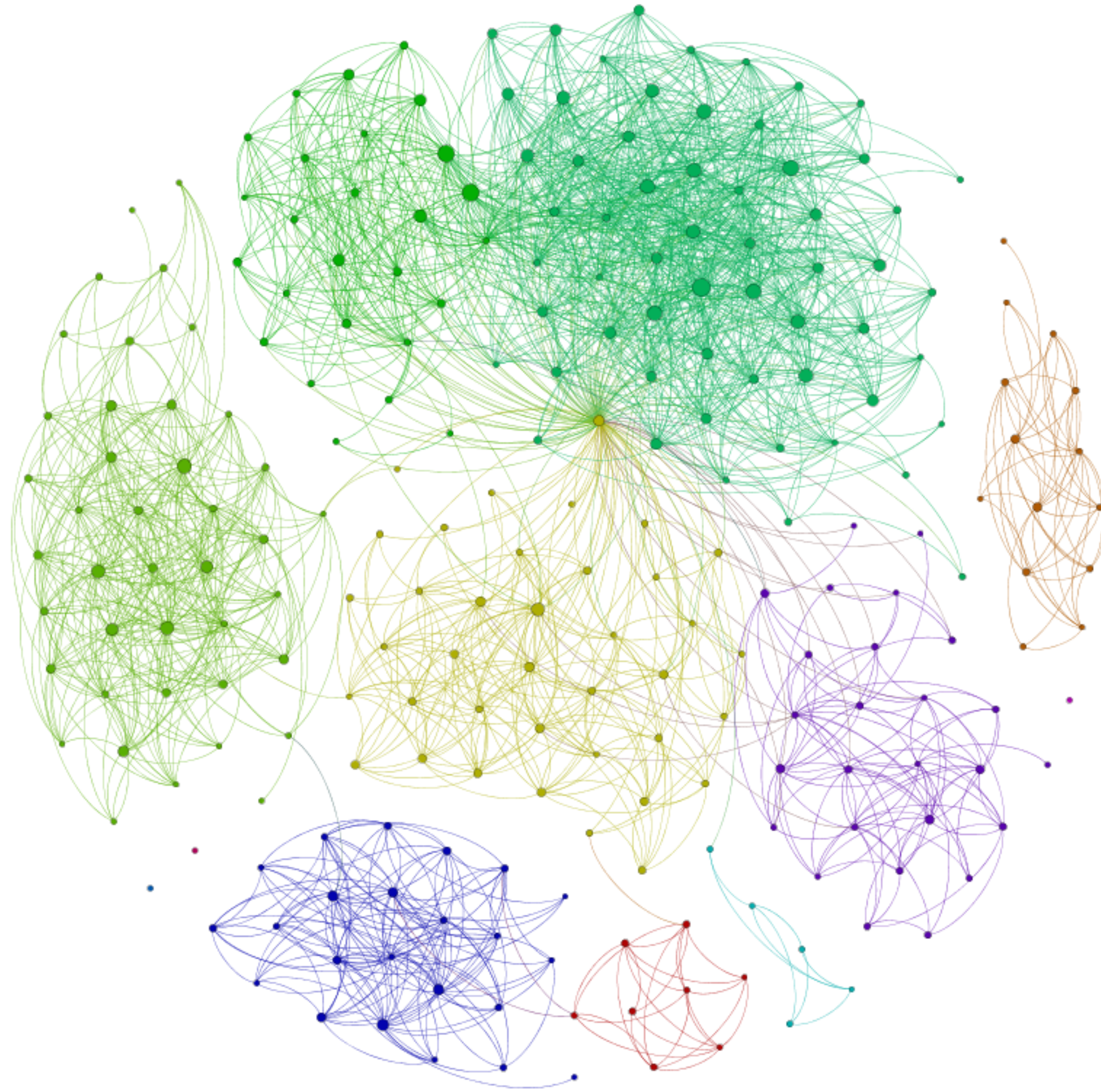
- plan du métro et le graphe qui relie les différentes stations

Graphes



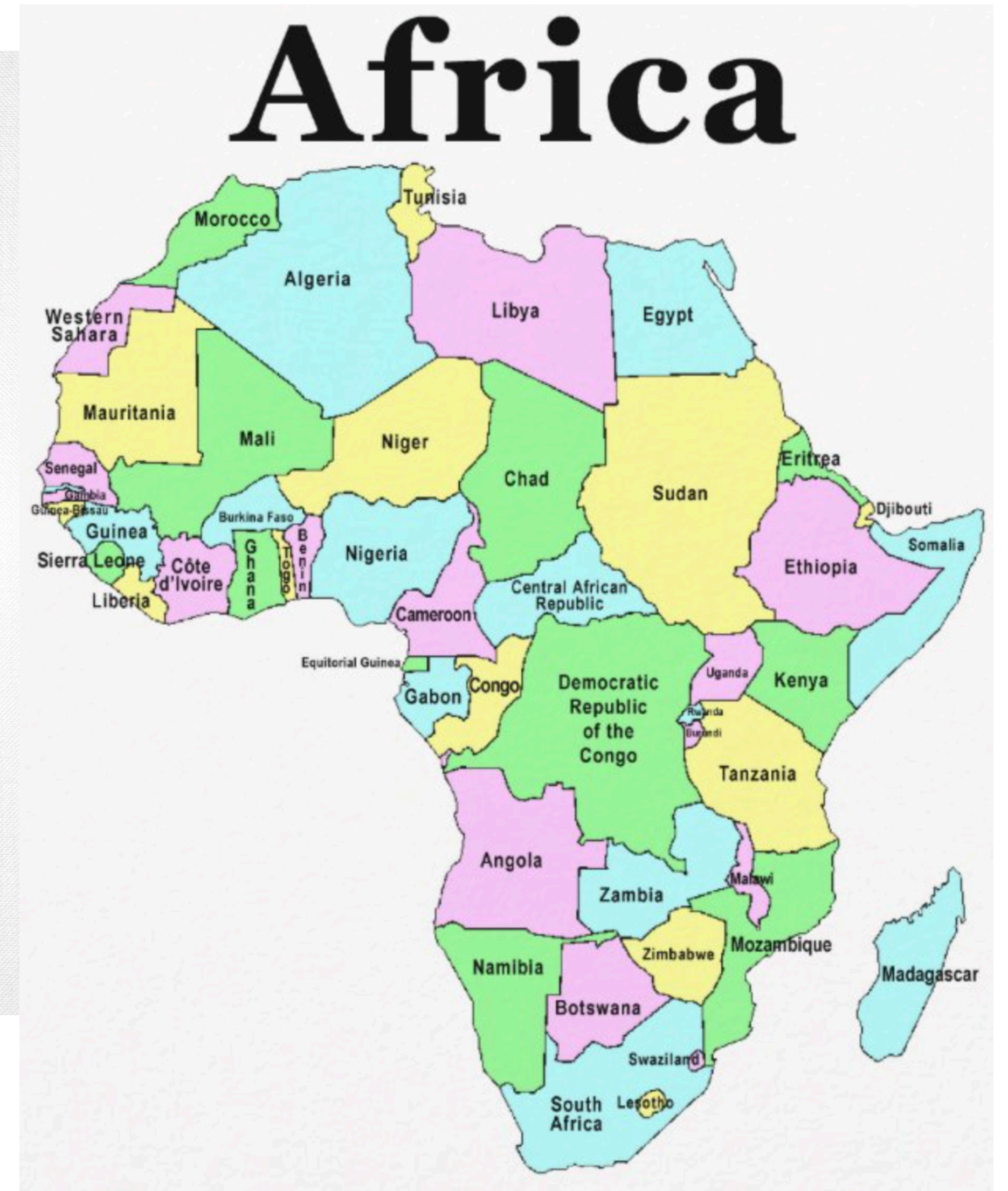
- Réseaux de neurone 2 couches et vrais réseaux de neurones biologiques

