

# Structures de Données & Algorithmes II

-

## Graphes

---

Pascal Mérindol (CM, TD, TP)

[merindol@unistra.fr](mailto:merindol@unistra.fr)

<http://www-r2.u-strasbg.fr/~merindol>

# Contenu

---

- **Les tris (~ 8h)**
- **Arbres & Forêts (~ 12h)**
- **Les graphes (12h ?)**
  - définitions & propriétés
  - opérations classiques, chemins, fermetures, équivalence, etc.
  - spécifications
  - représentations
  - algorithmes et complexité : parcours (largeur/profondeur), fermeture, chemins, etc.
    - recursif vs. itératif
- **Les tables (8h ?)**

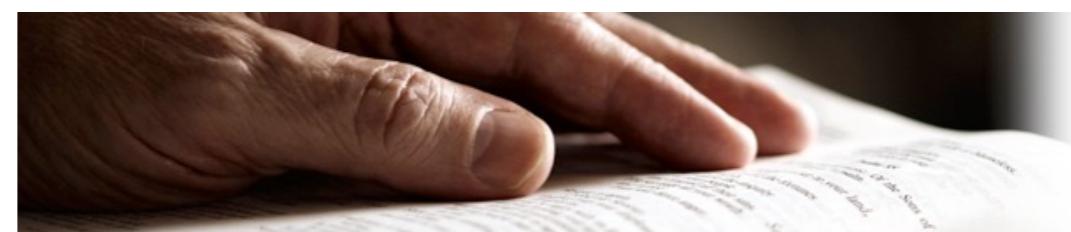
# Références

[http://en.wikipedia.org/wiki/Graph theory](http://en.wikipedia.org/wiki/Graph_theory)

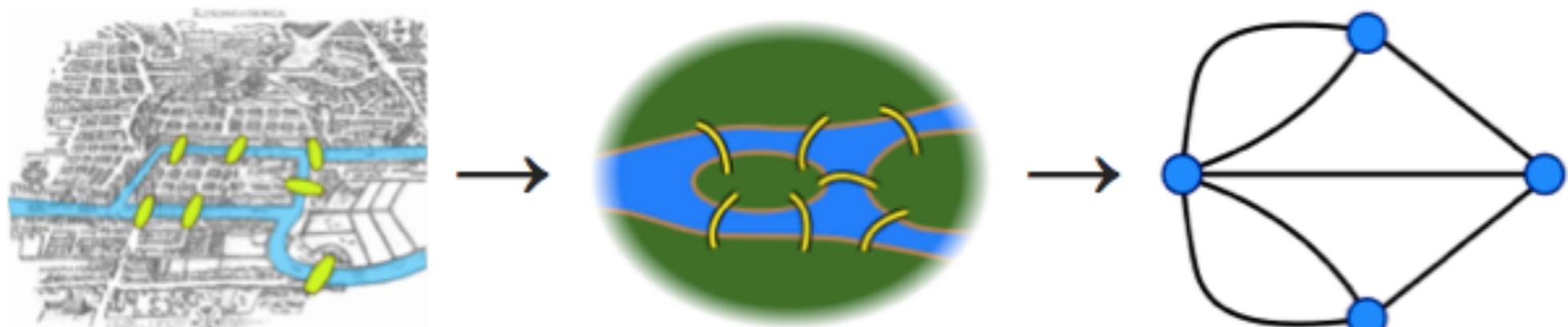
...



- \* ***Spécification algébrique, algorithmique et programmation*** : Jean-François Dufourd, Dominique Bechmann, Yves Bertrand.
- \* ***The Art of Computer Programming, Volume 4 (Chapter 7 in particular), Combinatorial Algorithms*** : Donald Knuth.
- \* ***Introduction to Algorithms*** : Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein.

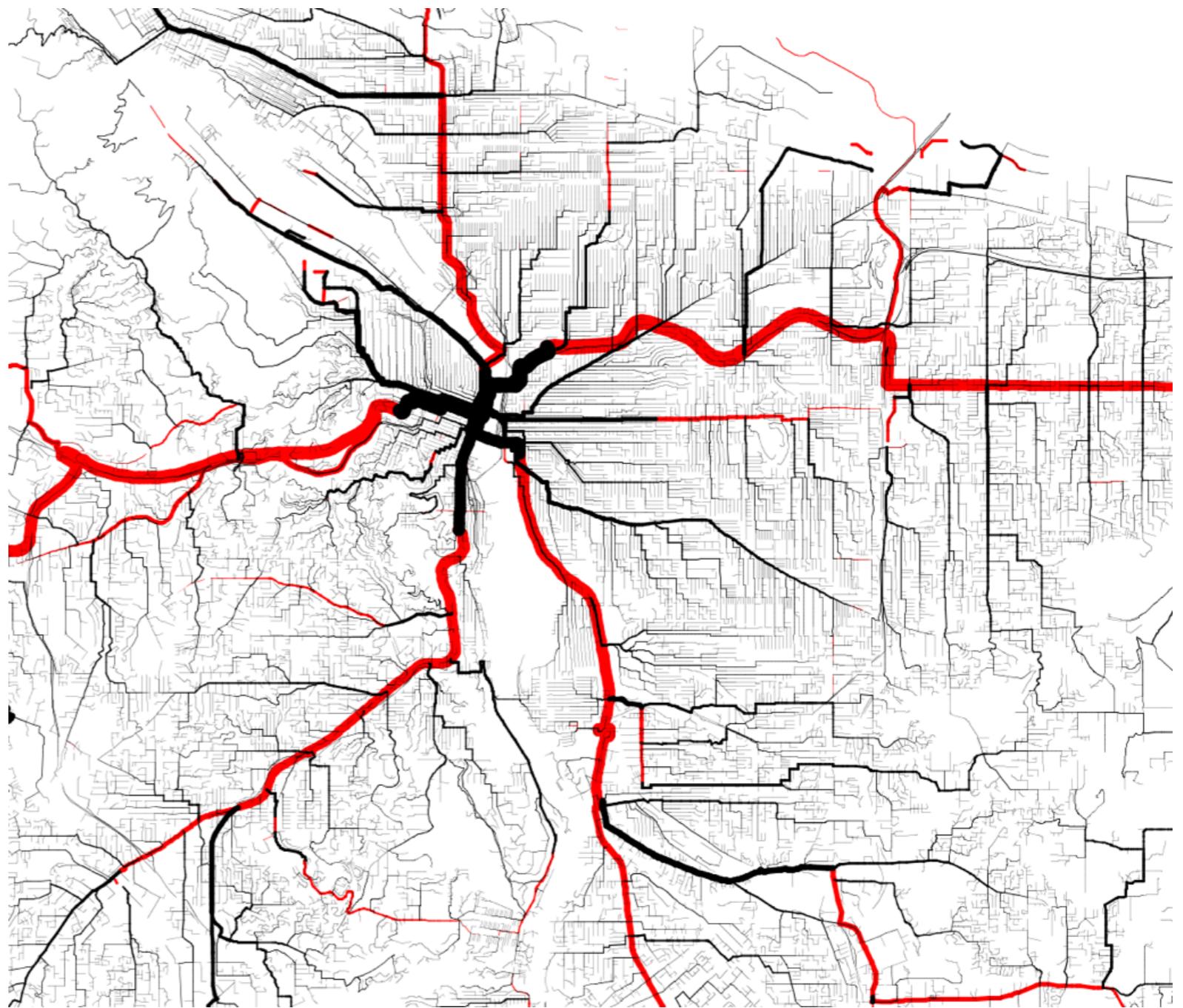


# Motivations

- **Les graphes sont partout...les graphes, c'est la vie !**
    - => L'UE Graphes : les graphes comme outil de résolution et modèle de formalisation
  - **C'est beau**
  - **C'est simple**
  - **C'est fort**
  - **Etc.**
  - **Et c'est surtout très utile dans presque tous les domaines de l'informatique et scientifique en général...**
  - **Mais c'est aussi bien pratique pour résoudre des énigmes, gagner des jeux, etc :)**
- 
- A diagram showing the abstraction process of a graph. It starts with a grayscale map of a city street layout with a river and green areas highlighted in yellow. An arrow points to a simplified version where the river is a blue line and the green areas are a green circle with yellow outlines. Another arrow points to a graph representation with four blue circular nodes and black lines connecting them in a complex network pattern.

# Quelques exemples

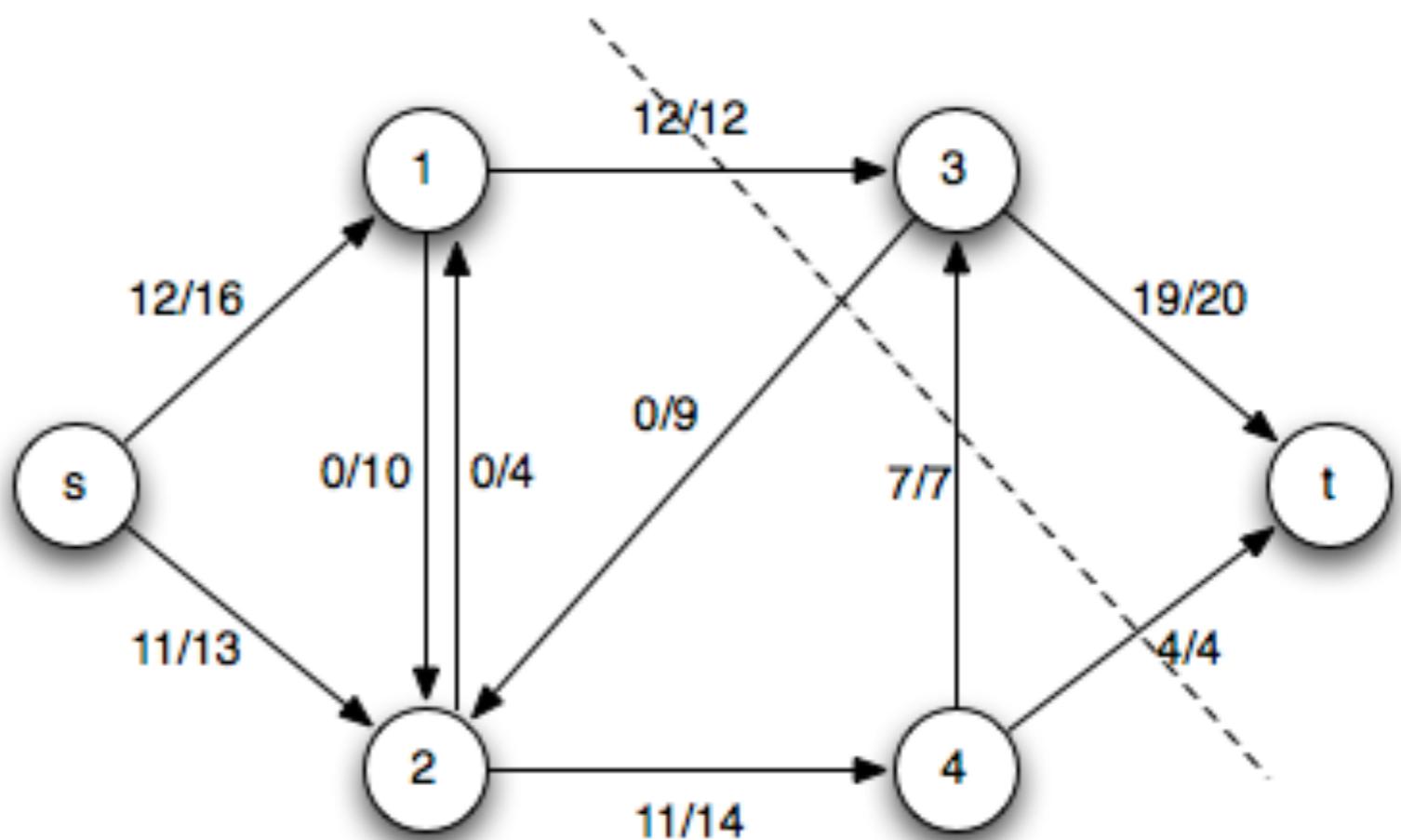
- **Calculer la meilleure route**
  - pondération des liens
  - métrique additive
    - $(\min, +)$



