Théorie des graphes

Souheib Baarir.

 sbaarir@epita.lrde.fr

I

Introduction

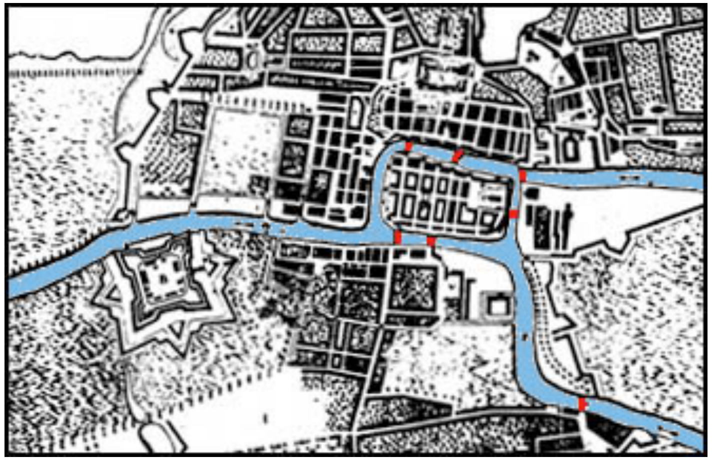
&

Définitions

Historique

- 1735 Leonhard Euler expose une solution formelle au problème des 7 ponts de Königsberg :

« Lors d'une promenade, est-il possible de passer sur tous les ponts de la ville une et une seule fois ? »



Domaines d'applications

- Chimie :
  Modélisation des molécules (A. Cayley en 1860)
- Mécanique :
  Treillis
- Biologie :
  Réseau de neurones
  Séquencement du génome
- Sciences sociales :
  Modélisation des relations
- Et bien sûr dans divers domaines de l'informatique

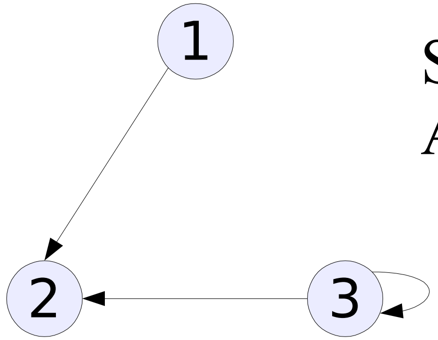
Définition : graphe

- Un **graphe orienté** G c'est un couple (S,A) avec :
  - S un ensemble fini : **ensemble des sommets**
  - A une relation binaire sur S : **ensemble des <u>arcs</u>**

- Un **graphe NON orienté** G c'est un couple (S,A) :
  - S un ensemble fini : **ensemble des sommets**
  - A paires non ordonnées : **ensemble des <u>arêtes</u>**

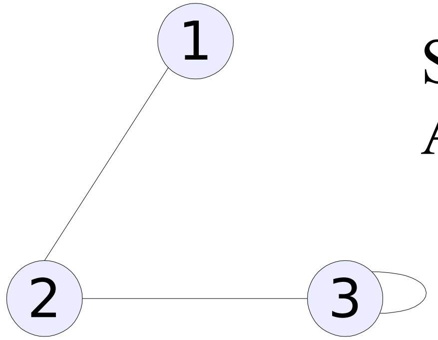
Exemple : graphe

- Cas orienté :



S = {1,2,3}
A ={(1,2) , (3,2) , (3,3)}

- Cas non-orienté :



S = {1,2,3}
A ={{1,2} , {3,2} , {3}}

Successeurs, prédécesseurs et voisins

- Les **sucesseurs** d'un sommet x sont définis par l'ensemble : $\delta^+(x) = \{y \mid (x,y) \in A\}$

- Les **prédécesseurs** d'un sommet x sont définis par l'ensemble : $\delta^-(x) = \{y \mid (y,x) \in A\}$

- Les **voisins** d'un sommet x sont définis par l'ensemble : $\delta(x) = \delta^-(x) \cup \delta^+(x)$

- Note : dans le cas non orienté, $\delta(x) = \delta^-(x) = \delta^+(x)$

Degré d'un sommet

- Dans un graphe :
  - On appelle **degré sortant** d'un sommet :
    le nombre d'arcs qui partent de ce sommet ($d^+(x) = |\delta^+(x)|$)
  - On appelle **degré entrant** d'un sommet :
    le nombre d'arcs qui arrivent à ce sommet ($d^-(x) = |\delta^-(x)|$)
  - On appelle **degré** d'un sommet :
    la somme des degrés entrant et sortant du sommet
    ($d(x) = |\delta(x)|$)

- Note : dans le cas non orienté, $d(x) = d^-(x) = d^+(x)$

Chemin (1/2)

- Un **chemin** d'un sommet *u* au sommet *u'* est une séquence de sommets $(v_0, v_1, v_2, \ldots\ v_{k-1}, v_k)$ tel que :

$$u = v_0\ , \quad u' = v_k \quad et \quad \forall i, (v_{i-1}, v_i) \in A$$

- On dit que ce chemin a une **longueur** *k*

- Ce chemin est **élémentaire** ssi $\forall i, j,\ v_i \neq v_j$

- Un sommet *u* **accessible** depuis un sommet *v* ssi : il existe un chemin du sommet *u* au sommet *v*

Chemin (2/2)

- Dans un graphe orienté :
  - Un chemin $(v_0,v_1, \dots ,v_k)$ forme un **circuit** ssi $v_0= v_k$
  - Ce circuit est **élémentaire** ssi $\forall\, i, j \in [1,\dots,k\text{-}1],\ v_i \neq v_j$
  - Une **boucle** est un circuit de longueur 1
  - Un graphe est **acyclique** ssi il ne contient aucun circuit

- Dans un graphe non orienté :
  - Un chemin $(v_0,v_1, \dots ,v_k)$ forme un **cycle** ssi $(v_0= v_k)$
  - Un graphe est **acyclique** ssi il ne contient aucun cycle

Propriétés

- On dit d'un graphe qu'il est :
  - **Réflexif** ssi :$\forall u_i \in S\ ,\ (u_i, u_i) \in A$
  - **Irréflexif** ssi :$\forall u_i \in S\ ,\ (u_i, u_i) \neq A$
  - **Transitif** ssi :
    $\forall u_i, u_j, u_k \in S\ ,\ (u_i, u_j) \in A \wedge (u_j, u_k) \in A \Rightarrow (u_i, u_k) \in A$

- On dit d'un graphe orienté qu'il est :
  - **Symétrique** ssi :$\forall u_i, u_j \in S\ ,\ (u_i, u_j) \in A \Rightarrow (u_j, u_i) \in A$
  - **Anti-Symétrique (Assymetrique)** ssi :
    $\forall u_i, u_j \in S\ ,\ (u_i, u_j) \in A \wedge (u_j, u_i) \in A \Rightarrow u_i = u_j$

Connexité

- On dit d'un graphe non orienté qu'il est :
  - **Connexe** ssi pour toute paire de sommets (*u,v*) il existe une chaîne entre les sommets *u* et *v*.
  - **Complet** ssi tous les sommets sont «reliés» 2 à 2 :
    $$\forall u,v \in S\ ,\ (u,v) \in A$$
- On dit d'un graphe orienté qu'il est :
  - **Connexe** ssi le graphe non-orienté correspondant est connexe
  - **Fortement connexe** ssi si pour tout *(u,v)* il existe un chemin de *u* à *v* et de *v* à *u*
  - **Complet** ssi tous les sommets sont «reliés» 2 à 2 :
    $$\forall u,v \in S\ ,\ (\ (u,v) \in A\ )\ \vee\ ((v,u) \in A\ )$$

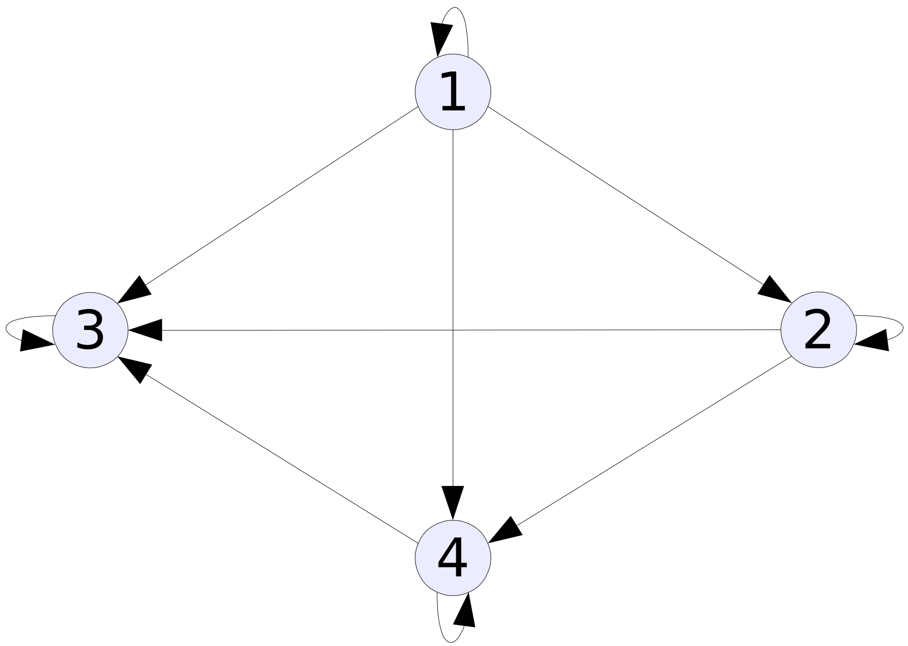
K-Connexe

- Un graphe non-orienté est **k-connexe** ssi :
  - il reste connexe après suppression d'un ensemble quelconque de k-1 arêtes et s'il existe un ensemble de k arêtes qui déconnecte le graphe.

- Un graphe orienté est **k-connexe** ssi :
  - le graphe non-orienté correspondant est k-connexe

- *Cette notion est utilisée :*
  - *en électronique pour le calcul de la fiabilité*
  - *dans l'étude de jeux de stratégie (cut and connect).*

Exemple



- Ce graphe orienté est-il :
  - Réflexif ?
  - Transitif ?
  - Antisymétrique ?
  - Connexe ?
  - Complet ?

ATTENTION :
Dans un graphe orienté :
Complet n'implique pas Fortement connexe.
Ex : il n'y a pas de chemin pour aller de 2 a 1

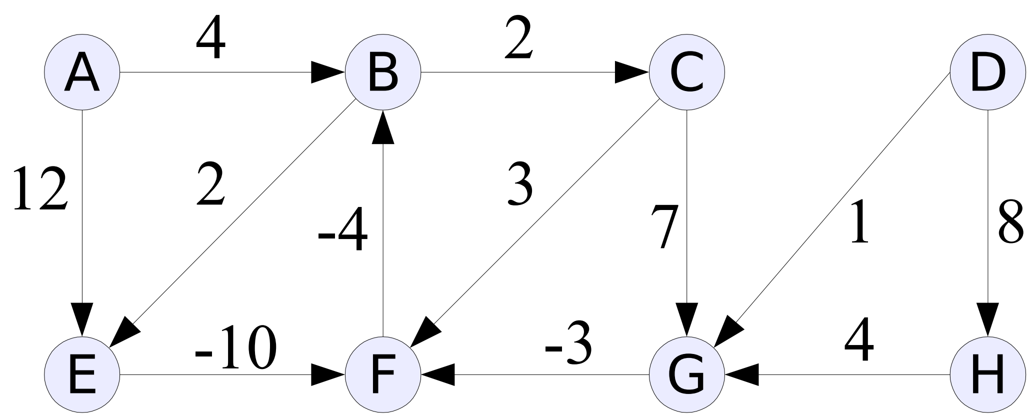
Graphes remarquables (1/2)

- Certains graphes portent des noms particuliers :
  - **Biparti =** graphe qui peut être partitionner en deux sous ensembles de sommets S_1 et S_2 tels que deux sommets du même ensemble ne sont jamais voisins.
  - **Hypergraphe =** graphe non orienté ou chaque arrête est une **hyperarête** qui relie un sommet à un sous ensemble de sommets.
  - **Forêt =** graphe non orienté acyclique.
  - **Arbre =** graphe connexe non orienté acyclique.

Graphes remarquables (2/2)

Graphe valué (pondéré) = c'est un graphe (orienté ou non) G =(S, A) muni d'une application

$$p: \quad A \to \mathbb{R}$$
$$(x, y) \to p(x, y)$$



Valuation (ou **Poids**)
de l'arc (x,y)

Représentation d'un graphe

- Il existe deux façons de représenter un graphe (S, A) :
  - **Liste adjacente** : pour les graphes peu denses

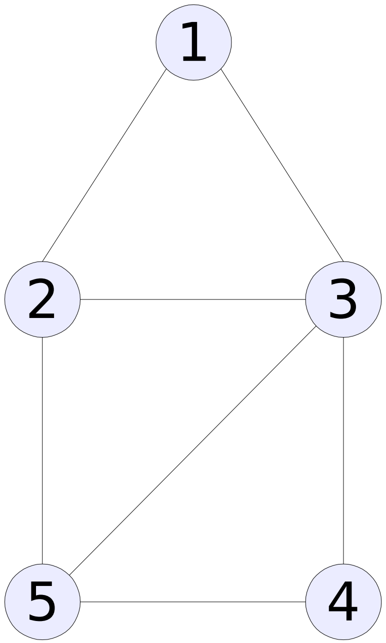
    $$Card\,(A) \ll (\,Card\,(S)\,)^2$$

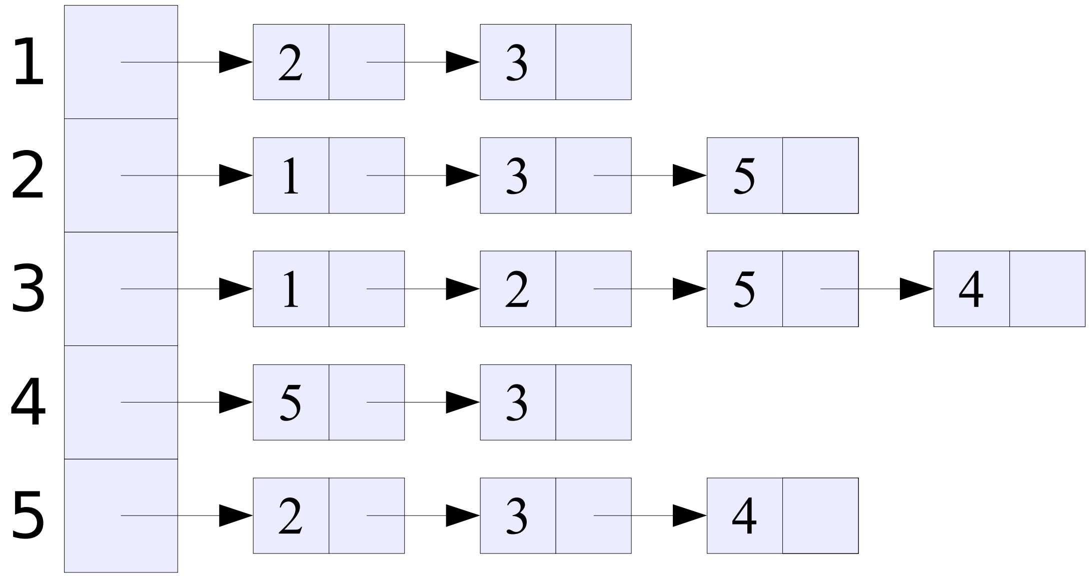
  - **Matrice d'incidence** : pour les graphes denses

    $$Card\,(A) \simeq (\,Card\,(S)\,)^2$$

Liste d'adjacence

- Pour chaque sommet $u \in S$ on a une liste d'adjacence :
  $Adj\,[u]$ liste des sommets $v \in S$ tel que $(u,v) \in A$





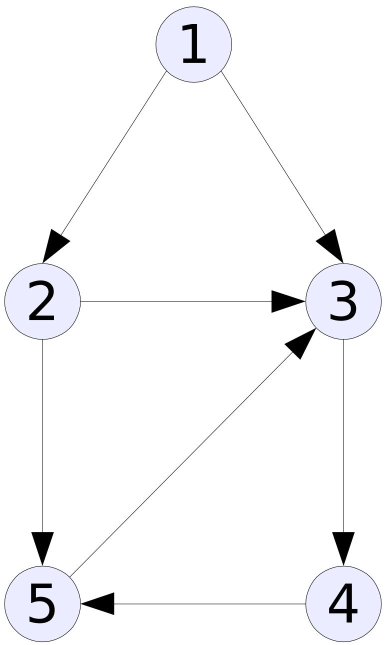
NB : Si le graphe est pondéré, on rajoute un champ avec le poids de l'arc (arête).

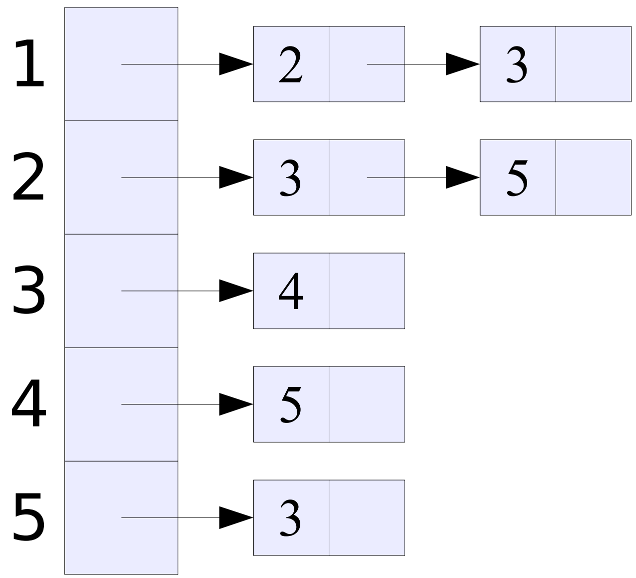
Matrice d'adjacence : quelques complexités

- Stockage : **$\Theta(|S|^2)$**
- Test d'existence arc : **$\Theta(1)$**
- Parcours des arcs incidents à un sommet : **$\Theta(|S|)$**
- Trouver un voisin : **$O(|S|)$**

Liste d'adjacence

- Pour chaque sommet $u \in S$ on a une liste d'adjacence : $Adj\,[u]$ liste des sommets $v \in S$ tel que $(u,v) \in A$



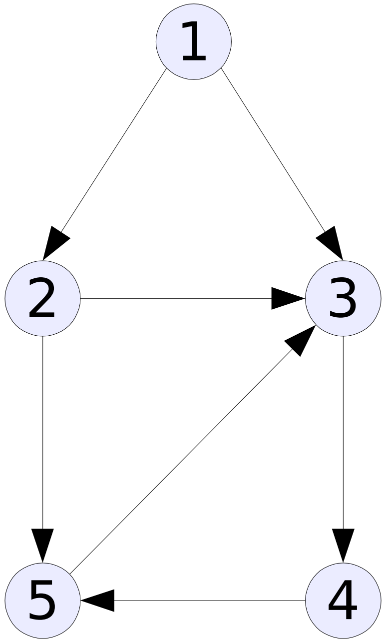


Liste d'adjacence : quelques complexités

- Stockage : $\Theta(|S|+|A|)$
- Test d'existence arc : $O(\max\{d^+(s) \mid s \in S\})$
- Parcours des arcs incidents à un sommet : $O(\max\{d^+(s) \mid s \in S\})$
- Trouver un voisin : $\Theta(1)$

Matrice d'adjacence

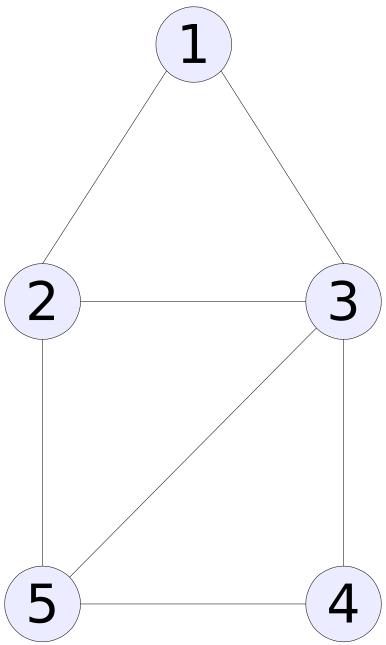
- *Pour un graphe orienté* : $\forall\, i\,,\, j \in S \quad a_{ij} = \begin{cases} 1 & si\,(i\,,\, j) \in A \\ 0 & sinon \end{cases}$



$$Mat(S\,,\,A) = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Matrice d'adjacence

- *Pour un graphe* non orienté : $\forall i, j \in S \quad a_{ij} = \begin{cases} 1 & si\,(i, j) \in A \\ 0 & sinon \end{cases}$



$$Mat(S, A) = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

NB : Si le graphe est pondéré, $a_{ij} = p(i,j)$

II

Algorithmes de recherche du plus court chemin

Motivation

Beaucoup de problèmes de la vie quotidienne peuvent être représentés sous forme de graphes…

Le calcul de distance (et donc un plus court chemin) en est un des plus courant :

- **Les logiciels de GPS calculant des itinéraires routiers**
- **Distribution de chaleur dans les alentours**
- **Connexion a haut débit par câble**
- **Routage dans des réseaux de télécommunications**
- **...**

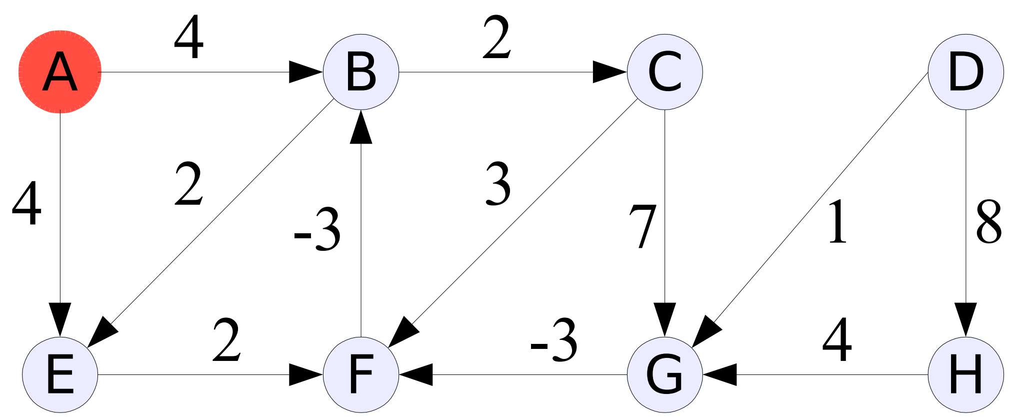
Quelques définitions

Définitions :

- La longueur d'un chemin est la somme des poids des arcs
- La distance entre x et y (*noté, d(x,y)*) est le minimum des longueurs sur tous les chemins.
- Un plus court chemin entre x et y est un chemin dont la longueur est égale à d(x,y).

Exemples :

- Longueur de (A,E,F,B) est $4 + 2 + (-3) = 3$
- $d(A, B) = 3$
- Plus court chemin entre A et B : (A, E, F, B)



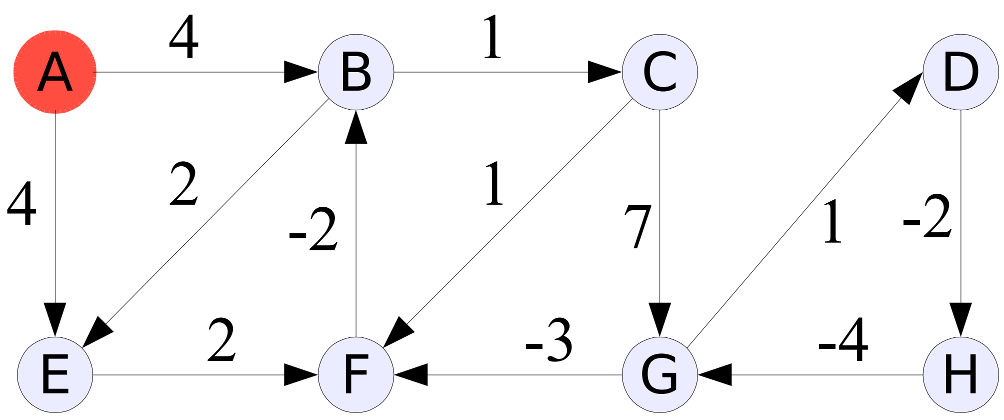
sbaarir@epita.lrde.fr

Remarques

Etant données deux sommets x et y, plusieurs cas se présentent :
1) il n'y a pas de chemins de x à y.
2) il existe un ou plusieurs plus courts chemins de x à y.
3) il existe des chemins de x à y mais pas de plus court.

Exemples :

1) il y a pas de chemins entre A et H (donc, pas de plus court chemin)
2) il existe deux plus courts chemins entre A et B : (A,B) et (A,E,F,B)
3) il existe une infinité de plus courts chemins entre B et F : (B,C,F), (B,C,F,B,C,F)....



4) Il existe des chemins entre D et G mais pas de plus court : les chemins (D,H,G,D,H,G....) sont arbitrairement courts.

Circuit absorbant

Définition :
Un circuit absorbant est un circuit de longueur négative

• Si un graphe possède un circuit absorbant, alors il n'existe pas de plus courts chemins entre certains de ses sommets.

Théorème : Soit G un graphe orienté pondéré n’ayant pas de circuits absorbants, et x et y deux sommets de G. Si il existe un chemin allant de x à y, alors la distance d(x,y) est bien définie et il existe au moins un plus court chemin d e x à y.

Attention : sauf indication contraire, les graphes que nous allons traiter par la suite sont sans circuit absorbant

Propriétés des plus courts chemins

Propriété 1 : Tout sous-chemin d'un plus court chemin est un plus court chemin.

Propriété 2 : Si il existe un plus court chemin entre deux sommets x et y, alors il existe un plus court chemin élémentaire entre x et y.

Calcul de distance : cas d'un graphe pondéré à 1

- C’est un cas particulier de calcul de distance, dans le cas où tous les arcs sont de valuation 1.
- Etant donné un sommet initial x, on cherche à déterminer d(x,y) pour tout sommet y.

- **Principe :**

Un sommet y est à distance n de x si :

- **il existe un chemin de longueur n de x à y,**
- **il n’existe pas de chemin de longueur strictement inférieure à n de x à y.**

Ces deux conditions peuvent se réécrire :

- **y est le successeur d’un sommet à distance n − 1 de x.**
- **La distance de x à y n’est pas plus petite que n.**

Calcul de distance : algorithme

Distance (graphe *G*, sommet *s*)

```
POUR CHAQUE v ≠ s  FAIRE couleur(v) ← Blanc ; distance(v) ← ∞
couleur(s) ← Rouge
distance(s) ← 0
F ← {s}
TANT-QUE  not (FileVide(F)) FAIRE
    s ← Défiler(F )
    POUR CHAQUE v ∈ δ(s)
        SI couleur(v) = Blanc ALORS
            couleur(v) ← Rouge
            distance(v) ← distance(s) + 1
            père(v) ← s
            Enfiler(F,v)
        FIN SI
    FIN POUR
    couleur(s) ← Noir
FIN-TANT-QUE
```

Calculer la complexité dans les deux cas :
1) liste d'adjacence ;
2) matrice d'adjacence

Calcul de distance : liste d'adj.

Distance (graphe *G*, sommet *s*)

POUR CHAQUE $v \neq s$ **FAIRE** couleur(v) ← Blanc ; distance(v) ← ∞	O(|S|)
couleur(s) ← Rouge	
distance(s) ← 0	O(1)
F ← {s}	
TANT-QUE not(FileVide(F)) **FAIRE**	O(|S|)
s ← Défiler(F)	O(|S|)
POUR CHAQUE $v \in \delta(s)$	O(|A|)
SI couleur(v) = Blanc **ALORS**	O(|A|)
couleur(v) ← Rouge	
distance(v) ← distance(s) + 1	
père(v) ← s	O(|A|)
Enfiler(F,v)	
FIN SI	
FIN POUR	
couleur(s) ← Noir	O(|S|)
FIN-TANT-QUE	O(|S| + |A|)

Calcul de distance : matrice d'adj.