Graphes



- Réseaux d'amis sur Facebook

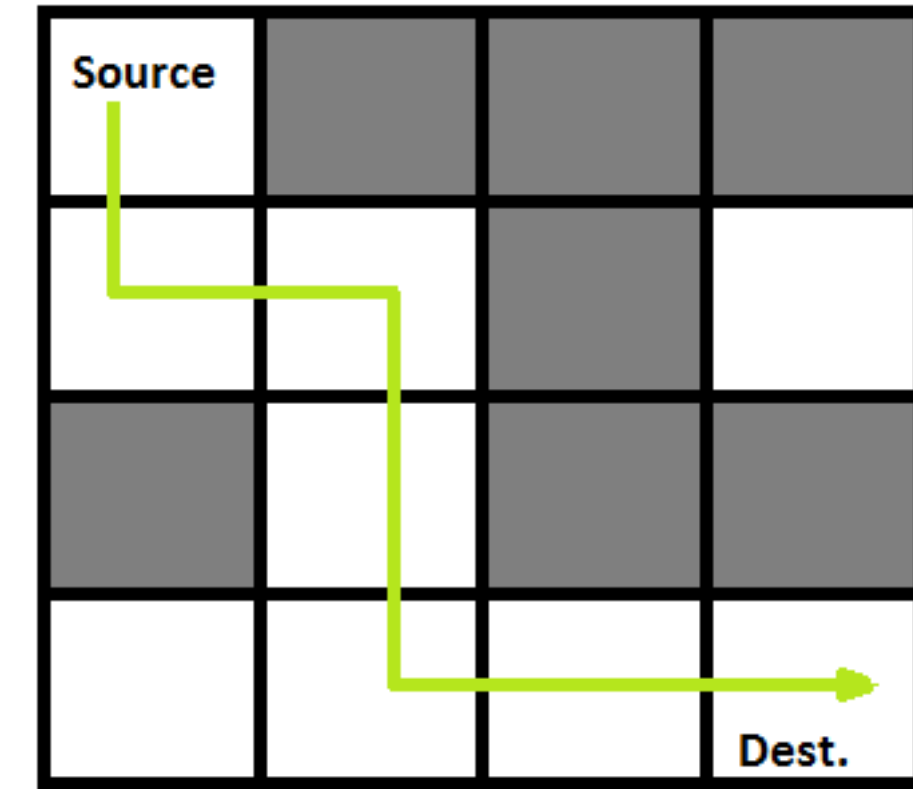
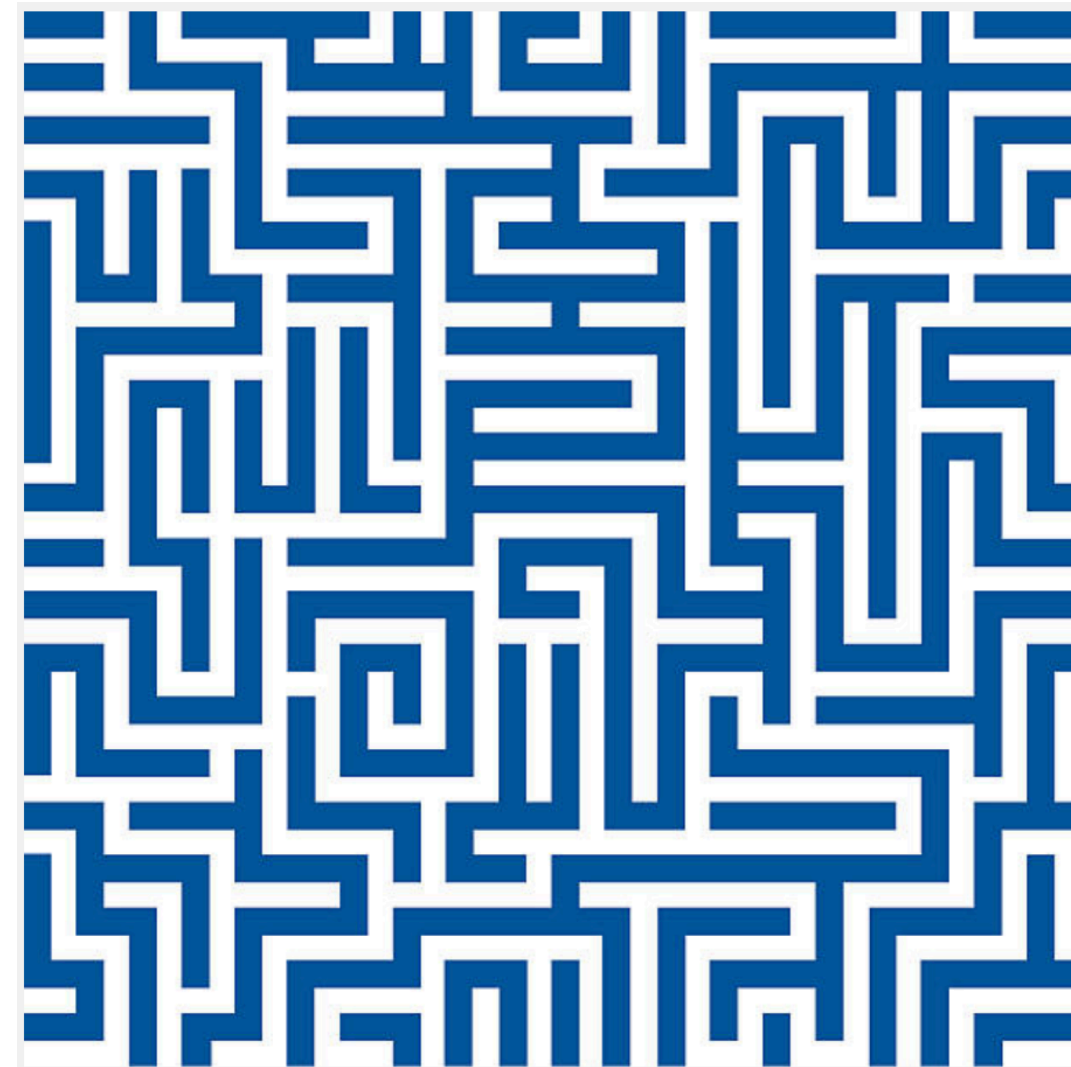
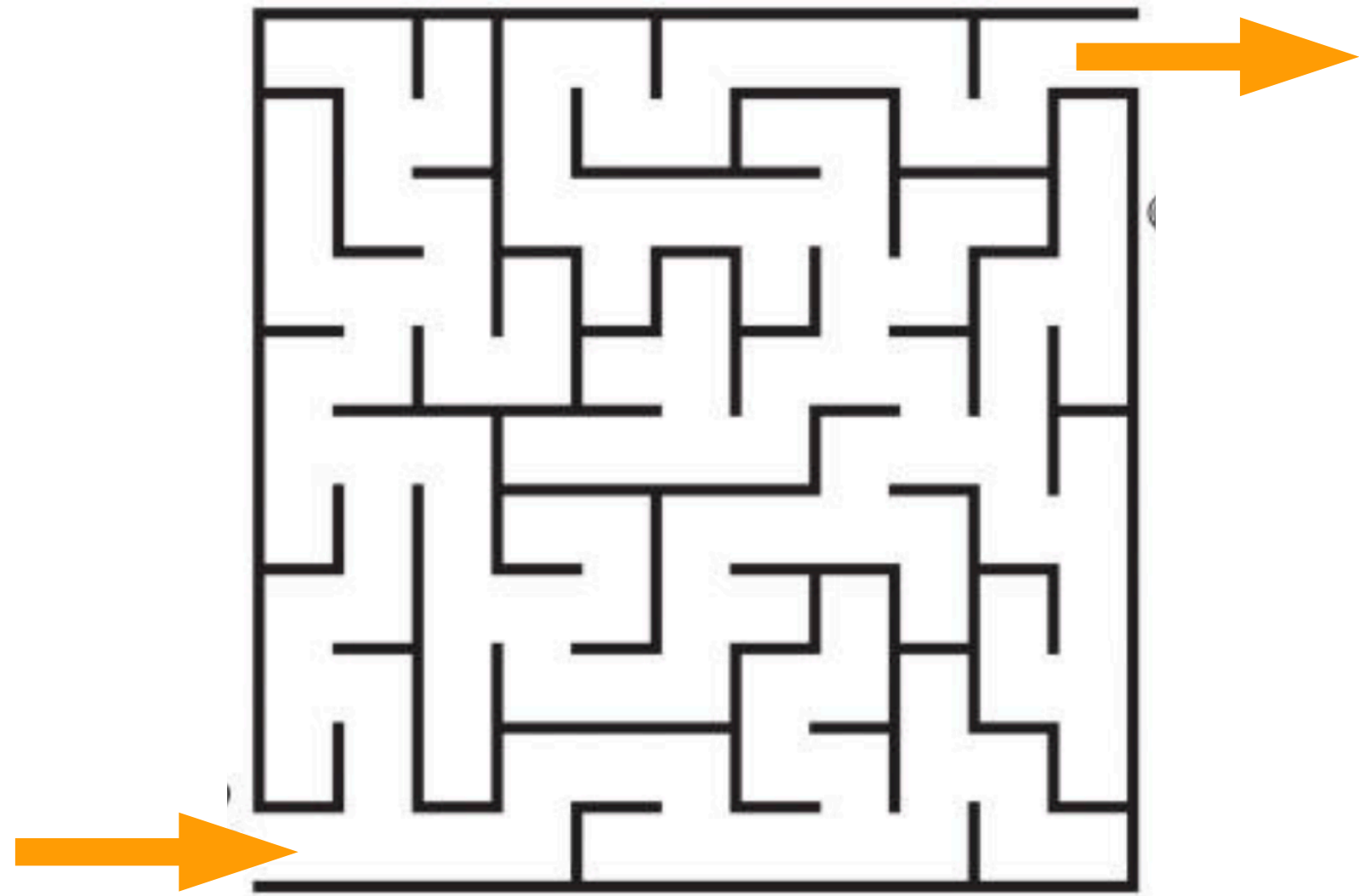
Graphes



