Autour des graphes

option informatique

Composantes fortement connexes

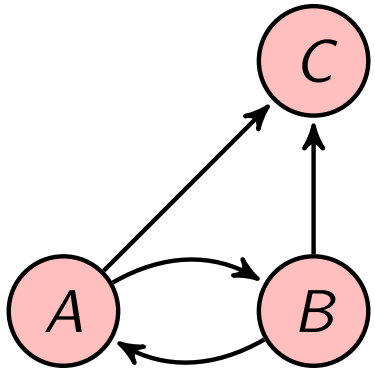
Composantes fortement connexes

Une **composante fortement connexe** d'un graphe orienté est un sous-ensemble **maximal** de sommets tels que deux quelconques d'entre eux soient reliés par un chemin.

- Les composantes fortement connexes d'un graphe forment une **partition du graphe**.
- Un **graphe** est **fortement connexe** si et seulement si il a une seule composante fortement connexe.
- Le sous-graphe induit par une composante fortement connexe est fortement connexe.

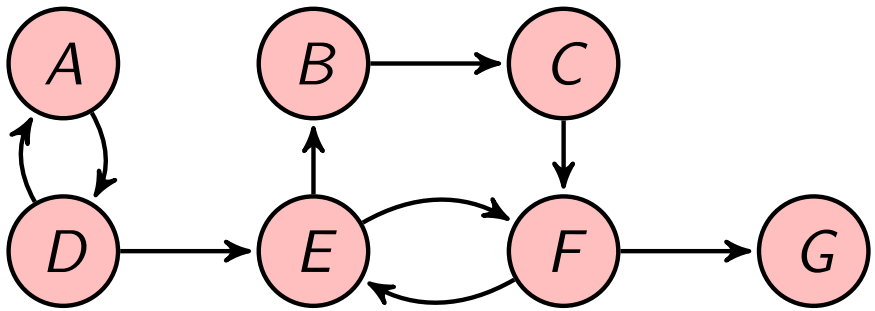
Composantes fortement connexes

Le graphe suivant a deux composantes fortement connexes.



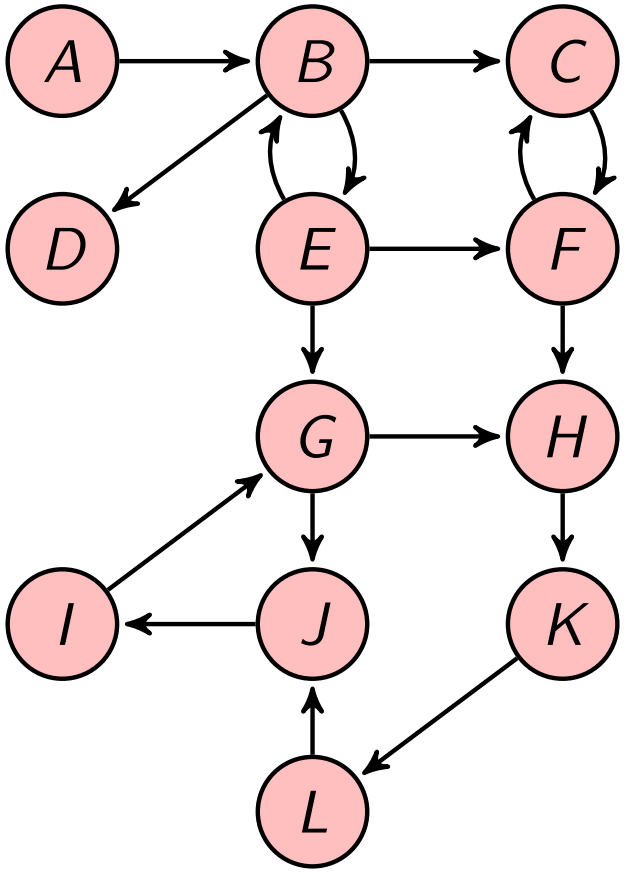
$\{A, B\}$ $\{C\}$

Le graphe suivant a trois composantes fortement connexes.