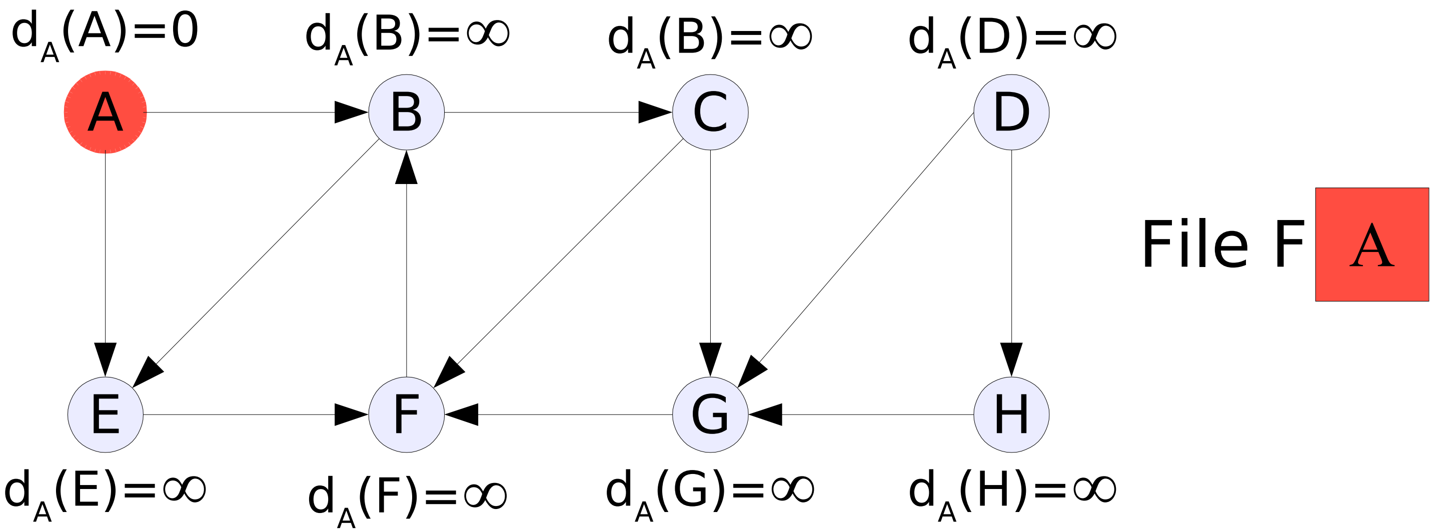
Distance (graphe *G*, sommet *s*)

POUR CHAQUE $v \neq s$ **FAIRE** couleur(v) $\leftarrow$ Blanc ; distance(v) $\leftarrow \infty$ — $O(|S|)$
couleur(s) $\leftarrow$ Rouge
distance(s) $\leftarrow$ 0 — $O(1)$
$F \leftarrow$ {s}
TANT-QUE not(FileVide(F)) **FAIRE** — $O(|S|)$
  $s \leftarrow$ Défiler(F) — $O(|S|)$
  POUR CHAQUE $v \in \delta(s)$ — $O(|S|^2)$
    SI couleur(v) = Blanc **ALORS** — $O(|A|)$
      couleur(v) $\leftarrow$ Rouge
      distance(v) $\leftarrow$ distance(s) + 1 — $O(|A|)$
      père(v) $\leftarrow$ s
      Enfiler(F,v)
    FIN SI
  FIN POUR
  couleur(s) $\leftarrow$ Noir — $O(|S|)$
FIN-TANT-QUE

$O(|S|^2 + |A|)$

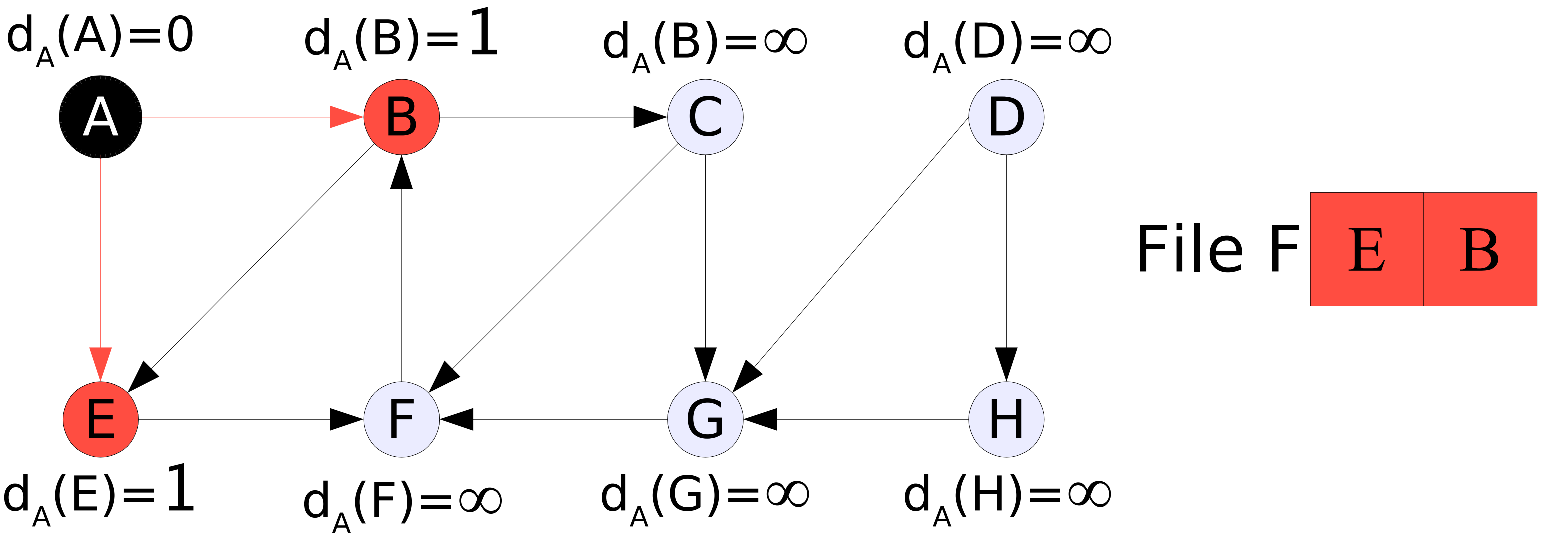
Exemple calcul de distance

- A l'état initial :
  - seul le sommet A est rouge
  - La file est réduite au site A



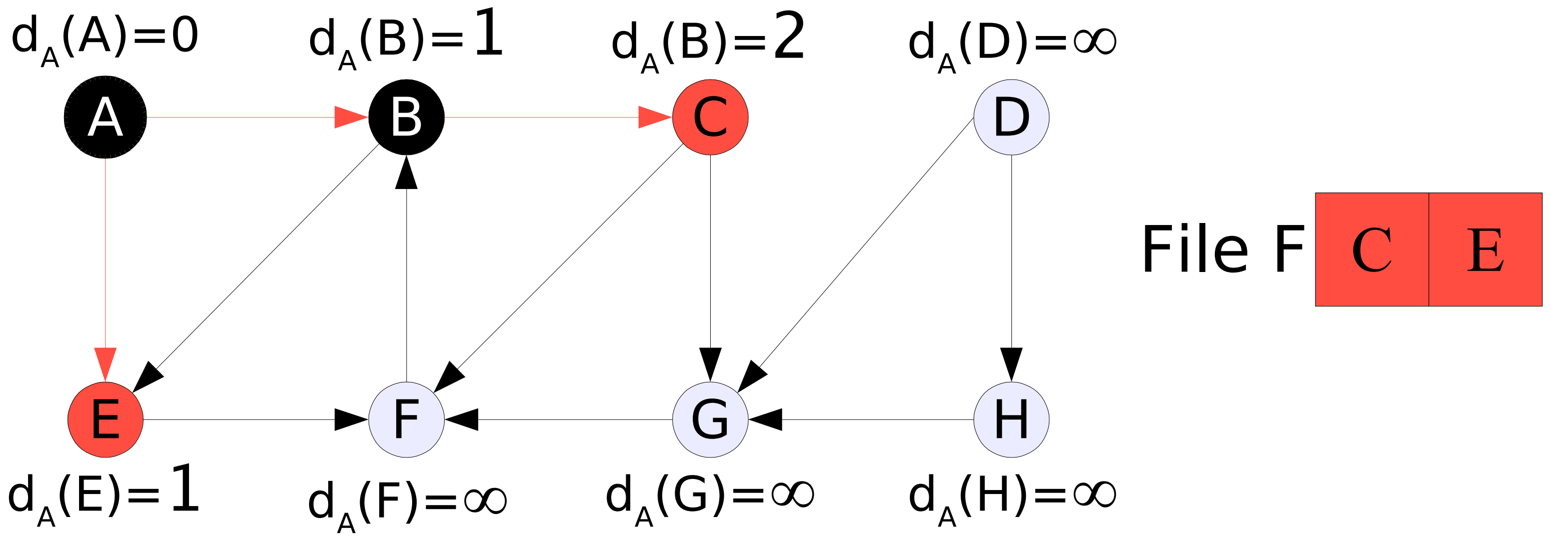
Exemple calcul de distance

- On défile le sommet A
- On visite les voisins blancs de A : B et E
- Le sommet A devient noir



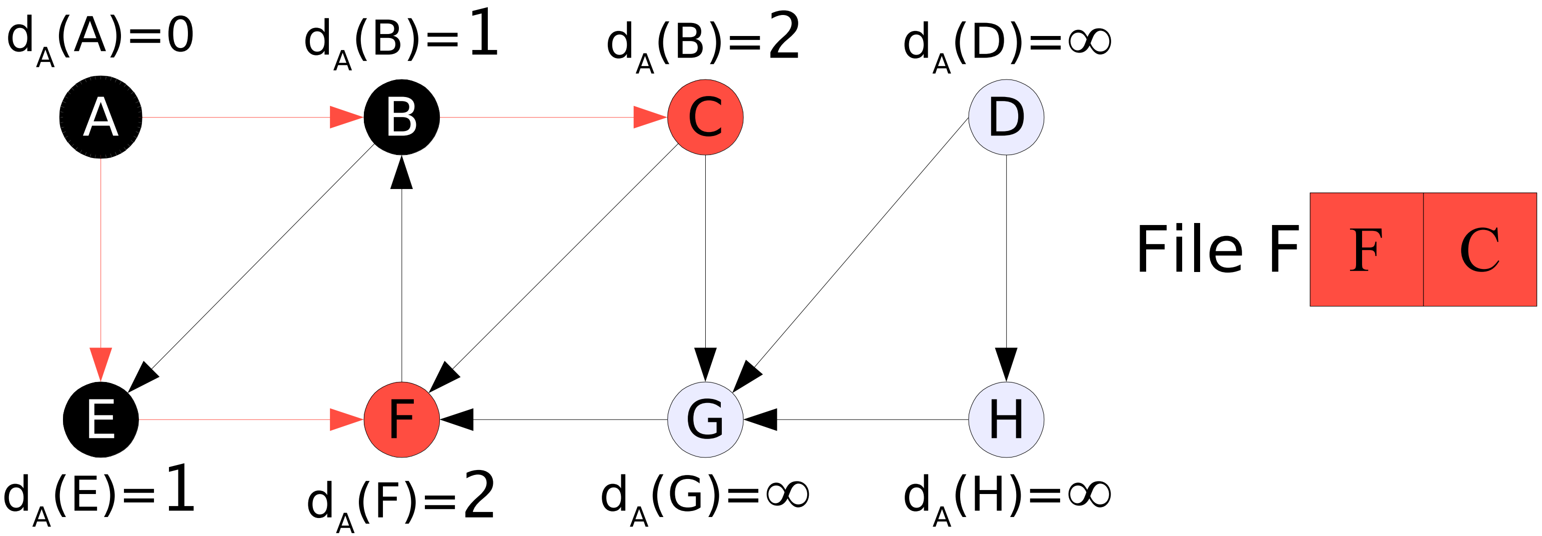
Exemple du BFS et le calcul de distance

- On défile le sommet B
- On visite le voisin blanc de B : C
- Le sommet B devient noir



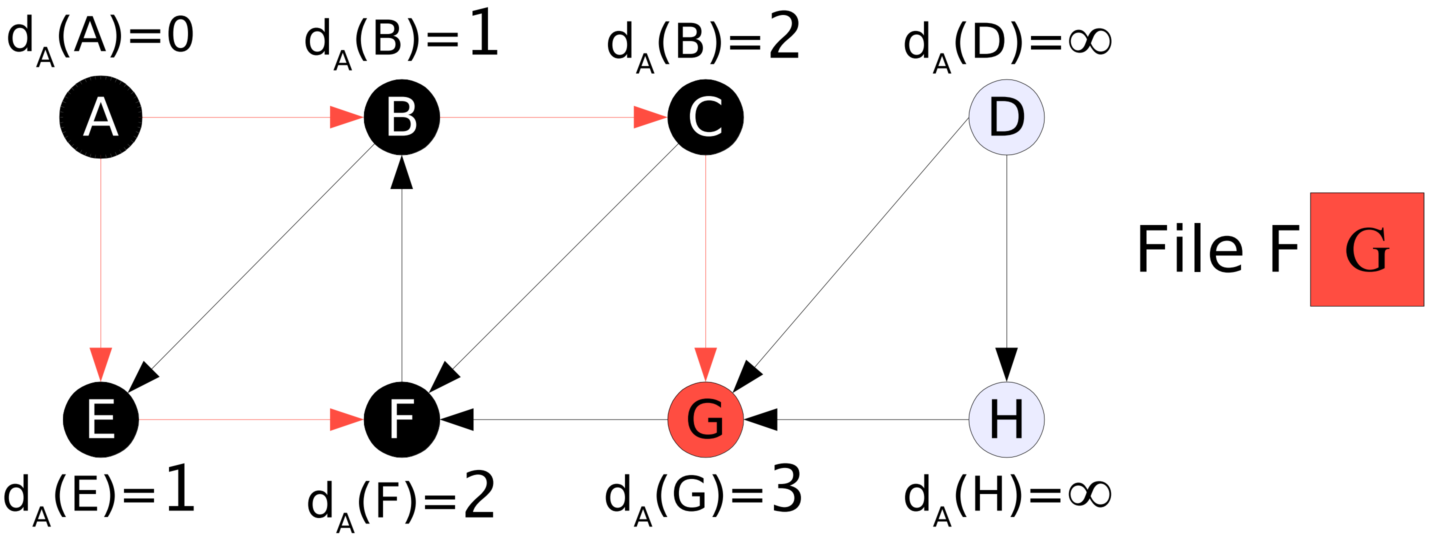
Exemple calcul de distance

- On défile le sommet E
- On visite le voisin blanc de B : F
- Le sommet B devient noir



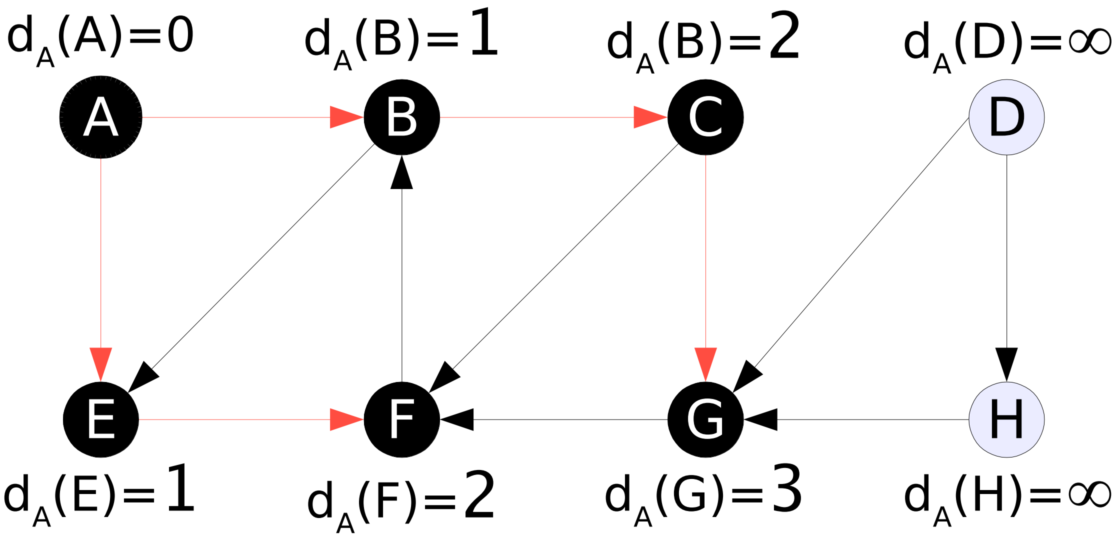
Exemple calcul de distance

- On défile le sommet F
- F n'a pas de voisin blanc
- Le sommet F devient noir



Exemple calcul de distance

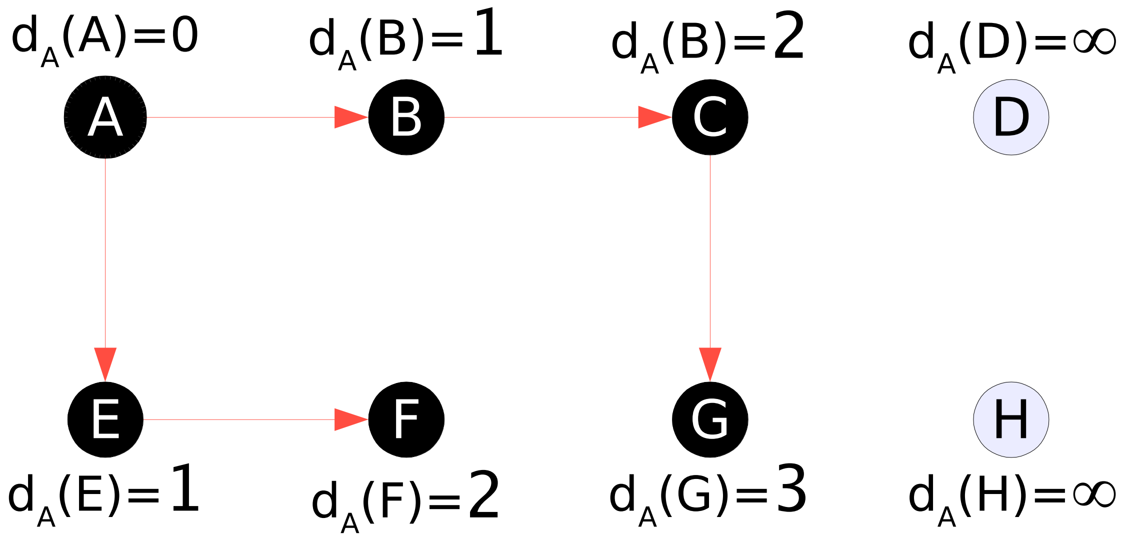
- On défile le sommet G
- G n'a pas de voisin blanc
- Le sommet G devient noir



File F : **VIDE**

Exemple calcul de distance

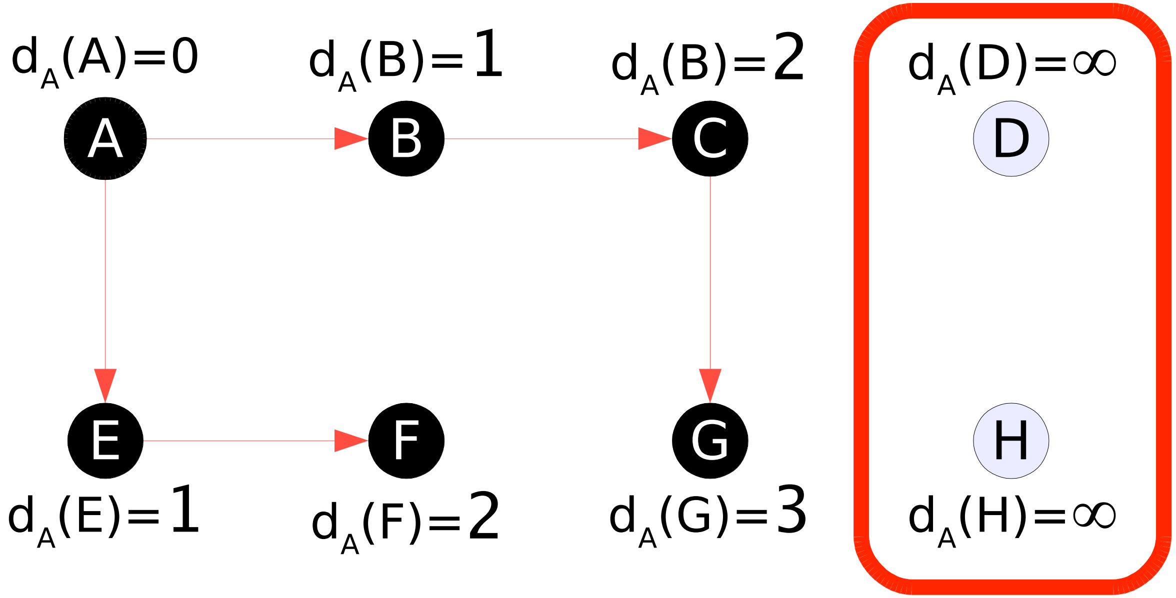
- Il n'y a plus de sommet à défiler : Fin de l'algorithme
- On obtient une **arborescence en largeur**
- $v \in S$, $d_A(v)$ = longueur du **plus court chemin** entre A et v



File F : **VIDE**

Exemple calcul de distance

ATTENTION :
Dans un parcours en largeur : tous les sommets ne sont pas visités
Ainsi les sommet inaccessibles depuis l'origine gardent une distance **∞**



Sommets inaccessibles depuis le sommet A

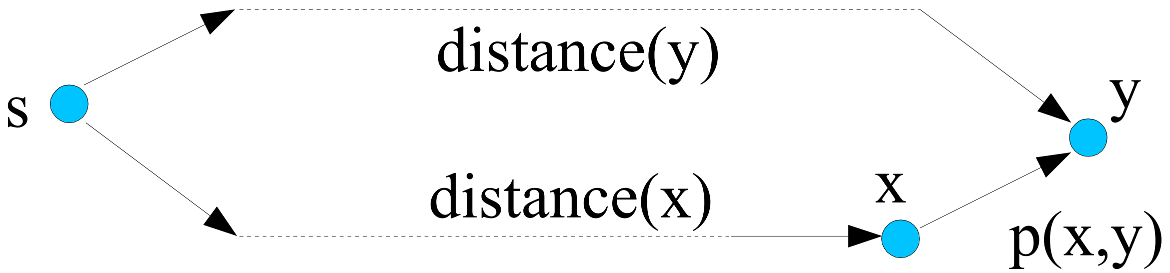
Principes des algorithmes dans le cas général (1/2)

Etant donnés un graphe pondéré et un sommet s, on veut déterminer pour chaque sommet x la distance et un plus court chemin (par rapport à s).

Les algorithmes de recherche de distance et de plus court chemin dans un graphe pondéré fonctionnent de la façon suivante.

•On calcule les distances d(s,x) par **approximations successives**. A un stade donné de l'algorithme on dispose d'estimations*, distance(s),* (éventuellement égales à +∞) pour ces distances, et de la donnée d'un prédécesseur Père(s) pour les plus courts chemins.

Principes des algorithmes dans le cas général (2/2)



- A chaque étape, on essaye d'améliorer les valeurs obtenues précédemment : on considère un sommet x et un successeur y de x. On compare la valeur *distance(y)* a celle que l'on obtiendrait en passant par x, *i.e., distance(x)+p(x,y)*. Si cette deuxième valeur est plus petite que distance(s), on remplace l'estimation *distance(y)* par *distance(x)+p(x,y)* et le père de y par x.

- Cette technique est appelée ***la technique de relaxation***.

- *La question est : comment appliquer la relaxation de façon efficace ?*

calcul de distance dans un graphe pondéré positif : complexité

Distance-pondere-pos (graphe *G*=<*S*,*A*>, sommet *s*)

POUR CHAQUE $v \neq s$ **FAIRE** distance(v) $\leftarrow \infty$ — $O(|S|)$

distance(s) $\leftarrow 0$

$E \leftarrow \varnothing$

TANT-QUE $E \neq S$ **FAIRE** — $O(|S|)$

$s \leftarrow$ choisir($e \in \{v \in S \mid \text{distance}(v)=\min\{\text{distance}(x) \mid x \in S\}\}$) — $O(|S|\log(|S|))$

$E \leftarrow E \cup \{s\}$

POUR CHAQUE $v \in \delta(s)$ — $O(|A|)$

SI distance(v) > distance(s) + p(s,v) **ALORS**

distance(v) $\leftarrow$ distance(s) + p(s,v) — $O(|A|\log(|S|))$

père(v) $\leftarrow$ s

FIN SI

FIN POUR

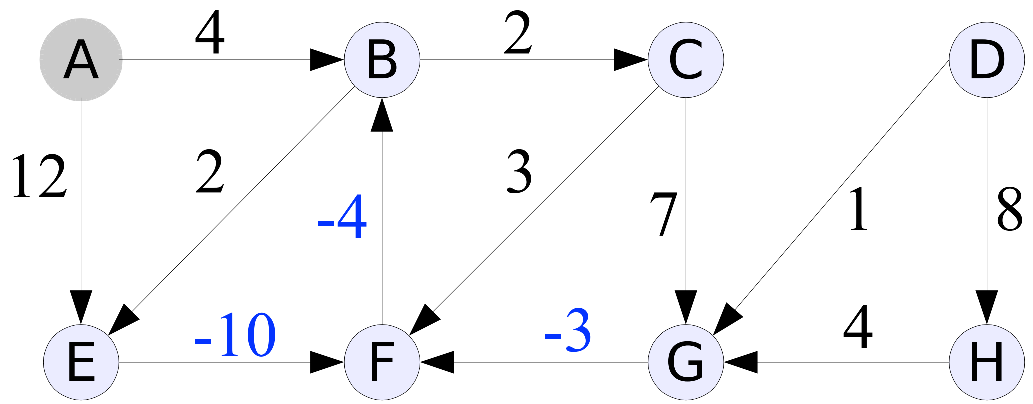
FIN-TANT-QUE

Cas d'un tas binaire

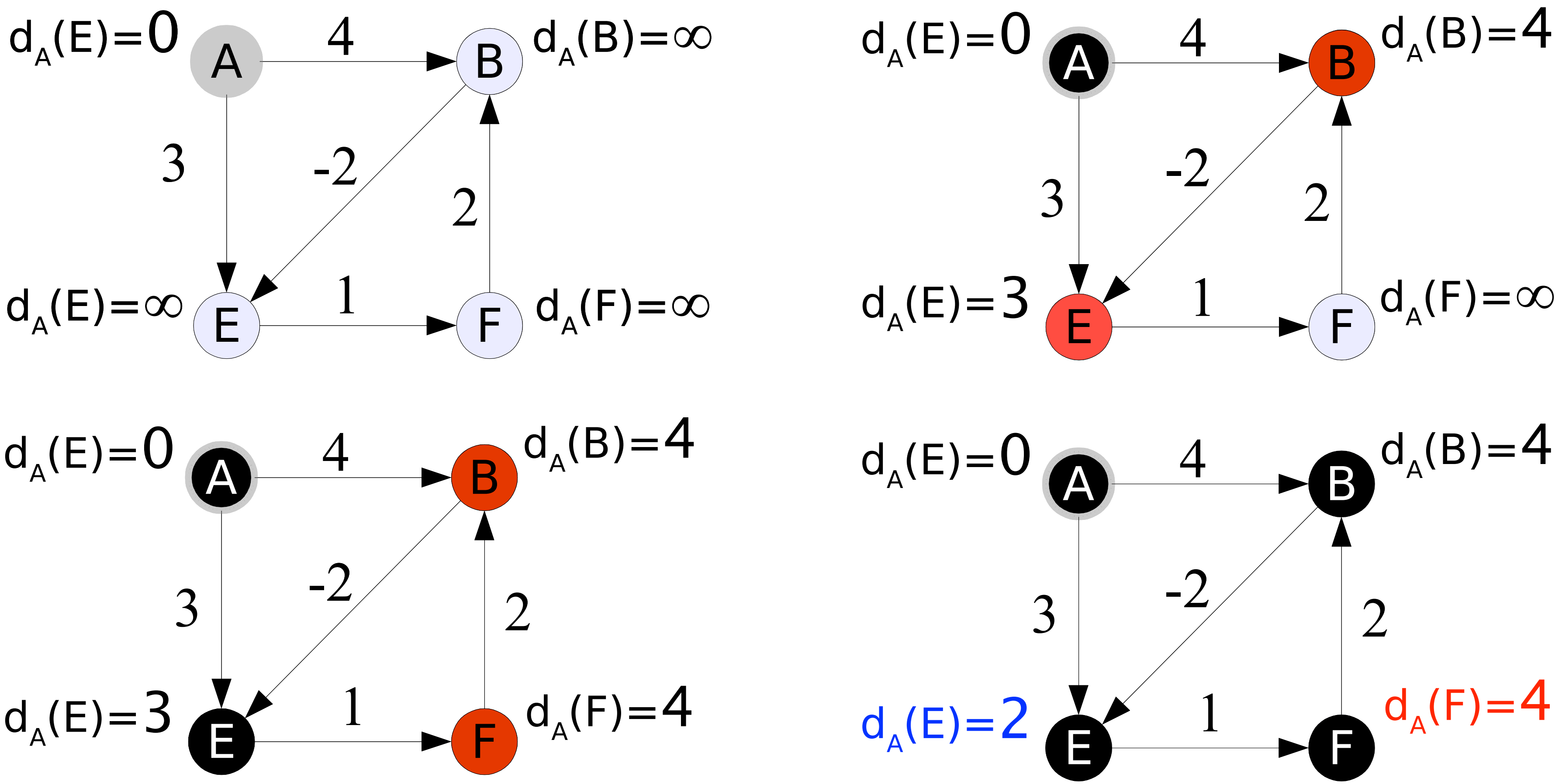
$O((|A|+|S|)\log(|S|))$

 sbaarir@epita.lrde.fr

Graphe pondéré avec poids négatif



L'algorithme de Dijkstra est-il applicable ?



La distance de F n'est pas correcte !

On peut modifier Dijkstra mais la complexité devient exponentielle.

Calcul de distance dans un graphe pondéré avec valeur négatives : Algorithme de Bellman-Ford

Principe :

- L'algorithme de Bellman-Ford utilise le principe général de l'approximation successive,

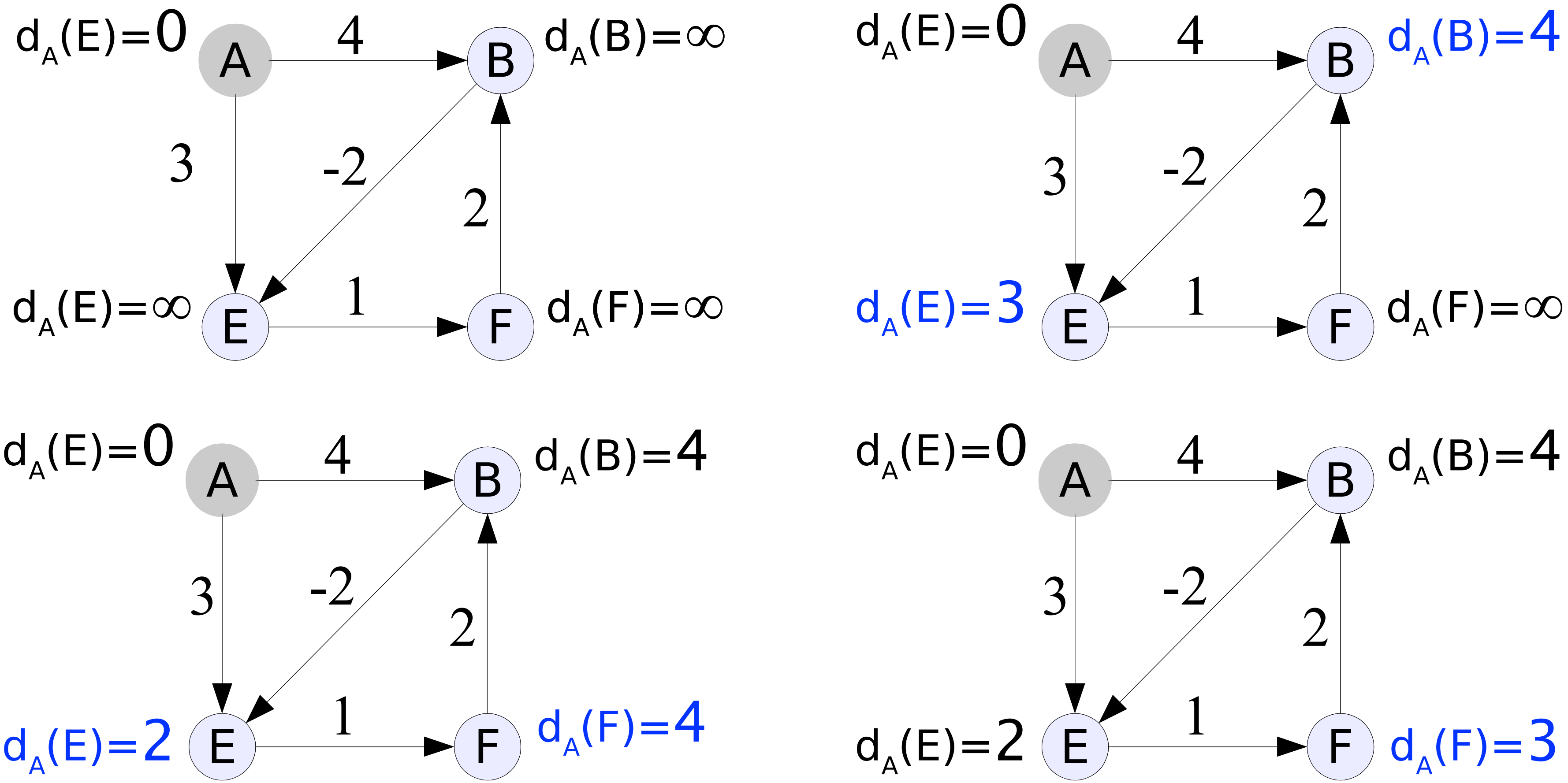
- Contrairement à l'algorithme de Dijkstra, qui sélectionne le minimum à chaque itération,

- Bellman-Ford applique l'approximation sur tous les arcs et ce |S|-1 fois pour garantir que tous les chemins soient visités.

sbaarir@epita.lrde.fr

Calcul de distance dans un graphe pondéré avec valeur négatives : Algorithme de Bellman-Ford

Exemple :



sbaarir@epita.lrde.fr

Calcul de distance dans un graphe pondéré avec valeur négatives : Algorithme de Bellman-Ford

Distance-pondere-neg (graphe G=<S,A>, sommet s)

POUR CHAQUE $v \neq s$ **FAIRE** distance(v) $\leftarrow \infty$
distance(s) $\leftarrow$ 0
POUR i = 1 **à** |S|-1 **FAIRE**
  POUR CHAQUE (s,v) $\in A$
    SI distance(v) > distance(s) + p(s,v) **ALORS**
      distance(v) $\leftarrow$ distance(s) + p(s,v)
      père(v) $\leftarrow$ s
    FIN SI
  FIN POUR
FIN-TANT-QUE

Preuve ?
Sa complexité ?