# Quelques exemples

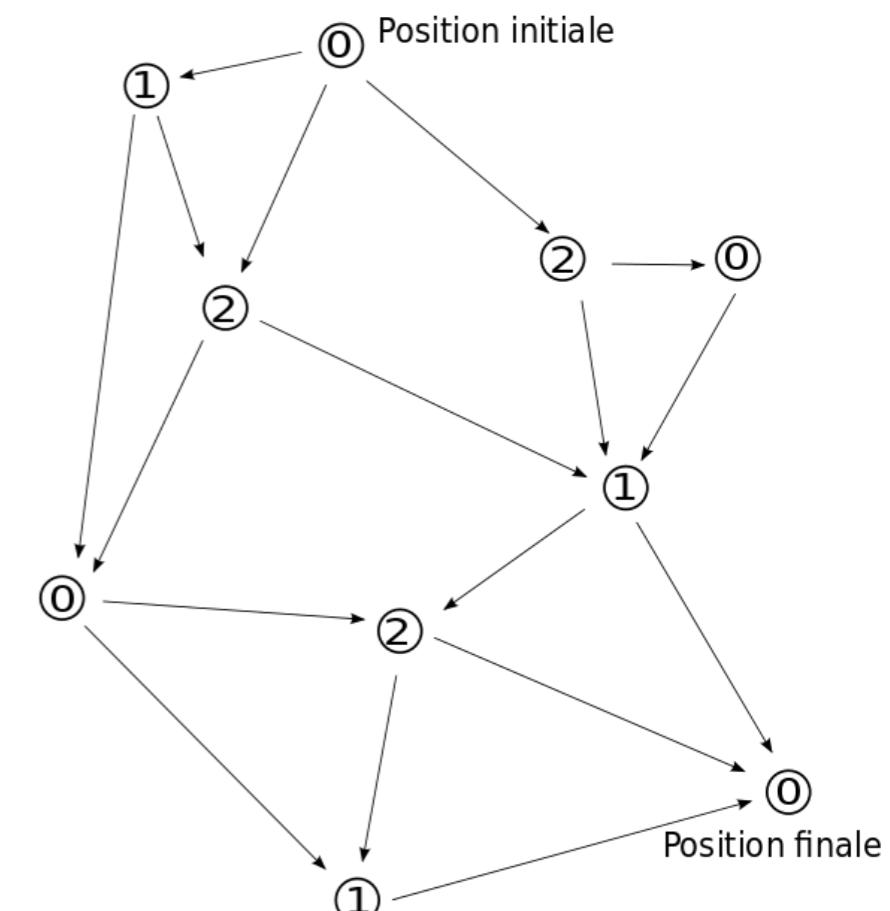
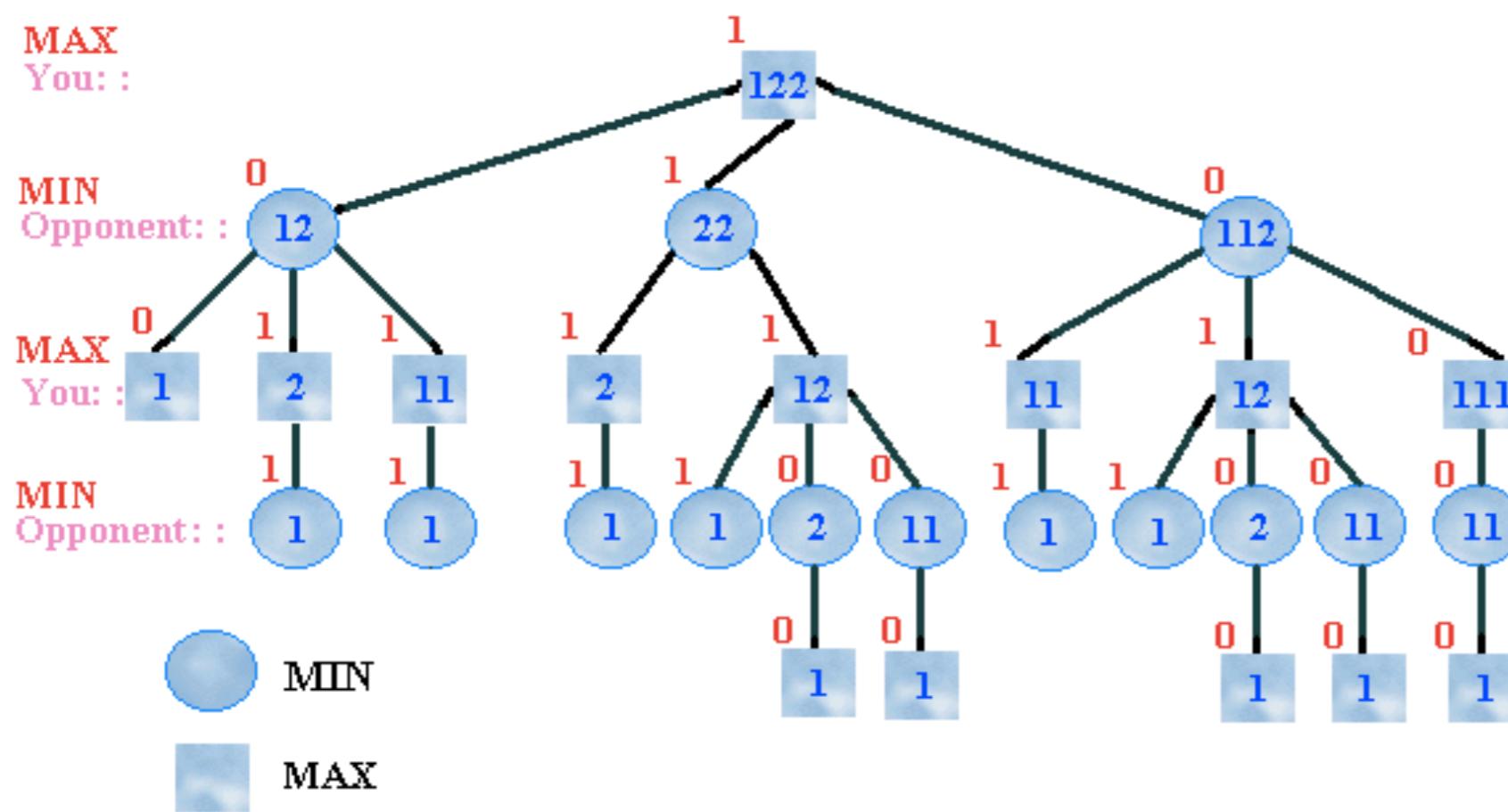
- **Déterminer un flot maximal (ou coupe minimale)**

- concept de flot augmentant
- tant que progression
- Ford-Fulkerson
- $\sim O(N^3)$



# Quelques exemples

- **Gagner à Fort Boyard face au père Fouras et ses maîtres de pacotille**
  - Jeu de Nim ou jeu de soustraction plus exactement



# Définitions

---

- **Un graphe orienté simple est un couple  $G=(S,R)$  où  $S$  est un ensemble de sommets et  $R$  une relation binaire dans  $S : R \subset S \times S$**
- **Exemples : déjà vus dans l'UE Graphes...**
- **Terminologie :**
  - arcs/arêtes/liens,
  - prédécesseur, successeur, degré, demi-degré intérieur/extérieur,
  - voisin,
  - origine, extrémité,
  - incidence, adjacence,
  - feuille, racine, sommet isolé, etc.

# Opérations globales

---

- **sous-graphe** : engendré par un sous ensemble de sommets
- **graphe partiel** : engendré par un sous ensemble d'arcs
- **union / disjonction (ou)** :  $R_1 \cup R_2 / R_1 \vee R_2$
- **intersection / conjonction (et)** :  $R_1 \cap R_2 / R_1 \wedge R_2$
- **composition** :  $R_1 \circ R_2$  (x R1  $\circ$  R2 y ssi  $\exists z$  i.e. x R1 z  $\wedge$  x R2 z)
- **symétrie** :  $R^{-1}$  graphe inverse, la relation  $R^{-1} \cup R$  est la fermeture symétrique de R
- **fermetures (plus petite relation qui contient R et qui vérifie certaines propriétés)**
  - transitive :  $R^+ = \bigvee_{k \geq 1} R^k$
  - réflexive transitive :  $R^* = \bigvee_{k \geq 0} R^k$
  - cas particulier si S est fini, l'union est aussi finie

# Chemins

---

- **Suite d'arcs adjacents tel que l'extrémité de l'arc  $i$  est l'origine de l'arc  $i+1$**
- **Composition  $k$  itéré de la relation  $R$  :  $R^k$** 
  - $x R^k y$  ssi  $\exists$  un chemin de longueur  $k$  de  $x$  à  $y$
- **Un circuit est un chemin non vide dont l'origine et l'extrémité sont confondus**
  - une boucle est un circuit de longueur 1
- **simple : tous les arcs sont distincts**
- **élémentaire : tous les sommets sont distincts**
  - élémentaire  $\rightarrow$  simple
  - circuit élémentaire autorise une exception pour l'origine/extremité confondus

# Équivalence forte, connexité

- **Deux sommets  $x$  et  $y$  sont fortement équivalents dans  $G$  si**
  - ils sont égaux ou
  - il existe un chemin de  $x$  à  $y$  et de  $y$  à  $x$  dans  $G$
- $\underset{f}{x} \sim y$  si et seulement si  $x R^* y \wedge y R^* x$ 
  - relation réflexive, symétrique et transitive -> relation d'équivalence
  - partition des sommets en classes d'équivalence
- **Une composante fortement connexe de  $G$  est le sous-graphe engendré par une classe de cette relation d'équivalence**
- **Graphe non orienté :**
  - chemin -> chaîne dans le graphe fermé par symétrie
  - $\underset{f}{x} \sim y$  si et seulement si  $x (R \vee R^{-1})^* y$