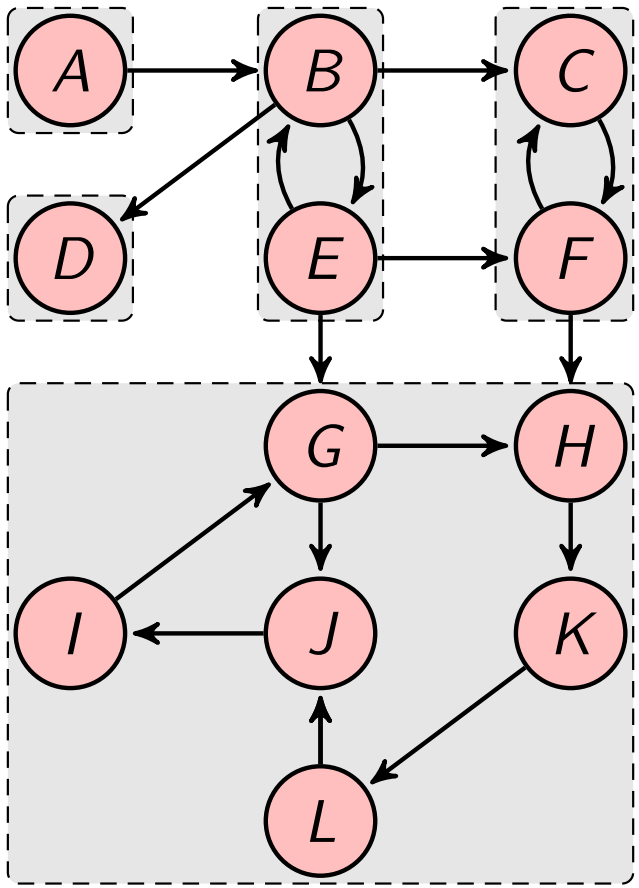


$\{A, D\}$ $\{B, C, E, F\}$ $\{G\}$

Décomposition en composantes fortement connexes



Décomposition en composantes fortement connexes



Définition d'un graphe

Type et fonctions élémentaires

```
(* type défini par un dictionnaire *)
type 'a graphe = ('a, 'a list) Hashtbl.t;;
(* voisins d'un sommet *)
let voisins = Hashtbl.find;;
(* existence d'une arête entre deux sommets *)
let existeArc graphe alpha beta =
  List.mem beta (Hashtbl.find graphe alpha);;
```

Création d'un graphe

```
let gr = Hashtbl.create 97;;
Hashtbl.add gr 1 [ ];;
Hashtbl.add gr 2 [ 1; 5 ];;
Hashtbl.add gr 3 [ 4; 5; 6 ];;
Hashtbl.add gr 4 [ 2 ];;
Hashtbl.add gr 5 [ 2; 4 ];;
Hashtbl.add gr 6 [ 3 ];;
```

Quelques fonctions pratiques (mais pas indispensables)

```
let rec print_list lst = match lst with
  | [] -> ()
  | t::q -> print_int t; print_string " "; print_list q;;

let print_values c k =
  print_string "voisins du sommet ";
  print_int c;
  print_string " -> ";
  print_list k;
  print_newline();;

let display hash_tbl =
  print_newline();
  print_string "----------";
  print_newline();
  print_string "graphe";
  print_newline();
  print_string "----------";
  print_newline();
  Hashtbl.iter print_values hash_tbl;
  print_string "----------";;
```

Composantes connexes

```
(* composantes connexes d'un graphe non-orienté *)
let composantes graphe liste_sommets =
  let deja_visite = Hashtbl.create 97 in
  (* Détermination d'une composante connexe *)
  (* dfs : 'a list -> 'a list *)
  let rec dfs = function
    | [] -> []
    | t::q when Hashtbl.mem deja_visite t -> dfs q
    | t::q -> Hashtbl.add deja_visite t true;
              t :: dfs (voisins graphe t @ q)
  (* Égrenne une liste de sommets et cherche *)
  (* les composantes connexes des sommets non visités *)
  and composantes = function
    | [] -> []
    | t::q when Hashtbl.mem deja_visite t -> composantes q
    | t::q -> let composante_de_t = dfs [t]
              in composante_de_t :: composantes q
  in composantes liste_sommets;;
```

Parcours DFS d'un graphe orienté

Principe

```
(* parcours dfs d'un graphe orienté - principe *)
 let dfs graphe action origine =
  (* Un dictionnaire pour mémoriser les sommets visités *)
  let deja_visite = Hashtbl.create 97 in
  let rec aux s =
    (* Si le sommet n'a pas encore été visité *)
    if not (Hashtbl.mem deja_visite s) then
    begin
      action s;                              (* On le visite *)
      Hashtbl.add deja_visite s true;        (* On le marque *)
      List.iter aux (voisins graphe s)       (* Et on considère *)
    end;                                     (* ses voisins *)
  in aux origine;;
```