Calcul de distance dans un graphe pondéré avec valeur négatives : Algorithme de Bellman-Ford (preuve (1/2))

Avant de donner une preuve de l'algorithme, nous allons donner quelques propriétés, dans le cas d'un graphe sans circuit absorbant, qui nous seront utiles :

Lemmes

1. Les valeurs dist(s) ne peuvent que diminuer pendant le déroulement de l'algorithme.
2. A chaque étape de l'algorithme, pour tout sommet s, la valeur dist(s) est soit $+\infty$, soit égale à la longueur d'un chemin de x0 à s.
3. A chaque étape de l'algorithme, dist(s) $\geq$ d(x0, s).
4. Quand la valeur dist(s) atteint d(x0, s), elle ne varie plus dans la suite de l'algorithme.

Preuve.

1. Ce point est évident, puisque les dist(s) ne sont modifiés que lors d'un éventuel relâchement, et ils sont alors diminués.

2. On démontre ce point par récurrence :
– A l'initialisation, tous les sont à infinie sauf dist(x0) qui vaut 0 = d(x0, x0) .
– Supposons que ce soit vrai à une étape de l'algorithme. A l'étape suivante, on remplace éventuellement dist(j) par dist(i)+v(i, j) . Par hypothèse de récurrence, dist(i) est la longueur d'un chemin C(x0, i) et donc dist(i) + v(i, j) est la longueur du chemin obtenu en ajoutant l'arc (i, j) au chemin C(x0, i) .

3. C'est une conséquence de 1 et 2, car le graphe ne contient pas de circuit absorbant.

4. Une fois la valeur d(x0, s) atteinte, la technique de relâchement n'a plus aucun effet.

Calcul de distance dans un graphe pondéré avec valeur négatives : Algorithme de Bellman-Ford (preuve (2/2))

Preuve. (Algorithme de Bellman-Ford) Nous devons prouver qu'à la fin de l'algorithme, si le graphe ne contient pas de circuit absorbant, le tableau contient les plus courtes distances à partir du sommet x_0. Pour cela, nous allons démontrer par récurrence sur k la propriété :

(P_k) Si un plus court chemin élémentaire de x_0 à un sommet s comporte k arcs, alors après k passages dans la boucle, on a

$$dist(s) = d(x_0, s).$$

– A l'initialisation, c'est clairement vrai.

– Soit p le prédécesseur de s dans un plus court chemin élémentaire comportant k arcs entre x_0 et s. Alors, il existe un plus court chemin élémentaire comportant $k - 1$ arcs entre x_0 et p, et donc, en utilisant l'hypothèse de récurrence, on en déduit qu'après $k - 1$ passages dans la boucle, on a $dist(p) = d(x_0, p)$.
Après le k-ième passage, on compare dist(s) et dist(p) + v(p, s), on a alors après changement éventuel de dist(s), l'inégalité $dist(s) \leq dist(p) + v(p, s)$. On en déduit que

$$\begin{aligned} dist\ (s) &\leq d(x_0, p) + v(p, s) \\ &\leq d(x_0, s) \end{aligned}$$

et donc, en utilisant le 3. du lemme, $dist(s) = d(x_0, s)$.

Le 4. du lemme nous dit qu'une fois les "bonnes" valeurs atteintes, elles ne changent plus.

Il reste à remarquer que dans un graphe à n sommets, un chemin élémentaire a au plus $n-1$ arcs et qu'un plus court chemin est nécessairement élémentaire. On est alors assuré, en au plus $n-1$ étapes, avoir $distance(s) = d(x_0, s)$ pour tous les sommets s. De plus, on a les prédécesseurs de chaque sommet dans un plus court chemin qui sont stockés dans père.

Calcul de distance dans un graphe pondéré avec valeur négatives : Algorithme de Bellman-Ford

Distance-pondere-pos (graphe *G*, sommet *s*)

POUR CHAQUE $v \neq s$ **FAIRE** distance(v) $\leftarrow \infty$ — $\Theta(|S|)$
distance(s) $\leftarrow 0$
POUR i = 1 **à** |S|-1 **FAIRE** — $\Theta(|S|)$
  POUR CHAQUE (s,v) $\in A$ — $\Theta(|S|*|A|)$
    SI distance(v) > distance(s) + p(s,v) **ALORS**
      distance(v) $\leftarrow$ distance(s) + p(s,v)
      père(v) $\leftarrow$ s
    FIN SI
  FIN POUR
FIN-TANT-QUE

$\Theta(|A|*|S|)$

Comment modifier l'algorithme de façon à avoir une complexité de O(|S|*|A|) ?

Calcul de toutes les distances dans un graphe pondéré avec valeur négatives

- **Problème :** calculer les distances entre toutes les paires de sommets

- **Première solution :**
  - Appliquer Bellman-Ford pour chaque sommet
  - Complexité : $\mathbf{O(|S|^2*|A|)}$ **….**Trop cher si le graphe est dense ($\mathbf{O(|S|^4)}$) !

- **Solution pour le cas sparse :**
  - Algorithme de Johnson...
  - Se réduire à un graphe dont les poids sont tous positifs, puis appliquer n fois l'algorithme de Dijkstra, une fois à partir de chaque sommet.

- **Solution pour le cas dense :**
  - Algorithme de Floyd-Warshall...

Calcul de toutes les distances dans un graphe pondéré avec valeur négatives : Algorithme de Johnson

Distances-Johnson (graphe *G*)

1. Ajouter un état q, relié à tous les autres avec un poids 0

2. Lancer **Bellman-Ford** à partir de q pour trouver les poids h de chaque état

3. Modifier le graphe avec $p(u,v) = p(u,v) + h(u) - h(v)$

4. Appliquer **Dijkstra |S|** fois sur le graphe (maintenant positif)

Preuve ?
Complexité ?

Calcul de toutes les distances dans un graphe pondéré avec valeur négatives : Algorithme de Johnson (preuve)

Lemme 1: les poids des arcs du graphe modifié sont positifs ou nul

Preuve
On sait que pour tout sommet, v, du graphe h(v) <= 0 : le sommet q est relié à chaque autre sommet par un arc de poids 0, donc un plus court chemin entre q et tout autre sommet v est forcement inférieur ou égale à 0 => (h(v) <=0). Si l'on prend un sommet v et prédécesseur u de ce sommet on a h(v) <= p(u,v)+ h(u) <=> 0 <= p(u,v)+ h(u) – h(v), et c'est la quantité qu'on rajoute à chaque arc.

Lemme 2:
Le poids de chaque chemin, entre la paire de sommets u et u', du graphe modifié est augmenté par la quantité $h(u) - h(u')$.

Preuve :
Soit c=(u,v1,...vn,u') un chemin entre u et u'. Son poids w est donné par l'expression suivante :
  w=(p(u,v1)+ h(u) – h(v1)) + (p(v1,v2)+ h(v1) – h(v2)) +....+(p(vn,u')+ h(vn) – h(u')).
On remarque ici, que chaque h(vi) est supprimé par le -h(vi) précédent, et ceci implique que :
  w= p(u,v1)+ p(v1,v2)+(p(vn,u') + h(u) – h(u').

Ainsi, Dijkstra est appliqué sur un graphe positif (lemme 1) et va donner les distances du graphe modifié. Pour retrouver les distances dans le graphe original il faut retrancher la quantité (h(u) – h(u')) de distance(u,u') pour chaque couple de sommets (conséquence directe du lemme 2).

Calcul de toutes les distances dans un graphe pondéré avec valeur négatives : Algorithme de Johnson

Distances-Johnson (graphe *G*)

1. Ajouter un état *q*, relié à tous les autres avec un poids 0 — $\Theta(|S|)$

2. Lancer **Bellman-Ford** à partir de *q* pour trouver les poids *h* de chaque état — $O(|S|*|A|)$

3. Modifier le graphe avec *w(u,v) = h(u) – h(v) + w(u,v)* — $\Theta(|A|)$

4. Appliquer **Dijkstra |S|** fois sur le graphe (maintenant positif) — $|S|\ O((|A|+|S|)\log(|S|))$

$O((|A||S|+|S|^2)\log(|S|))$

$= O((|S|^3)\log(|S|))$

(si graphe dense)

Calcul de toutes les distances dans un graphe pondéré avec valeur négatives : Algorithme de Floyd-Warshall

AU TABLEAU...

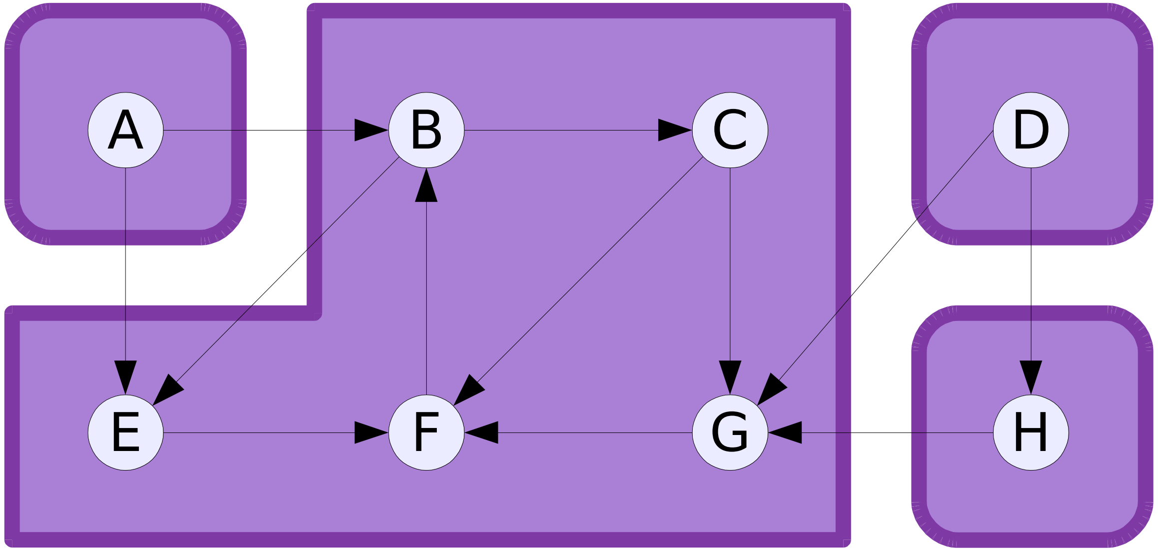
III

Algorithmes de calcul de compostes fortement connexes

Motivation

Le calcul compostantes fortement connexes présente un interêt majeur dans plusieurs domaines :

- **Identifier des relations fortes entre les groupes sur les réseaux sociaux.**
- **La base de plusieurs techniques de vérification de systèmes (2-SAT, *Model-Checking*.**
- **...**



Algorithme récursif DFS

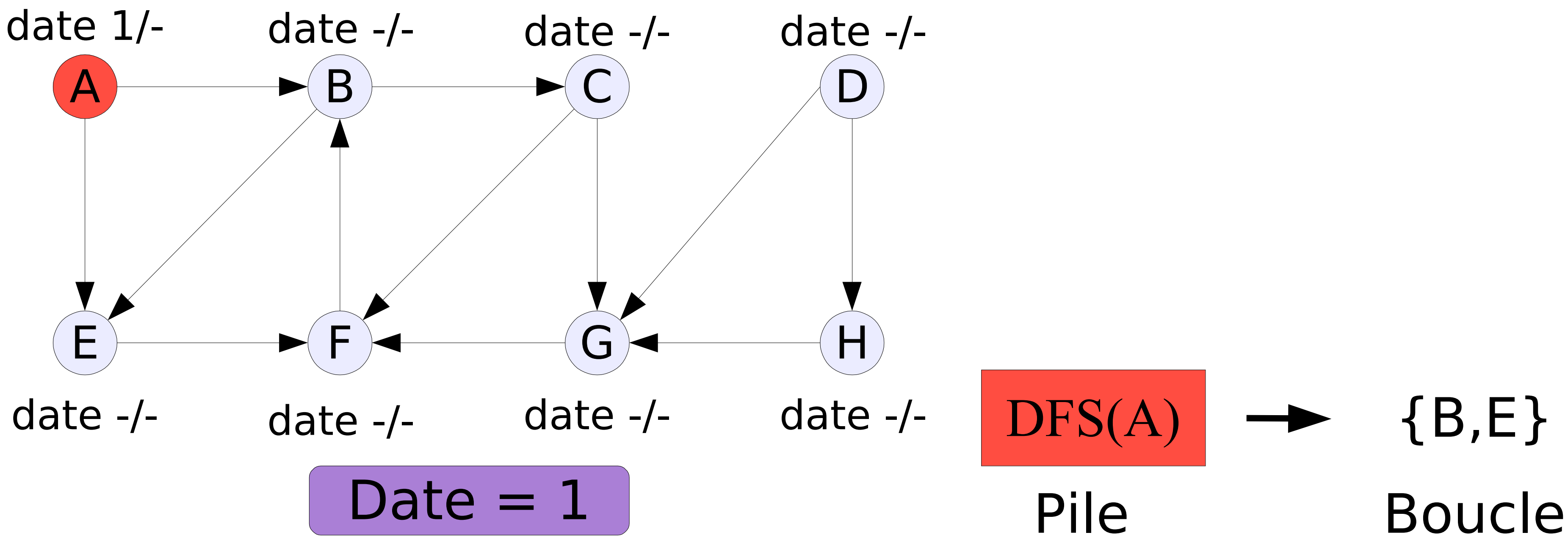
VARIABLE
date : un compteur d'étape

DFS_run (graphe *G*)
POUR CHAQUE $s \in S$ **FAIRE**
couleur(s) ← Blanc
FIN POUR
date ← 0
POUR CHAQUE $s \in S$ **FAIRE**
SI couleur(s) = Blanc **ALORS**
DFS (G , s)
FIN SI
FIN POUR

DFS (graphe *G*, sommet *s*)
couleur(s) ← Rouge
dateDebut(s) ← ++date
POUR CHAQUE $v \in \delta(s)$
SI couleur(v) = Blanc **ALORS**
DFS (G,v)
FIN SI
FIN POUR
dateFin(s) ← ++date

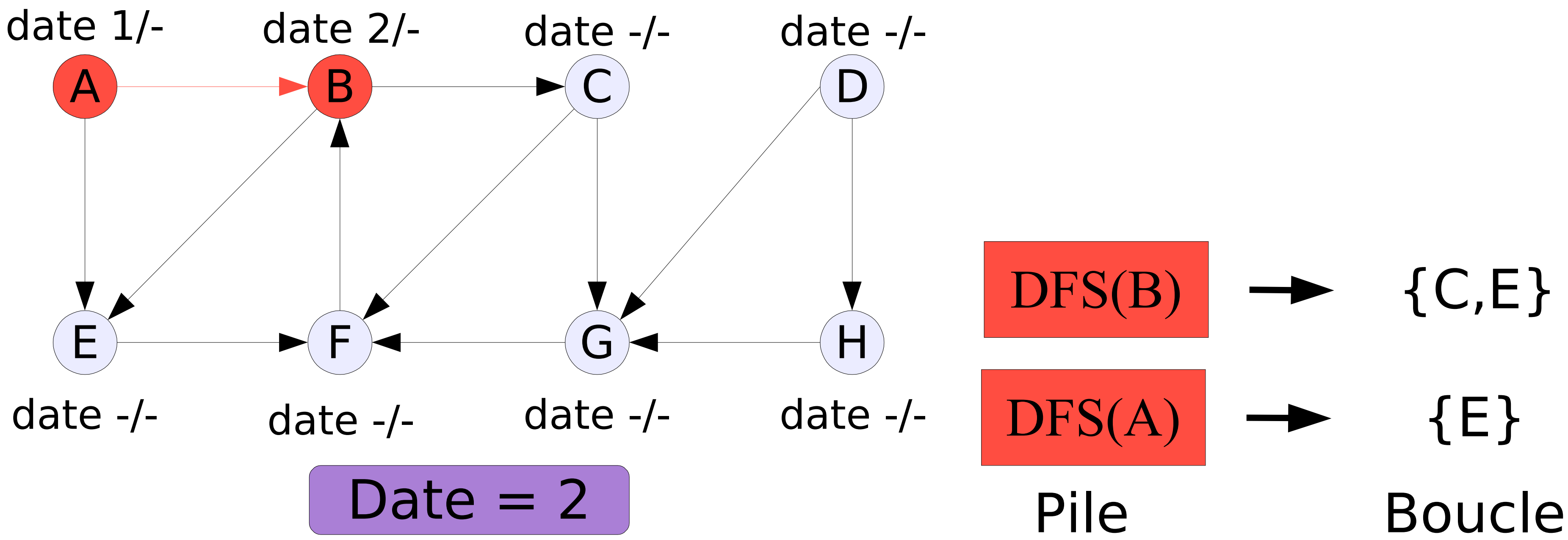
Exemple de l'algorithme DFS

- DFS_run appelle de DFS(A).
- Seul le sommet A est rouge
- La pile des appels de fonction est réduite au DFS(A)



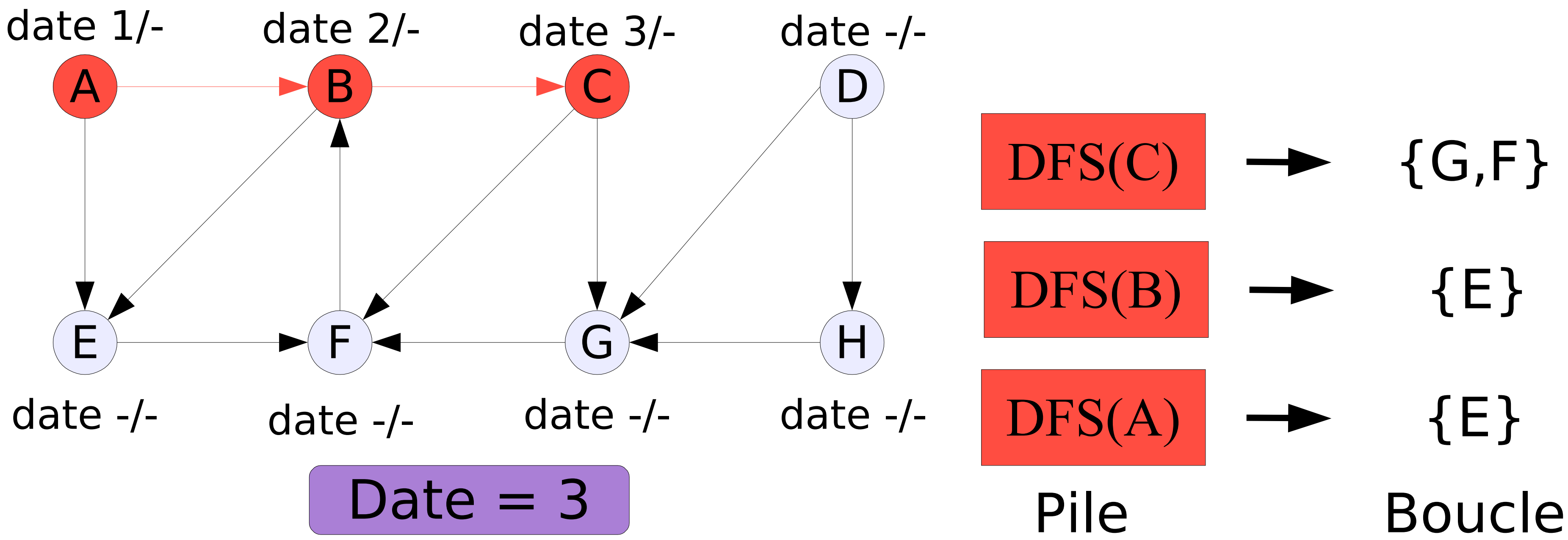
Exemple de l'algorithme DFS

- On empile la fonction DFS(B)
- B devient rouge et on note la date : dateDebut(B) ← 2



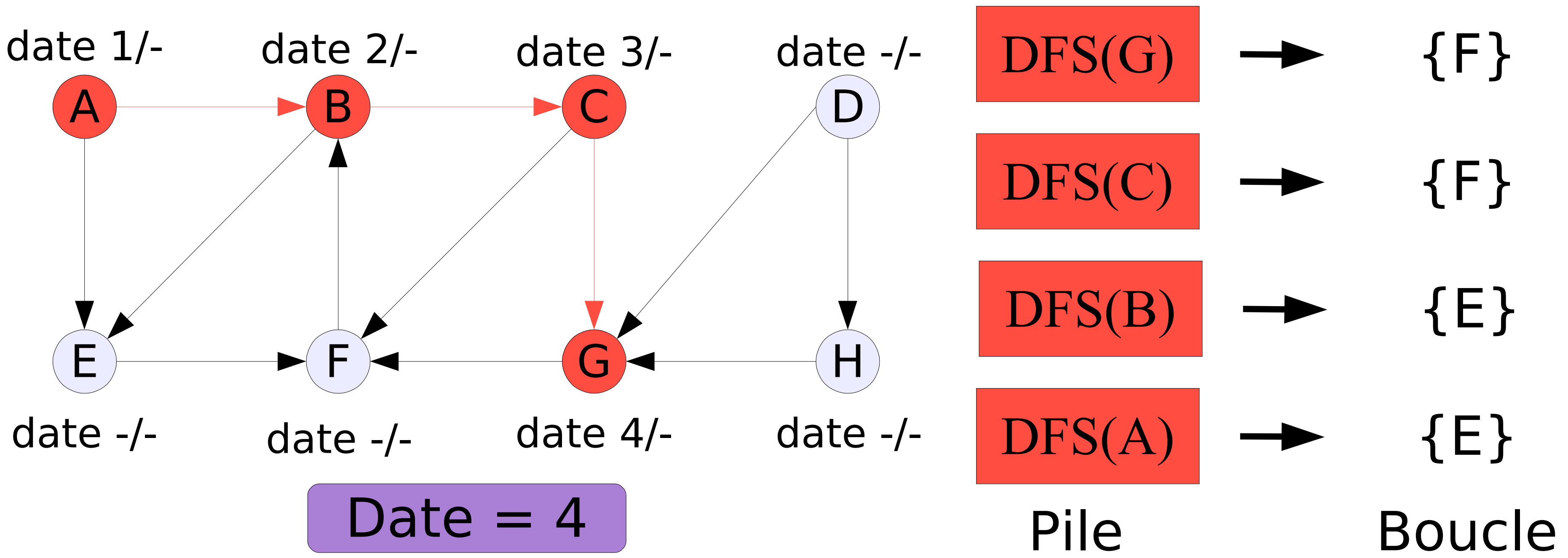
Exemple de l'algorithme DFS

- On empile la fonction DFS(C)
- C devient rouge et on note la date : dateDebut(B) ← 3



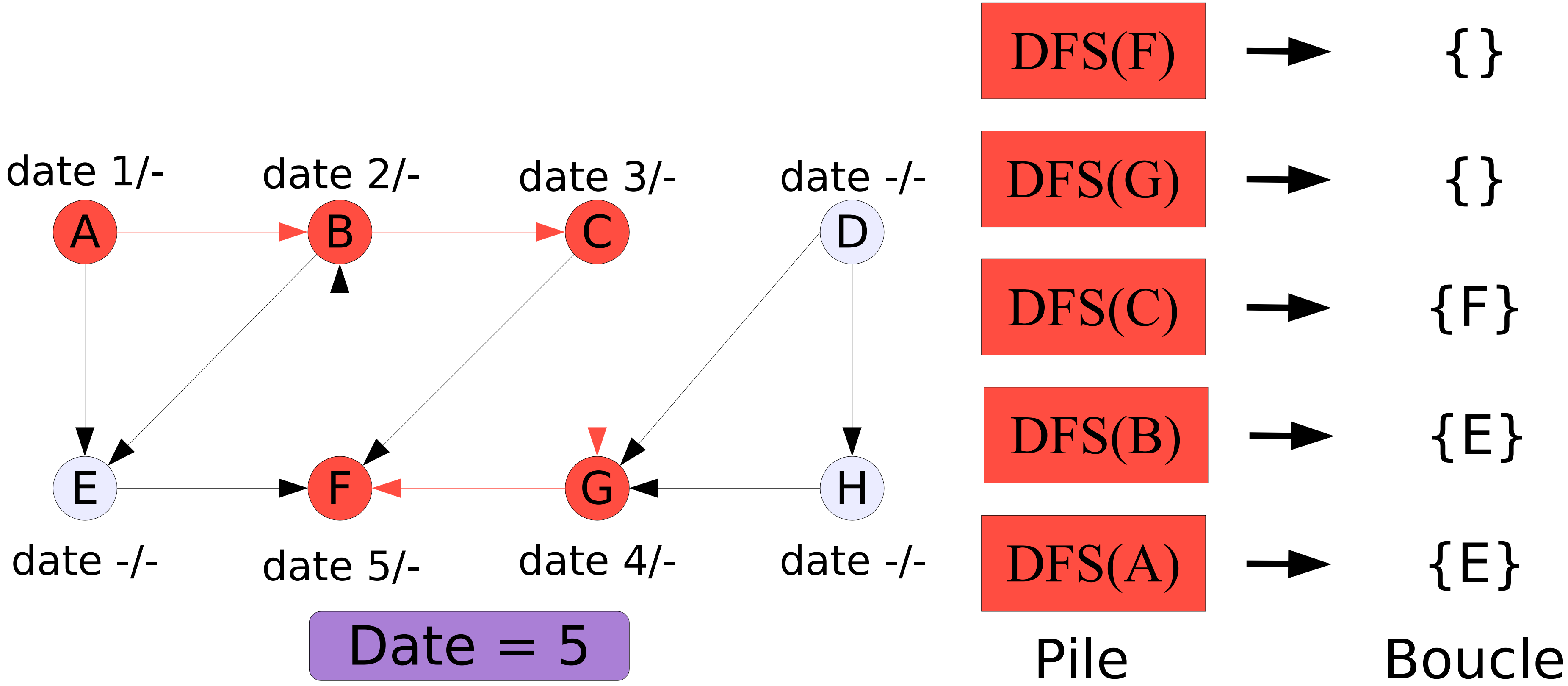
Exemple de l'algorithme DFS

- On empile la fonction DFS(G)
- G devient rouge et on note la date : dateDebut(G) ← 4



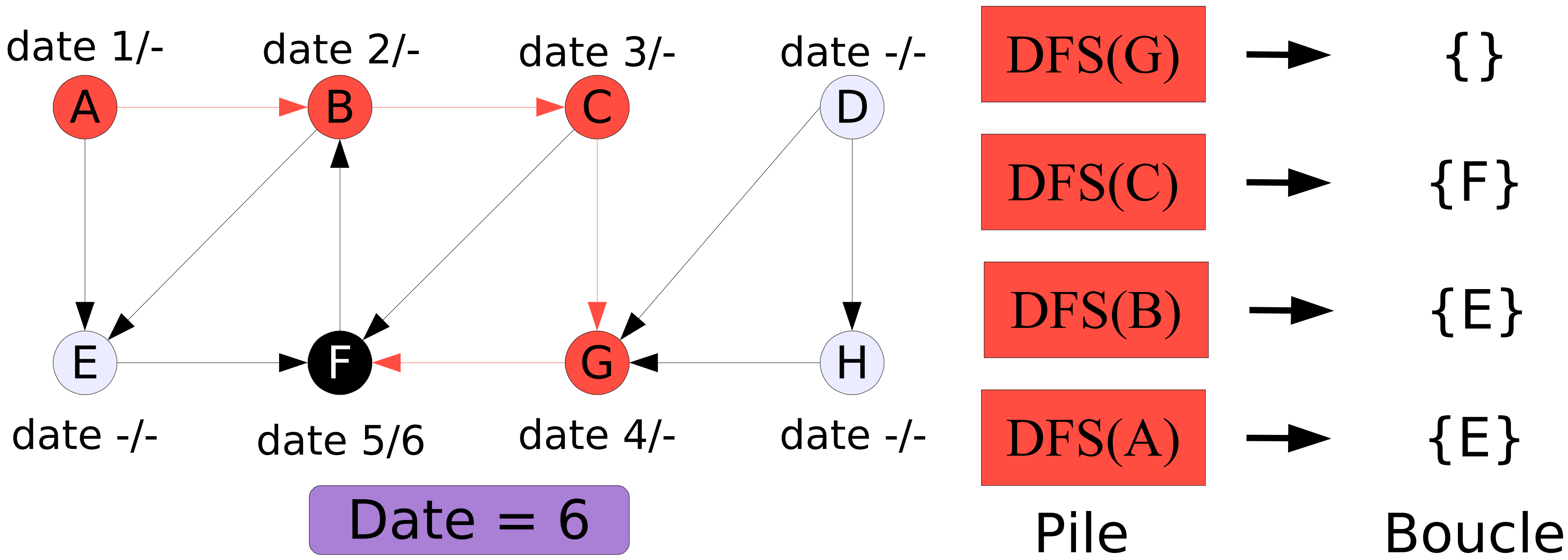
Exemple de l'algorithme DFS

- On empile la fonction DFS(F)
- F devient rouge et on note la date : dateDebut(F) ← 5