# Graphes valués

---

- **Valuation sur les sommets dans un ensemble Vs**
- **Valuation sur les arcs dans un ensemble Va**
- **Deux fonctions vs et va :**
  - $vs : S \rightarrow Vs$
  - $va : R \rightarrow Va$
- **L'ensemble Vs décrit généralement un identifiant ou une adresse**
  - nom, coordonnées, date, couleur, etc.
- **L'ensemble Va décrit lui souvent une mesure**
  - durée, poids, coût, équation, etc.
- **Notation graphe simple orienté et valué :  $G = (S, R, Vs, Va, vs, va)$**

# Spécification algébrique (simple)

- **spec GRAPHE étend EG // opérations d'égalité sur une sorte S décrivant les sommets**
- **sorte Graphe**
- **opérations**
  - grnouv :  $\text{-->} \text{Graphe}$  /\* Graphe vide \*/
  - adjs :  $\text{Graphe } S \rightarrow \text{Graphe}$  /\* Adjonction d'un sommet \*/
  - adja :  $\text{Graphe } S \times S \rightarrow \text{Graphe}$  /\* Adjonction d'un arc \*/
  - sups :  $\text{Graphe } S \rightarrow \text{Graphe}$  /\* Suppression d'un sommet \*/
  - supa :  $\text{Graphe } S \times S \rightarrow \text{Graphe}$  /\* Suppression d'un arc \*/
  - exs :  $\text{Graphe } S \rightarrow \text{Booléen}$  /\* Existence d'un sommet \*/
  - exa :  $\text{Graphe } S \times S \rightarrow \text{Booléen}$  /\* Existence d'un arc \*/
  - di :  $\text{Graphe } S \rightarrow \text{Nat}$  /\* Demi-degré intérieur \*/
  - de :  $\text{Graphe } S \rightarrow \text{Nat}$  /\* Demi-degré extérieur \*/

# Spécification algébrique (ensembliste)

- **Second niveau de spécification pour la gestion des ensembles (sommets, arcs)**

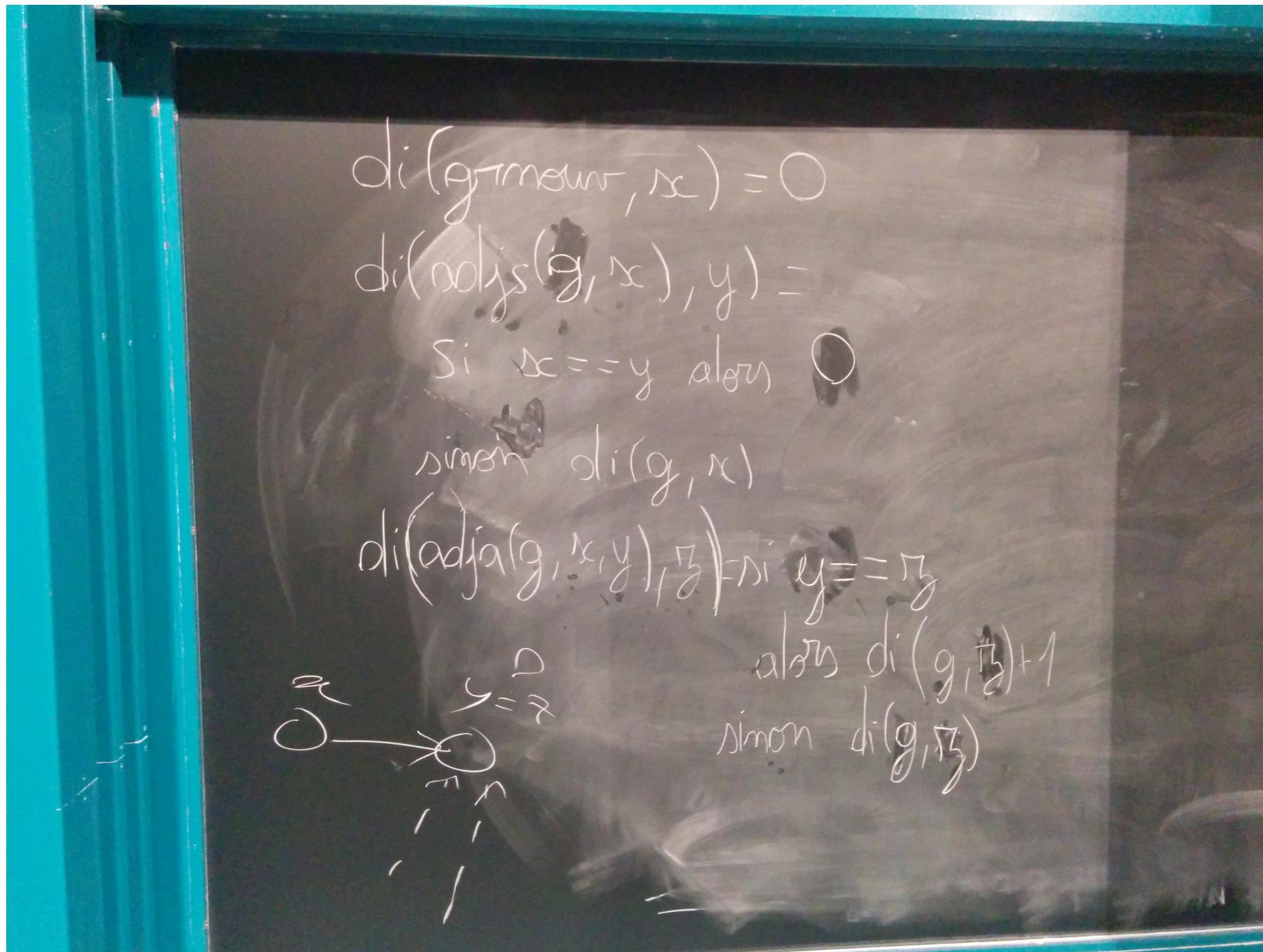
- **opérations**

- es : Graphe  $\rightarrow$  Ens /\* Ensemble des sommets \*/
- ea : Graphe  $\rightarrow$  Ens /\* Ensemble des arcs \*/
- esuc : Graphe S  $\rightarrow$  Ens /\* Ensemble des successeurs \*/
- epred : Graphe S  $\rightarrow$  Ens /\* Ensemble des prédecesseurs \*/

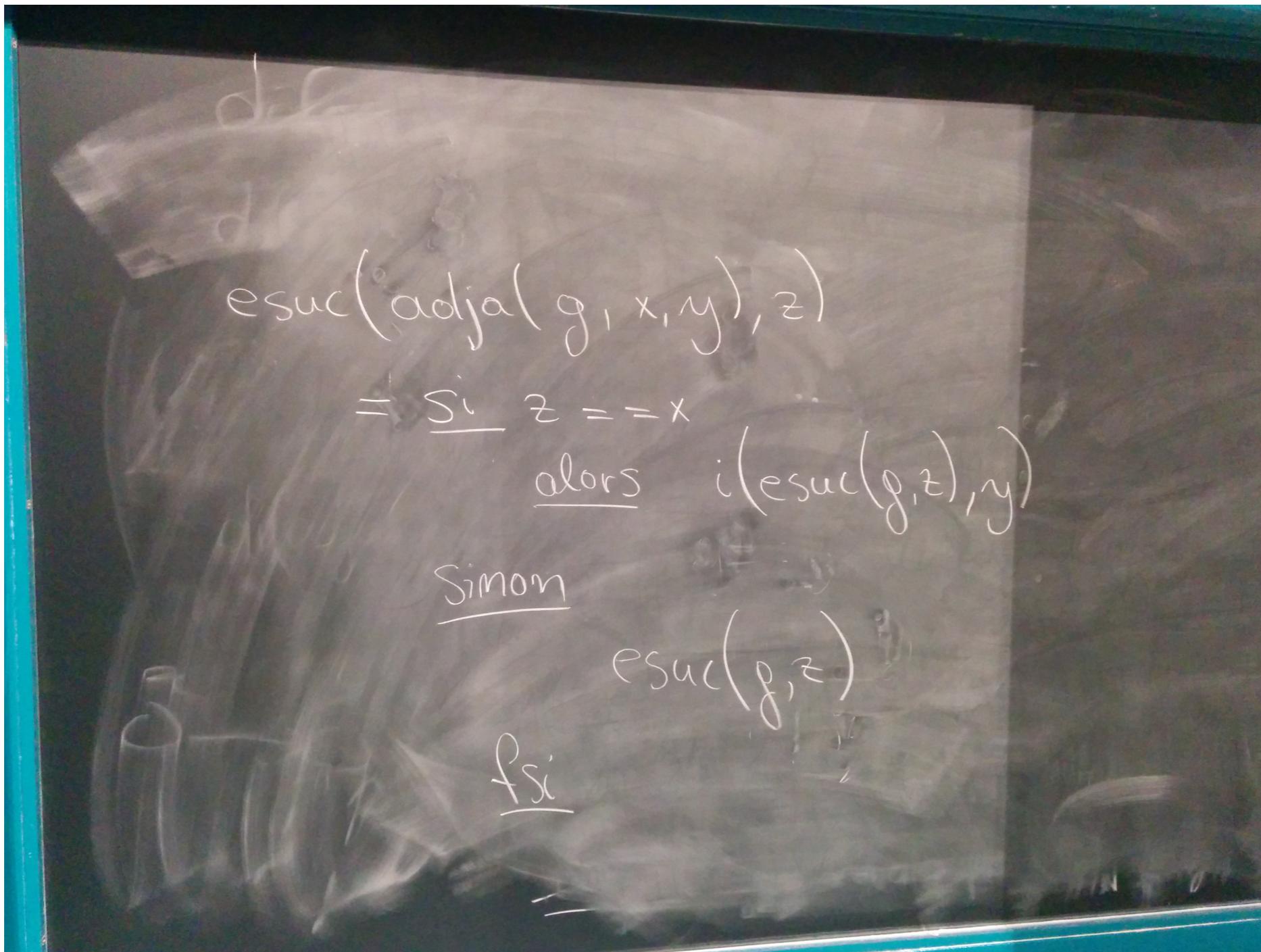
- **axiomes**

- es(grnouv) =  $\emptyset$
- es(adj<sub>s</sub>(g, x)) = i(es(g), x) // i opérateur d'insertion
- es(ajda(g, x, y)) = es(g)
- ...