- Graphes planaires coloriables avec 4 couleurs

Labyrinthes

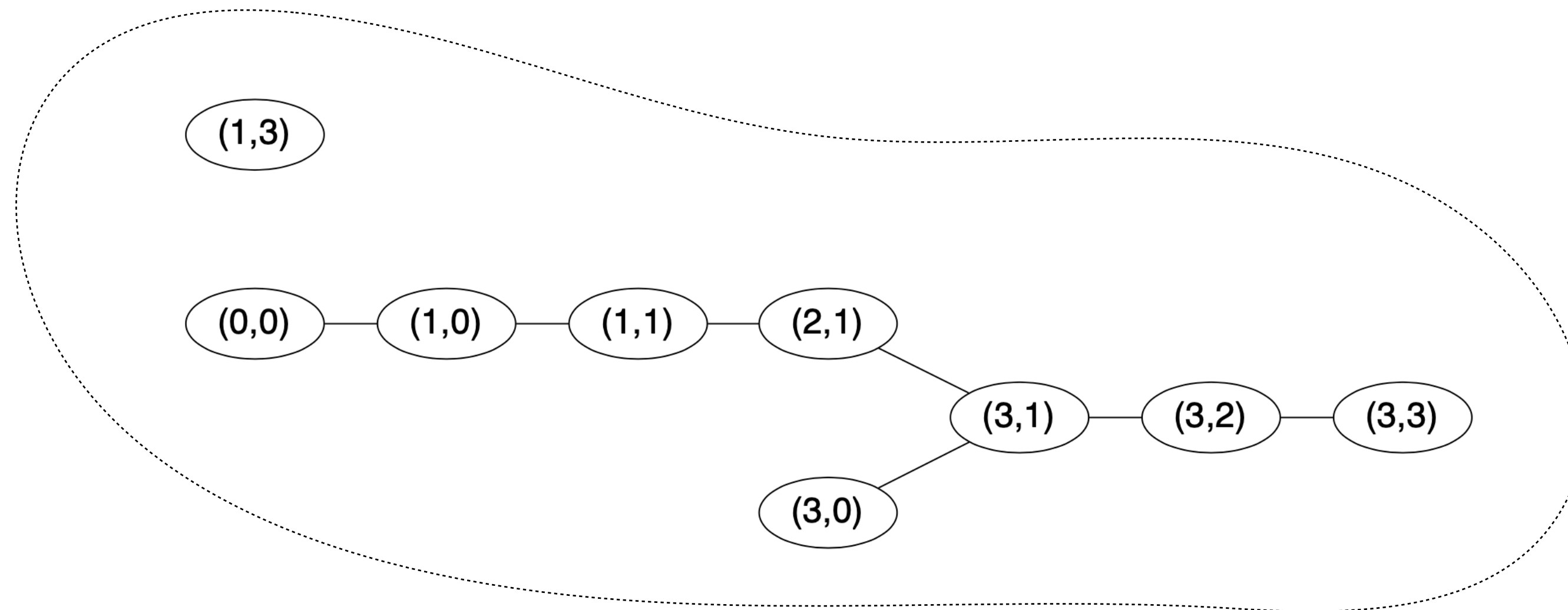
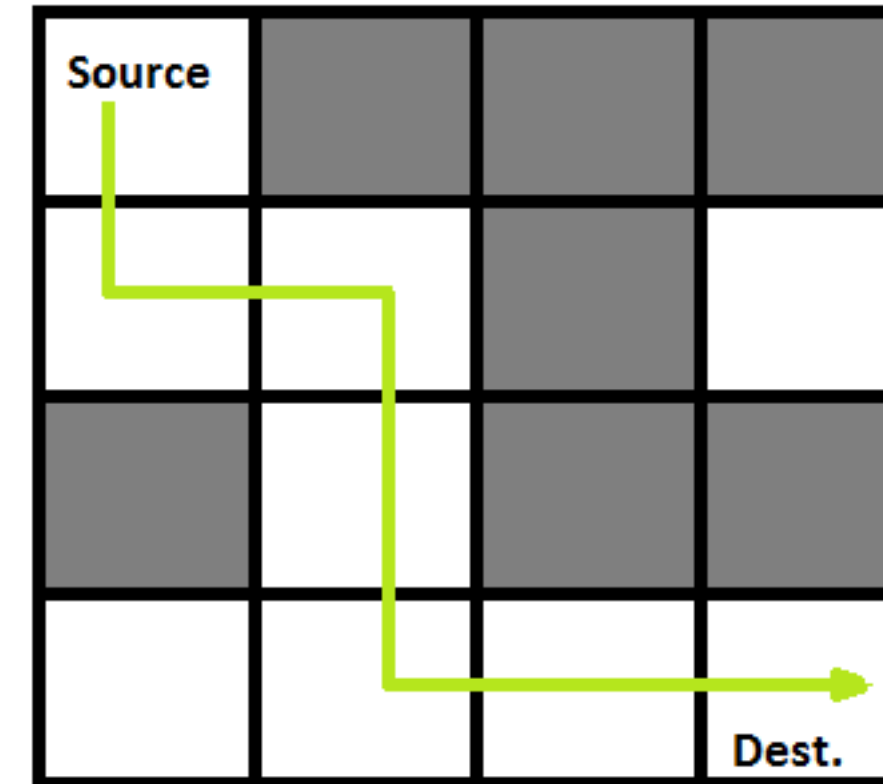
- trouver le chemin vers la sortie !