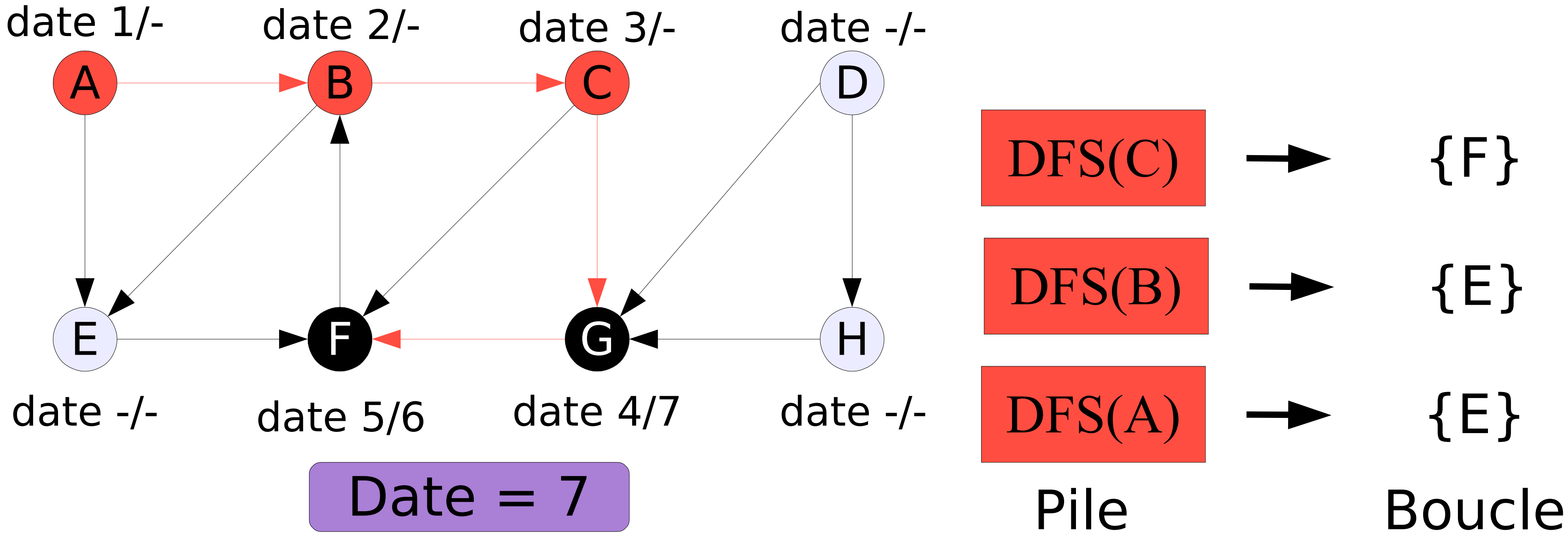
Exemple de l'algorithme DFS

- Fin de la boucle dans la fonction DFS(F)
- F devient noir et on note la date : dateFin(F) ← 6



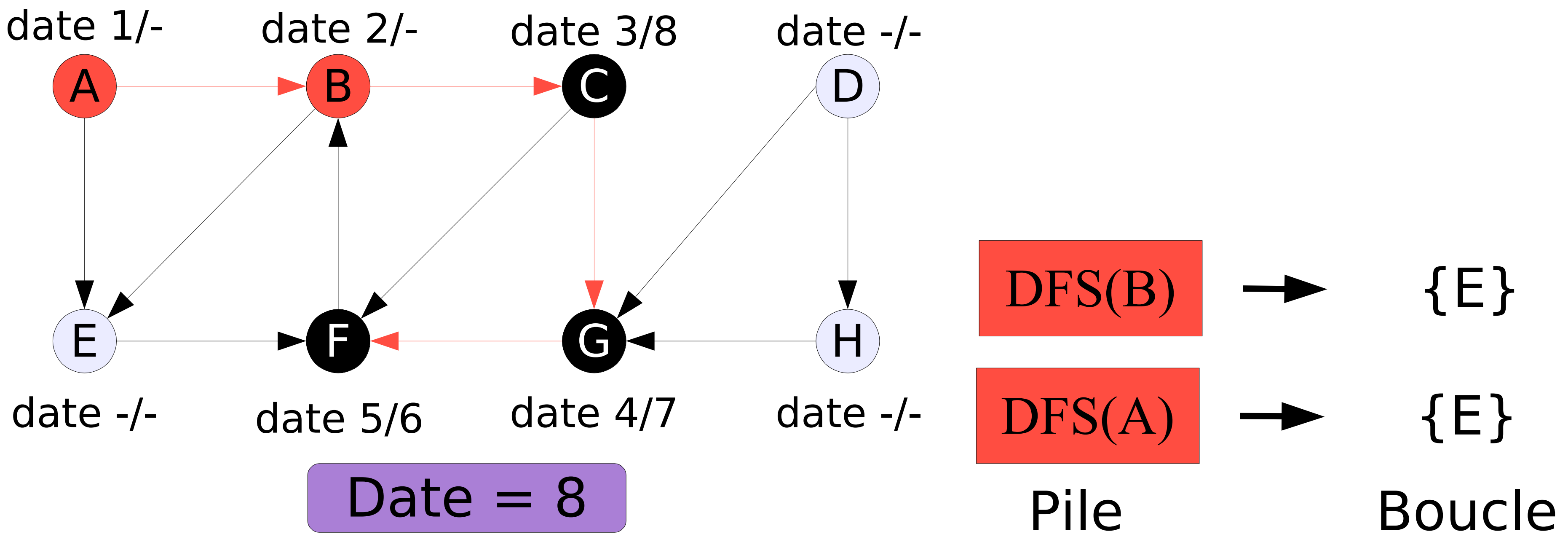
Exemple de l'algorithme DFS

- Fin de la boucle dans la fonction DFS(G)
- F devient noir et on note la date : dateFin(G) ← 7



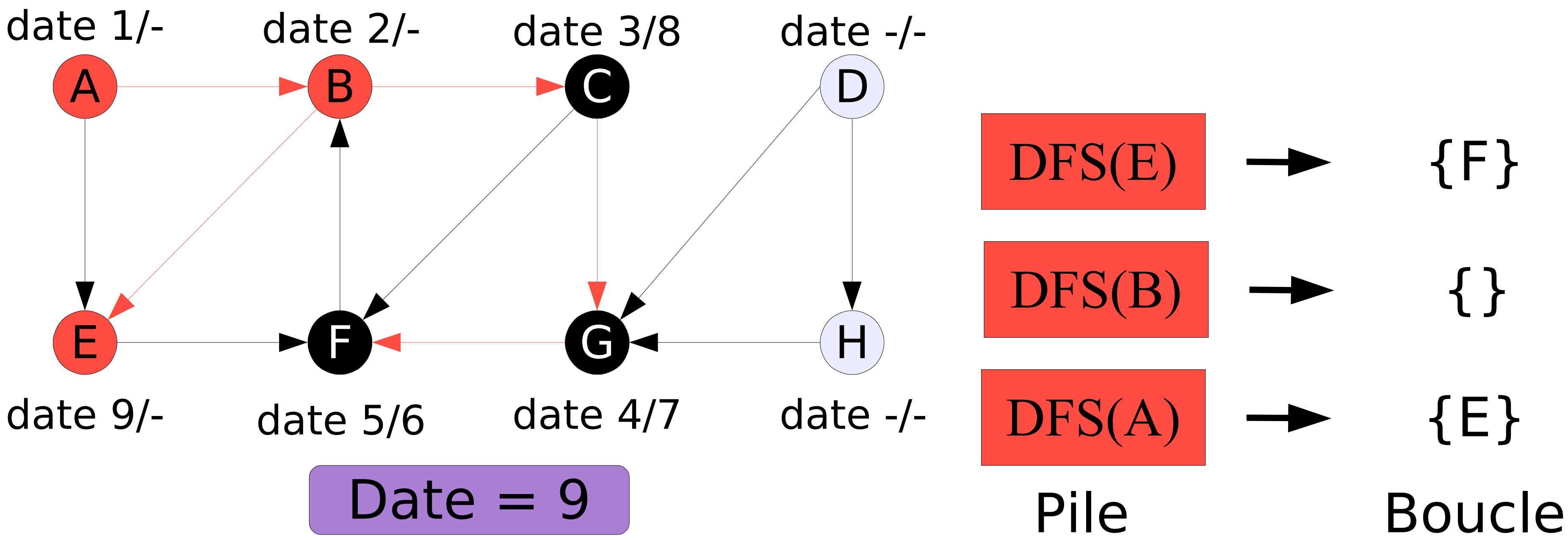
Exemple de l'algorithme DFS

- Le sommet F n'est pas blanc ⇒ pas d'appel à DFS(F)
- Fin de la boucle dans la fonction DFS(C)
- C devient noir et on note la date : dateFin(C) ← 8



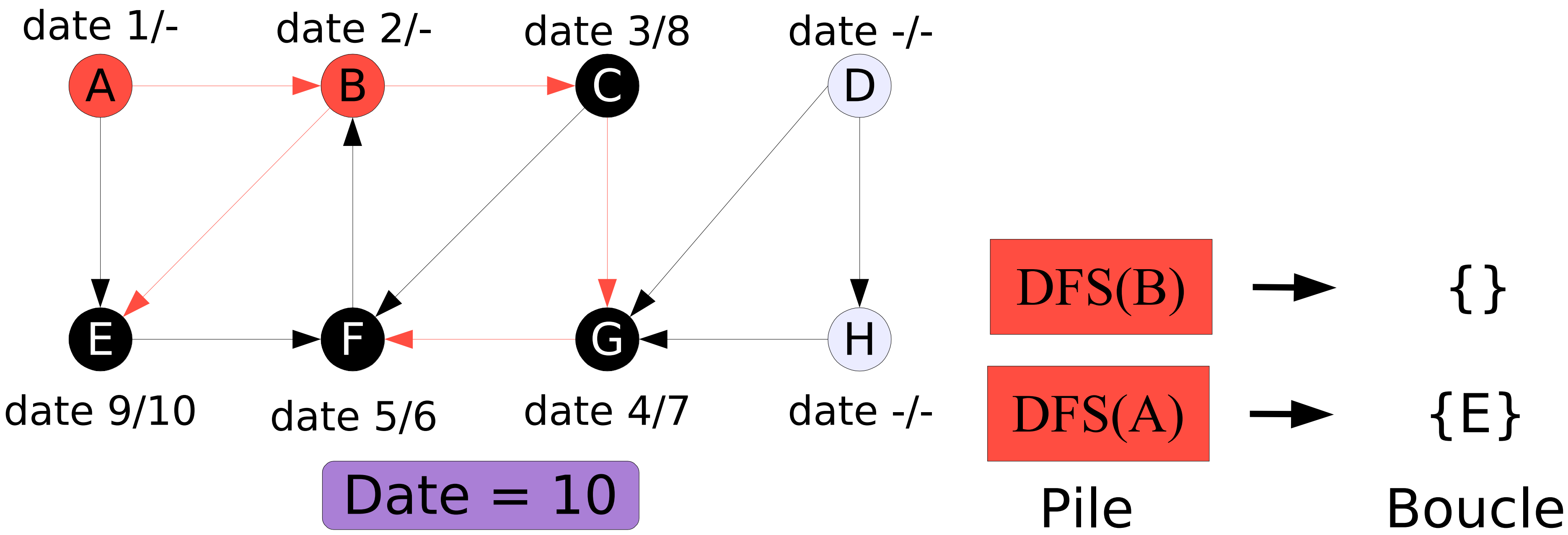
Exemple de l'algorithme DFS

- Retour à la boucle dans la fonction DFS(B)
- On empile la fonction DFS(E)
- E devient rouge et on note la date : dateDebut(E) ← 9



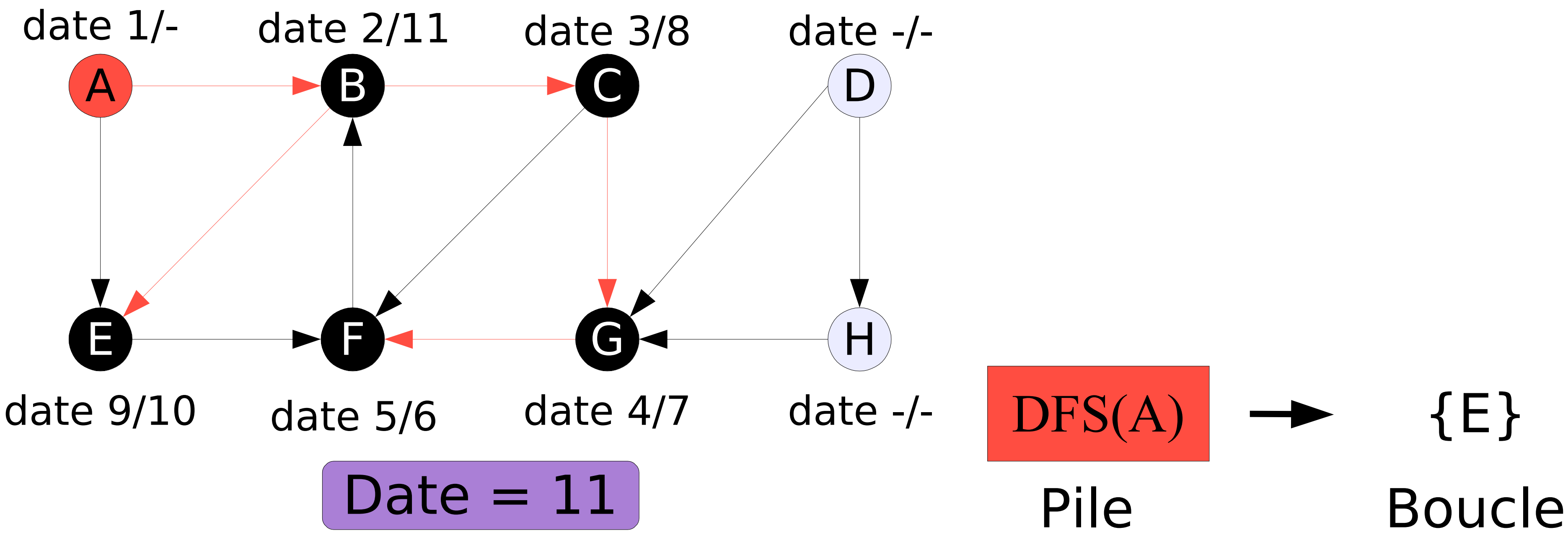
Exemple de l'algorithme DFS

- Le sommet F n'est pas blanc ⇒ pas d'appel a DFS(E)
- Fin de la boucle dans la fonction DFS(E)
- E devient noir et on note la date : dateFin(E) ← 10



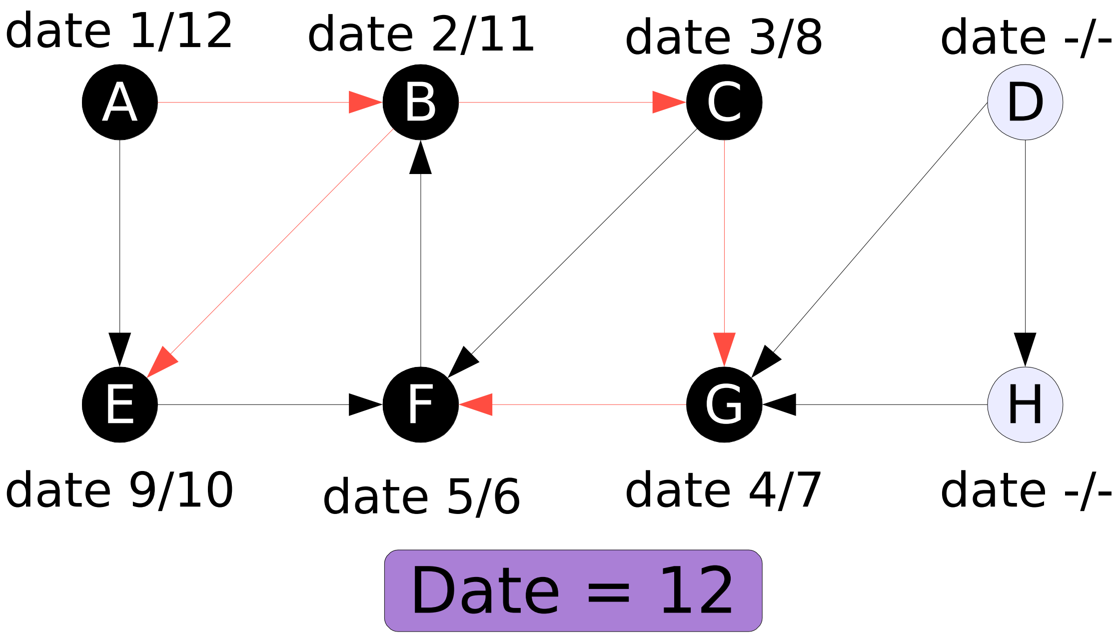
Exemple de l'algorithme DFS

- Fin de la boucle dans la fonction DFS(B)
- B devient noir et on note la date : dateFin(B) ← 11



Exemple de l'algorithme DFS

- Le sommet E n'est pas blanc ⇒ pas d'appel à DFS(A)
- Fin de la boucle dans la fonction DFS(A)
- A devient noir et on note la date : dateFin(A) ← 12

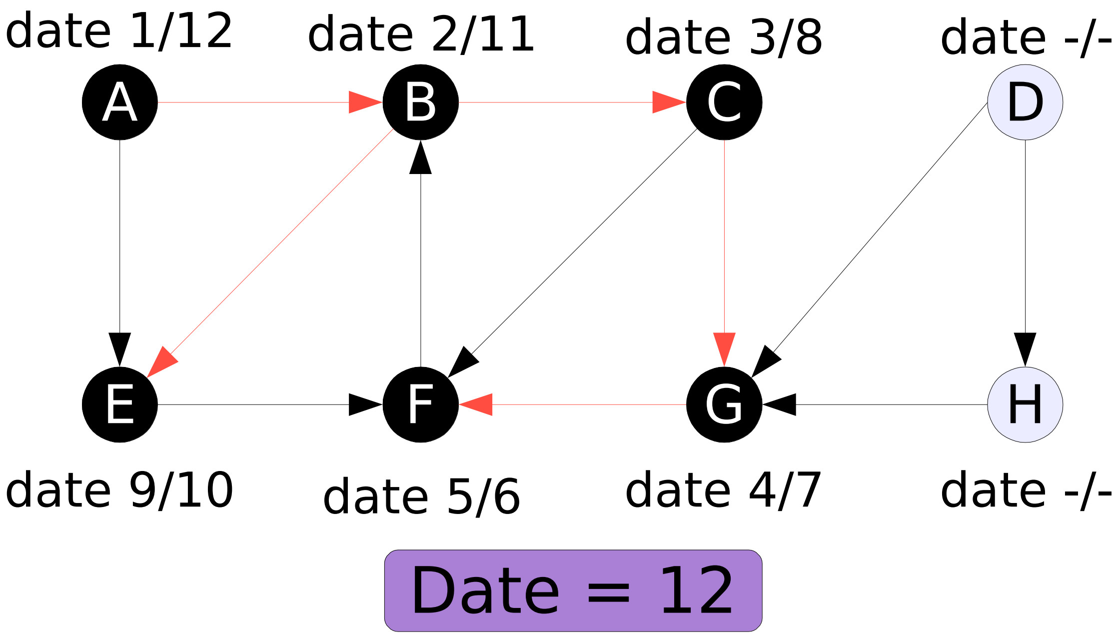


Pile

Boucle

Exemple de l'algorithme DFS

- L'appel à la fonction DFS(A) est terminé.
- La boucle principale de DFS_run() appel ensuite DFS(A) et DFS(B) qui ce terminent tout de suite : « pas de voisin blanc »



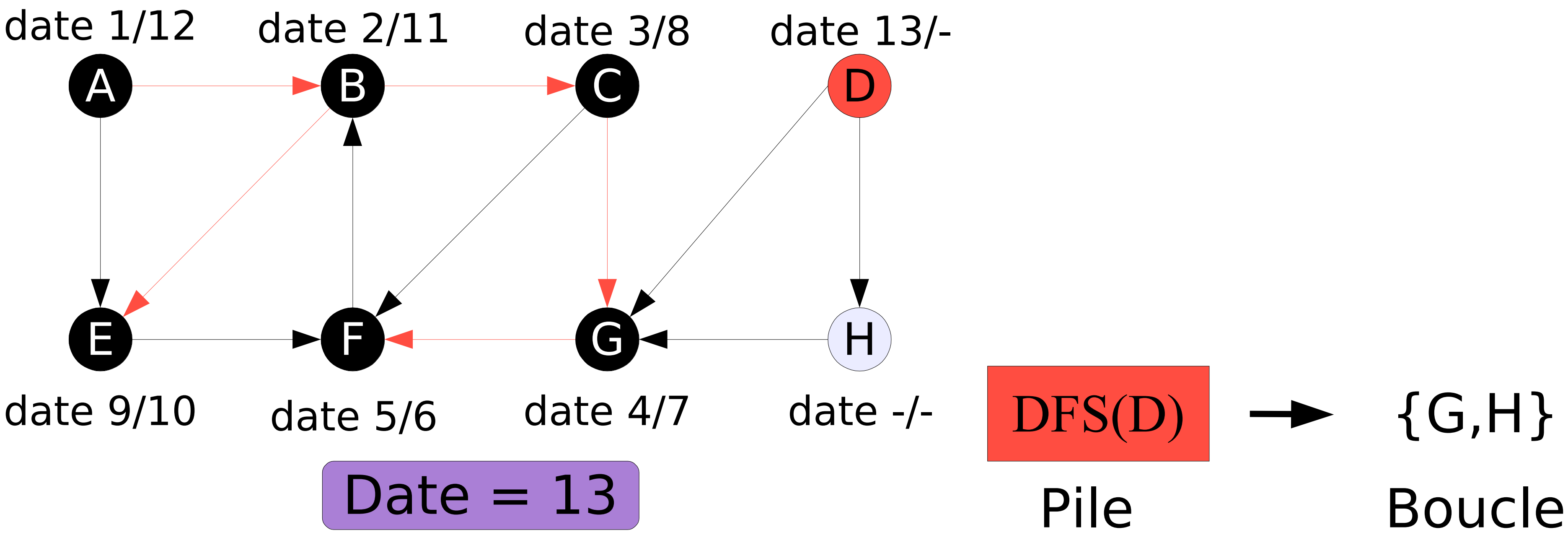
ATTENTION :
La variable date n'est pas réinitialisée

Pile

Boucle

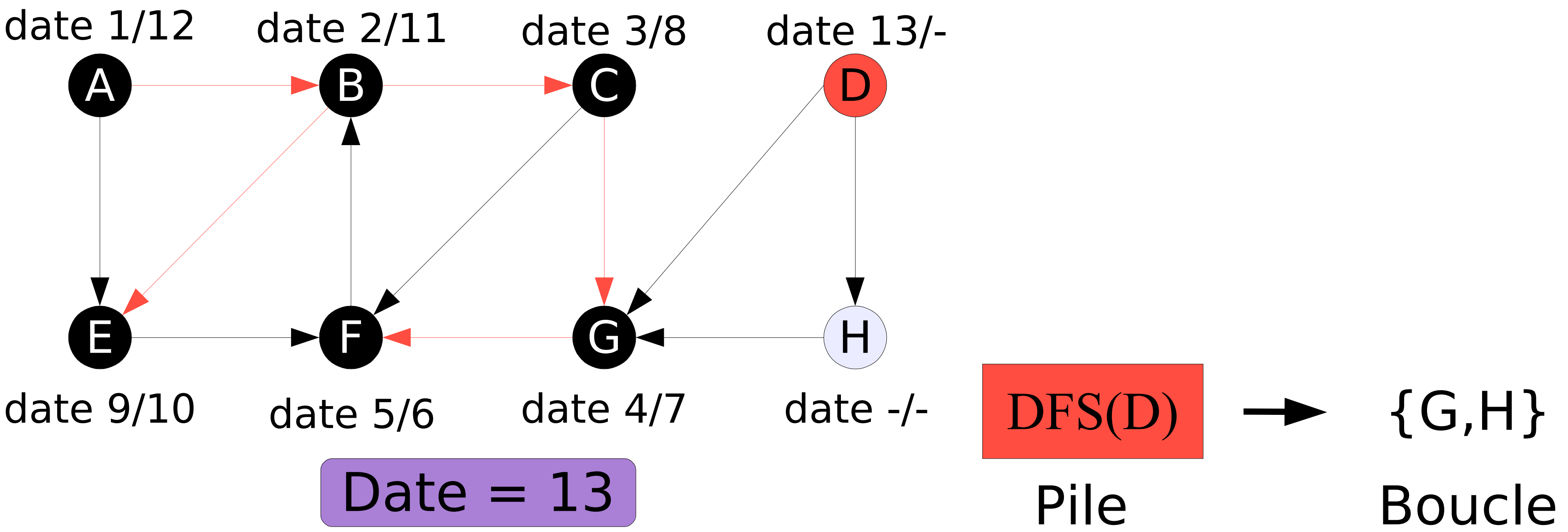
Exemple de l'algorithme DFS

- On empile la fonction DFS(D)
- F devient rouge et on note la date : dateDebut(F) ← 13



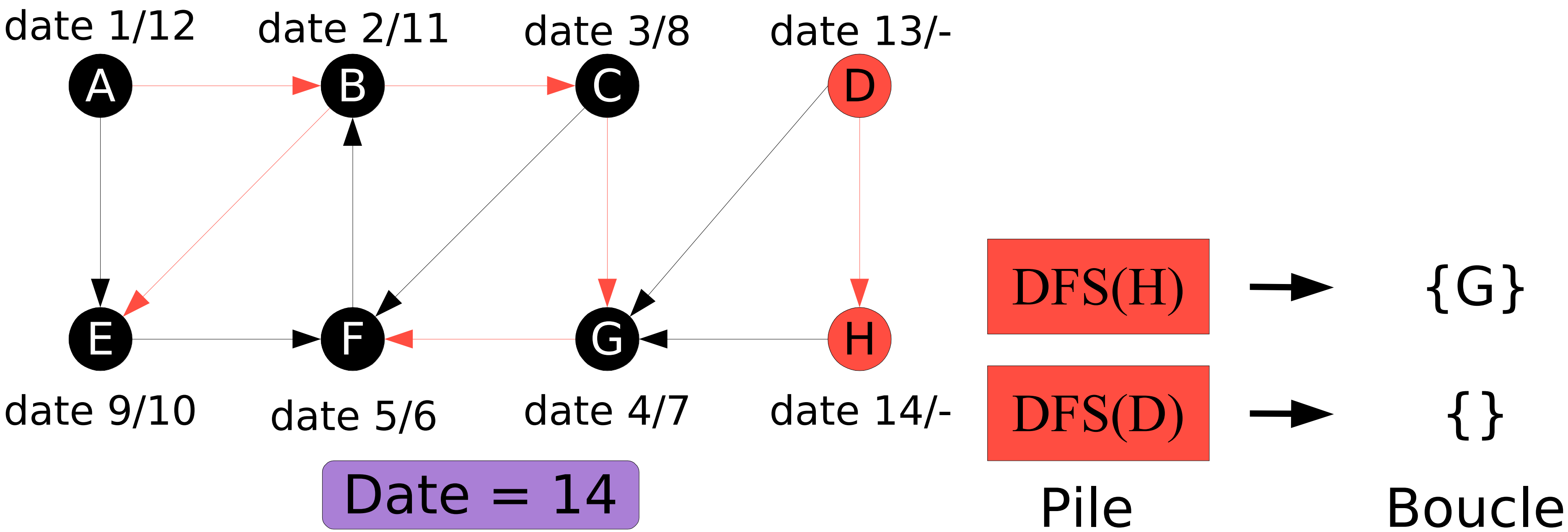
Exemple de l'algorithme DFS

- On empile la fonction DFS(D)
- F devient rouge et on note la date : dateDebut(F) ← 13



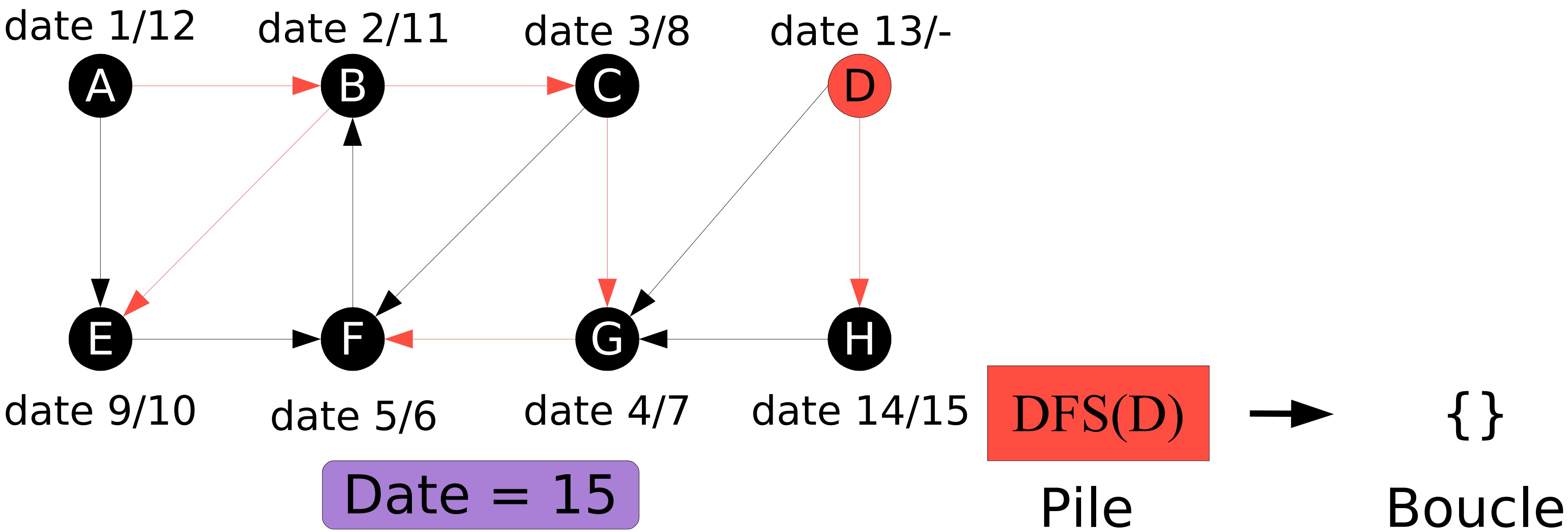
Exemple de l'algorithme DFS

- Le sommet G n'est pas blanc ⇒ pas d'appel a DFS(G)
- On empile la fonction DFS(H)
- H devient rouge et on note la date : dateDebut(H) ← 14