# Axiome di



# axiome esuc (pour ajda uniquement)



# Spécification algébrique (avec valuation)

- **Troisième niveau de spécification pour les graphes valués**
  - modification des opérateurs d'adjonction et opérations liées à la valuation
- **opérations**
  - adjs : Graphe S Vs -> Graphe /\* adjonction d'un sommet valuée \*/
  - adja : Graphe S S Va -> Graphe /\* adjonction d'un arc valué \*/
  - vs : Graphe S -> Vs /\* consultation de la valeur d'un sommet \*/
  - va : Graphe S S -> Va /\* consultation de la valeur d'un arc \*/
  - modvs : Graphe S Vs -> Graphe /\* modification « côté » sommet \*/
  - modva : Graphe S S Va -> Graphe /\* modification « côté » arc \*/

# Représentations

---

- **Matrice d'adjacence (matrice booléenne associée)**
- **Matrice d'incidence aux arcs**
- **Ensemble de sommets et d'arcs**
- **Ensemble de successeurs et de prédecesseurs**
- ...

# Matrice d'adjacence

---

- **Les n sommets  $\{x_1, \dots, x_n\}$  du graphe peuvent être numérotés dans  $[1, n]$** 
  - soit « naturellement » par modélisation
  - soit via une table de correspondance, i.e. une fonction de  $S$  vers  $[1, n]$ 
    - ... et sa table inverse, fonction de  $[1, n]$  vers  $S$
- **La relation binaire  $R$  peut être décrite via une matrice carrée  $n$  par  $n$  tel que**
  - $m_{ij} = \text{exa } (G, x_i, x_j) = x_i \ R \ x_j$
- **Bien pour la manipulation des arcs, moins bien pour les sommets**
- **Pratique pour la description d'opérations matricielles :**  $m^+ = \bigoplus_{k=1}^n m^k$

# Fermeture transitive et réflexive avec rep. matricielle

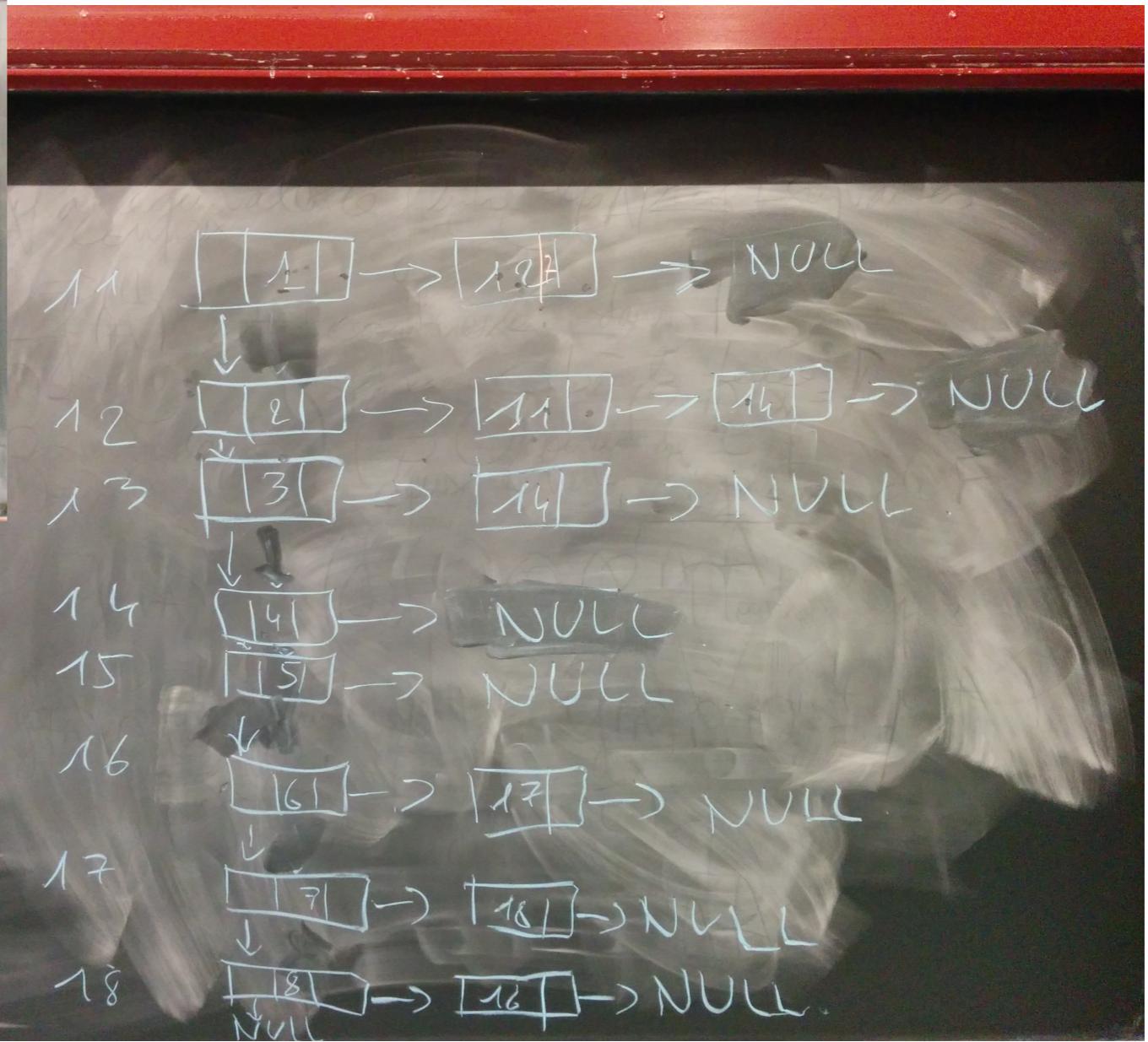
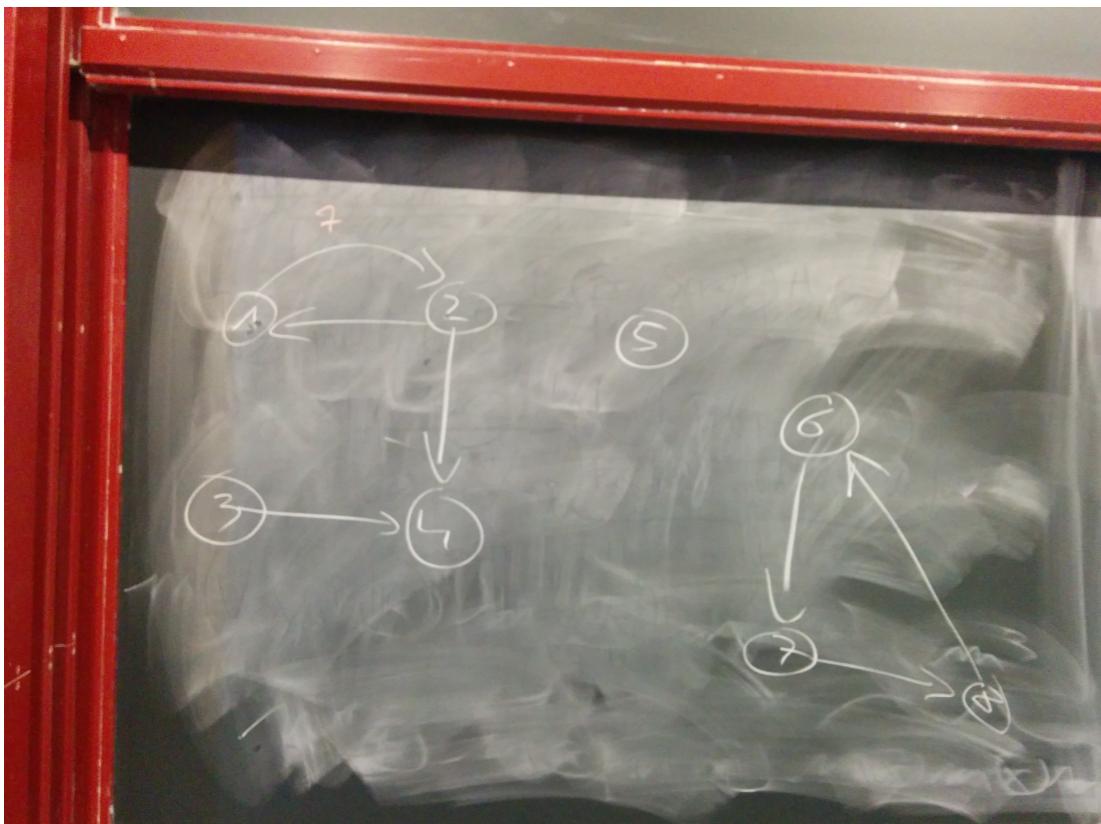
- **Exercice : écrire un algorithme dont la complexité est en O( $n^3$ )**
- **procedure PHI (var m: GrapheMat; x: IntSom; n: integer);**
- **var u, v: IntSom;**
- **for u := 1 to n do**
- **if (m[u, x] = 1) then**
- **for v := 1 to n do**
- **if (m[x, v] = 1) then**
- **m[u, v] := 1;**
- **procedure FermetureTransitive (var m: GrapheMat; n: integer);**
- **var x: IntSom;**
- **for x := 1 to n do**
- **PHI(m, x, n);**

# Listes d'adjacence

---

- L'ensemble des sommets est décrit par une liste (contiguë ou chaînée)
  - L'ensemble des successeurs (et éventuellement des prédecesseurs) de chaque sommet est décrit par une liste chaînée appelée liste d'adjacence.
  - Plutôt que de chainer les valeurs des sommets, on peut chainer leurs adresses (pointeurs dans la liste des sommets)
- 
- Très pratique pour les parcours dans les graphes
  - Exercice : représenter une telle structure sur l'exemple de votre choix

# Représentation 2-chaînée avec pointeurs sur adresses



# Parcours de graphes

---

- **Parcours en profondeur d'abord (Depth First Search - DFS)**
- **Numérotation des sommets dans l'ordre du parcours**
  - on part d'un sommet  $x$  et on applique une procédure  $pr(z)$  pour chaque sommet  $z$  accessible depuis  $x$
  - prolongement du chemin de  $x$  à  $z$
  - pour simplifier,  $pr(z)$  range  $z$  dans une liste lors de sa première visite après ceux déjà visités
- **parcprof : Graphe S → Liste**
- **pré parcprof(g,x) = exs(g,x)**
- **/\* parcprof(g,x) = parcprof1(g,esuc(g,x),adjq(⊤,e)) \*/**

# Version récursive : DFS ordre préfixe

- Utilisation d'une fonction auxiliaire avec comme inputs : ensemble et liste en cours
- `ch(e) /* choix dans élément dans un ensemble */`
- `app(l,z) /* test si z est dans l */`
- `s(e,z) /* suppression de z dans e */`
- `parcprof1 : Graphe Ens Liste -> Liste`
- `parcprof1(g,e,l) =`
  - si `v(e)` alors `l`
  - sinon `parcprof1(g, s(e,z), l1)`
    - avec `z = ch(e)`
    - avec `l1 = si app(l,z) alors l`
      - sinon `parcprof1(g, esuc(g,z), adjq(l,z))`