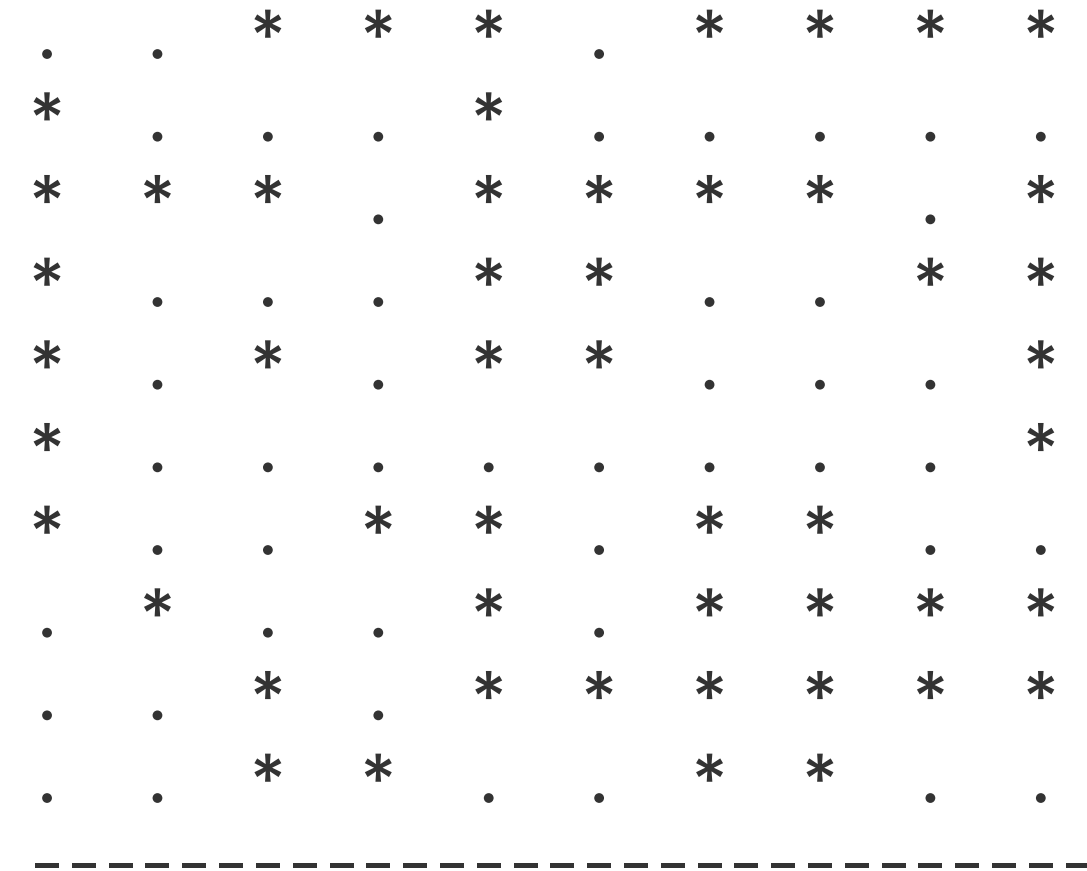
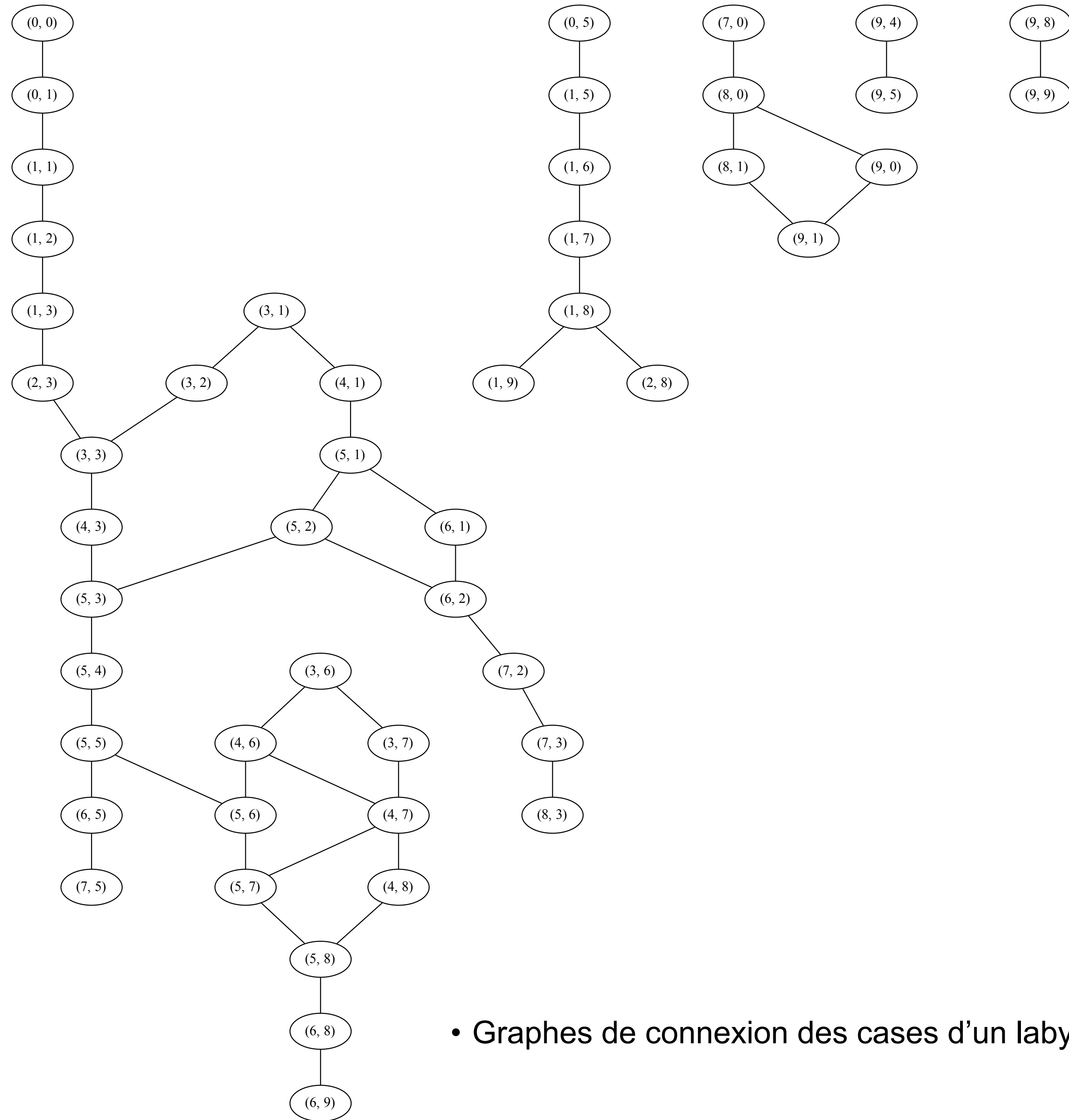
Graphes