- arrtes@gmail.com

Parcours DFS d'un graphe orienté

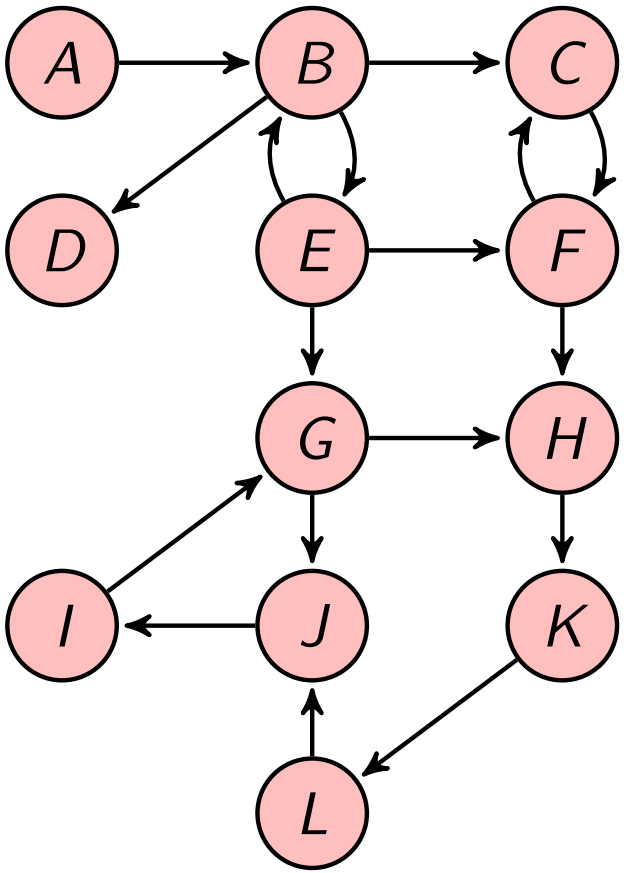
Une utilisation pratique

```
(* parcours dfs d'un graphe orienté avec affichage des nœuds visités *)
let dfs graphe origine =
  (* Un dictionnaire pour mémoriser les sommets visités *)
  let deja_visite = Hashtbl.create 97
  and lst = ref [] in
  let rec aux s =
    (* Si le sommet n'a pas encore été visité *)
    if not (Hashtbl.mem deja_visite s) then
    begin
      Hashtbl.add deja_visite s true;
      lst := s::!lst;
      List.iter aux (voisins graphe s)
    end;
  in aux origine;
  print_string "pacours en profondeur depuis le nœud ";
  print_int origine;
  print_string " - > ";
  print_list (List.rev !lst);;
```

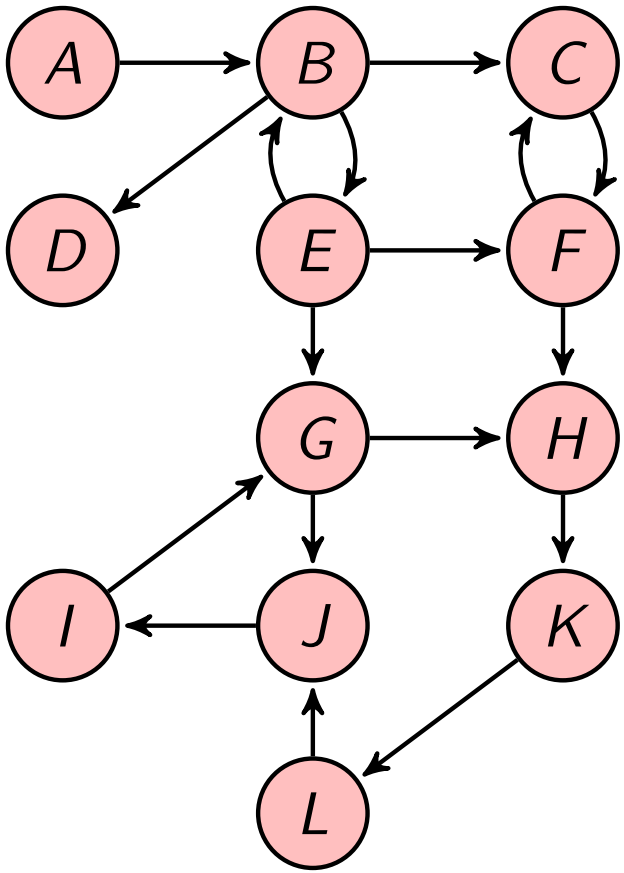

- arrtes@gmail.com

Graphe transposé

Le **transposé d'un graphe orienté** G est le graphe ayant les mêmes sommets que G et les arêtes de G parcourues en sens inverse.



(a) Graphe orienté...



(b) ...et sont transposé

Algorithme de Kosaraju

- L'**algorithme de Kosaraju** détermine les composantes fortement connexes d'un graphe orienté.
- Il effectue deux parcours en profondeur du graphe et de son transposé.
- Sa complexité temporelle est linéaire en la taille du graphe.

Algorithme de Kosaraju

Soit G un graphe orienté.

- Effectuer un parcours en profondeur sur G.
- Effectuer un parcours en profondeur sur ${}^tG$ (graphe transposé de G) en explorant les sommets dans l'ordre inverse de l'ordre suffixe donné par le premier parcours en profondeur.