# Version récursive : DFS ordre préfixe

---

- **Quelques remarques**

- retours en arrière (backtracking)
- on peut généraliser sans point de départ (pour tout parcourir)
- notion de marquage des sommets
  - si déjà marqué alors arrêt et choix d'un autre sommet
- plusieurs choix possible pour passer au sommet suivant
- récursivité multiple : appels imbriqués
- version fonctionnelle (pas de pr avec effet de bord)
- on peut mélanger récursif et itératif
  - -> exercice sans la fonction choix

# Mélange récursif-itératif

PartProf1 : Graphe Ens Liste  $\rightarrow$  Liste

PartProf1 ( $g, \epsilon, l$ ) =  $l_1$   
Avec  $l_1 = \text{init}(l)$   
Pour tout  $z$  dans  $\epsilon$   
| Si app( $l, z$ ) alors  $l_1$   
| Sinon PartProf1 ( $g, \text{esuc}(g, z), \text{adj}(l, z)$ )

# Version itérative : DFS ordre préfixe

- Utilisation d'une pile P d'ensembles (pointeurs en pratique)
- //remplacer ( $p, s(e, z)$ ) = empiler (depiler ( $p$ ) ,  $s(e, z)$ )
- parcprof( $g, x$ ) = 1
- avec ( $p, l$ )=init(empiler(pilenouv, esuc( $g, x$ )) , adjq( $l, x$ )) ttq !vide( $p$ )
  - répéter si  $v(e)$  alors (dépiler ( $p$ ) ,  $l$ )
    - sinon si app ( $l, z$ )
      - alors (remplacer ( $p, s(e, z)$ ) ,  $l$ )
      - sinon (empiler(remplacer ( $p, s(e, z)$ ) , esuc( $g, z$ )) , adjq( $l, z$ ))
    - avec  $z = \text{choix}(e)$
  - avec  $e = \text{sommet}(p)$

# Déroulement de l'algo. sur un exemple de votre choix



# Avec pile de sommets (et pas d'ensembles de sommets)

---

- Définir une fonction prem pour parcourir la liste des successeurs d'un sommet  $x$  dans un ordre prédéfini (tel que le suivant  $y$  d'un premier voisin  $z$  soit compris entre  $\min_x$  et  $\max_x$  et plus grand que  $z$ ) // appelé avec  $\min_x$  la première fois, puis avec la dernière valeur renournée ensuite...
  - **prem(g, x, z) = min**
    - avec  $\min = \text{init}(\max_x)$
    - pour tout  $y$  dans  $\text{esuc}(x)$ 
      - si  $y > z$  ET  $y \leq \min$
      - $\min = y$
- Sur cette base, définir un parcours sur une pile de sommets

# Remarques

---

- **On peut généraliser à tout le graphe sans point de départ**
- **Le marquage peut se faire dans la structure**
  - il peut s'agir d'un pointeur pour chainer le parcours
- **On peut généraliser aux graphes non orientés**
  - composantes connexes et fortement connexes
- **Complexités (attention aux fonctions de base comme le test d'appartenance)**
  - $O(n+m)$  avec listes d'adjacences
  - $O(n^2)$  avec matrice d'adjacence
  - $O(n)$  en complexité spatiale dans le pire cas (si bien implémentée)

# Forêt couvrante associée au parcours

---

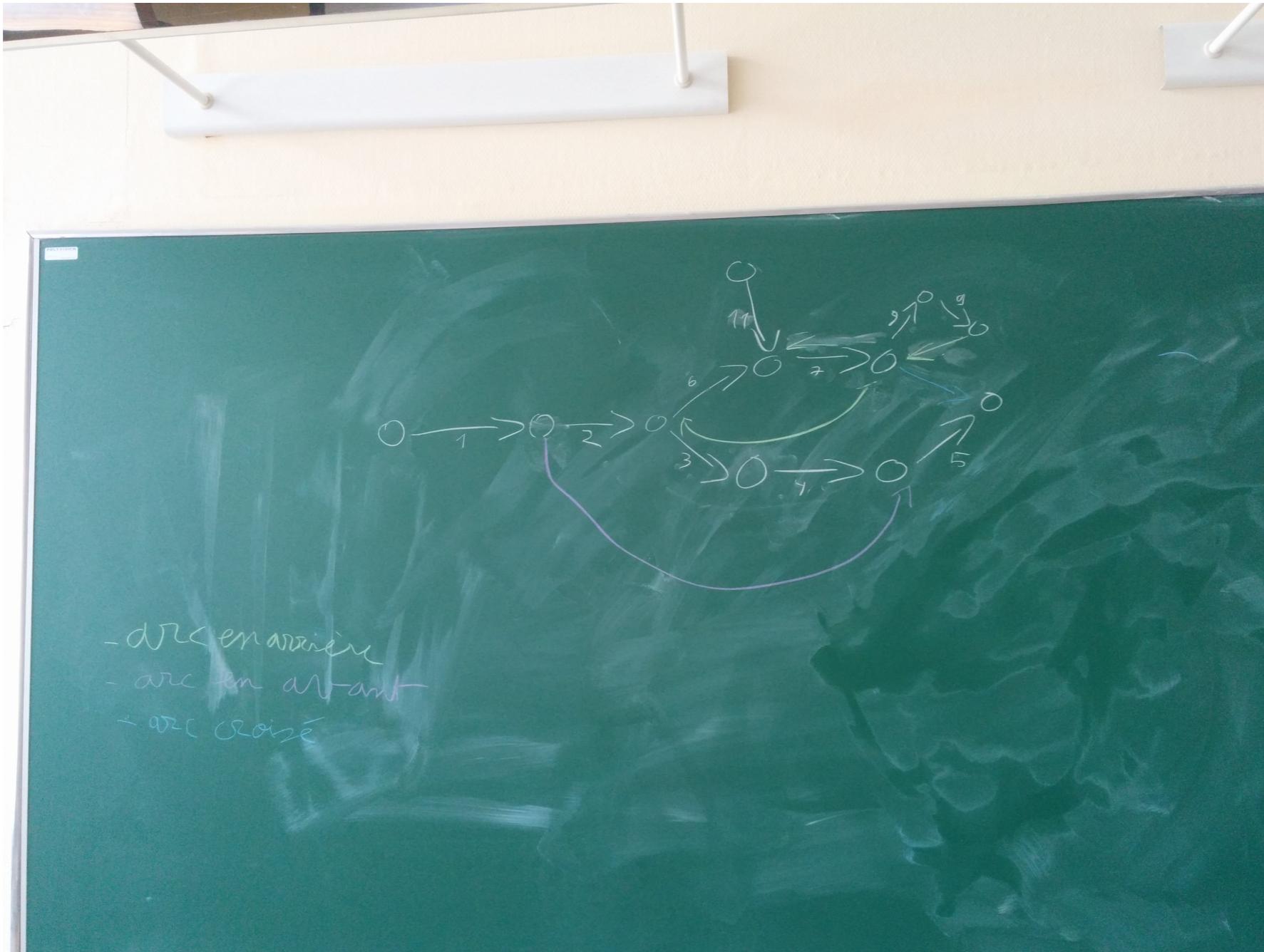
- **Classification des arcs**

- les arcs couvrants (la forêt de recouvrement correspondant au parcours)
- les arcs non couvrants
  - en arrière
  - en avant
  - croisé

- **Exercice sur le graphe de votre choix**

- + définitions
- + tous les cas sur le même exemple

# Arcs couvrants vs. arcs non couvrants



# Recherche de chemins entre deux sommets

---

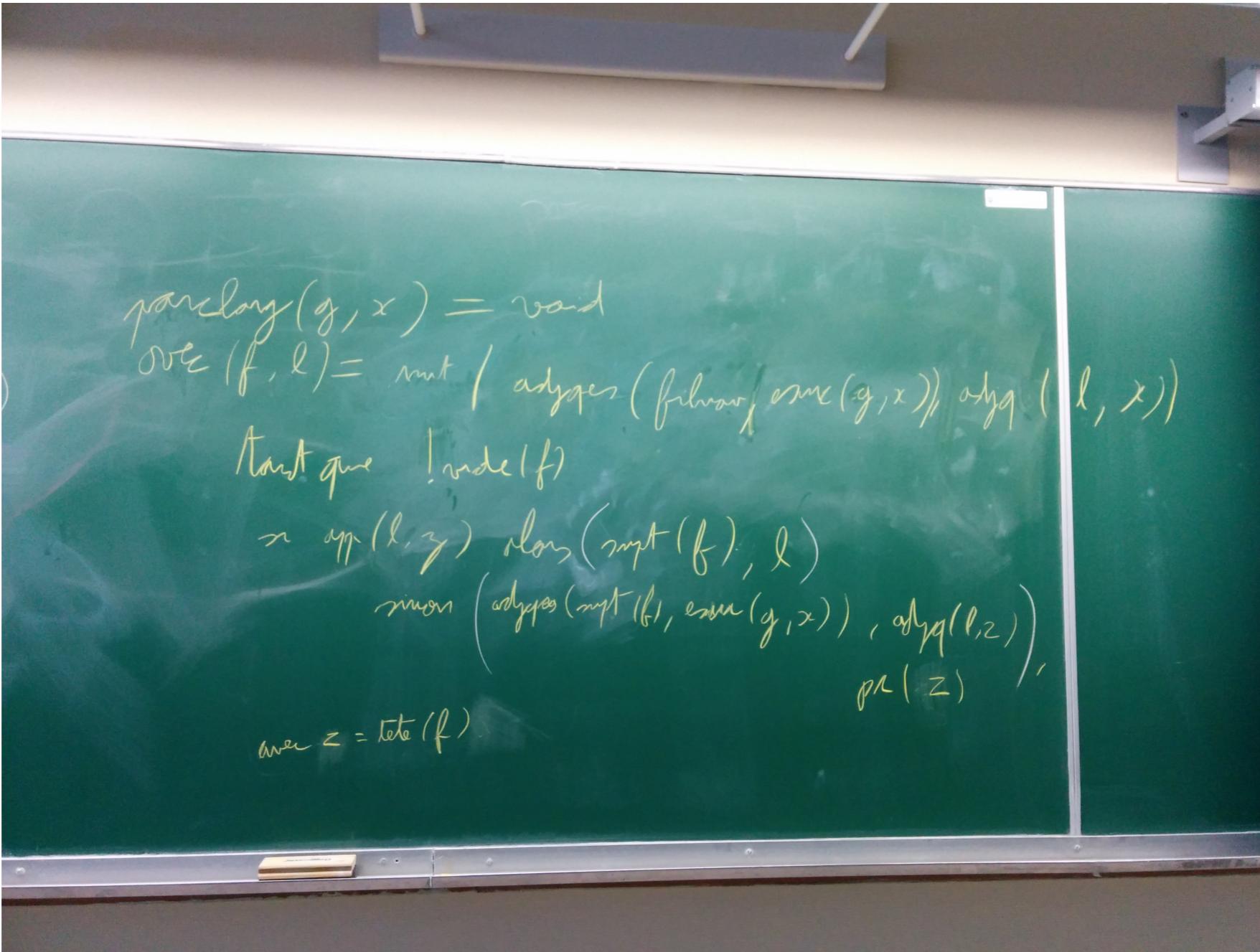
- **Spécifiez un algorithme qui donne tous les chemins élémentaires entre deux sommets**
  - procédure écrire\_chemin = “extension” de pr avec effet de bord
- **Faire de même pour savoir s'il existe un chemin entre x et y**