- Graphes de connexion des cases d'un labyrinthe

```
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 1)

- Représentation par matrice d'adjacence

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

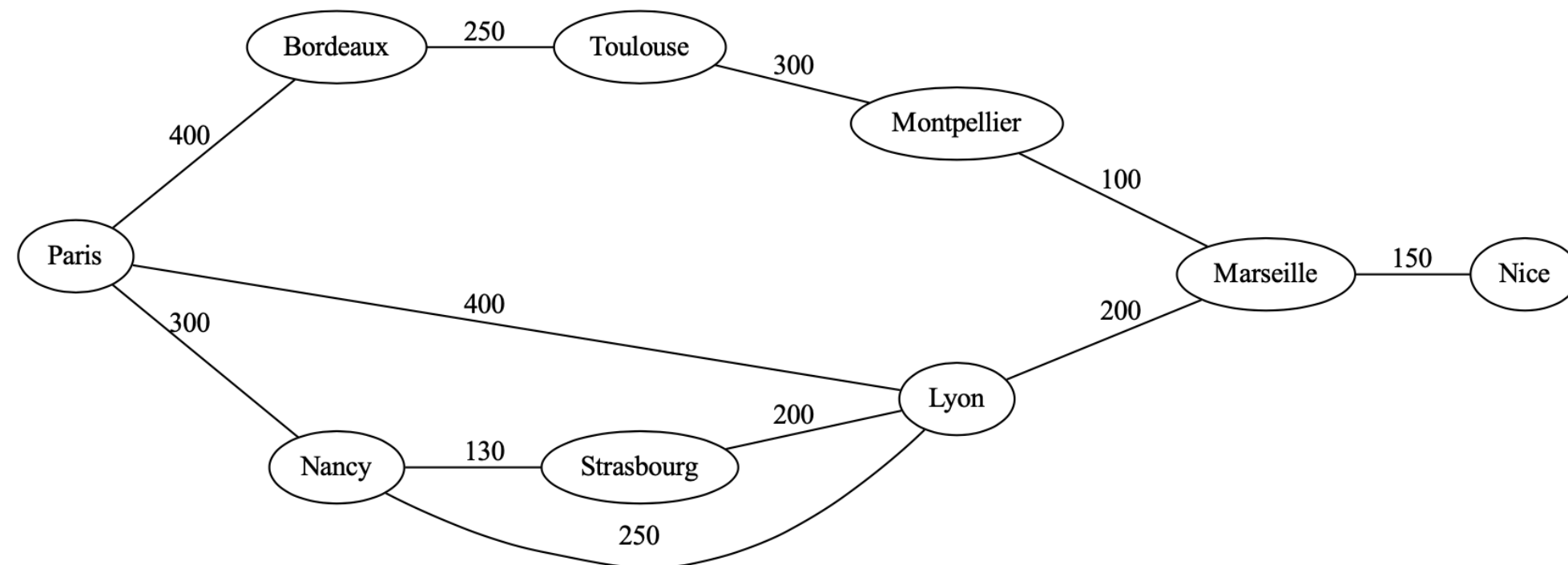
```
let g = let n = Array.length (villes) in  
        Array.make_matrix n n None ;;
```

```
g.(0).(1) = Some 400 ;;
```

```
g.(0).(5) = Some 300 ;;
```

```
g.(0).(7) = Some 400 ;;
```

```
...
```



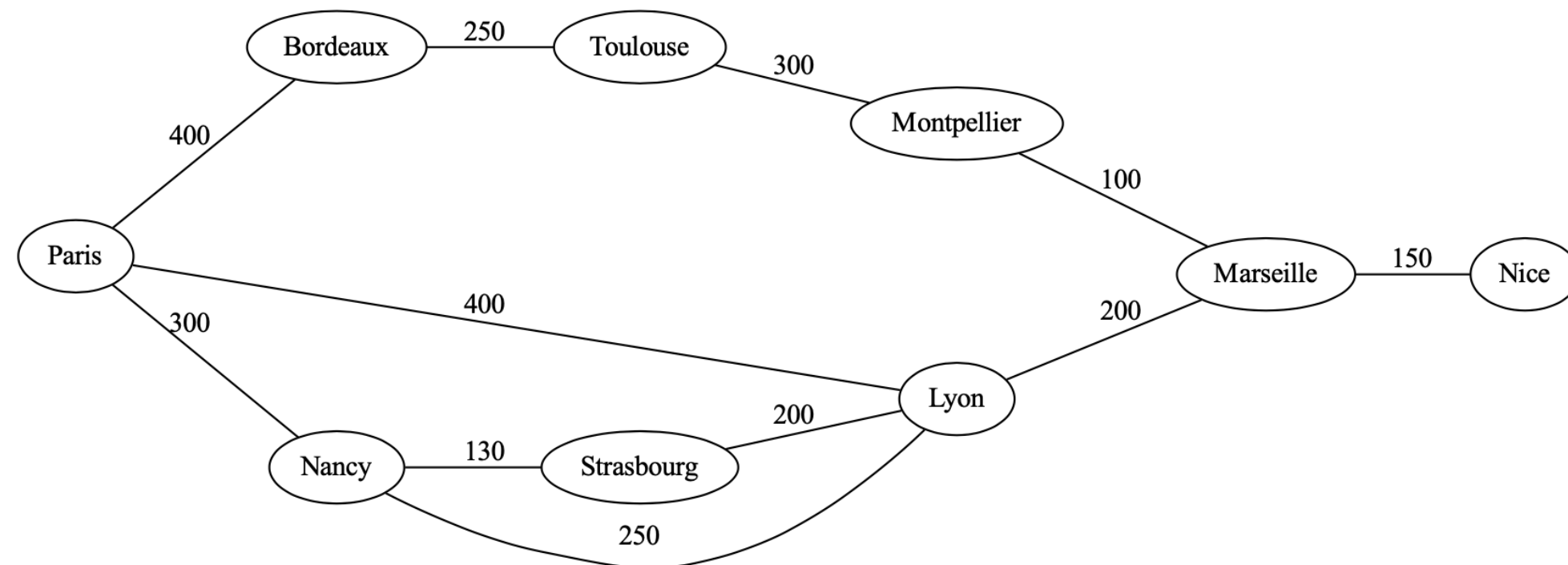
	0	1	2	3	4	5	6	7	8
0:		400				300		400	
1:	400		250						
2:		250		300					
3:			300		100				
4:				100				200	150
5:	300						130	250	
6:						130		200	
7:	400				200	250	200		
8:					150				

Graphes (représentation 2)

- Représentation par tableau de listes d'adjacence

```
module type GRAPHV = sig
  type t
  type vertex
  type weight
  val order : t -> int
  val make : int -> t
  val add_edge : t -> vertex -> weight -> vertex -> unit
  val succ : t -> vertex -> (vertex * weight) list
end ;
```

```
module Graph: GRAPHV = struct
  type vertex = int
  type weight = int
  type t = (vertex * weight) list array
  let make n = Array.make n [ ]
  let order g = Array.length g
  let add_edge g x w y = g.(x) <- (y,w) :: g.(x)
  let succ g x = g.(x)
end ;;
```

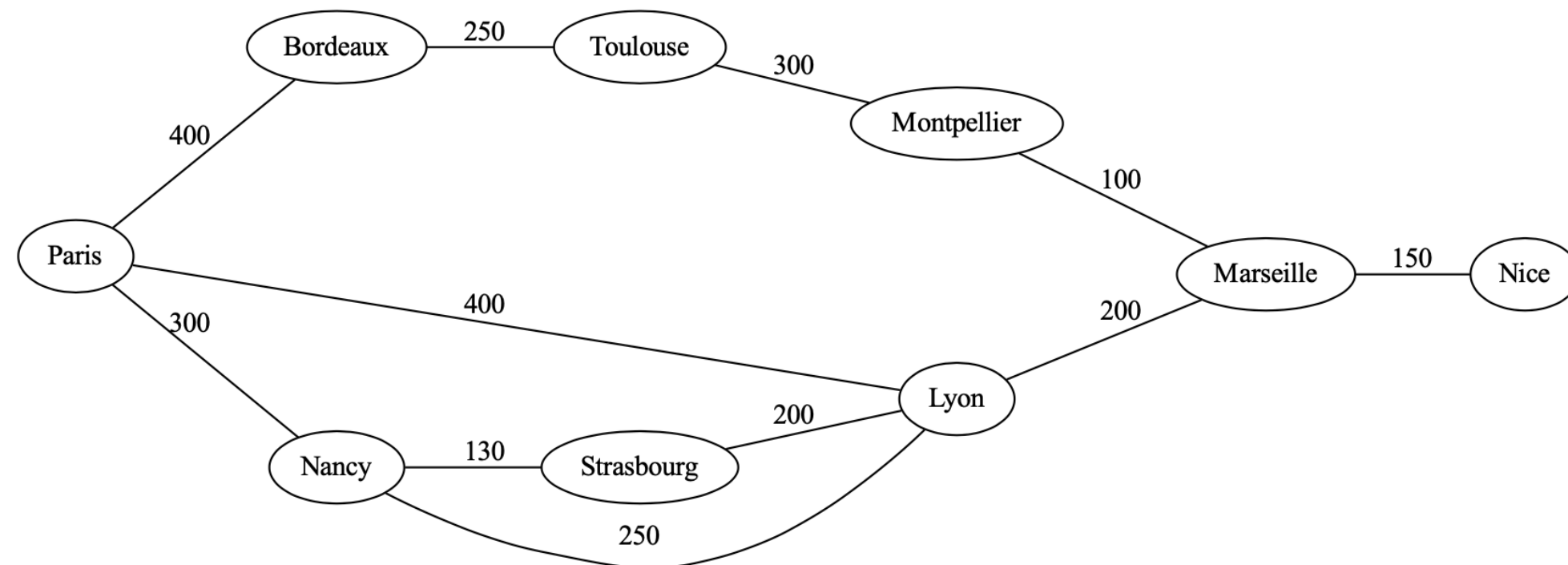


- Représentation compacte de la matrice de connexion

Graphes (représentation 2)

- Représentation par tableau de listes d'adjacence