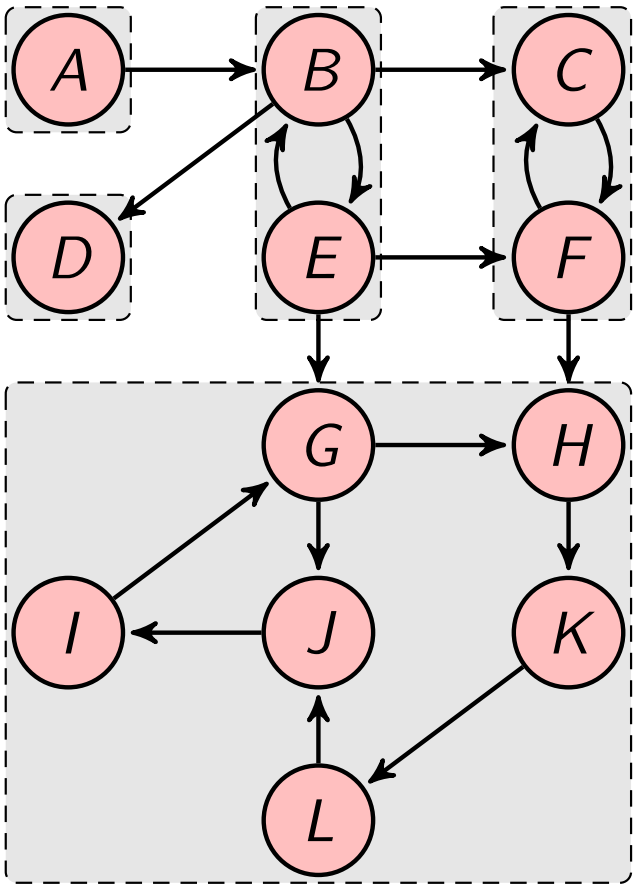
Les arbres produits par le second parcours sont les composantes fortement connexes de G.

Graphes orientés acycliques

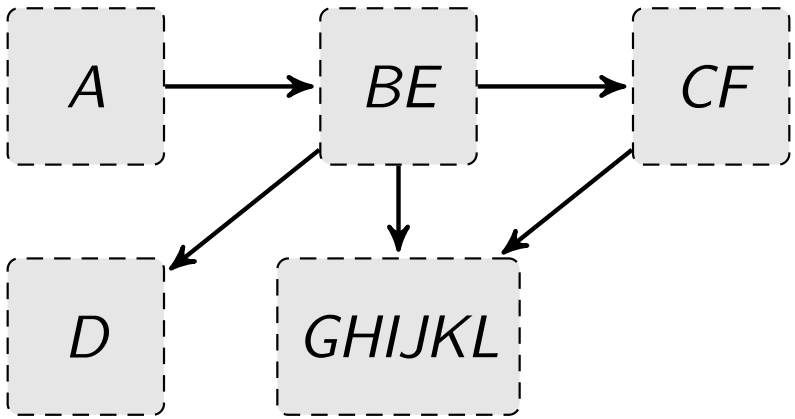
DAG

- Un **graphe orienté acyclique** (DAG) est un graphe orienté qui ne possède pas de circuit, c'est-à-dire une suite d'arcs consécutifs (chemin) dont les deux sommets extrémités sont identiques.
- En remplaçant chaque composante fortement connexe d'un graphe par un (meta-)sommet, on obtient un DAG.
- Les DAG permettent de modéliser les relations causales, les structures hiérarchiques et les dépendances chronologiques.

Meta-graphe



(a) Graphe orienté...