# Parcours en largeur d'abord (Breadth First Search - BFS)

- Traitement des sommets par “ondes successives” (niveau par niveau)
- On utilise une file avec placement en queue et lecture en tête (mode FIFO)
- `parclarg(g,x) = parclargf(g, adjq(∅, x), ∅)`
- `parclargf : Graphe File Liste -> Liste`
- `parclargf(g,f,l) =`
- `si vide(f) alors l`
- `sinon si app(l,z) alors parclargf(g,supt(f),l)`
  - `sinon parclargf(g,adjques(supt(f),esuc(g,z)),adjq(l,z))`
- `avec z = tete(f)`
- `/* adjques := ajout en queue d'un ensemble */`

# Version itérative avec fonction pr

- **Exercice (facile car récursivité terminale)**



# Remarques

---

- **Sur tout le graphe ?**
  - parclarg sur chaque sommet en transmettant la liste I
  - on appelle parclarg avec une file contenant tous les sommets du graphe (ou au moins toutes les racines)
- **Complexité : comme parcours en profondeur...**
- **Gestion d'une file et d'une liste ?**
  - une structure unique comme une liste pointée suffit...
    - liste avec un élément courant (tête de la file des éléments à traiter) sans supprimer les précédents afin de conserver les éléments déjà traités
    - on ajoute seulement en queue les successeurs non traités
  - ou une file d'ensemble de sommets successeurs (anologue à la pile)
- **On peut associer une forêt couvrante, composantes (fortement) connexes, mais moins bien adapté spatialement à la recherche de chemins que les parcours en profondeur**

# Parcours multiples

---

- **On peut traiter un sommet z autant de fois qu'on le rencontre**
  - (et pas seulement à sa première visite...)
  - soit la procédure  $\text{pr}(z,k)$ 
    - si  $\text{esuc}(g,z) = \emptyset$  alors  $\text{pr}(z,0)$
    - sinon, à la  $k^{\text{ème}}$  visite (le  $k^{\text{ème}}$  successeur) :  $\text{pr}(z,k)$
    - après le traitement de tous les successeurs, alors  $\text{pr}(z,\text{de}(g,z)+1)$
- **On peut aussi être plus restrictif et travailler :**
  - en ordre préfixe :  $\text{pr}(z,0)$  ou  $\text{pr}(z,1)$  seulement
  - en ordre suffixe : traitement de z après la dernière visite :  $\text{pr}(z,0)$  ou  $\text{pr}(z,\text{de}(g,z)+1)$ 
    - -> rappel : parcours d'arbres

# Exercices : spécifications

---

- **Parcours en profondeur en ordre suffixe**
- **Parcours en largeur avec une liste pointée**
- **Chainage des sommets lors du parcours en profondeur**
- ...

# Fermeture transitive

- **Algorithme de Warshall**
- Si on ordonne les sommets du graphe en un ensemble  $\{x_1, \dots, x_i, \dots, x_n\}$ , alors on peut définir une fonction booléenne  $\text{exch}(g, x, y, i)$  qui renvoie vrai ssi il existe une chemin de  $x$  à  $y$  dont tous les sommets intermédiaires ont un indice  $\leq i$
- $\text{exch}(g, x, y, 0) = \text{exa}(g, x, y)$
- $\text{exch}(g, x, y, i) =$
- $\text{exch}(g, x, y, i-1) \vee (\text{exch}(g, x, x_i, i-1) \wedge \text{exch}(g, x_i, y, i-1))$  pour  $1 \leq i \leq n$
- La fermeture transitive de  $g$  est alors donné par  $\text{exch}(g, x, y, n)$
- Notons qu'à chaque étape  $i$  (ou chaque sommet  $x_i$ ), la fonction booléenne  $\text{exch}$  correspond à l'application de la fonction  $\text{exa}$  sur un graphe augmenté  $g_i$  tel que :  $\text{exa}(g_i, x, y) = \text{exch}(g, x, y, i)$
- Pour construire ces graphes emboités augmentant, il suffit de partir de  $g$  et de considérer les prédecesseurs et les successeurs de  $x_i$  à chaque étape  $i$ .

# Warshall - spécification : $G^+ = (S, R^+)$

---

- warshall( $g$ ) =  $g'$
- avec  $g' = \text{init } g$  pour tout  $x_i$  dans  $\text{es}(g)$  /\* nouvel itérateur ! \*/
  - rep  $g''$ 
    - avec  $g'' = \text{init } g'$  pour tout  $x$  dans  $\text{epred}(g', x_i)$
    - rep  $g'''$ 
      - avec  $g''' = \text{init } g''$  pour tout  $y$  dans  $\text{esuc}(g'', x_i)$
      - rep si  $\text{!exa}(g''', x, y)$ 
        - alors  $\text{adja}(g''', x, y)$
        - sinon  $g'''$

# Recherche des plus courts chemins

- **Algorithme de E.W. Dijkstra**

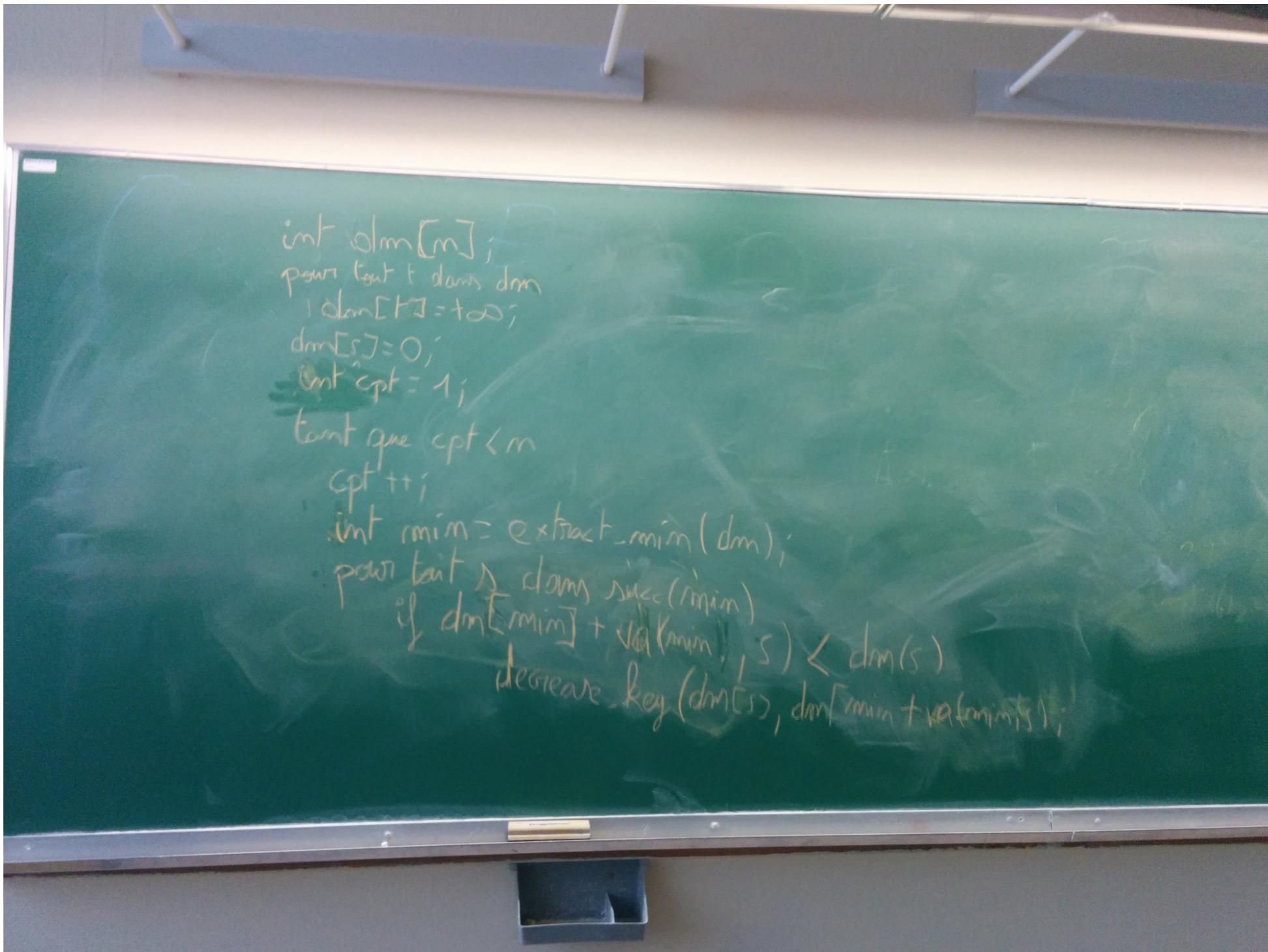
- opère sur un graphe simple orienté et valué par une distance  $va(g,x,y) \geq 0$
- plus courts chemins d'un sommet  $x$  à tous les autres sommets  $z$  du graphe
- notion d'arbre des plus courts chemins (si on "force" l'unicité du plus court chemin)
  - cet arbre est représenté par une fonction  $(i)père : Graphe S \rightarrow S$
  - la distance est donnée par la fonction  $(i)dm : Graphe S \rightarrow Rat$
- la stratégie consiste à définir une suite croissante d'ensemble  $e_i$  de sommets pour lesquels les deux fonctions précédentes sont effectivement connues
  - à chaque étape  $i (>= 2)$ , on choisit un noeud  $x_i$  dans  $E / \{e_{i-1}\}$  tel que celui-ci soit l'extrémité sortante d'un arc  $(y, x_i)$  dans  $e_{i-1}$  vérifiant la propriété suivante
    - $x_i = \arg \min (dm(g, y) + va(g, y, z))$  pour tout  $z$  dans  $E / \{e_{i-1}\}$
    - on augmente alors  $e_i$  avec ce noeud et on met à jour les fonctions dm et père