```
module type GRAPHV = sig
  type t
  type vertex
  type weight
  val order : t -> int
  val make : int -> t
  val add_edge : t -> vertex -> weight -> vertex -> unit
  val succ : t -> vertex -> (vertex * weight) list
end ;
```



```
module G = Graph;;
```

```
let g = let n = Array.length (villes) in G.make n ;;
let add_edges g =
  List.iter (fun (i, j, w) -> G.add_edge g i w j) ;;
```

```
add_edges g
[0,5,300; 0,7,400; 0,1,400;
 1,2,250; 1,0,400;
 2,3,300; 2,1,250;
 3,4,100; 3,2,300;
 4,8,150; 4,7,200; 4,3,100;
 5,7,250; 5,6,130; 5,0,300;
 6,7,200; 6,5,130;
 7,4,200; 7,5,250; 7,6,200; 7,0,400;
 8,4,150 ] ;;
```

- Représentation compacte de la matrice de connexion

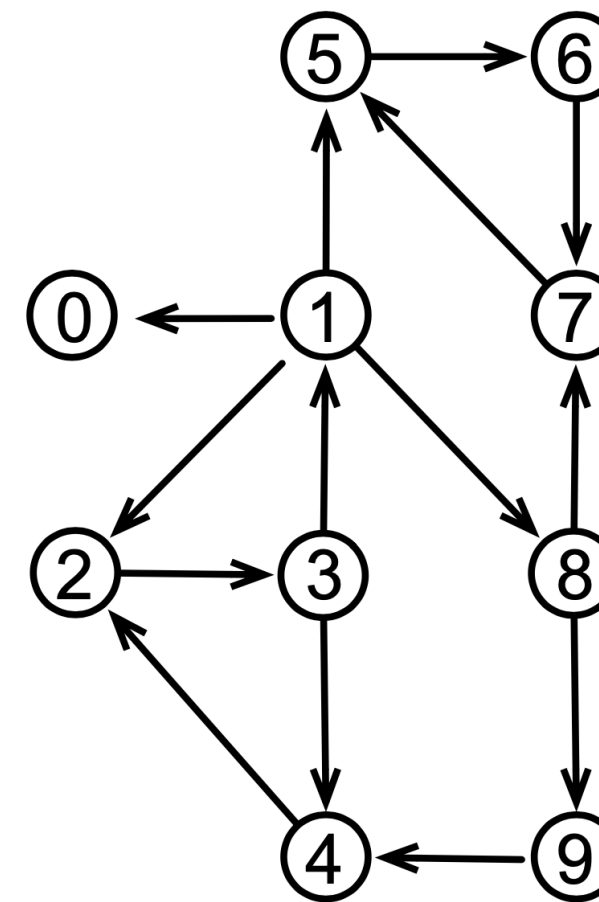
Graphes (représentation 2)

- Représentation par tableau de listes d'adjacence (connexions seules)

```
module type GRAPH = sig
  type t
  type vertex = int
  val order : t -> int
  val make : int -> t
  val add_edge : t -> vertex -> vertex -> unit
  val succ : t -> vertex -> vertex list
end ;;
```

```
module Graph: GRAPH = struct
  type vertex = int
  type t = vertex list array
  let make n = Array.make n [ ]
  let order g = Array.length g
  let add_edge g x y = g.(x) <- y :: g.(x)
  let succ g x = g.(x)
end ;;
```

```
module G = Graph ;;
```



```
let add_edges g =
  List.iter (fun (i, j) ->
    Graph.add_edge g i 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];;
```


Graphes (représentation 2bis)

- Représentation par tableau de listes d'adjacence (connexions seules et sous-module des sommets)

```
module type GRAPH = sig
  module V : VERTEX
  type vertex = V.t
  type t
  val order : t -> int
  val make : int -> t
  val add_edge : t -> vertex -> vertex -> unit
  val succ : t -> vertex -> vertex list
end ;;
```

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

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


Graphes (représentation 3)

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

```
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
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)
end ;;
```

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

Graphes (représentation 3)

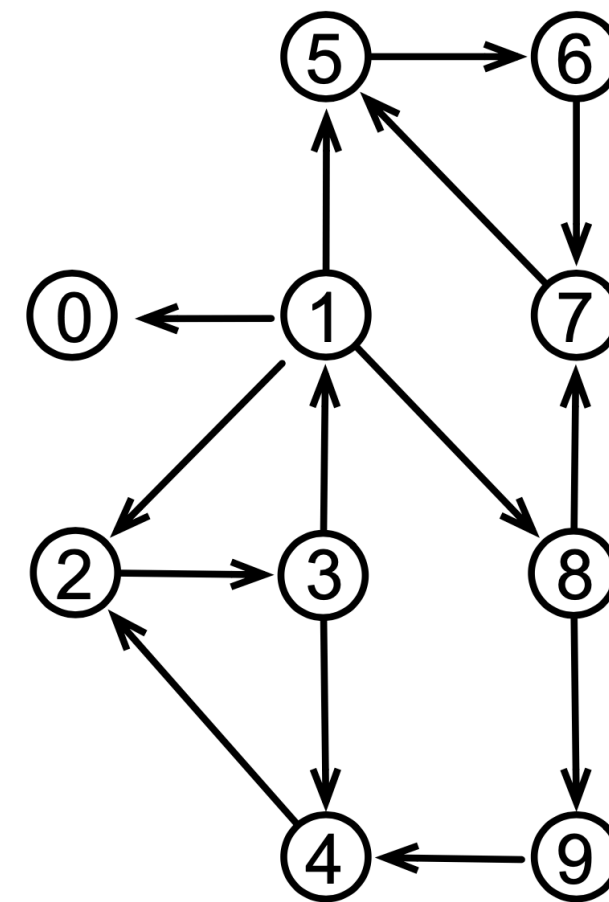
- Représentation par tableau de listes d'adjacence (connexions seules et en abstrayant les sommets)