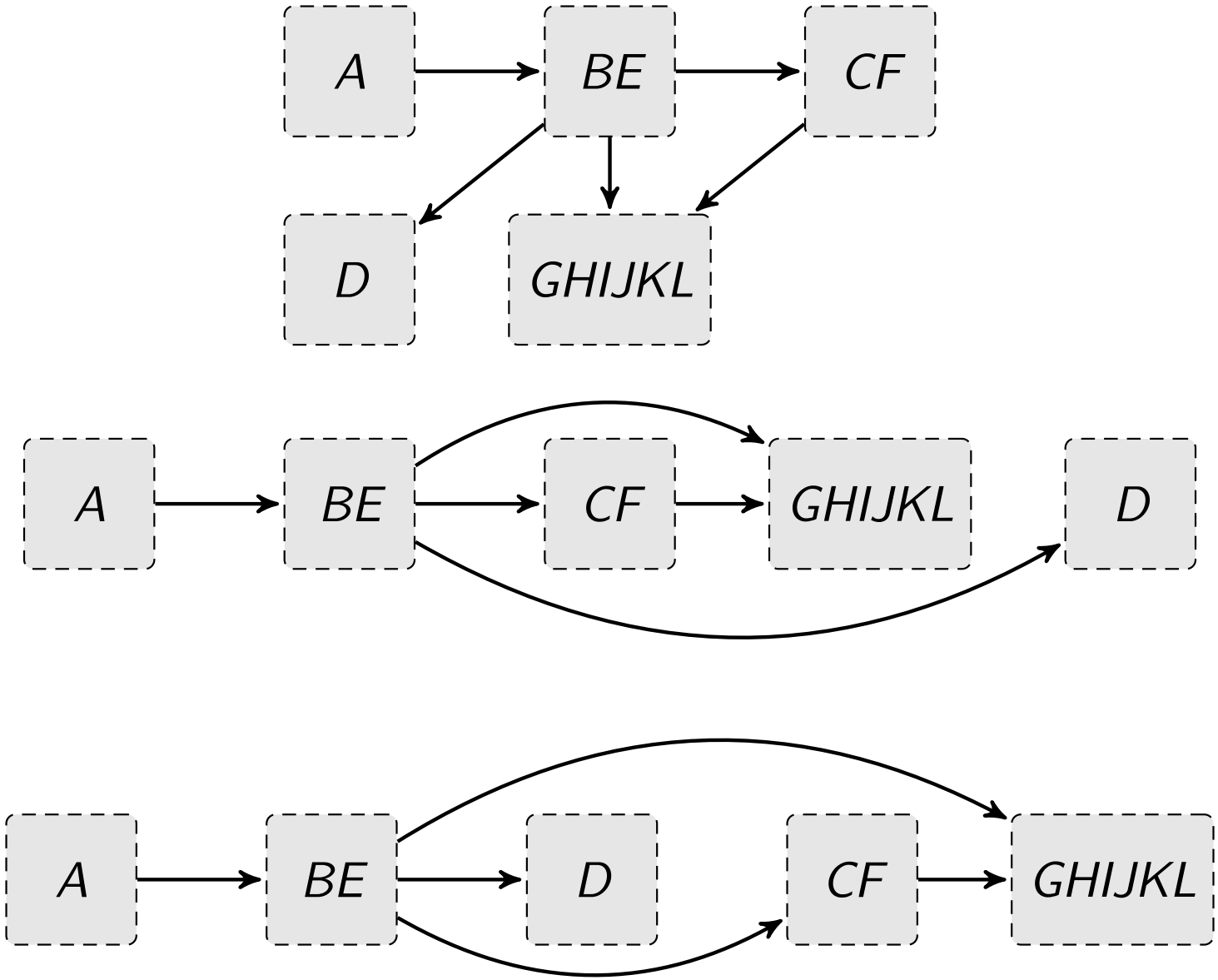


(b) ...et son meta-graphe (DAG)

Tri topologique

- Un **tri topologique** d'un DAG est un ordre linéaire des sommets du graphe. Si le graphe contient l'arête (u, v), alors u apparaît avec v.
- Le tri topologique peut être représenté par un schéma linéaire des sommets tels que toutes les arêtes soient orientées dans un même sens.
- Noter quel le tri topologique d'un DAG n'est pas nécessairement unique.