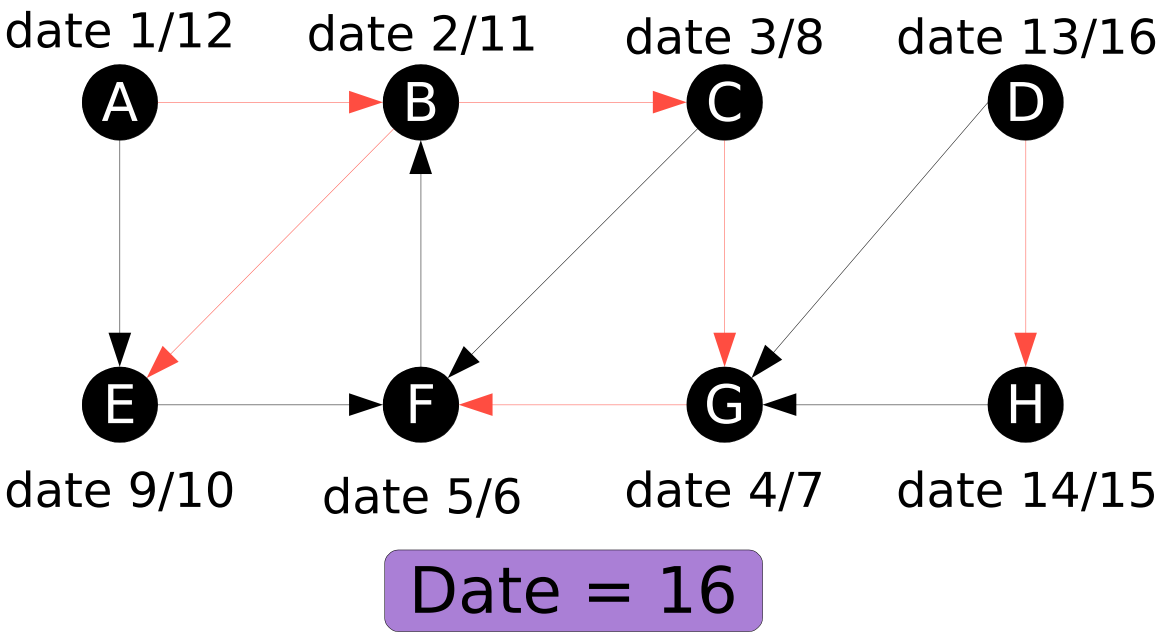
Exemple de l'algorithme DFS

- Le sommet G n'est pas blanc ⇒ pas d'appel a DFS(G)
- Fin de la boucle dans la fonction DFS(H)
- H devient noir et on note la date : dateFin(H) ← 15



Exemple de l'algorithme DFS

- Fin de la boucle dans la fonction DFS(H)
- H devient noir et on note la date : dateFin(H) ← 16

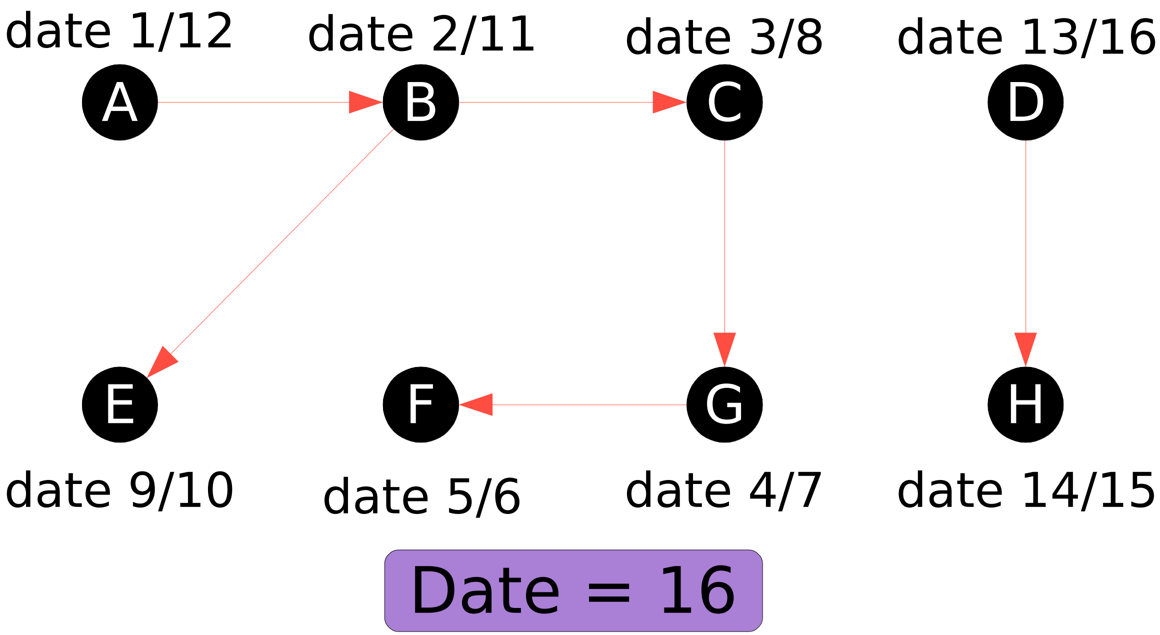


Pile

Boucle

Exemple de l'algorithme DFS

- Toutes les fonctions lancées par DFS_init() avortent.
- Parcours en profondeur ⇒ tous les sommets sont visités
- On obtient une **foret en profondeur :** par exemple, l'arbre de DFS de A, **DFS-tree(A) = {B, E, C, G, F}**



Pile

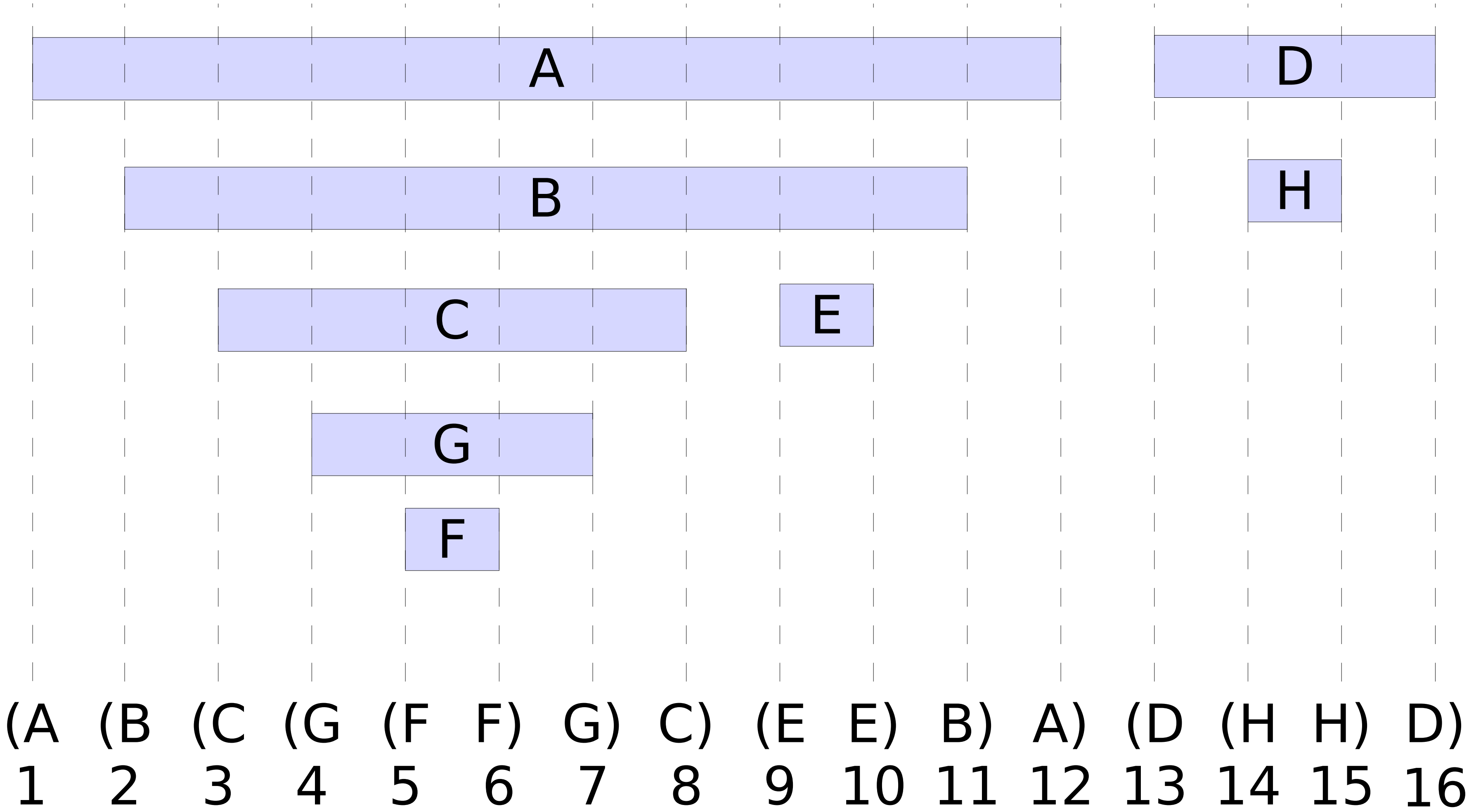
Boucle

Théorème des parenthèses

- Les dates de découvertes et fin de traitement ont : **une structure parenthésée.**

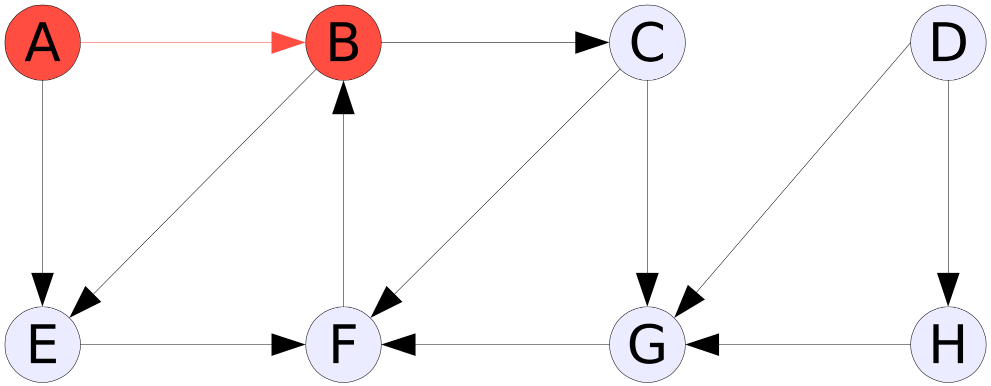
- $\forall u, v \in S$ une seul des 3 propositions suivantes est vraie :
  - *[début[u], fin[u]]* $\cap$ *[début[v], fin[v]]* = $\emptyset$
  - *[début[u], fin[u]]* $\subset$ *[début[v], fin[v]]*
    ET u descendant de v dans une arborescence de la foret
  - *[début[v], fin[v]]* $\subset$ *[début[u], fin[u]]*
    ET v descendant de u dans une arborescence de la foret

Théorème des parenthèses



Théorème du chemin blanc

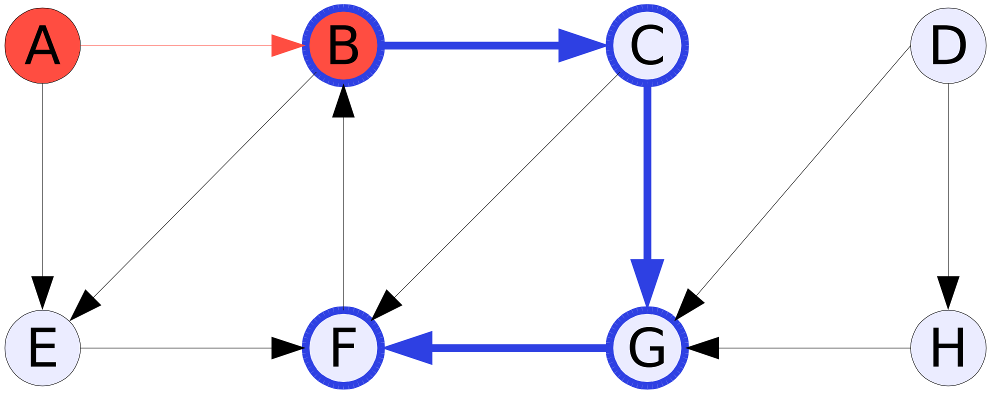
- Dans une foret en profondeur, un sommet *u* est un descendant d'un sommet *v* ssi :
  lorsque l'on découvre *v* (*dateDebut[v]*),
  il existe un **chemin de sommet BLANC** entre *v* et *u*



- On vient de découvrir B

Théorème du chemin blanc

- Dans une foret en profondeur, un sommet *u* est un descendant d'un sommet *v* ssi :
  lorsque l'on découvre *v* (*dateDebut[v]*),
  il existe un **chemin de sommet BLANC** entre *v* et *u*



- On vient de découvrir B
- **(B, C, G, F)** chemin BLANC
- Donc : F descendant de B

Calcul de compostantes fortement connexes (SCC) : algorithme de Kosaraju

- L'algorithme utilise :

  deux parcours en profondeur successifs

- Basé sur le théorème suivant :

  Soit **G** un graphe et $\mathbf{G^{-1}}$ son inverse. Soit **O** l'ordre descendant des sommets dans un parcours **DFS(G)**.

  - Chaque arbre de la forêt construite par un $\mathbf{DFS(G^{-1})}$, dont l'ordre de parcours des sommets est **O**, couvre les sommets **d'une et une seule** compostante fortement connexe de **G**.

- La preuve utilise les théorèmes des parenthèses et du chemin blanc.

A vous de jouer !

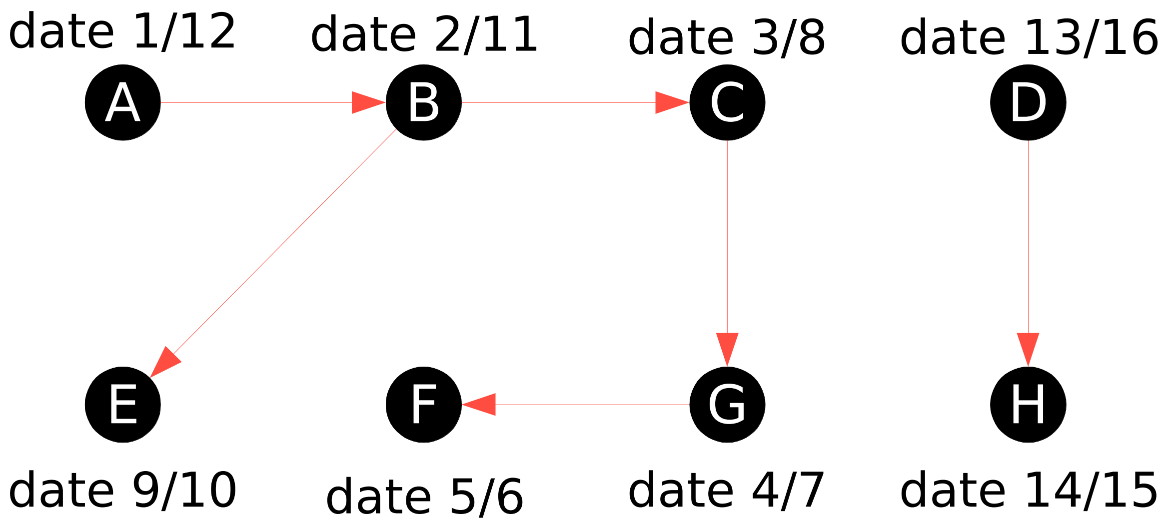
Calcul de compostantes fortement connexes : algorithme de Kosaraju

CFC (graphe *G*)

1. **DFS_run(G)**
2. **Calculer ^{t}G : transposé de G (inversion du sens de tous les arcs)**
3. **DFS_run(^{t}G) :** dans la boucle principale qui appelle DFS(s), on parcourt les sommets par ordre décroissant des *dateFin* calculées lors du premier DFS(G)

Exemple de l'algo. de Kosaraju

- A la fin du premier appel DFS (G), on obtient :



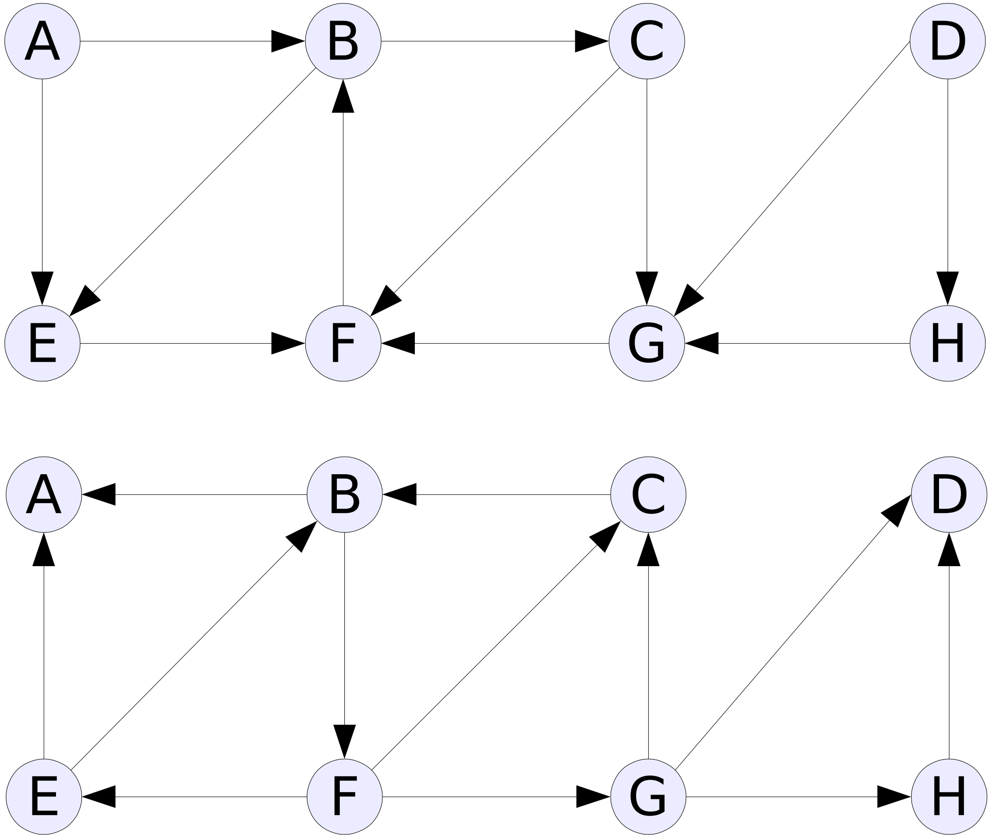
- On ordonne les sommets suivants *dateFin* (décroissant):

D – H – A – B – E – C – G – F

- C'est l'ordre qui sera utilisé dans la boucle du 2^{em} DFS_run

Exemple de l'algo. de Kosaraju

- On inverse les arcs pour obtenir : le graphe transposé



Graphe G

Transposition : Inversion des arcs

Graphe ${}^tG$

Exemple de l'algo. de Kosaraju

- La transposition est une opération matricielle
- La matrice d'adjacence du graphe transposé ${}^tG$ est : la transposée de la matrice d'adjacence du graphe G
- *Ex : L'inversion de l'arc* $\overrightarrow{BC} \Leftrightarrow \begin{cases} a_{23}=0 \\ a_{32}=1 \end{cases}$

Graphe G

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 1 & 0 & 0 & 0 \\ 0 & \mathbf{0} & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Transposition :

$$\longrightarrow$$

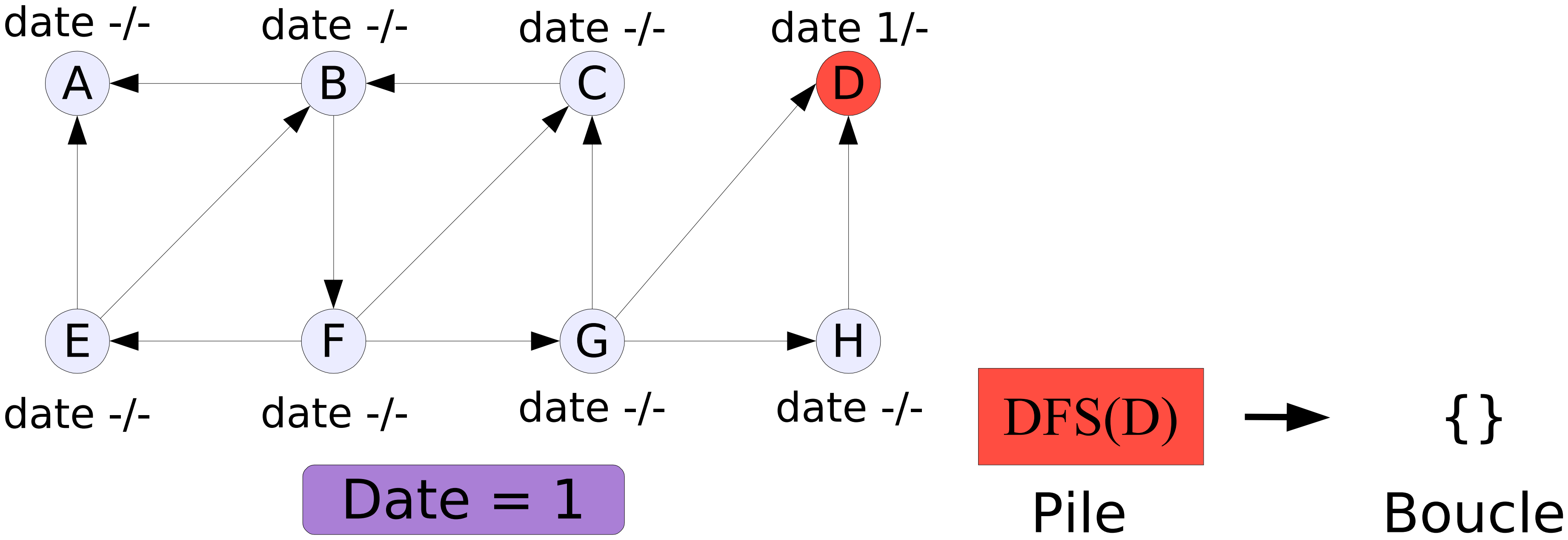
$$\forall i, j \in S : a_{ij} \leftarrow a_{ji}$$

Graphe ${}^tG$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & \mathbf{0} & 0 & 0 & 1 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

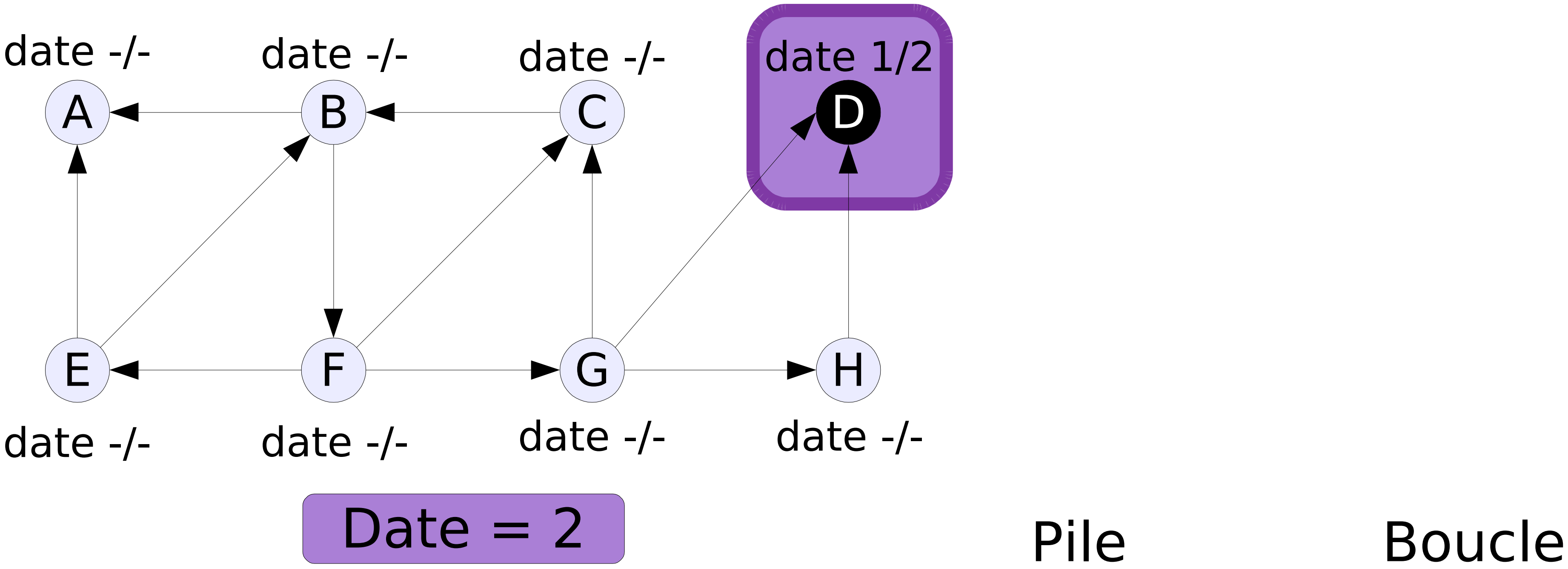
Exemple de l'algo. de Kosaraju

- On lance une deuxième fois le parcours en profondeur
- D est le premier sommet dans la boucle de DFS_run()
- Appel a DFS(D); D devient rouge.



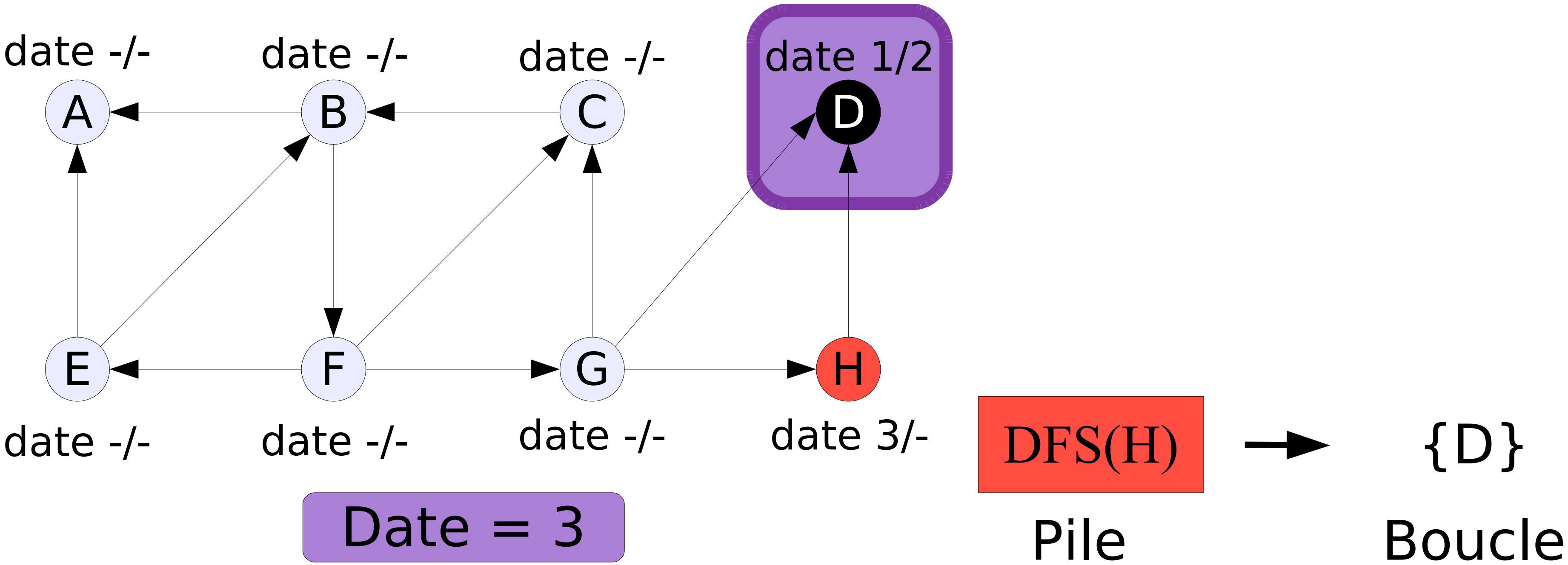
Exemple de l'algo. de Kosaraju

- D n'a pas de voisins blancs
- D devient noir
- Fin du 1er appel de DFS() dans la boucle de DFS_run()



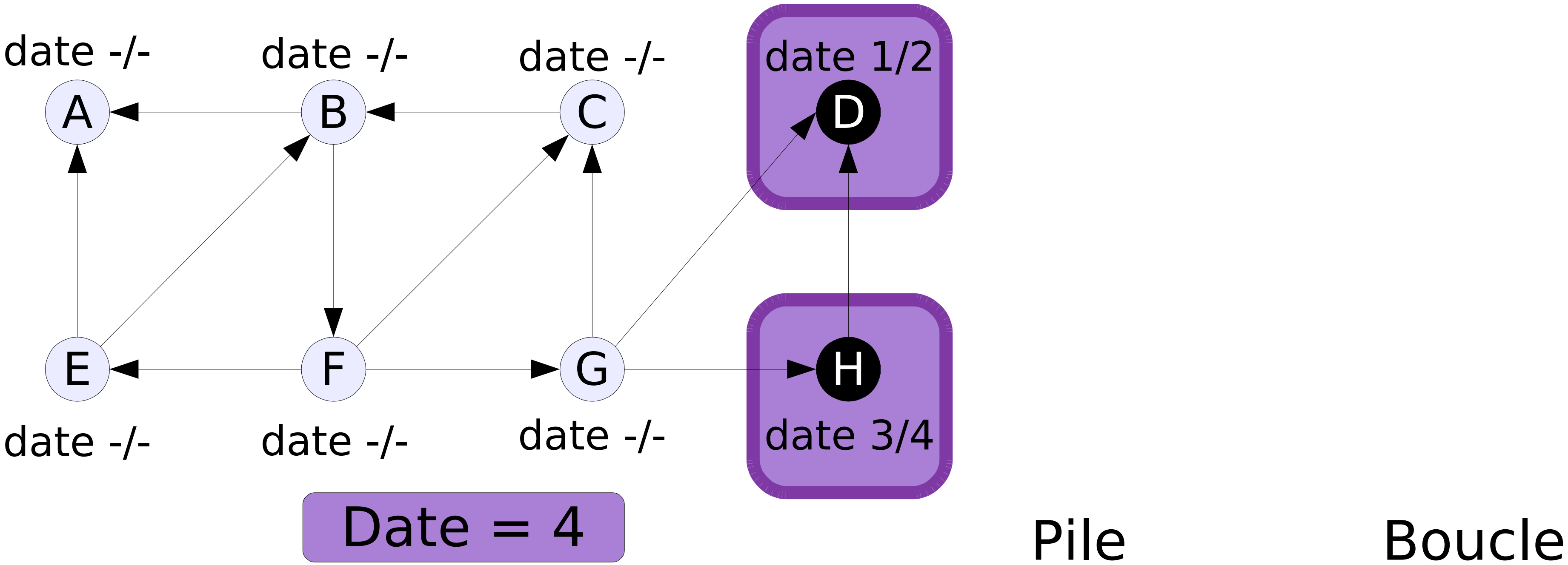
Exemple de l'algo. de Kosaraju

- H est le 2em sommet dans la boucle de DFS_run()
- Appel à DFS(H); H devient rouge.