```
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) ;;
```

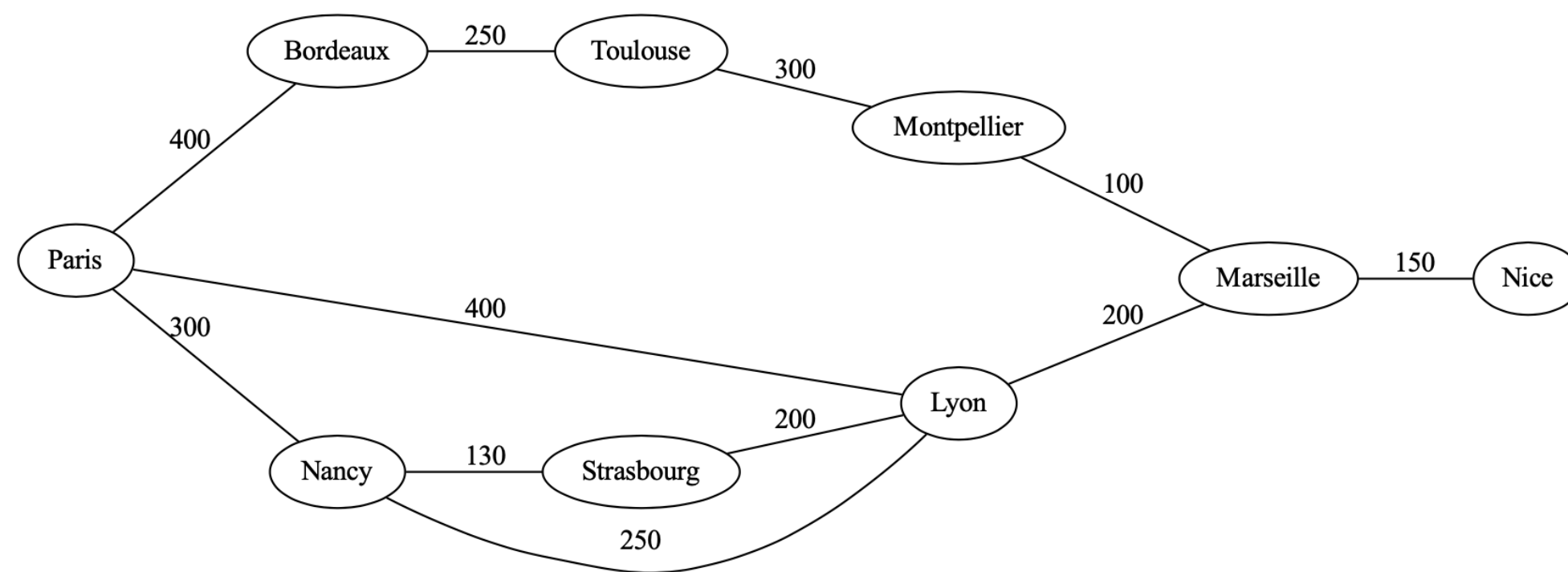
```
let add_edges g =  
  List.iter (fun (v1, v2) ->  
    G.add_edge g (Villes.lab (pos v1 villes))  
              (Villes.lab (pos v2 villes)) );;
```

```
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 ;;
```

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 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 ;;
```


Parcours de graphes

- Parcours en profondeur d'abord de tous les sommets

```
let dfs g f =  
  let n = G.order g in  
  let visited = Array.make n false in  
  let rec dfs1 x =  
    if not visited.(V.ord x) then begin  
      visited.(V.ord x) <- true;  
      f x;  
      G.iter_succ dfs1 g x  
    end in  
  G.iter_vertex dfs1 g ;;
```

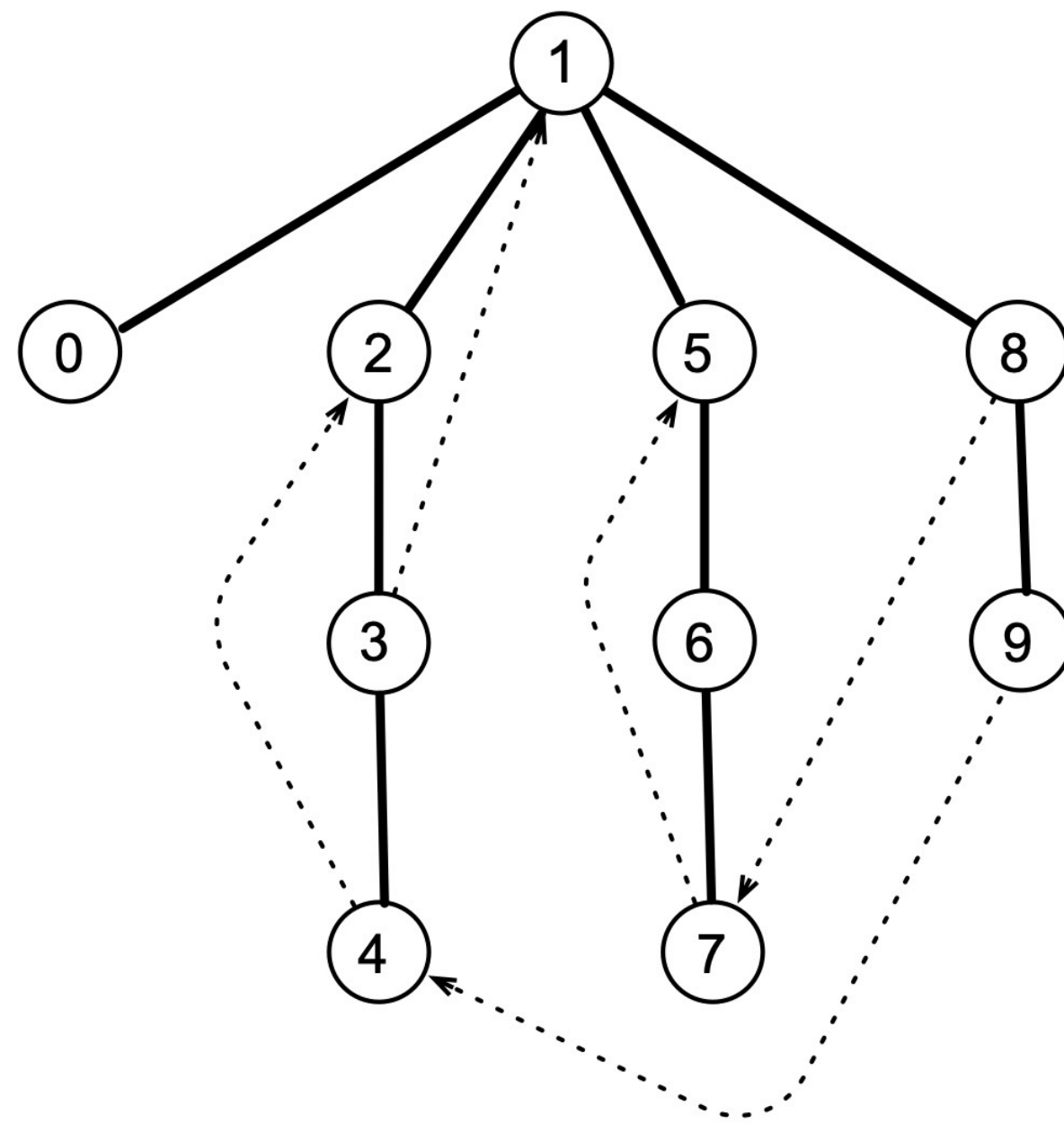
(* On marque les noeuds
visités pour ne pas boucler *)

```
dfs g (fun x -> Printf.printf "%d\n" (V.ord x));;
```