Tri topologique

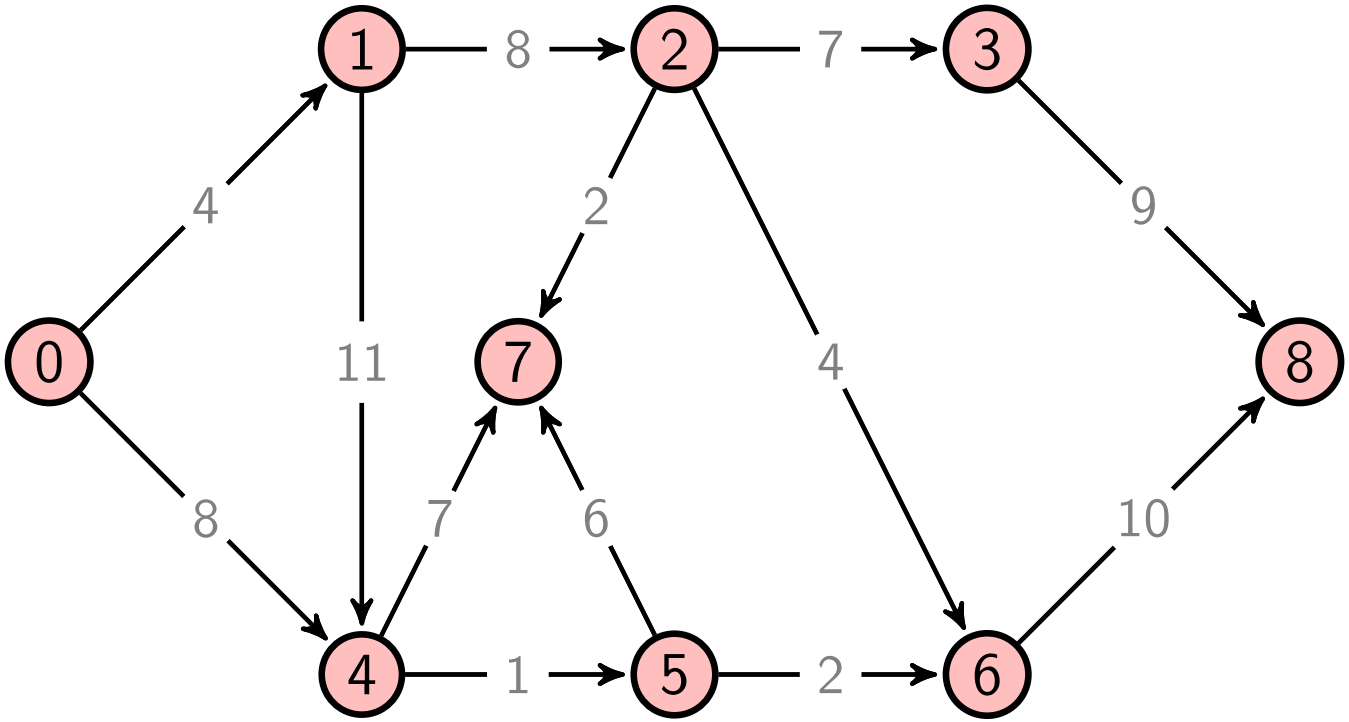


Arbres couvrants minimaux

Objectifs

Étant donné un graphe connexe $G = (V, E)$ non orienté pondéré, comment construire un sous-graphe de G de poids minimal qui recouvre exactement G ? C'est le problème de la recherche de l'**arbre couvrant minimal** dont l'**algorithme de Prim** est une réponse.

Graphe exemple



$G = (V, E)$

V : ensemble des sommets

E : ensemble des arêtes pondérées

Graphe partiel et arbre couvrant

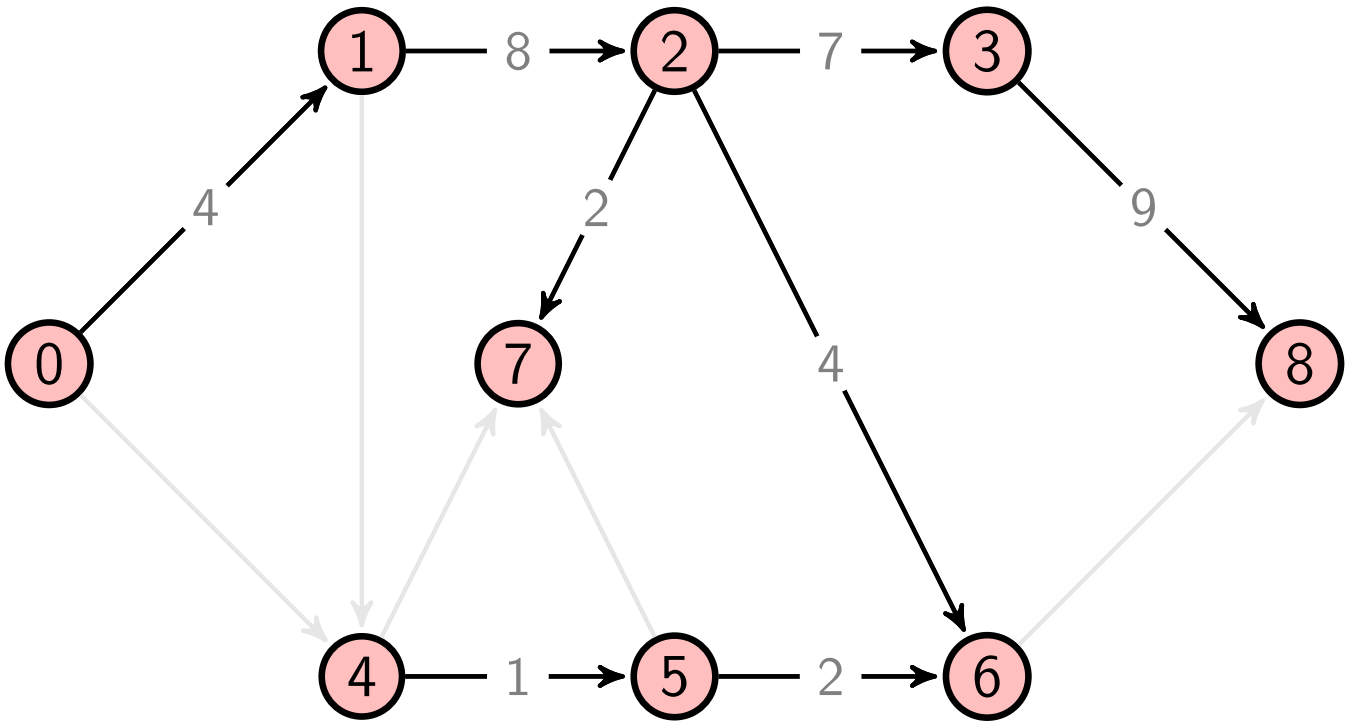
- Un **graphe partiel** $G_p = (V_p, E_p)$ d'un graphe $G = (V, E)$ a les mêmes sommets V et un sous-ensemble E_p des arêtes E de G.

$$V_p = V \qquad E_p \subset E$$

- Un **arbe couvrant** d'un graphe $G = (V, E)$ est un graphe partiel sans cycle.