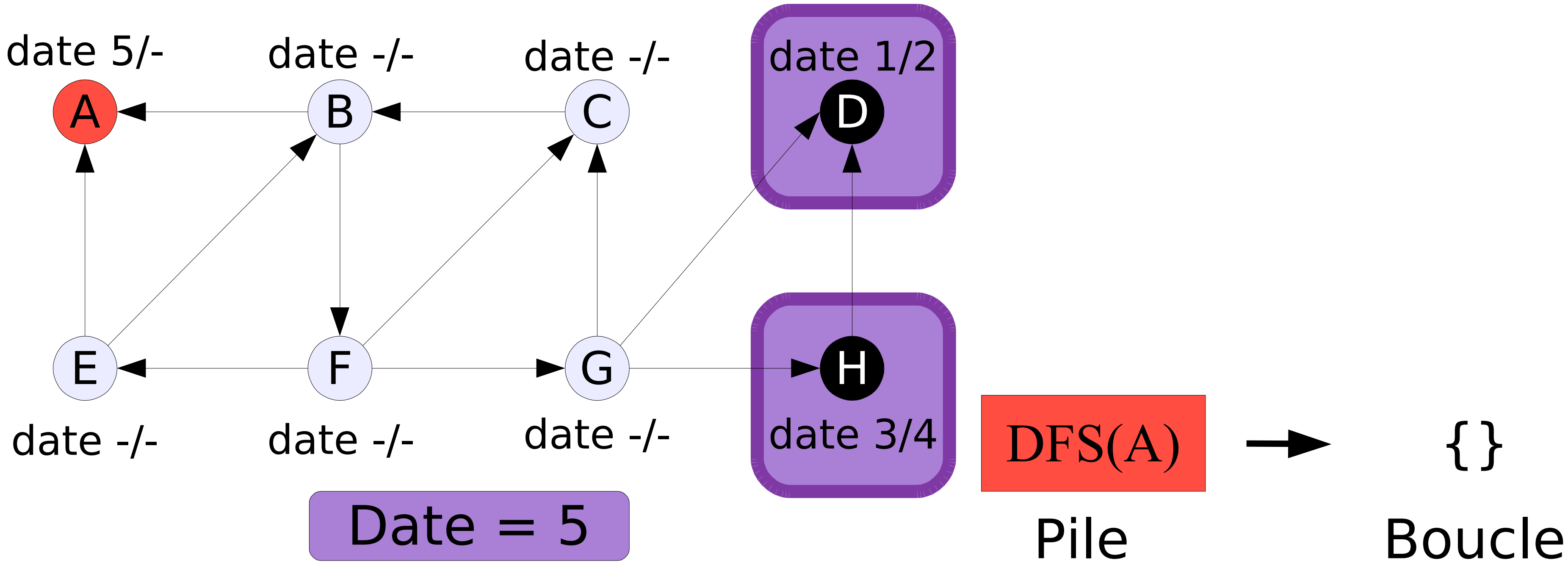
Exemple de l'algo. de Kosaraju

- H n'a pas de voisins blancs
- H devient noir
- Fin du 2[em] appel de DFS() dans la boucle de DFS_run()



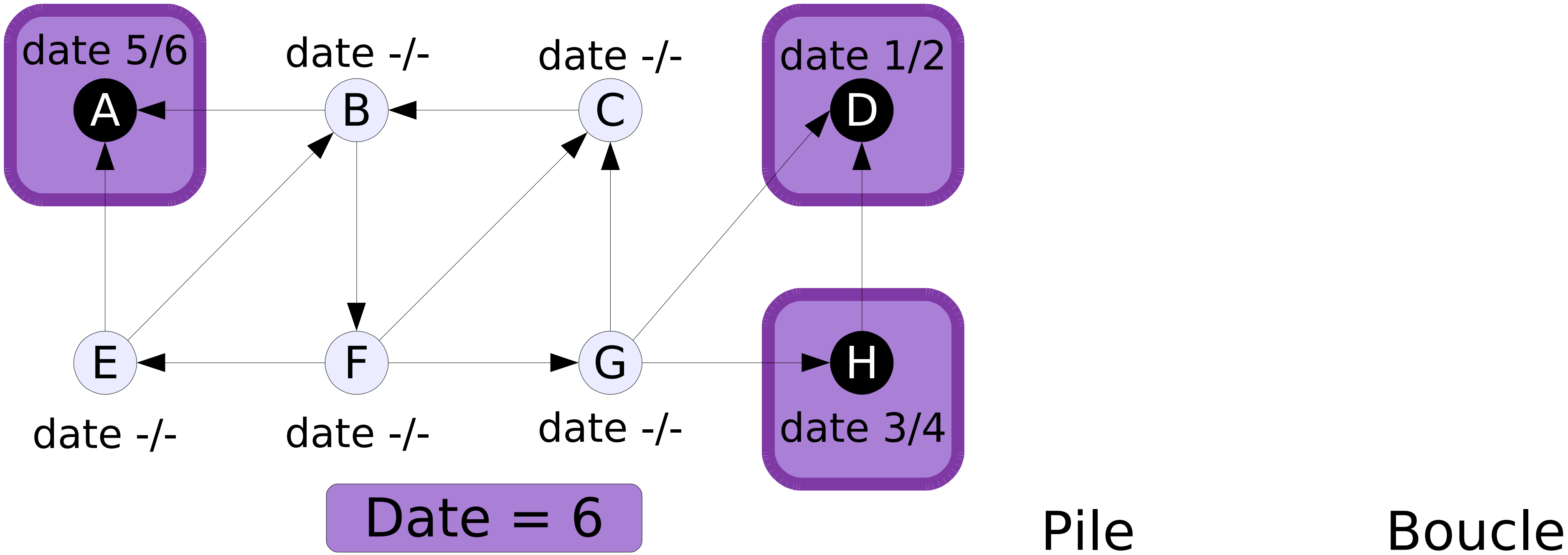
Exemple de l'algo. de Kosaraju

- A est le 3[em] sommet dans la boucle de DFS_run()
- Appel a DFS(A)
- A devient rouge.



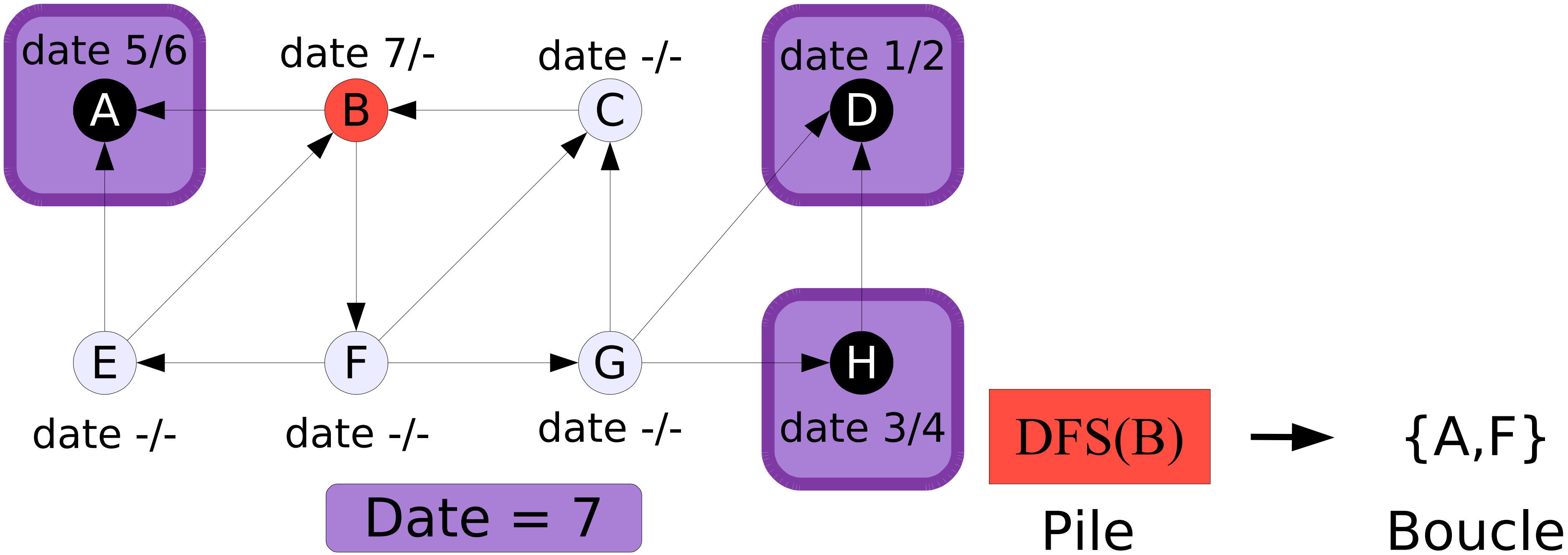
Exemple de l'algo. de Kosaraju

- A n'a pas de voisins blancs
- A devient noir
- Fin du 3em appel de DFS() dans la boucle de DFS_run()



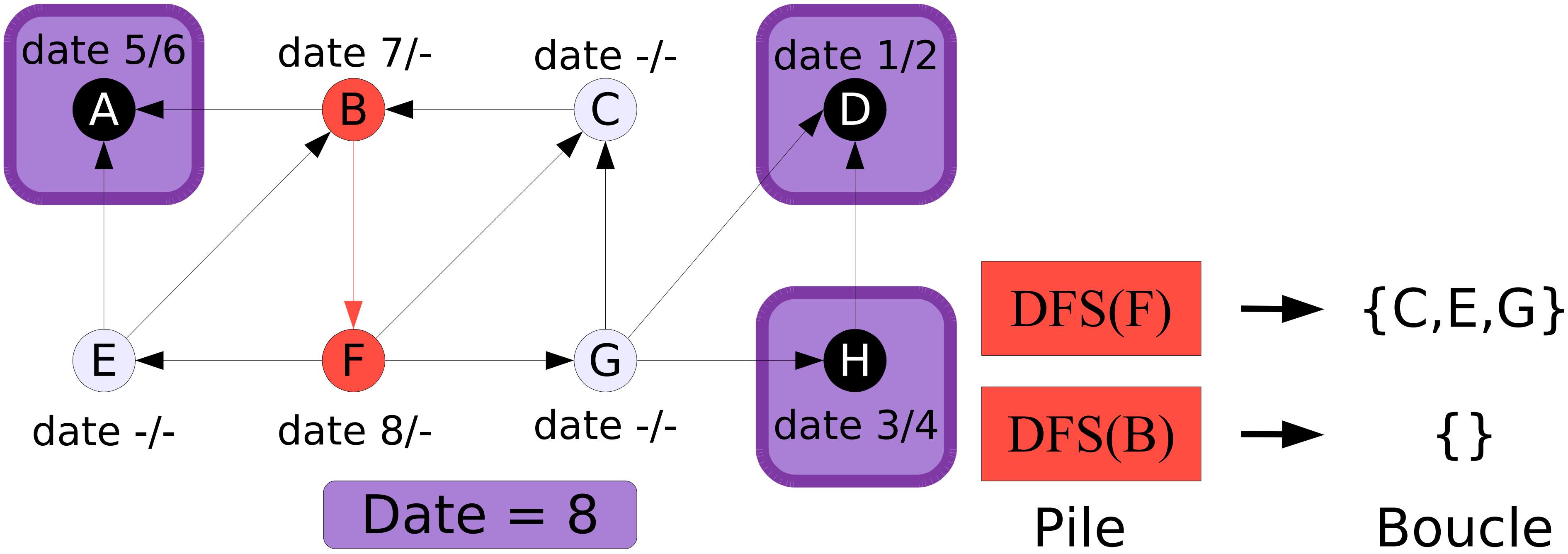
Exemple de l'algo. de Kosaraju

- A est le 4em sommet dans la boucle de DFS_run()
- Appel à DFS(B)
- B devient rouge.



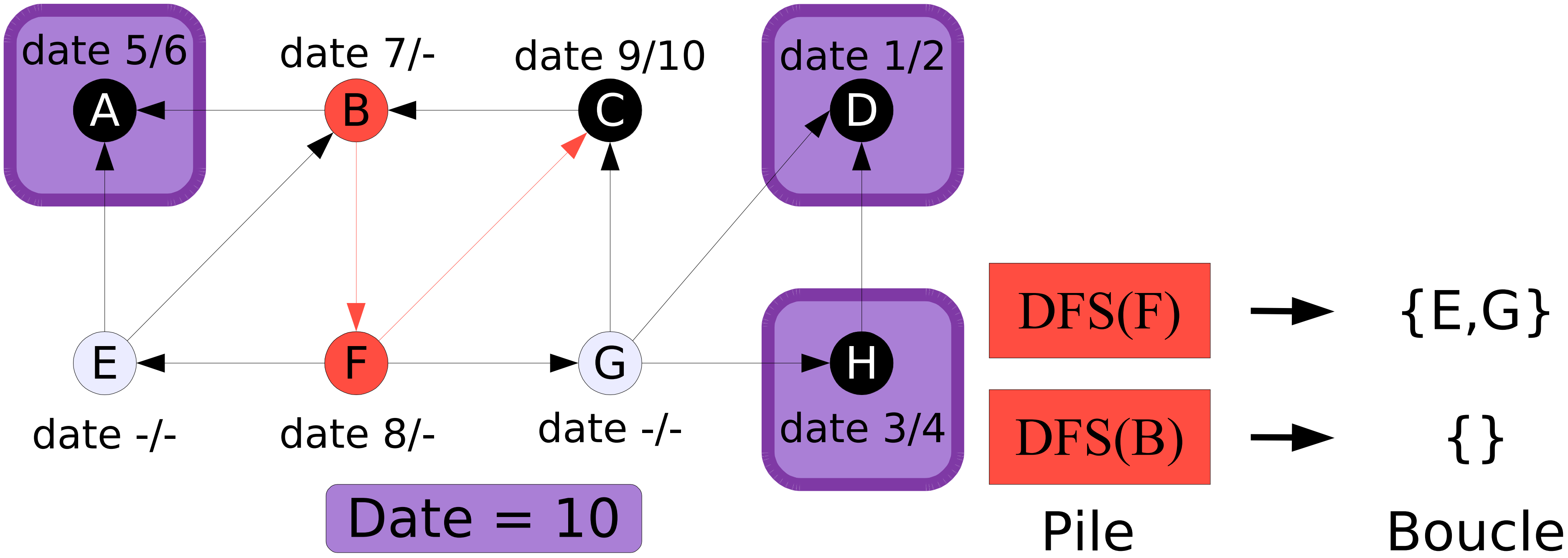
Exemple de l'algo. de Kosaraju

- Le sommet A n'est pas blanc ⇒ pas d'appel à DFS(A)
- Appel à DFS(F)
- F devient rouge



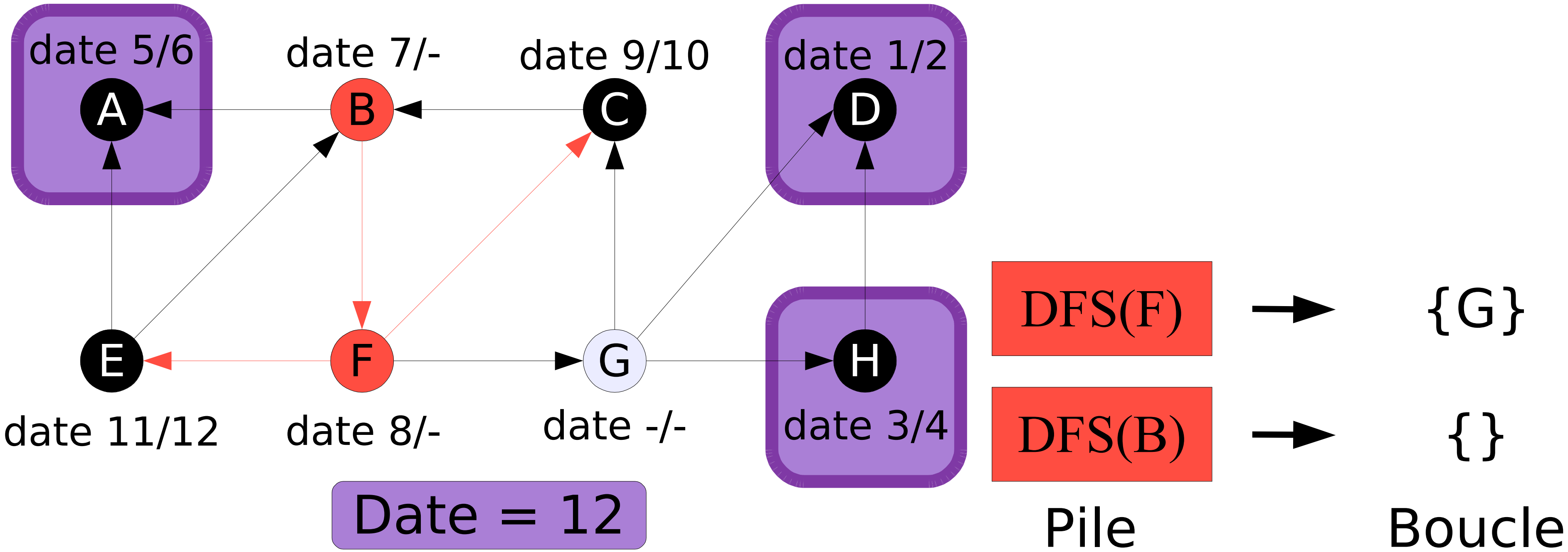
Exemple de l'algo. de Kosaraju

- Appel a DFS(C)
- C devient rouge
- C n'a pas de voisins blancs ⇒ C devient noir



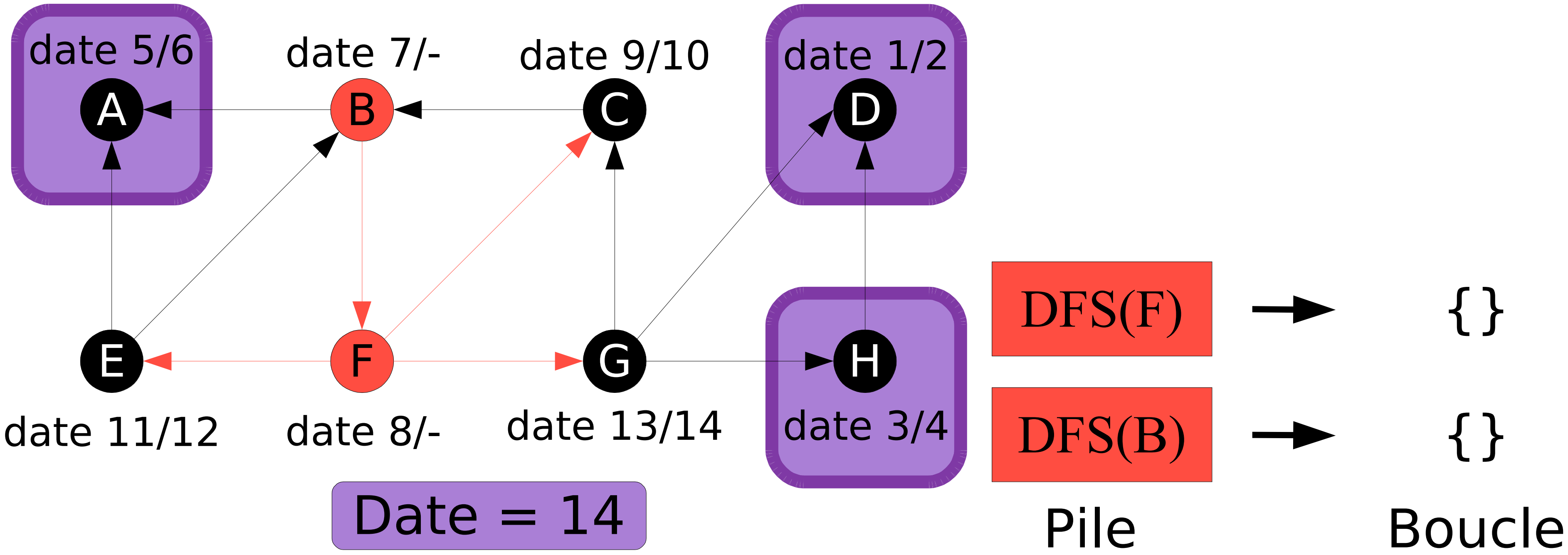
Exemple de l'algo. de Kosaraju

- Appel à DFS(E)
- E devient rouge
- E n'a pas de voisins blancs ⇒ E devient noir



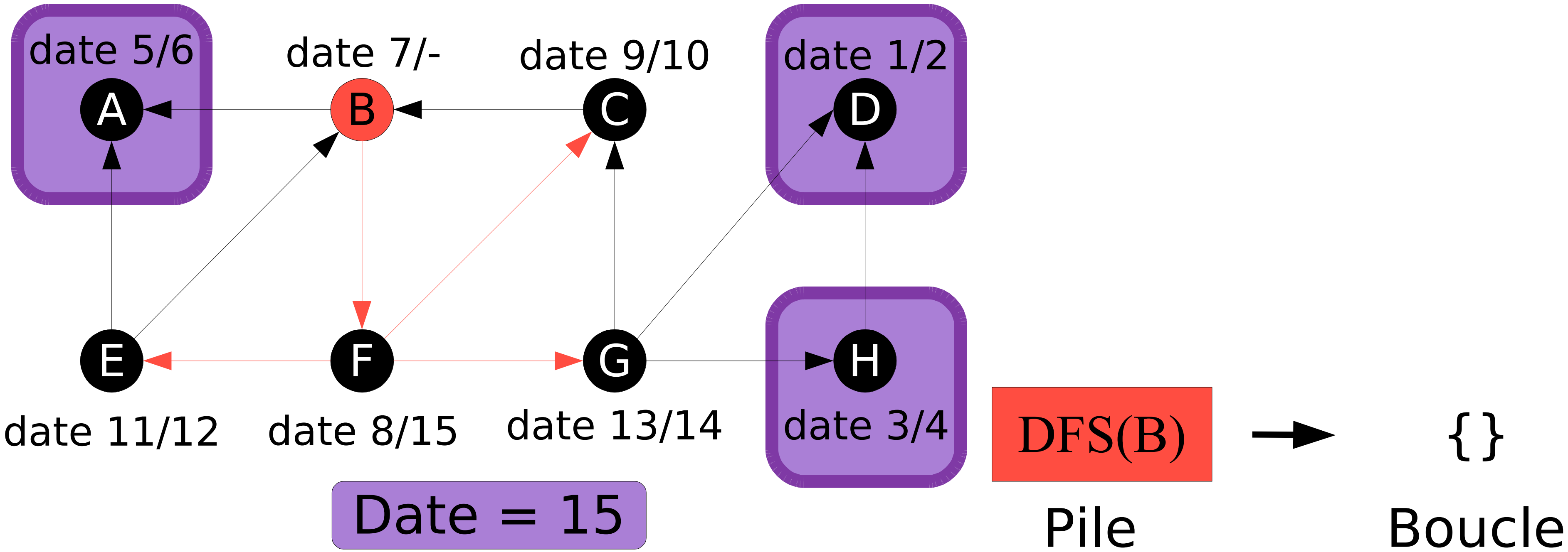
Exemple de l'algo. de Kosaraju

- Appel à DFS(G)
- G devient rouge
- G n'a pas de voisins blancs ⇒ G devient noir



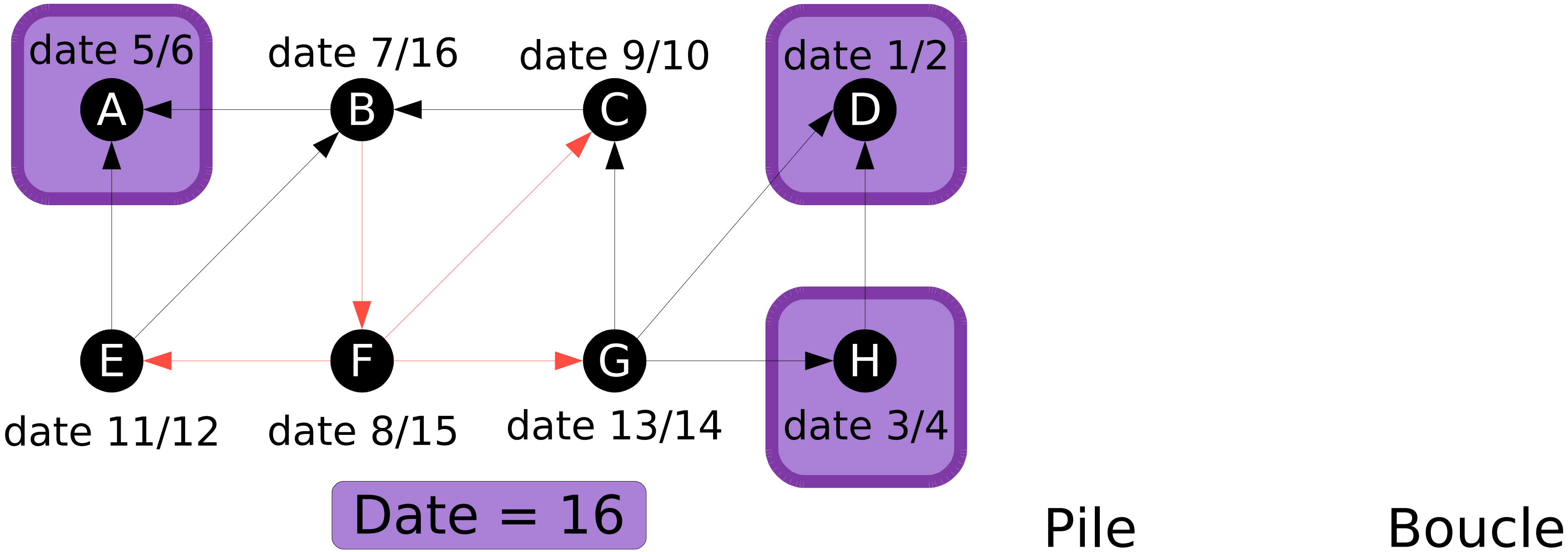
Exemple de l'algo. de Kosaraju

- Fin de la boucle dans la fonction DFS(F)
- F devient noir et on note la date : dateFin(F) ← 15



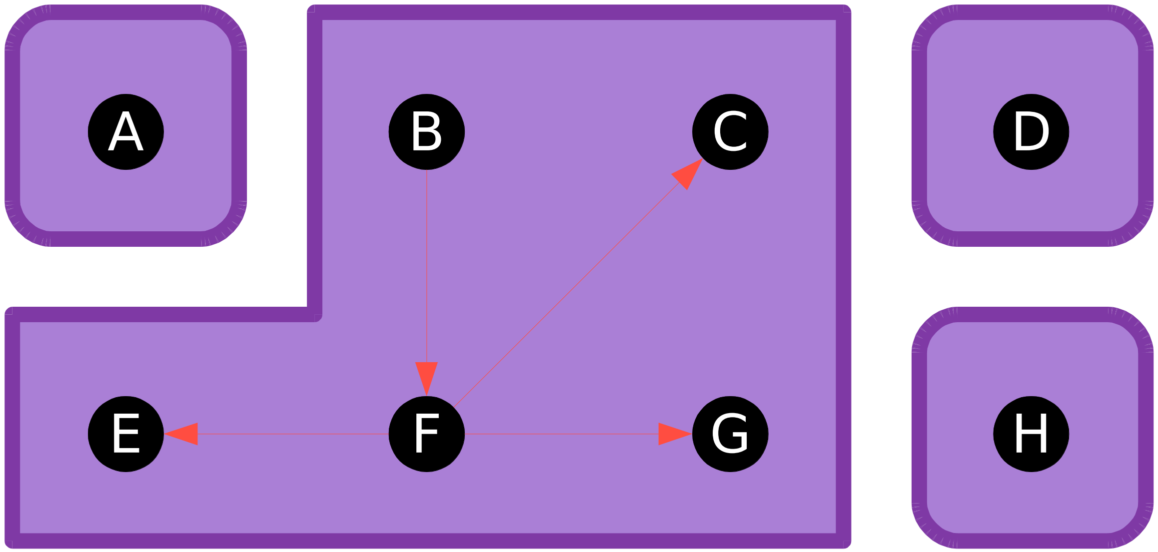
Exemple de l'algo. de Kosaraju

- Fin de la boucle dans la fonction DFS(F)
- F devient noir et on note la date : dateFin(F) ← 15



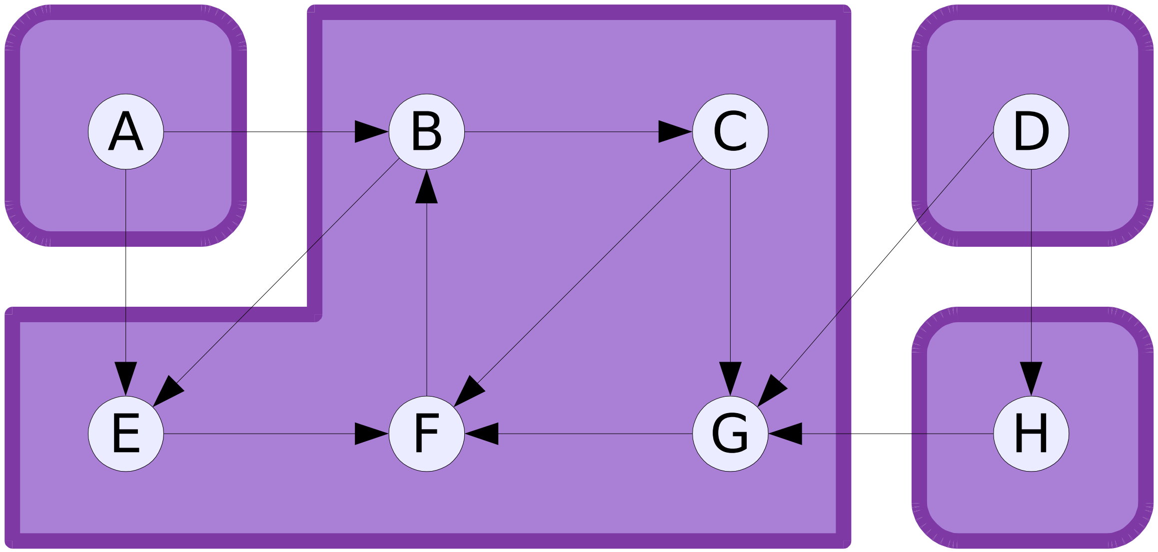
Exemple de l'algo. de Kosaraju

- Le résultat du deuxième parcours en profondeur est : une forêt de 4 arborescences



Exemple de l'algo. de Kosaraju

- Chacune de ces arborescences correspondent à :
 1 **composante fortement connexe** du graphe de départ



Calcul de compostantes fortement connexes : algorithme de Tarjan (1/2)

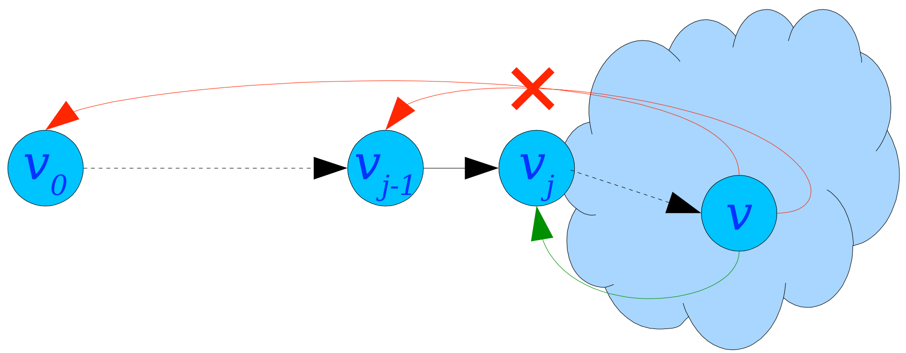
- L'algorithme utilise : **un seul parcours en profondeur.**

- Basé sur les observations suivantes :
  - Dans un DFS, chaque arbre de la foret recouvre les sommets d'une/plusieurs SCC.
  - Dans un arbre de la foret, chaque sommet peut-être la « racine » d'une SCC (le premier sommet de la SCC rencontré lors du DFS).
  - Soit $(v_0,\ldots,v_j)$ le chemin parcouru par le DFS jusqu'au sommet v_j (la pile du DFS). v_j est la racine d'une SCC, si :
    1. $\forall\ v \in$ DFS-tree(v_j), $\forall\ i \in \{0,\ldots,j\text{-}1\}$, $(v,v_i) \notin E$.
       (Pas d'arc vers un sommet de $(v_0,\ldots,v_{j-1})$)

Calcul de compostantes fortement connexes : algorithme de Tarjan (2/2)

- Soit $(v_0,\ldots,v_j)$ le chemin parcouru par le DFS jusqu'au sommet v_j (la pile du DFS). v_j est la racine d'une SCC, si :

  1. $\forall v \in$ DFS-tree(v_j), $\forall\ i \in \{0,\ldots,j\text{-}1\}$, $(v, v_i) \notin E$.
     (Pas d'arc vers un sommet de $(v_0,\ldots,v_{j-1})$)



Algorithme de Tarjan (version sans pile) : num et lowlink

- La condition citée plus haut, peut être vérifiée en associant, lors d'un DFS du graphe, à chaque sommet *v*, deux informations :

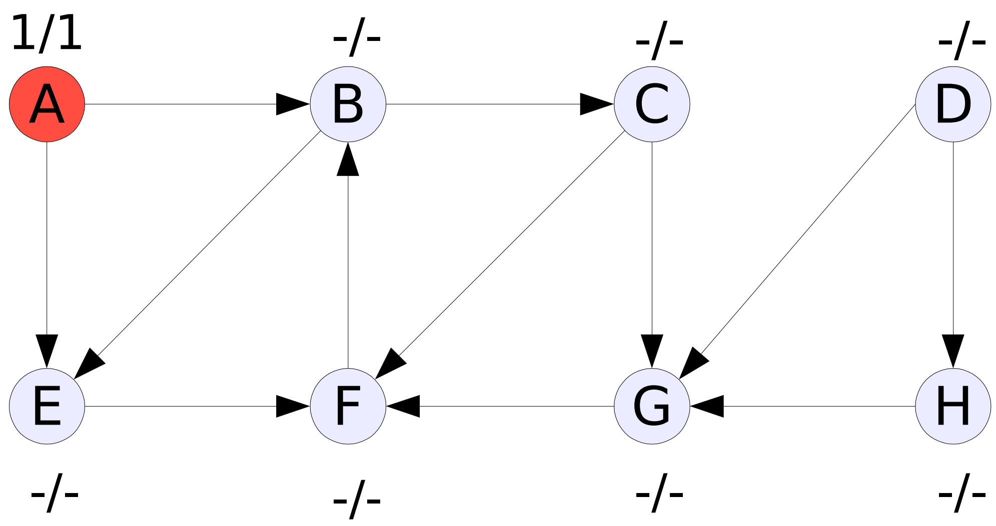
  - *v.num* : représentant *l'ordre du parcours* de *v* dans un DFS.
    - Valeurs spéciales :
      - \- : le sommet n'a pas encore été visité,
      - * : sommet déjà visité et ces informations ne sont plus utiles

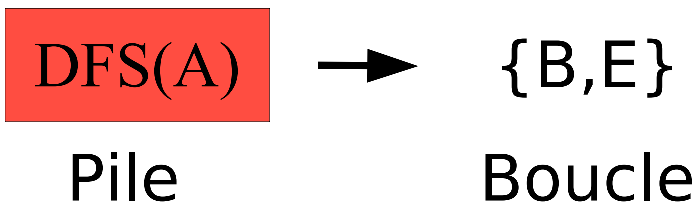
  - *v.lowlink* : représentant *le plus petit « num » des sommets accessibles* par l'ensemble DFS-tree(*v*)

  - La mise à jour des deux informations est effectuée lors de la décente et la remonté du DFS...