# Spécification récursive

---

- Il faut donc définir une fonction `amin` qui permet de faire croître l'ensemble  $e_i$ 
  - en pratique très dépendant de la structure choisie (gros impact sur les perfs)
  - admettons que cette fonction renvoie l'arc  $(y, x_i)$  et positionne donc  $\text{père}(x_i)$  à  $y$
- `dijkstra` : Graphe S -> Graphe
- `dijkstrae` : Graphe S Ens -> Graphe Ens
- `dijkstra(g, x) = dijkstra(g, x, ∅)`
- `dijkstrae(g, x, e) = si exa(g, amin(g, e))`
  - alors `dijkstrae(ipère(idm(g, xi, dmin), xi, y), x, i(e, xi`
  - avec  $(y, x_i) = \text{amin}(g, e)$ ,  $dmin = \text{dm}(g, y) + \text{va}(g, y, x_i)$

# Exercice : traduire tout ça en pseudo-code itératif !



# Correction

---

- **dijkstra\_iteratif : Graphe S -> Graphe /\* Un arbre en pratique... \*/**
- **dijkstra\_iteratif(g, x) = g' /\* sdm := sd pour stocker les dm \*/**
- **avec (g', n, sdm) = init (grnouv, x, sdmnouv) tant que n ≠ Ø**
  - **répéter (adjqa(adjqs(g', n), (père(n), n)), extract\_min(sdm), sdm')**
    - **avec sdm' = init sdm pour tout y dans esuc(n, g)**
      - **si key(sdm', y) = ∞ /\* Première Visite \*/**
        - **alors insert\_key(sdm', y, nv)**
        - **sinon si nv < key(sdm', y) /\* M.A.J \*/**
          - **alors decr\_key(sdm', y, nv)**
          - **avec nv = key(sdm', n) + va(n, y)**

# Dijkstra : complexité

---

- **Trois opérations élémentaires pour une implémentation efficace**
  - extract\_min : m ( $\sim \alpha_m$ )
  - insert\_key : i (ajouter un dm et père courant – idm et ipère)
  - decrease\_key : d (mettre à jour dm et père)
- **Soit n le nb de noeuds et m le nombre d'arcs**
- **Complexité temporelle en  $O((m+i) \times n + d \times m)$** 
  - en fonction de la queue de priorité utilisé, on obtient (dans le cas général) :
    - $O(n^2)$  avec des listes
    - $O(n\log n + m\log n)$  avec un tas binaire
    - $O(n\log n + m)$  avec un tas de Fibonacci
  - NB : en général,  $n < m \ll n^2$

# Floyd : tous les plus courts chemins (APSP)

---

- **Notion de table à double entrée**
- **père** : Graphe  $S \times S \rightarrow S$  /\* retourne  $z$  le père de  $y$  dans le chemin de  $x$  vers  $y$ \*/
- complexité en  $O(n^3)$  avec une matrice d'adjacence
- **floyd** : Graphe  $\rightarrow$  Graphe
- **floyd( $g$ ) =  $g'$**
- avec  $g' = \text{init } g$  pour tout  $z$  dans  $\text{es}(g)$ 
  - **rep  $g''$**
  - avec  $g'' = \text{init } g'$  pour tout  $x$  dans  $\text{epred}(g', z)$ 
    - **rep  $g'''$**
    - avec  $g''' = \text{init } g''$  pour tout  $y$  dans  $\text{esuc}(g'', z)$ 
      - **rep si  $nv < \text{va}(g''', x, y)$** 
        - alors **ipère(modv( $g''', x, y, nv$ ), x, y, z)**
        - avec  $nv = \text{va}(g''', x, z) + \text{va}(g''', z, y)$

# Plus courts chemins : résumé

---

- **Bellman-Ford**
  - pas de circuit absorbant (mais capable de détection !)
  - pire des cas “au mieux” :  $O(n*m)$
- **Dijkstra**
  - poids positifs
  - pire des cas “au mieux” :  $O(n\log n + m)$
  - avec l’algo. de Floyd dans les mêmes conditions
    - tous les meilleurs chemins :  $O(n^3)$
- **Bellman**
  - pas de circuit (un graphe acyclique orienté)
  - pire des cas “au mieux” :  $O(m)$

# Exercices : spécifications + implémentations

## Graphes

-> parcours en profondeur suffixe

- > quelle structure et quelle méthode ?
- > spécification !

-> tri topologique (partition circuit-niveaux)

- > Bellman (simple)

- > Composantes fortement connexes (2DFS avec Kosaraju ou, plus efficace, Tarjan avec une seule passe DFS avec tri topologique comme effet de bord et décomposition implicite des arêtes selon les quatre types vu dans ce cours)

-> implémentation (et spécification si pas dans le cours) d'algos sur les graphes

- > Dijkstra

- > Bellman-Ford

- > Prim

- > Kruskal

## Arbres (révisions chapitre précédent)

-> complexités

-> parcours itératif sur arbre parfait (dérécursion)

-> parcours infixé, préfixé, suffixé / postfixé

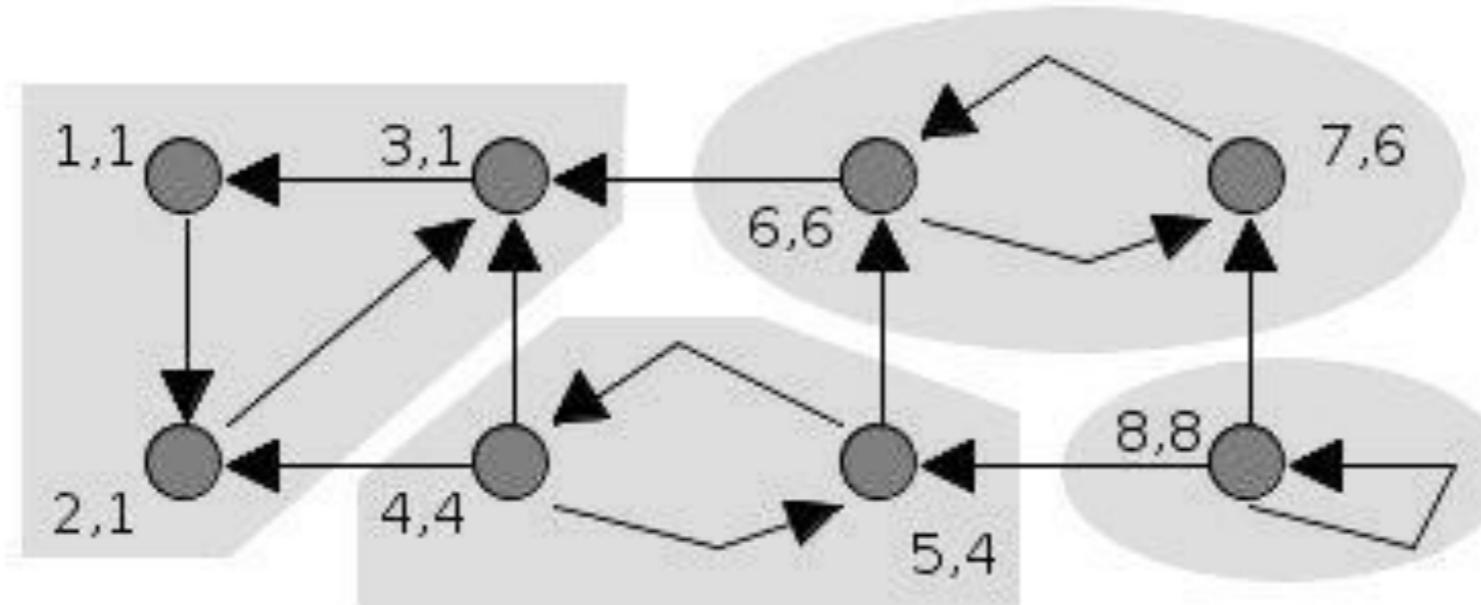
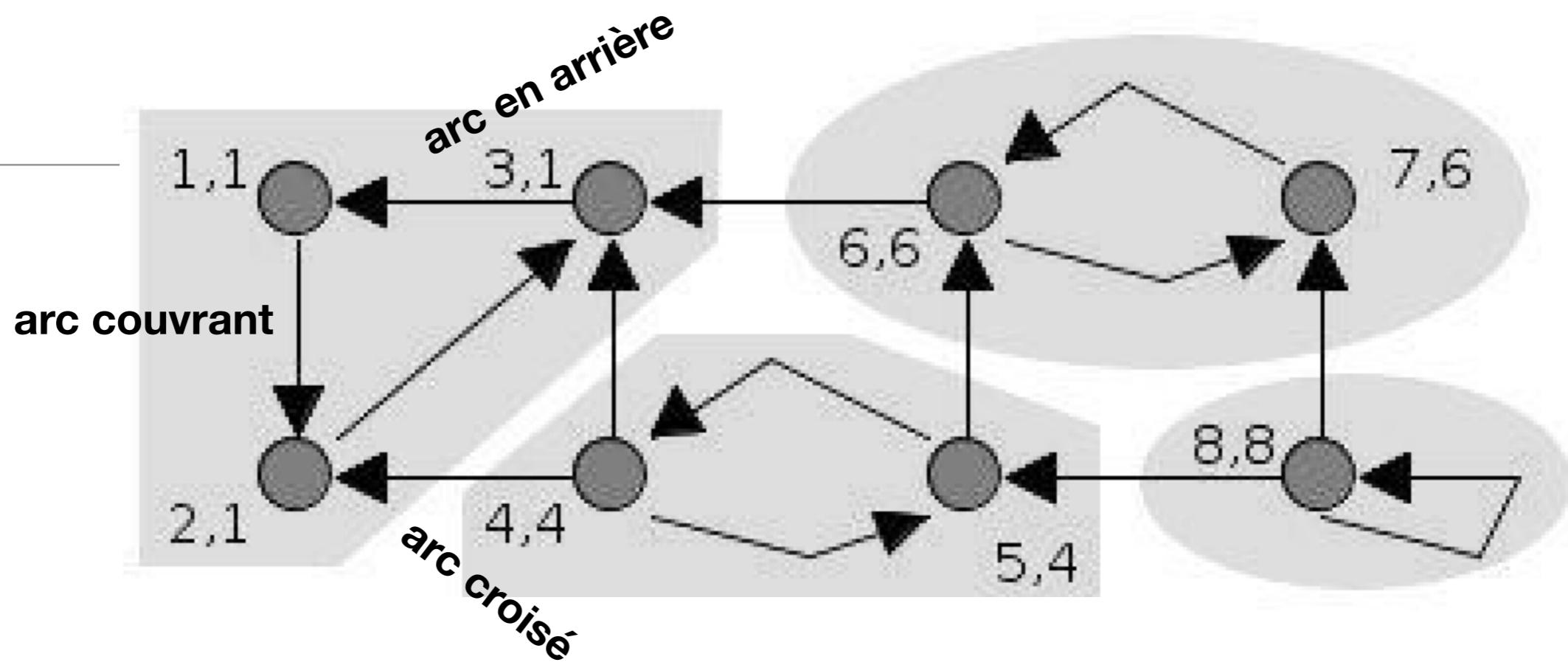
-> le reste...