Arbre couvrant minimal

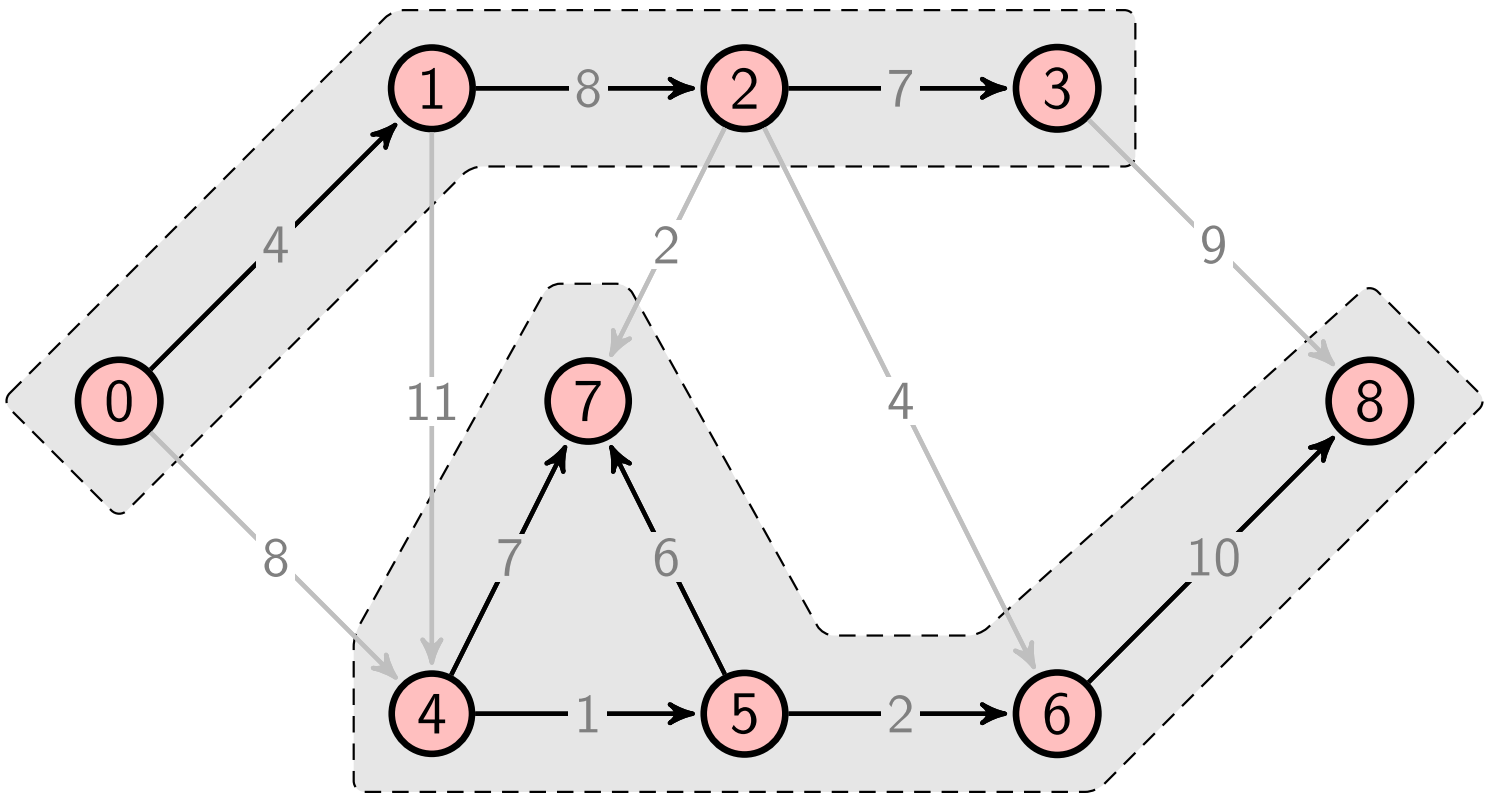
Un **arbre couvrant minimal (ACM)** T d'un graphe $G = (V, E)$ est un arbre couvrant dont la somme des poids de ses arêtes est minimale.



Poids total : $\omega(T) = 37$

Coupure

Une **coupure** d'un graphe $G = (V, E)$ est un ensemble d'arêtes dont une extrémité est dans un sous-ensemble $U \subset V$ et l'autre extrémité est dans $V \setminus U$. La coupure est notée $(U, V \setminus U)$.



Coupure matérialisée par les arêtes grises.

Algorithme de Prim

- Cet algorithme met en œuvre une **stratégie gloutonne**.
- À partir d'un sommet quelconque, l'algorithme construit un ACM par **croissance progressive**.
- Si G est de taille n, un ACM de G comporte $n-1$ arêtes.
- L'ACM est unique si et seulement les poids du graphe sont tous distincts. Sinon, plusieurs ACM peuvent être associés à un graphe.