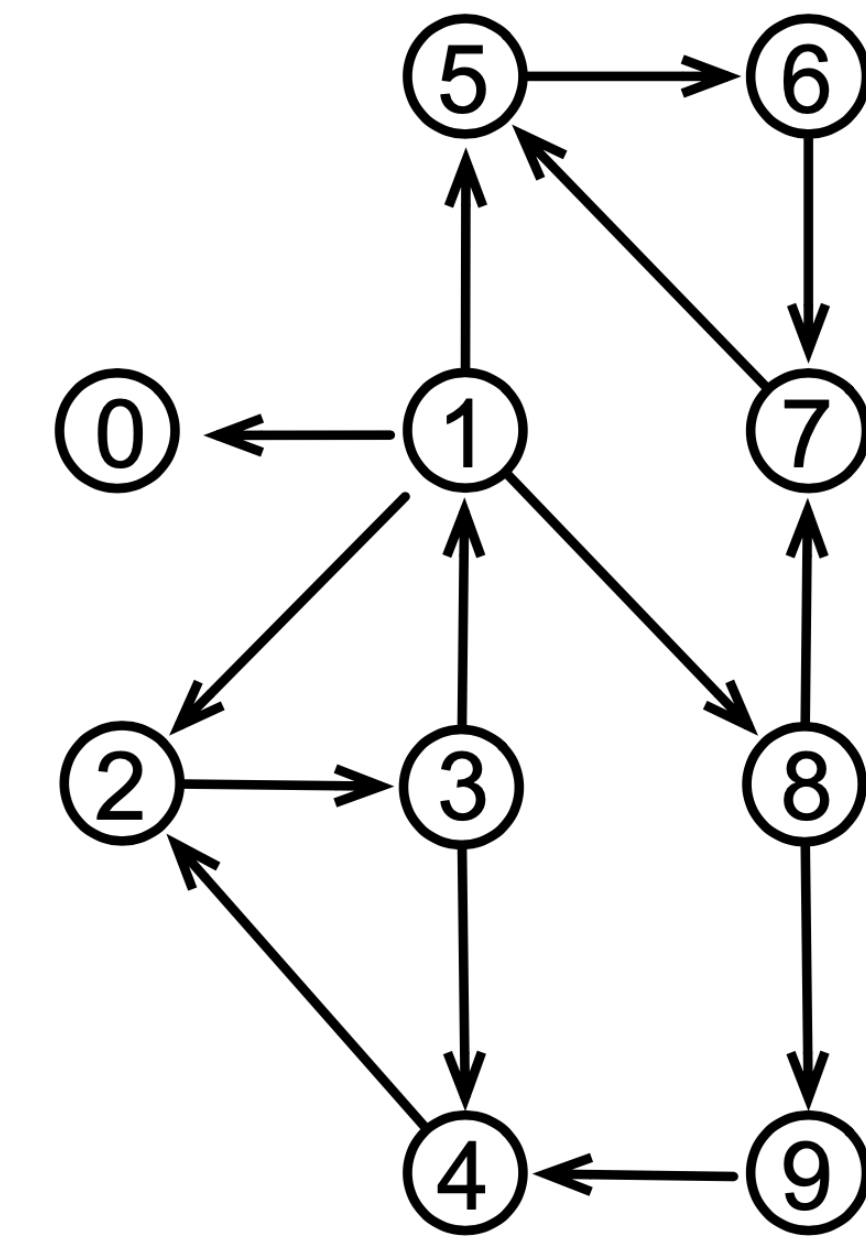
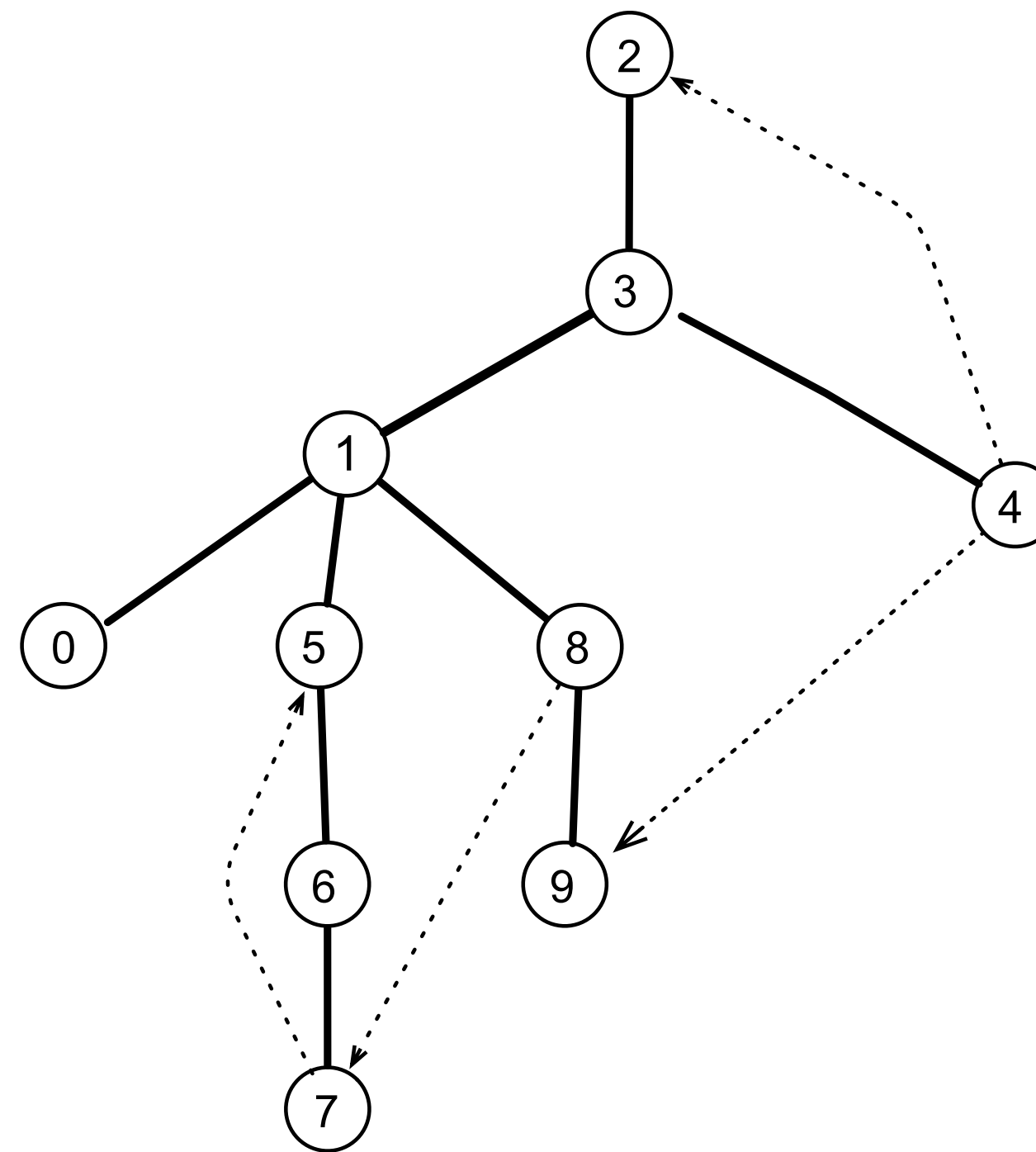
- Il existe aussi le parcours en largeur d'abord (avec une file FIFO)

Arbre de recouvrement

- un arbre de recouvrement (*spanning tree*) est l'arbre des appels récursifs de dfs
- un arbre (une forêt) de recouvrement recouvre tous les sommets d'un graphe



2 arbres de recouvrement



graphe

Parcours de graphes

- Parcours en profondeur d'abord de tous les sommets itératif (avec une pile)

```
let dfs' g f =  
  let n = G.order g in  
  let visited = Array.make n false in  
  let s = Stack.create () in  
  let push_unvisited x =  
    if not visited.(V.ord x) then  
      begin visited.(V.ord x) <- true; Stack.push x s end in  
  let dfs1 () =  
    while not (Stack.is_empty s) do  
      let x = Stack.pop s in  
      f x;  
      G.iter_succ push_unvisited g x ;  
    done in  
  G.iter_vertex (fun x -> push_unvisited x; dfs1()) g ;;  
  
dfs' g (fun x -> Printf.printf "%d\n" (V.ord x));;
```

(* On marque les noeuds visités pour ne pas boucler *)

- Il existe aussi le parcours en largeur d'abord (avec une file FIFO)

Parcours de graphes

- Parcours en largeur d'abord de tous les sommets itératif (avec une file d'attente)

```
let bfs g f =  
  let n = G.order g in  
  let visited = Array.make n false in  
  let q = Queue.create () in  
  let push_unvisited x =  
    if not visited.(V.ord x) then  
      begin visited.(V.ord x) <- true; Queue.push x q end in  
  let bfs1 () =  
    while not (Queue.is_empty q) do  
      let x = Queue.take q in  
      f x;  
      G.iter_succ push_unvisited g x ;  
    done in  
  G.iter_vertex (fun x -> push_unvisited x; bfs1()) g ;;  
  
bfs g (fun x -> Printf.printf "%d\n" (V.ord x));;
```

(* On marque les noeuds visités pour ne pas boucler *)

- Il existe aussi le parcours en largeur d'abord (avec une file FIFO)

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