# Exercices

---

- Spécifiez un algorithme capable de déterminer les composantes fortement connexes dans un graphe orienté
  - Facile !
- Faites en sorte qu'il soit linéaire en S et A
  - Moins facile : 2DFS
    - surtout en un seul passage DFS seulement...

## Tarjan



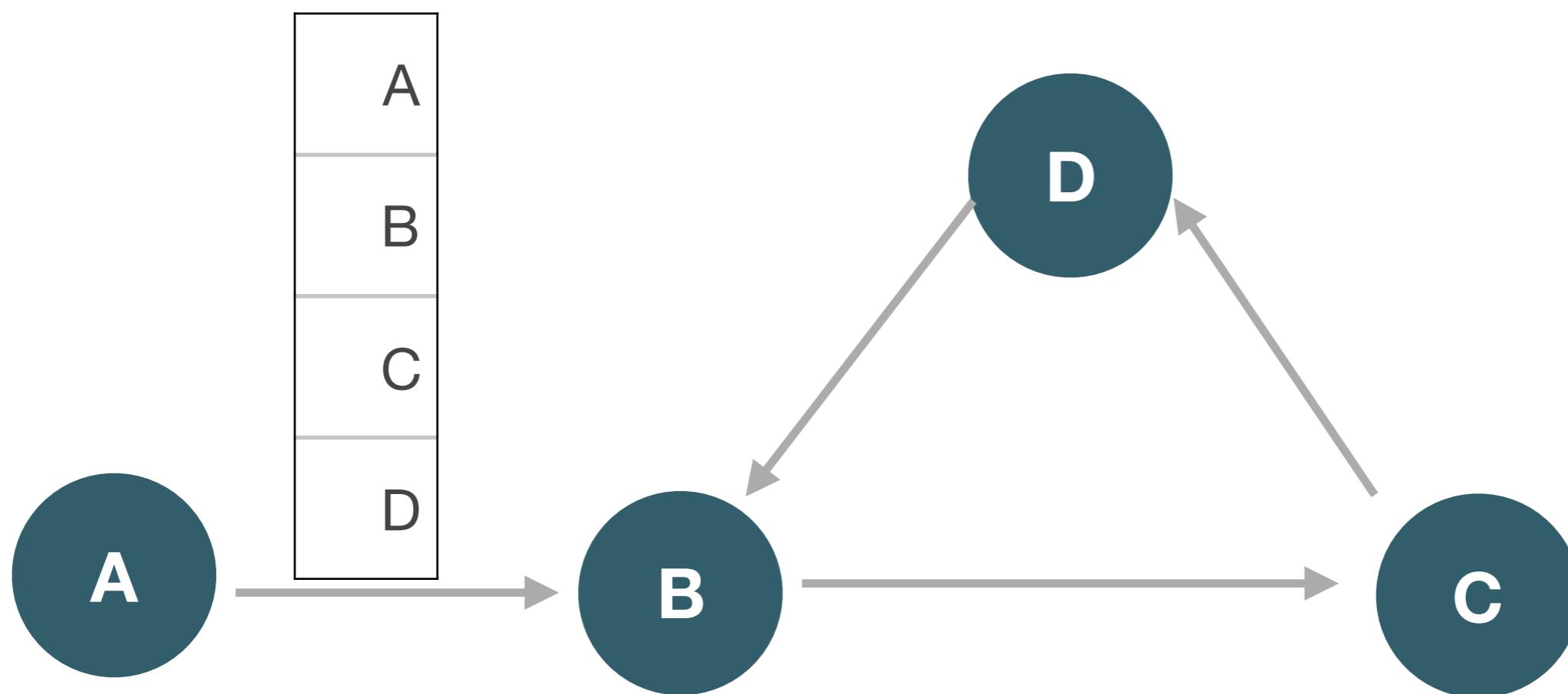
- ParcoursCFC : sommet v, entier k, pile p -> rien
- v.num = k; v.numAccessible = k; k++; empiler(p,v);
- pour chaque w dans esuc(v,g)
  - si w.num = -1
  - ParcoursCFC (w, k)
  - v.numAccessible = min(v.numAccessible, w.numAccessible)
  - sinon si app(w,p) // possible en O(1)
  - v.numAccessible = min(v.numAccessible, w.num)
  - si v.numAccessible = v.num
    - CFC =  $\emptyset$
    - répéter
      - w = dépiler(p); CFC = CFC  $\cup$  w;
      - tant que w  $\neq$  v
      - afficher CFC

- Tarjan : Graphe g -> rien
- k = 0; p = pilenouv; partition =  $\emptyset$
- pour chaque sommet v de g
  - v.num = -1
  - pour chaque sommet w de g
    - si v.num = -1
    - parcoursCFC (v, k, p)

# L'algorithme de Kosaraju

- **Soit  $g$  un graphe orienté et  $p$  une pile vide**
- **Tant que  $p$  ne contient pas tous les sommets, faire:**
  - Choisir le premier sommet  $v$  non dans  $p$ ;
  - Lancer un parcours en profondeur à partir de  $v$ .
    - Chaque fois que la récursion est finie pour tous les successeurs d'un sommet  $w$ , ajouter  $w$  à  $p$ .
- **Inverser l'orientation de tous les arcs de  $g$  pour obtenir le graphe symétrique.**
  - Tant que  $p$  non vide:
    - Soit  $v =$  dépiler  $p$ .
    - Lancer un parcours en profondeur à partir de  $v$  dans le graphe symétrique.
    - L'ensemble des sommets visités depuis  $v$  décrit alors la composante fortement connexe contenant  $v$  : l'afficher/l'enregistrer et effacer tous ses sommets dans  $g$  et dans la pile  $p$ .

# Dérouler les deux algorithmes sur le même exemple



# Corrigé CC1

---



# Corrigé CC 1

## • Questions de compréhension

---

- Pour les tris efficaces, les algos de parcours avec optimalité des chemins (Disjktra)
- Voir les exemples donnés dans le TD
- Plusieurs options possibles (voir le cours) :
  - binariser la foret => premier fils à gauche puis second fils à droite et le reste en liste
  - presque pareil mais juste une liste chainée de fils
  - à l'envers : pointeur vers le père...peu pratique
- $O(\log(|S|))$  au pire pour toutes les opérations sauf pour obtenir le min ou le max  $O(1)$ 
  - c'est liée à la hauteur maximale d'un arbre binaire complet
- $O(\log(|S|))$  au pire pour toutes les opérations sauf pour obtenir le min et le max  $O(1)$ 
  - hauteur maximale limitée grâce au ré-équilibrage
- $O(|S|)$  au pire mais  $O(\log(|S|))$  en moyenne
  - car dans le cas moyen, peu de chances d'avoir le pire des cas (valeurs pré-triées)

# Corrigé CC1

- **Parcours postfixe en impératif itératif**

---

- **desgd : Arbin Nat Pile -> Pile**

- **desgd(a, n, p)**

- **empiler(p, (a, n)) ;**
- **Tant que  $\neg f(a)$** 
  - **si  $v(ag(a))$  alors  $a = ad(a)$  ;  $empiler(p, (a, 2))$  ;**
  - **sinon  $a = ag(a)$  ;  $empiler(p, (a, 1))$  ;**

- **posfix(a)**

- **$p = desgd(a, 0, pilenouv)$  ;**
- **Tant que  $\neg vide(p)$** 
  - **$(a, n) = sommet(p)$  ;  $afficher rac(a)$  ;  $dépiler(p)$  ;**
  - **si  $vide(p)$  alors  $return$  ;**
  - **sinon si ( $n == 2 \vee (v(ad(sommet(p).arbre)) \wedge n == 1)$ )**
    - **continue** ;
    - **sinon  $desgd(ad(sommet(p).arbre), 2, p)$**

# Corrigé CC1

## • Parcours postfixe en fonctionnel itératif

- `desgd : Arbin Nat Pile -> Pile /* descendre à gauche si possible...`
- `pré desgd(a, n, p) = ~v(a) ... à droite sinon */`
- `desgd(a, n, p) = p' // couple (arbre, position=0,1,2)`
- `avec (a', p') = init (a, empiler(p, (a, n)) tant que ~f(a')`
  - répéter ( $a''$ , `empiler(p', (a'', n''))`)
  - avec ( $a''$ ,  $n''$ ) = si  $v(ag(a'))$  alors `(ad(a'), 2)` sinon `(ag(a'), 1)`
- `posfix(a, pr) = rien`
- `avec (p, todo) = init (desg(a, 0, pilenouv), rien) ttq ~vide(p)`
  - répéter si `vide(p')` alors  $(p', todo')$  // FIN :  $n == 0$ 
    - sinon si ( $n == 2 \vee (v(ad(a'')) \wedge n == 1)$ ) alors  $(p', todo')$ 
      - sinon `(desgd(ad(a''), 2, p'), todo')`
      - avec ( $a''$ ,  $n'$ ) = `sommet (p')`
  - avec ( $a'$ ,  $n$ ) = `sommet (p)`
  - avec  $p' = depiler(p)$  et  $todo' = pr(r(a'))$

# Corrigé CC1

---

- **Illustration**
  - à faire en TD pendant le corrigé
- **Complexité :  $O(|S|+|A|)$  ou  $O(\max(|S|,|A|))$** 
  - Linéaire, comme tout parcours en profondeur simple...

# Corrigé CC1

## Partition circuits-niveaux en fonctionnel

- **partition** : Graphe (acyclique)  $\rightarrow$  tableau
- **partition** ( $G$ ) =  $t$
- soit  $(d, x) = \text{init}(\text{tabnouv}, 0)$  tant que  $x < |S|$ 
  - répéter  $(d', x++)$
  - avec  $d' [x] = |\mathcal{T}^{-1}(x)|$
- soit  $(S_0, x, t_i, i) = \text{init}(\text{listnouv}, 0, \text{tabnouv}, 0)$  tant que  $x < |S|$ 
  - répéter  $(S_0', x++, t', i')$
  - avec  $(t', i', S_0') = \text{si } d[x]==0 \text{ alors } (S_0' \cup x, t'[i'++]=x)$
- avec  $(d, N, t, j, S_{k+1}, S_k) = \text{init}(d, |S_0|, t_i, i, \emptyset, S_0)$  tant que  $N < |S|$ 
  - répéter  $(d', N', t', j', S_{k+1'}, S_{k'})$
  - avec  $(d', N', t', j', S_{k+1'}) =$ 
    - pour tout  $x$  dans  $S_k$ 
      - pour tout  $y$  dans  $T(x)$ 
        - répéter  $(d' [y]--, N'', t'', j', S_{k+1''})$
        - avec  $(N'', t'', j', S_{k+1''}) = \dots$
        - ...si  $d' [y] == 0$  alors  $(N''++, S_{k''} \cup y, t'' [j''++]=y)$
    - avec  $S_{k'} = S_{k+1'}$