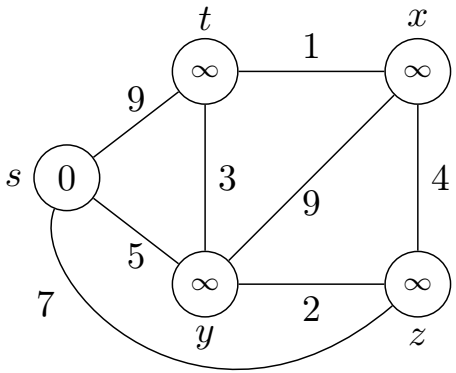
Algorithme

Soit $G = (V, E)$ de taille $n = |V|$.

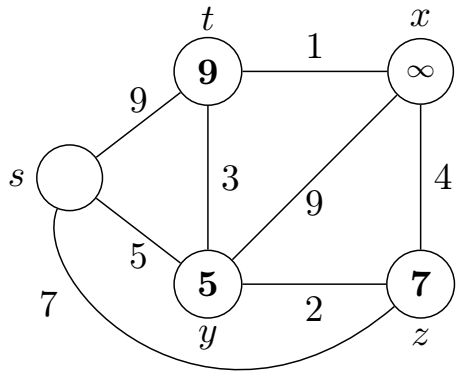
- Choisir un sommet $v_{i_0} \in V$ et poser $U_0 = \{v_{i_0}\}$.
- Dans la coupure $(U_0, V \setminus U_0)$, trouver une arête $\{v_{i_0}, v_{i_1}\}$ de poids minimal puis poser $U_1 = \{v_{i_0}, v_{i_1}\}$.
- Dans la coupure $(U_1, V \setminus U_1)$, trouver une arête $\{v, v_{i_2}\}$ de poids minimal où $v \in U_1$ puis poser $U_2 = \{v_{i_0}, v_{i_1}, v_{i_2}\}$.
- Répéter cette procédure jusqu'à obtenir l'ensemble U_{n-1}.

U_{n-1} est un **arbre couvrant minimal** de G.

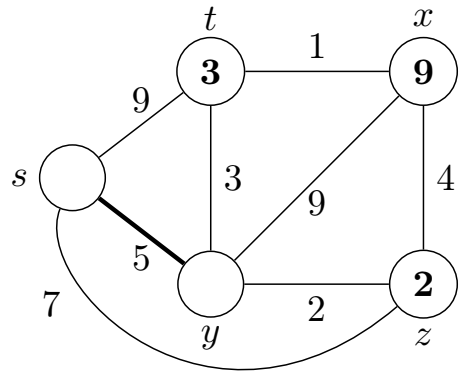
Exemple



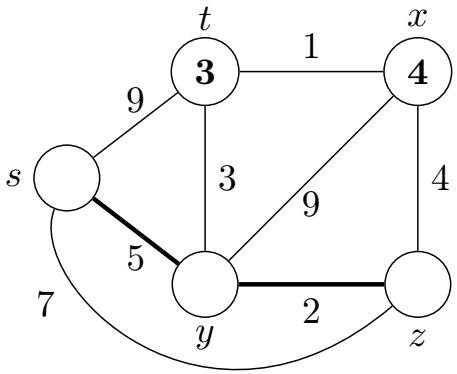
$H = \emptyset,\ F = \{s\}$



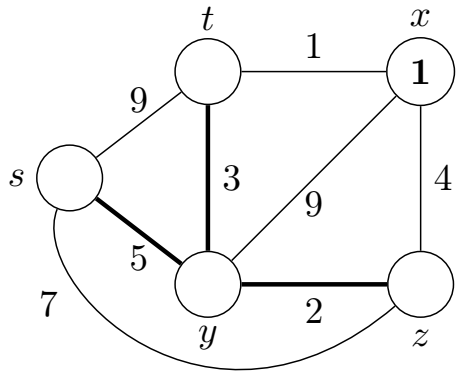
$H = \{s\},\ F = \{y, t\}$



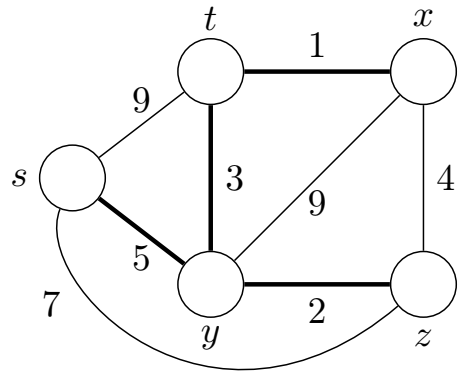
$H = \{s, y\},\ F = \{t, z, x\}$



$H = \{s, y, z\},\ F = \{t, x\}$



$H = \{s, y, z, t\},\ F = \{x\}$



$H = \{s, y, z, t, x\},\ F = \{\}$

Mise en œuvre de l'algorithme de Prim.
(poids minimal d'un ACM = 11)