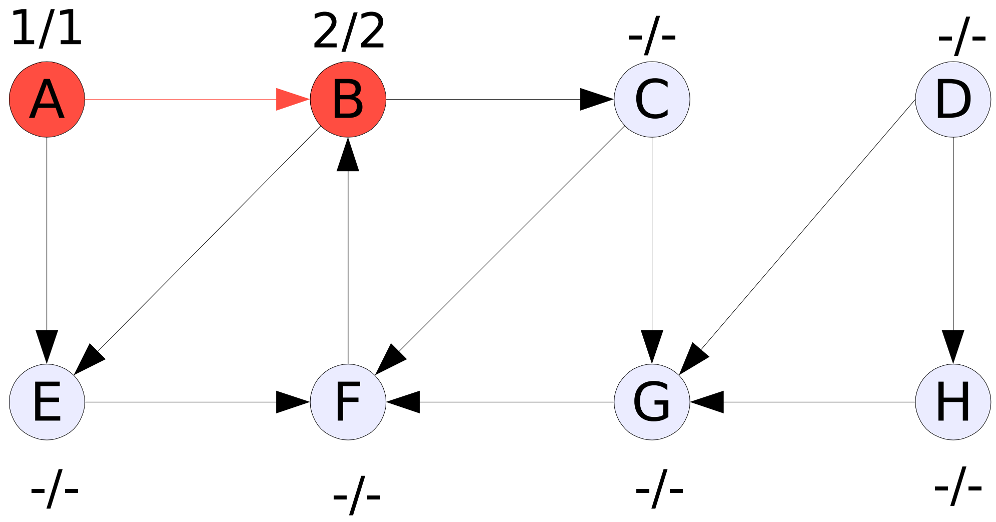
Algorithme de Tarjan (version sans pile) : exemple de mise à jour de num et lowlink

num/lowlink



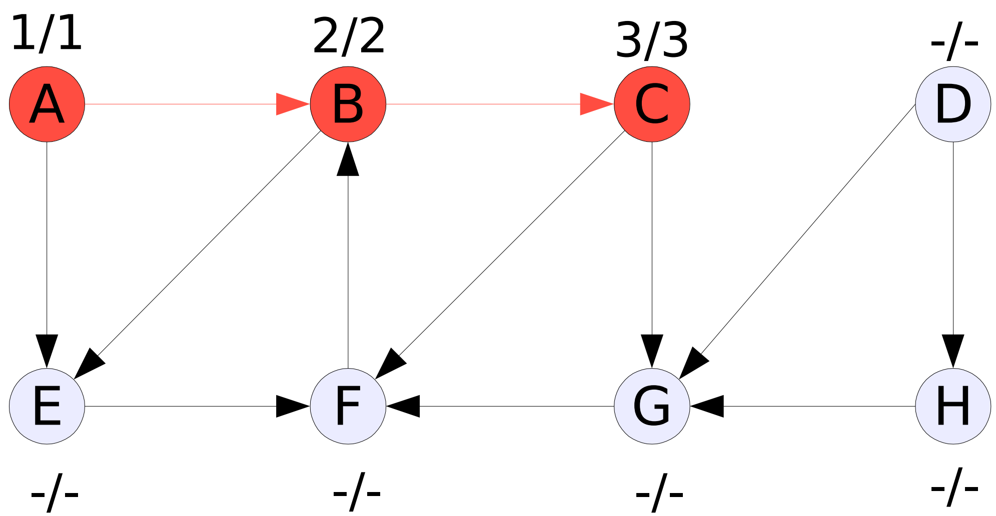


Algorithme de Tarjan (version sans pile) : exemple de mise à jour de num et lowlink



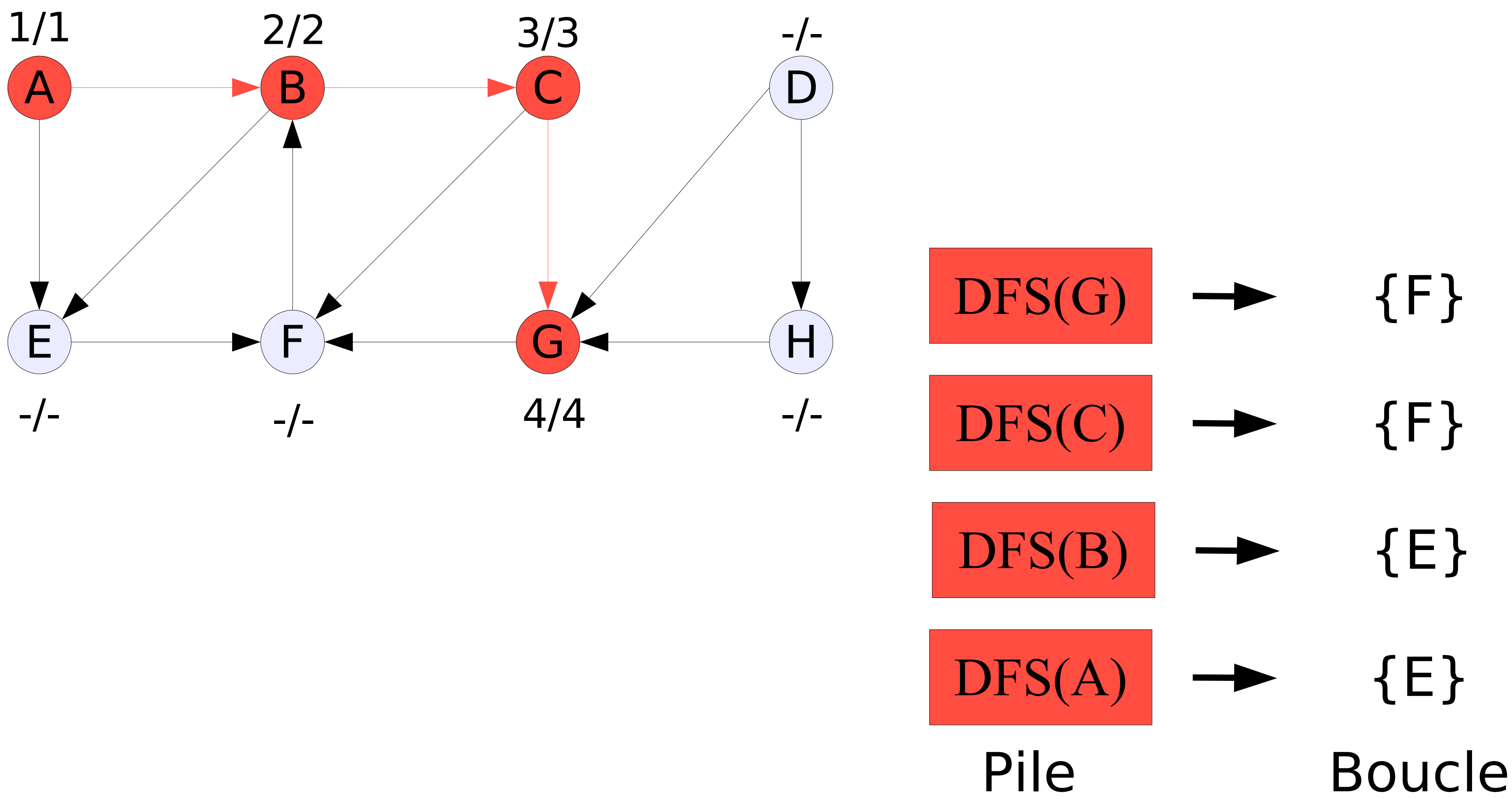
Pile		Boucle
DFS(B)	→	{C,E}
DFS(A)	→	{E}

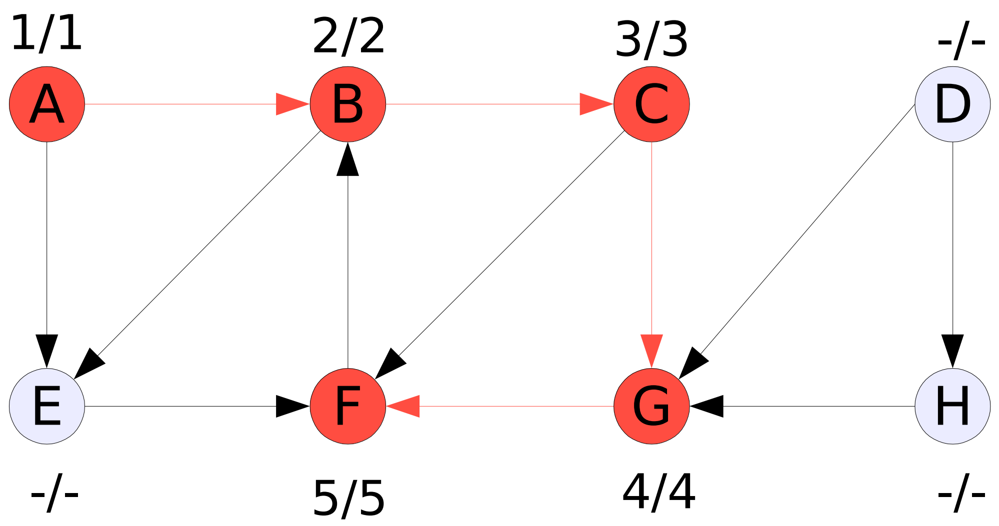
Algorithme de Tarjan
(version sans pile) : exemple de mise à jour de num et lowlink



Pile		Boucle
DFS(C)	→	{G,F}
DFS(B)	→	{E}
DFS(A)	→	{E}

Algorithme de Tarjan (version sans pile) : exemple de mise à jour de num et lowlink





DFS(F) → {}

DFS(G) → {}

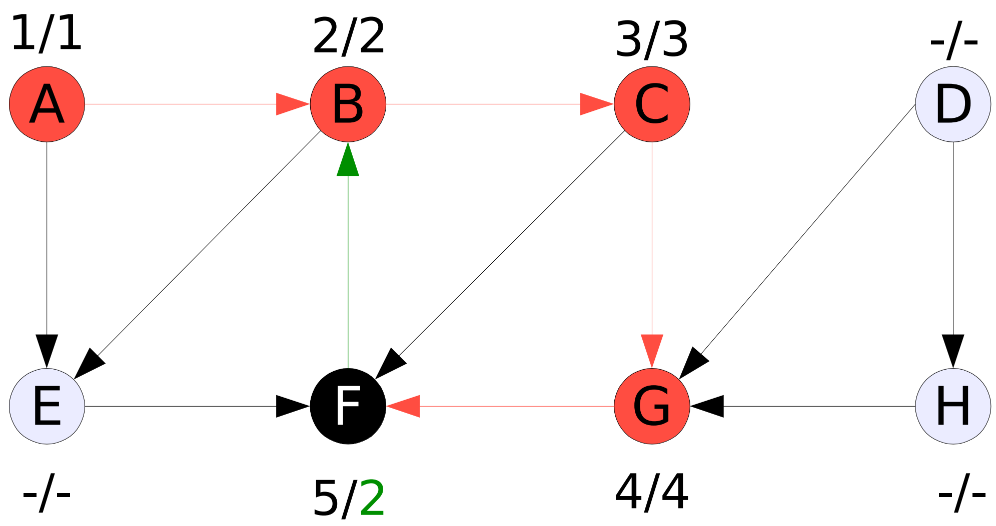
DFS(C) → {F}

DFS(B) → {E}

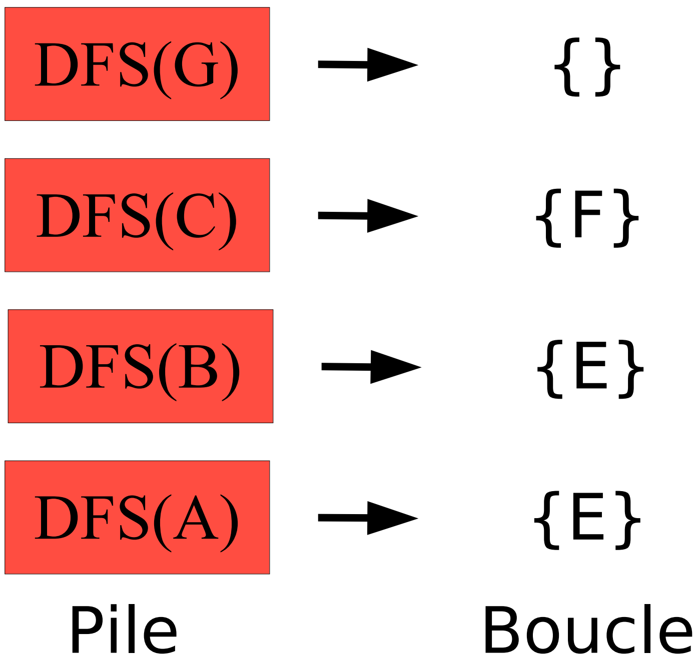
DFS(A) → {E}

Pile — Boucle

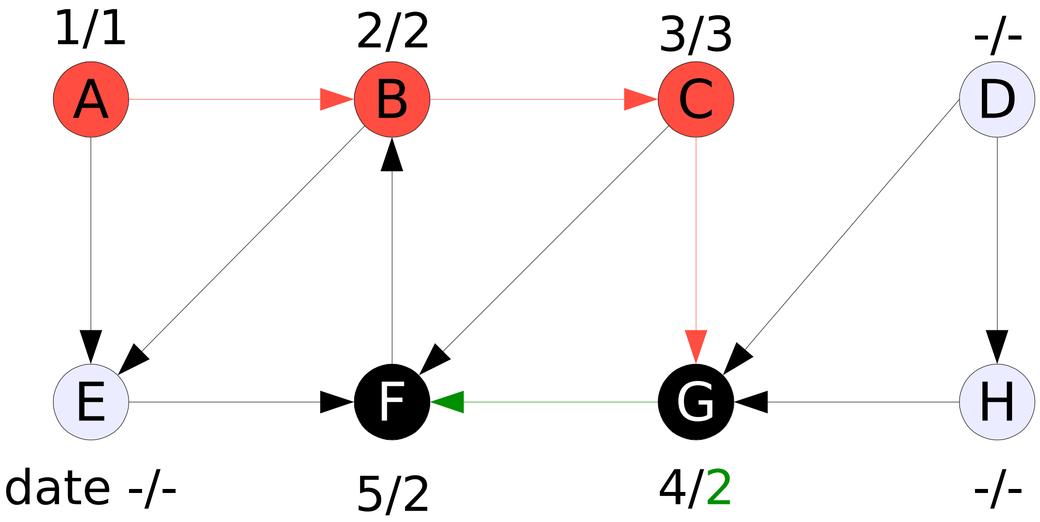
Algorithme de Tarjan
(version sans pile) : exemple de mise à jour de num et lowlink



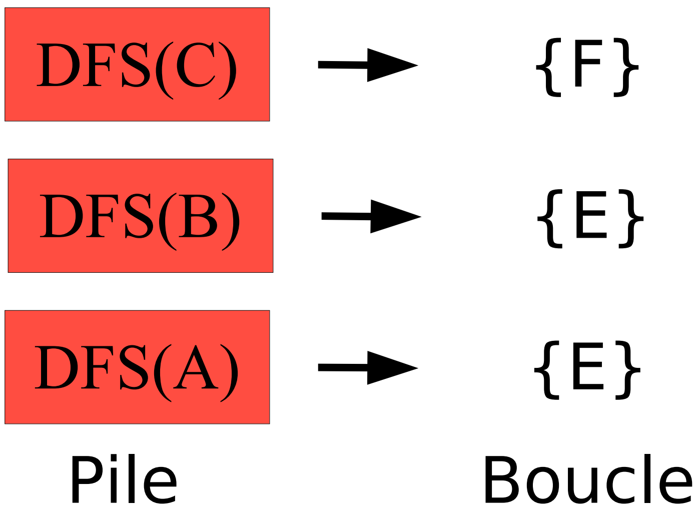
F.lowlink = min (F.lowlink, B.lowlink)



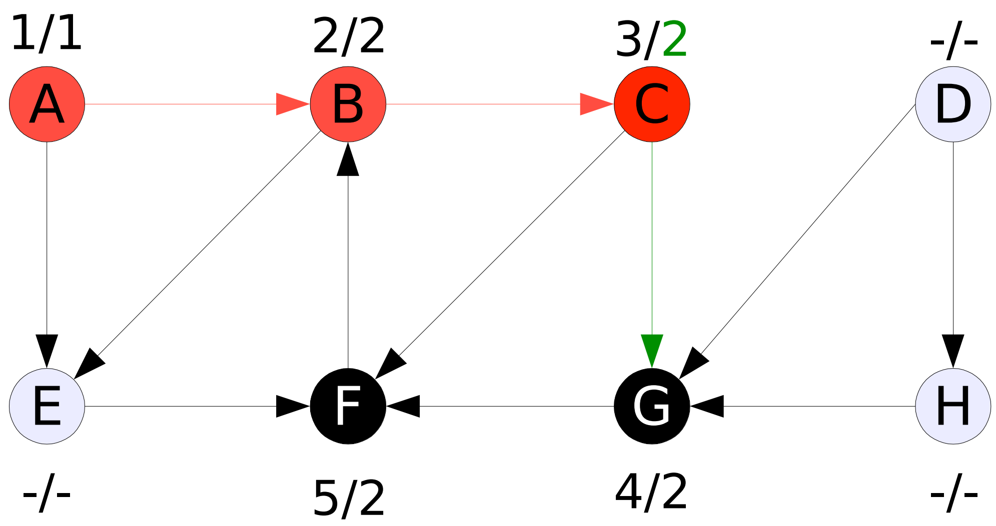
Algorithme de Tarjan
(version sans pile) : exemple de mise à jour de num et lowlink



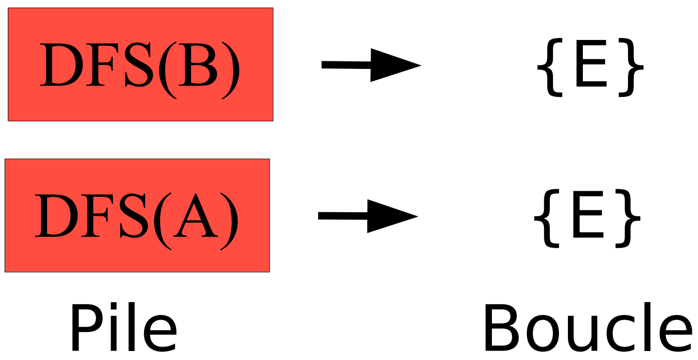
G.lowlink = min (G.lowlink, F.lowlink)



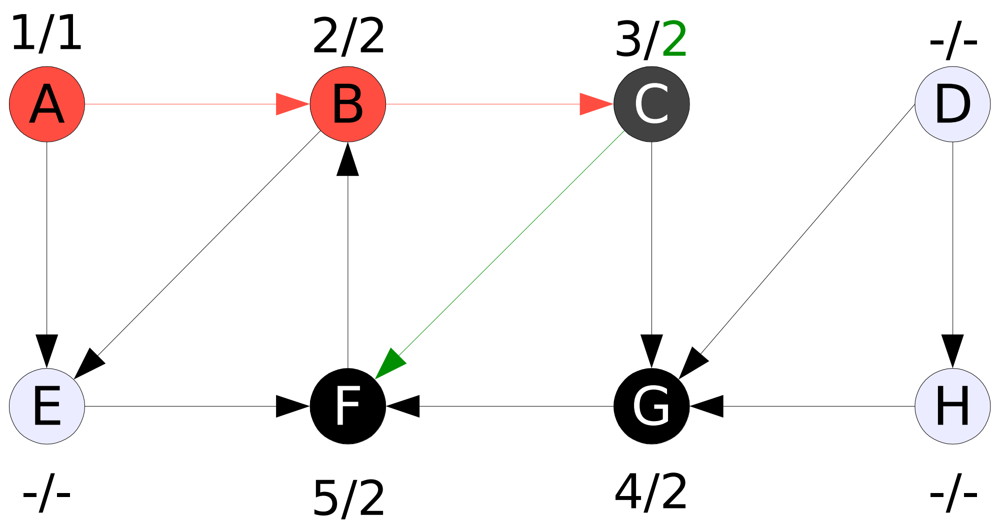
Algorithme de Tarjan (version sans pile) : exemple de mise à jour de num et lowlink



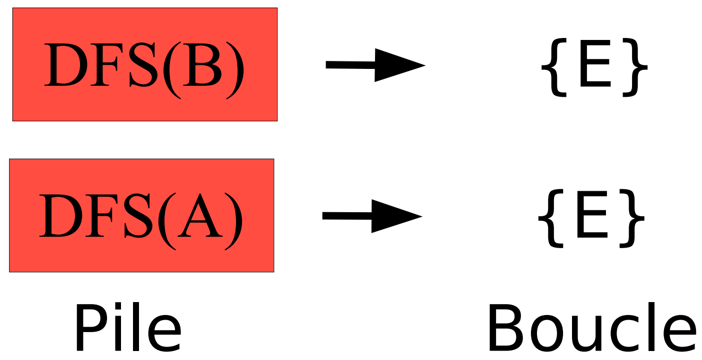
C.lowlink = min (C.lowlink, G.lowlink)

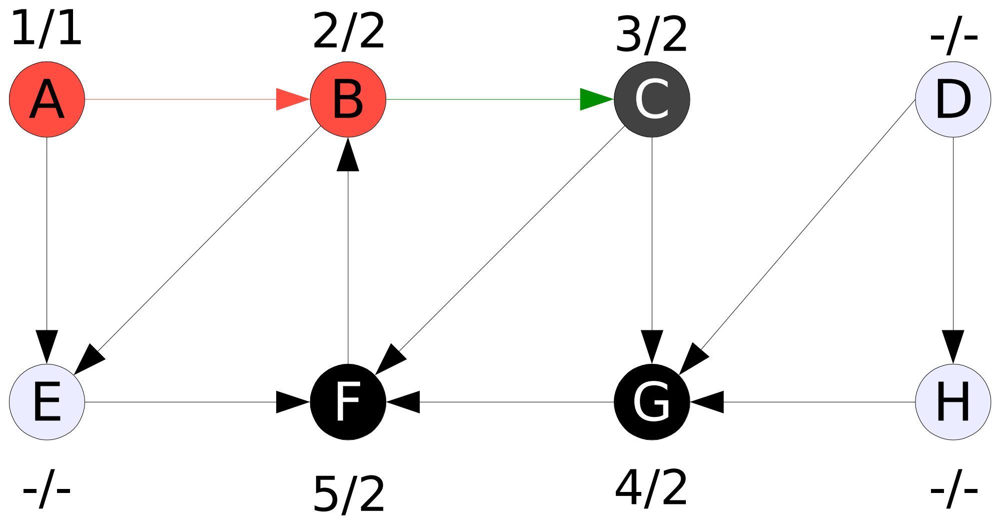


Algorithme de Tarjan (version sans pile) : exemple de mise à jour de num et lowlink

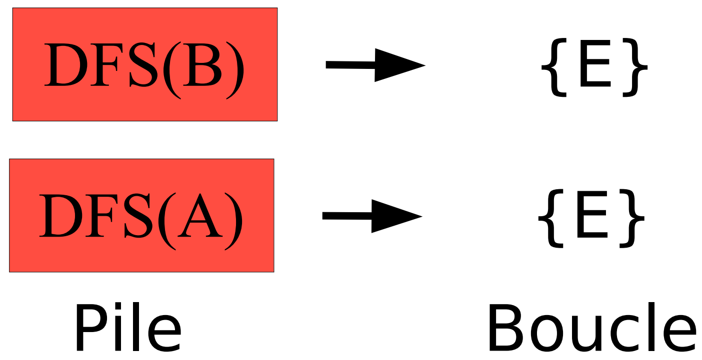


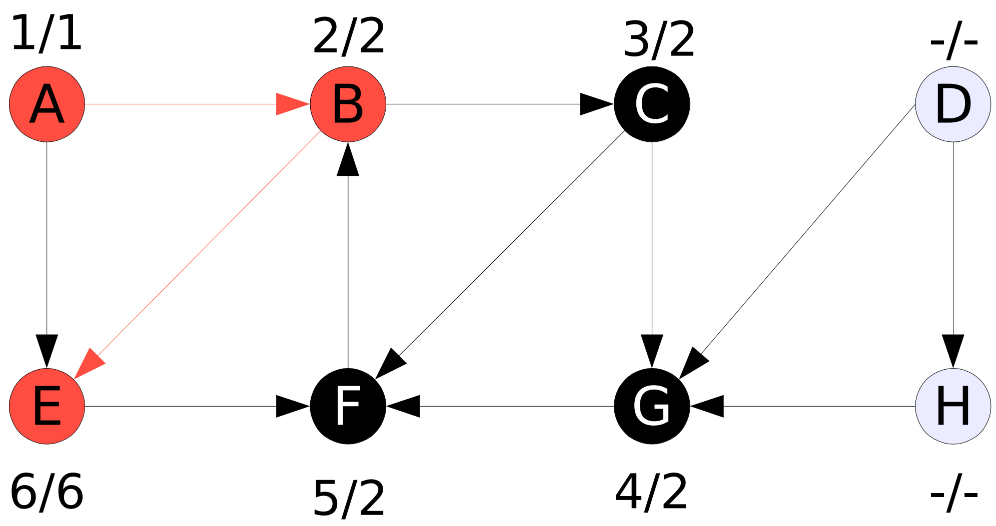
C.lowlink = min (C.lowlink, F.lowlink)



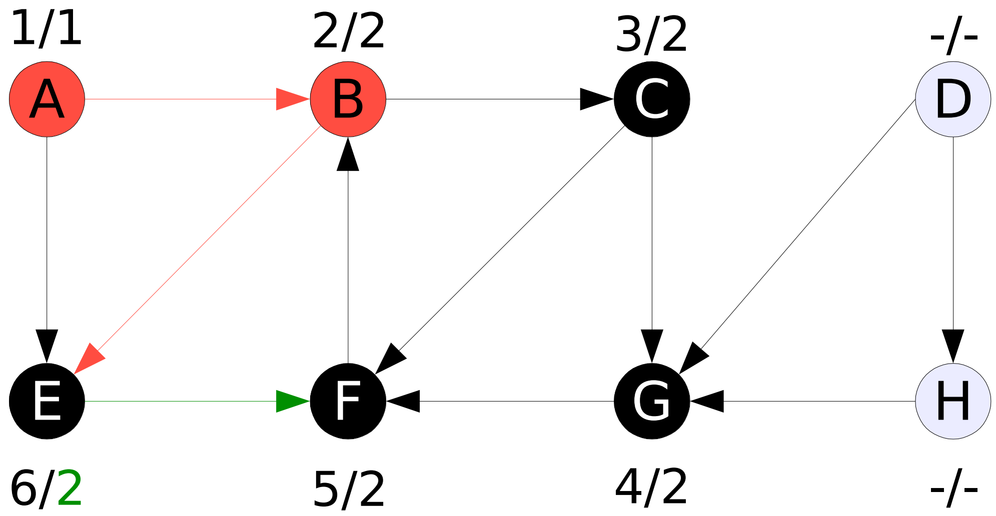


B.lowlink = min (B.lowlink, C.lowlink)

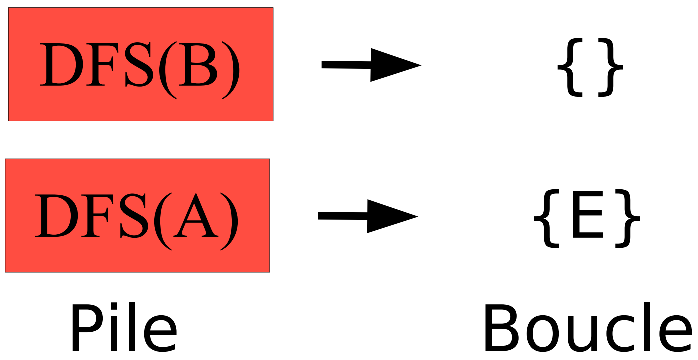




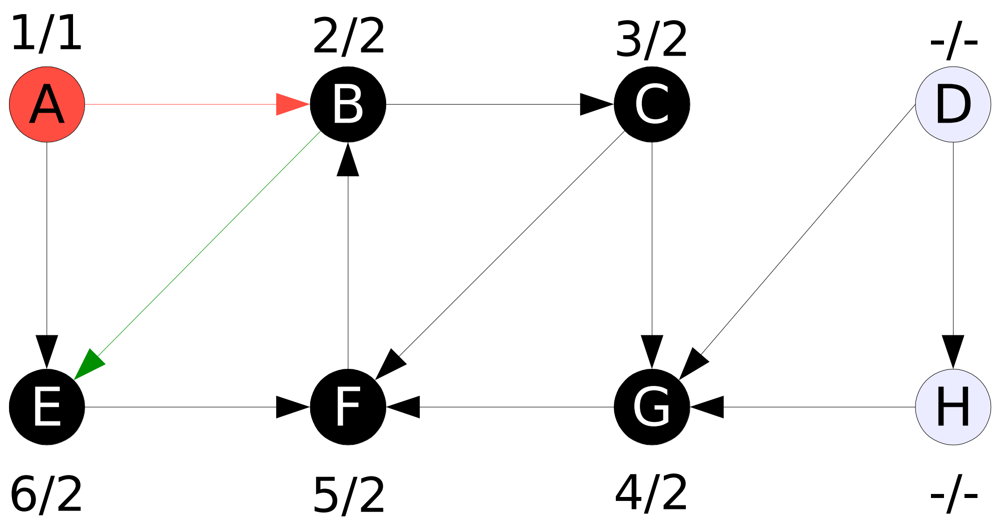
Pile		Boucle
DFS(E)	→	{F}
DFS(B)	→	{}
DFS(A)	→	{E}



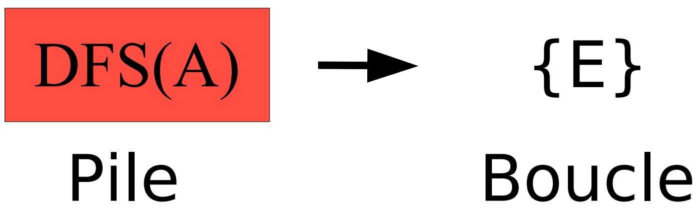
E.lowlink = min (E.lowlink, F.lowlink)

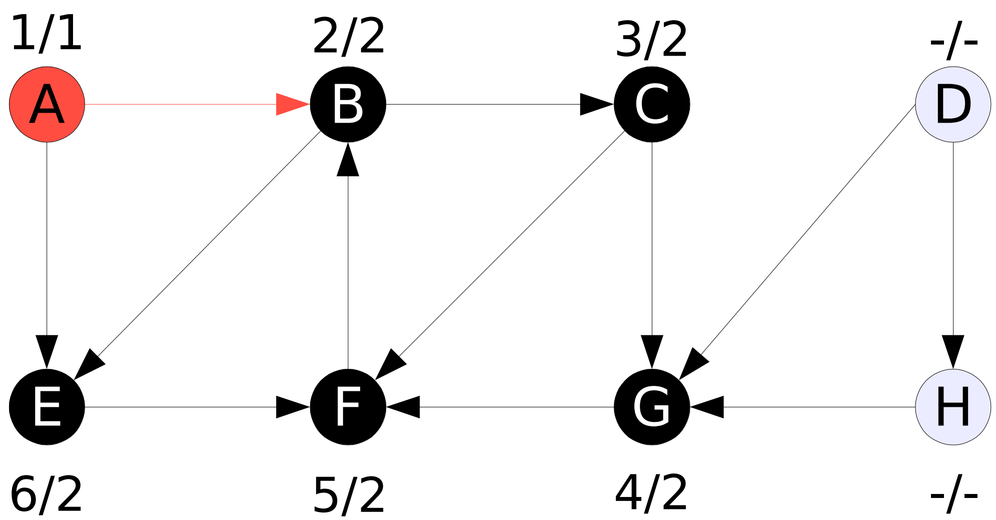


Algorithme de Tarjan (version sans pile) : exemple de mise à jour de num et lowlink



B.lowlink = min (B.lowlink, E.lowlink)





- Les DFS-tree(B) ne contient aucun sommet ayant un arc vers un sommet de la pile du DFS (ici, A)

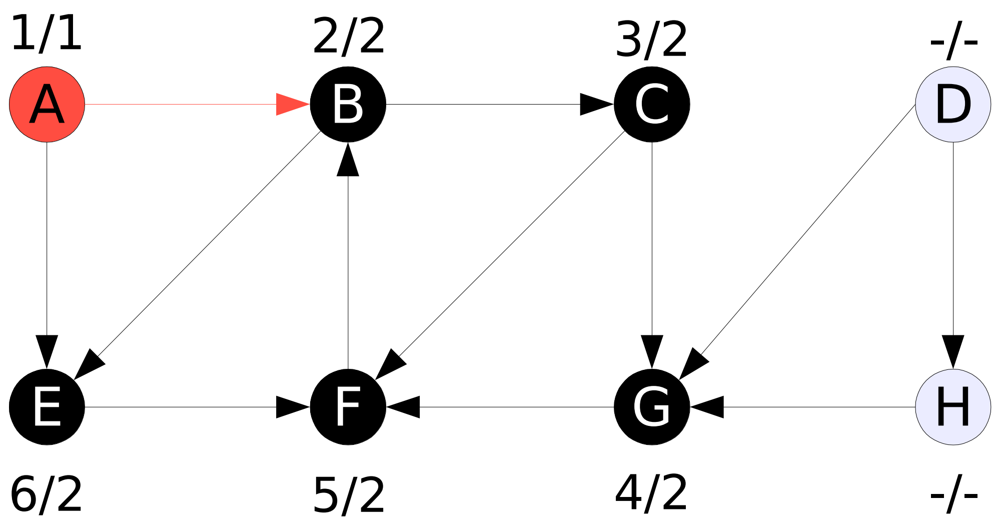
=> B est la racine d'une SCC.

=> Caractérisation :

- $\nexists\, v' \in \text{DFS-tree}(B),\ \ v'.\text{lowlink} < B.\text{num}$
- ou aussi, **B.lowlink = *B*.num**

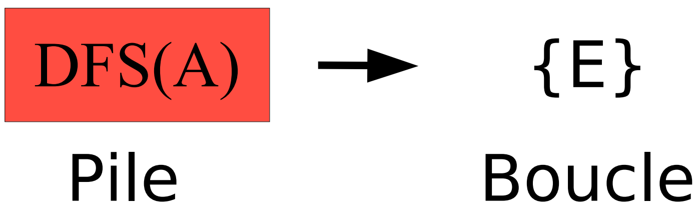
DFS(A) → {E}

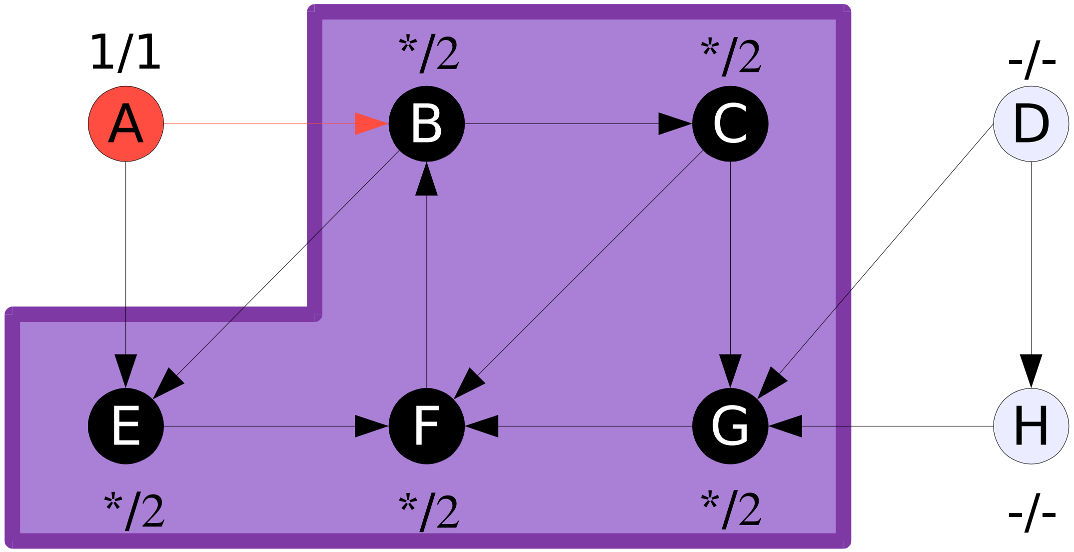
Pile   Boucle



Comment obtenir les sommets couvrant la SCC ?

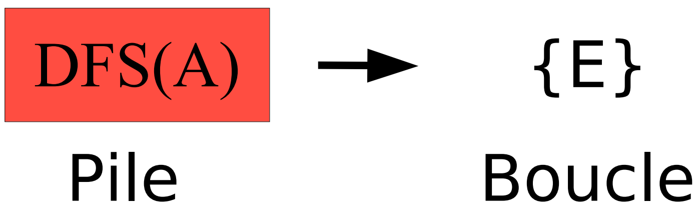
- Roots(B) = {$v \in$ DFS-tree(B) | $v \neq$ B $\wedge$ v.lowlink=v.num}
- num-Roots(B) = { $v.num$ | $v \in$ Roots(B) }
- **SCC-Set(B) = {$v \in$ DFS-tree(B) | v.lowlink $\notin$ num-Roots(B)** }



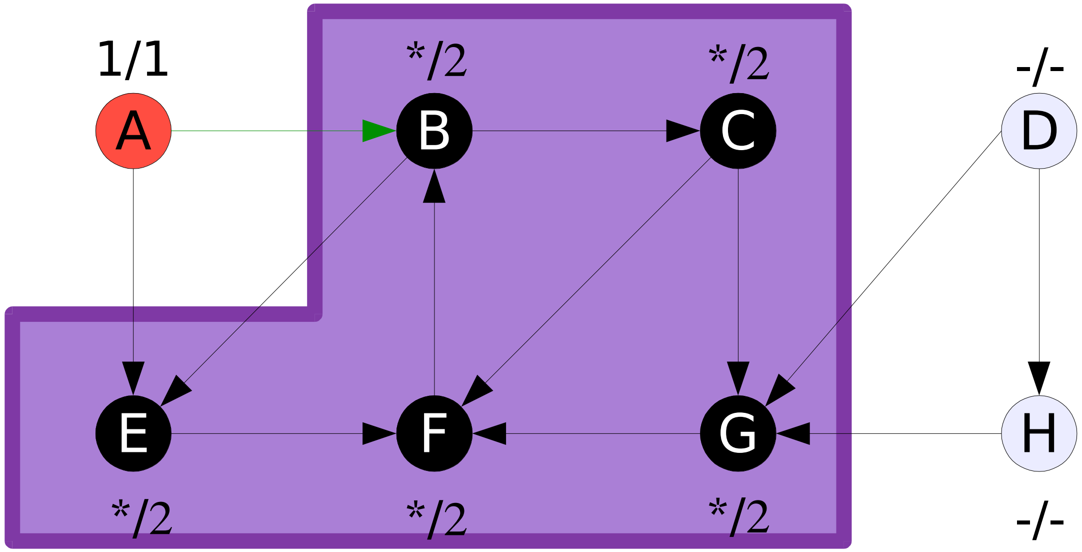


- SCC-Set(B) ={B, C, E, F, G}

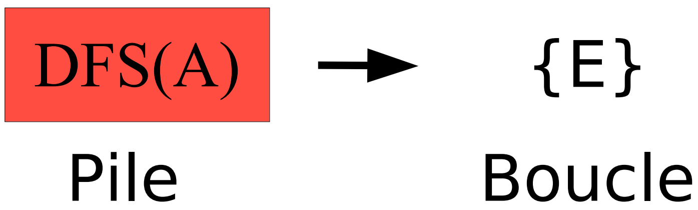
- Mettre à jour chaque *v*.num de SCC-Set(B) par * pour signifier son appartenance à une SCC déjà construite.



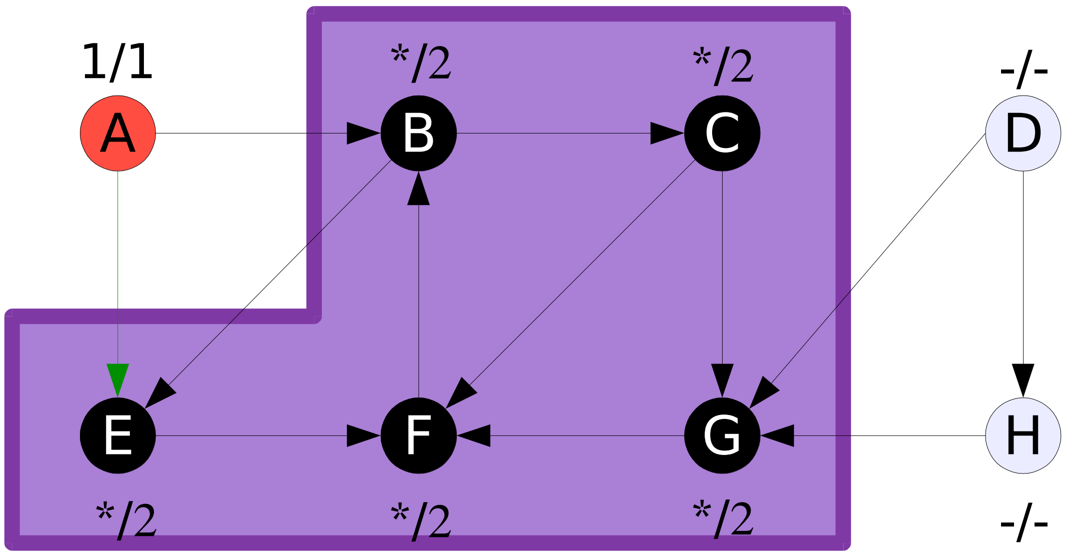
Algorithme de Tarjan
(version sans pile) : exemple de mise à jour de num et lowlink



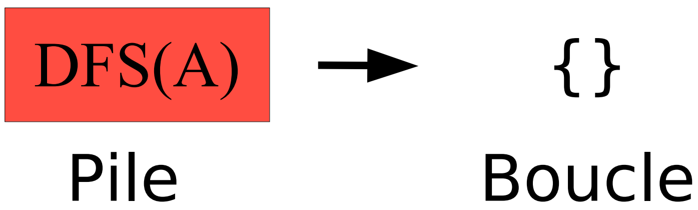
A.lowlink = min (A.lowlink, B.lowlink)

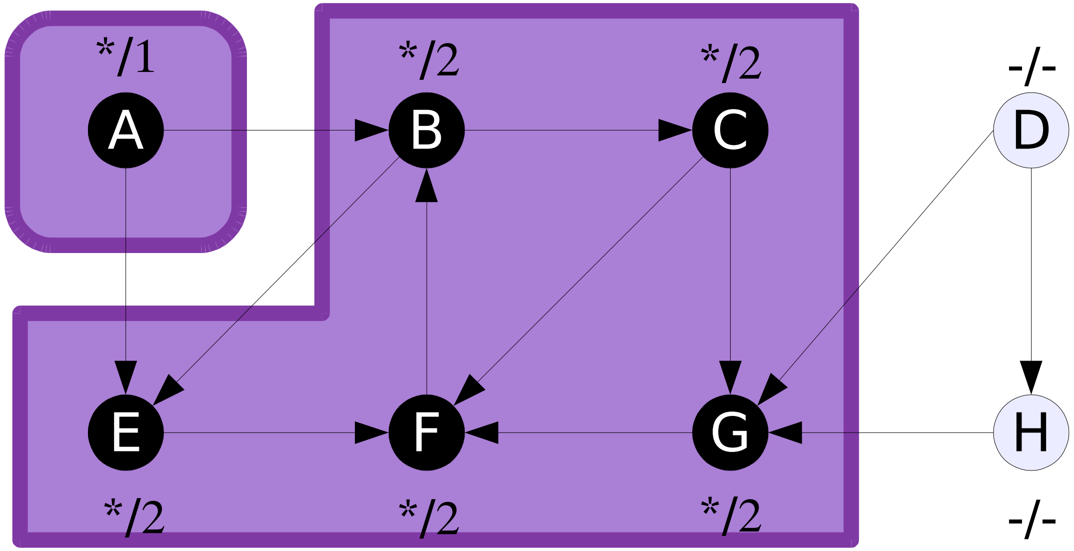


Algorithme de Tarjan (version sans pile) : exemple de mise à jour de num et lowlink



A.lowlink = min (A.lowlink, E.lowlink)





• SCC-Set(A) ={A}

Pile Boucle

Algorithme de Tarjan (version sans pile) : récap.

- La condition citée plus haut, peut être vérifiée en associant, lors d'un DFS du graphe, à chaque sommet v, deux informations :
  - *v.num* : représentant *l'ordre du parcours* de v dans un DFS.
  - *v.lowlink* : représentant *le plus petit num des sommets accessibles* par l'ensemble DFS-tree(v)
- Un sommet v est une racine d'une SCC ssi à la fin du DFS de v :
  $v.lowlink = v.num$
- L'ensemble des sommets formant la SCC d'un sommet v est défini par
  $SCC\text{-}Set(B) = \{v \in DFS\text{-}tree(B) \mid v.lowlink \notin num\text{-}Roots(B)\}$ avec,
  $Roots(B) = \{v \in \text{DFS-tree}(B) \mid v \neq B \wedge v.\text{lowlink} = v.\text{num}\}$ et
  $\text{num-Roots}(B) = \{ v.num \mid v \in \text{Roots}(B) \}$

Algorithme de Tarjan (version sans pile) : formellement (1/2)

Variables globales

num = 0
sccs = ϕ

Tarjan-SCC-VSP (graphe <*S*, *E*>)

```
POUR CHAQUE v ∈ S FAIRE
    v.num = v.lowlink = -
FIN POUR

POUR v ∈ S FAIRE
    SI v.num == - ALORS
        SCC(<S,E>, v)
    FIN SI
FIN POUR
```

SCC (graphe <*S*, *E*>, sommet *v*)

```
v.num = v.lowlink = ++num

POUR CHAQUE (v, w) ∈ E FAIRE
    SI w.num == - ALORS
        SCC(<S,E>, w)
    FIN SI

    SI w.num != * ALORS
        v.lowlink = min(v.lowlink, w.lowlink)
    FIN SI
FIN POUR

SI v.lowlink == v.num ALORS
    v.num = * ; scc = {v}
    SCC-SET(<S,E>, v, scc)
    sccs = sccs U scc
FIN SI
```

Algorithme de Tarjan (version sans pile) : formellement (2/2)

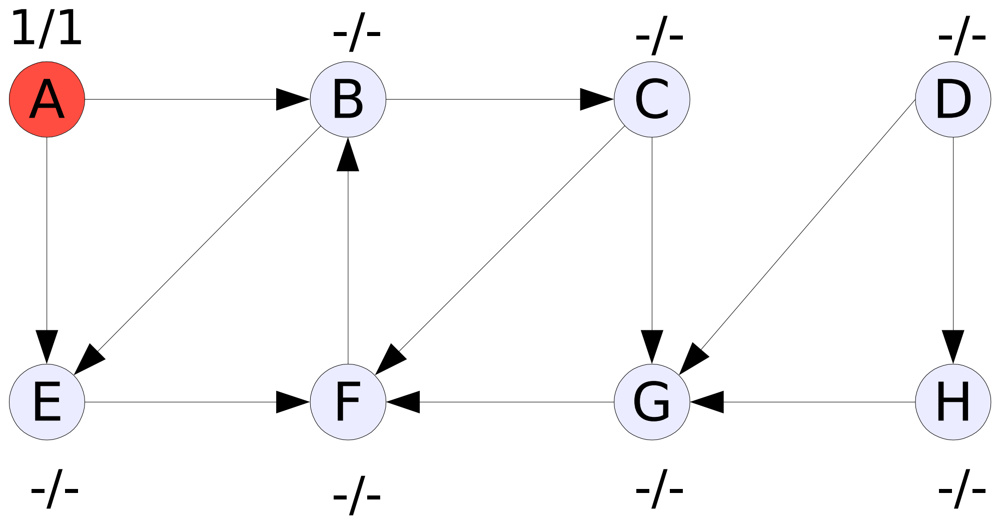
SCC-SET (graphe <*S*, *E*>, *sommet* v, *CFC* scc)

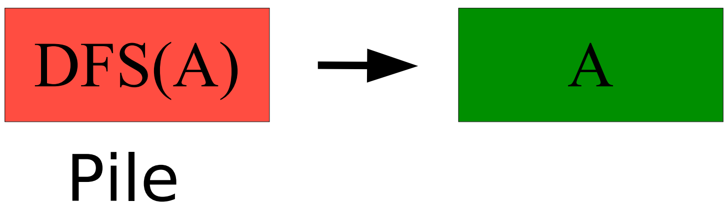
POUR CHAQUE $(v, w) \in E$ **FAIRE**
  SI w.num != * **ALORS**
    w.num = *
    scc = scc $\cup \{w\}$
    SCC-SET(<S,E>,w)
  FIN SI

On vient d'écrire un algorithme qui fait deux parcours pour chaque états et chaque arc, alors qu'on a annoncé que l'algorithme de Tarjan faisait **un seul parcours en profondeur** !!
Où est le problème ?

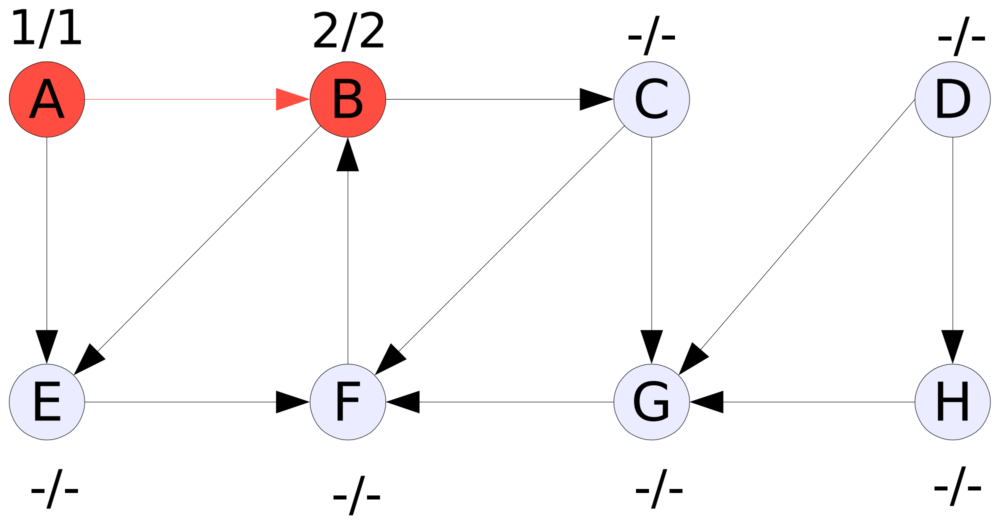
En maintenant une pile (explicite) des sommets lors du DFS, on peut écrire l'algorithme avec un seul parcours DFS....D'ailleurs c'est le vrai algorithme de Tarjan !

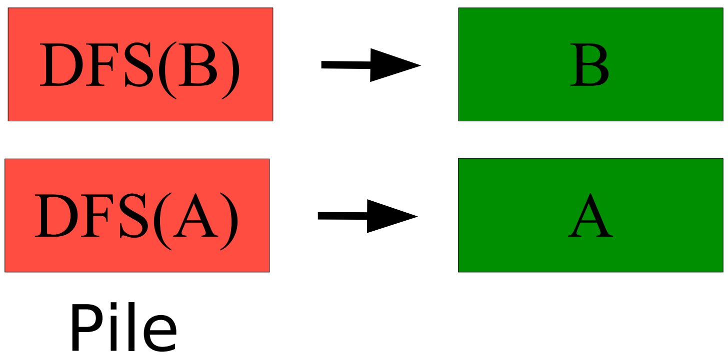
Algorithme de Tarjan : exemple de la version avec pile



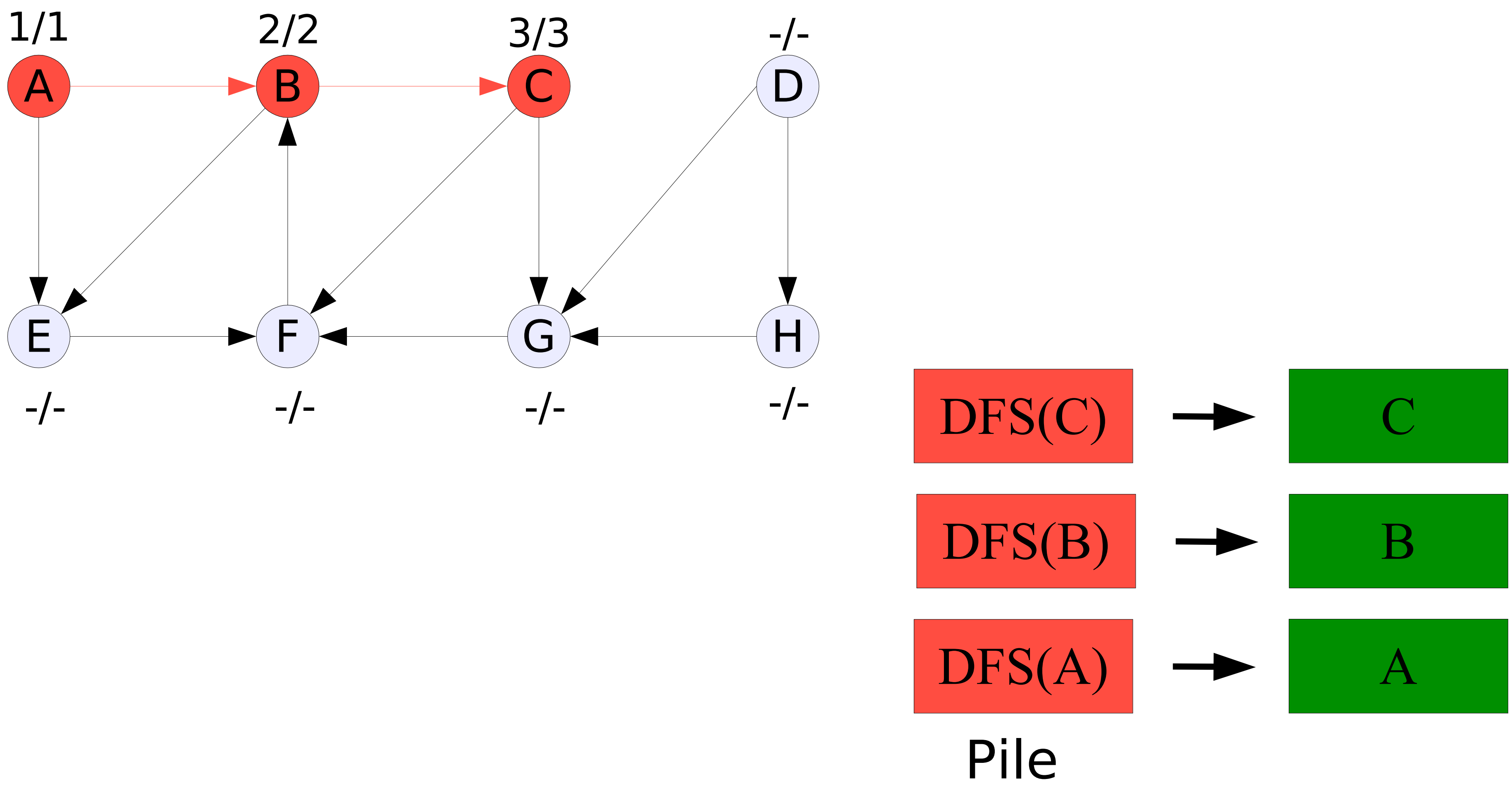


Algorithme de Tarjan : exemple de la version avec pile

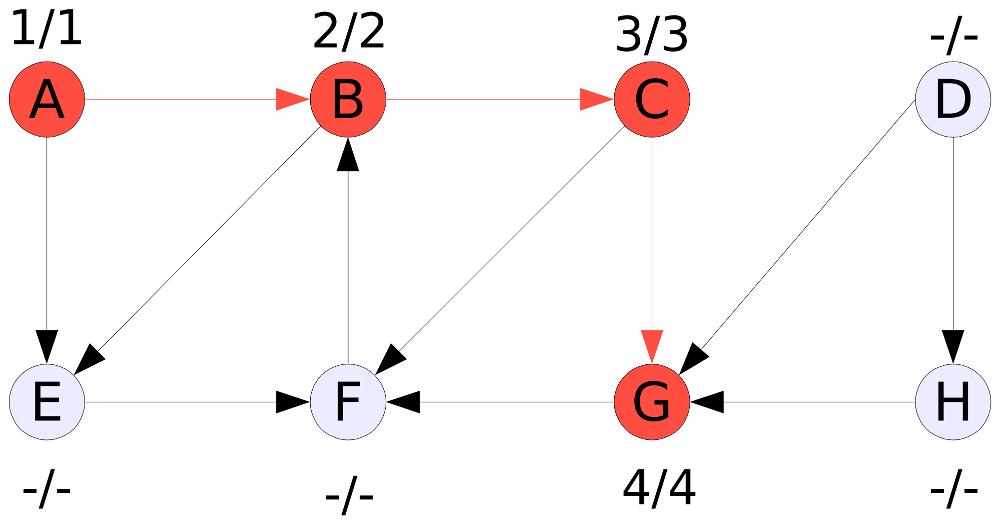


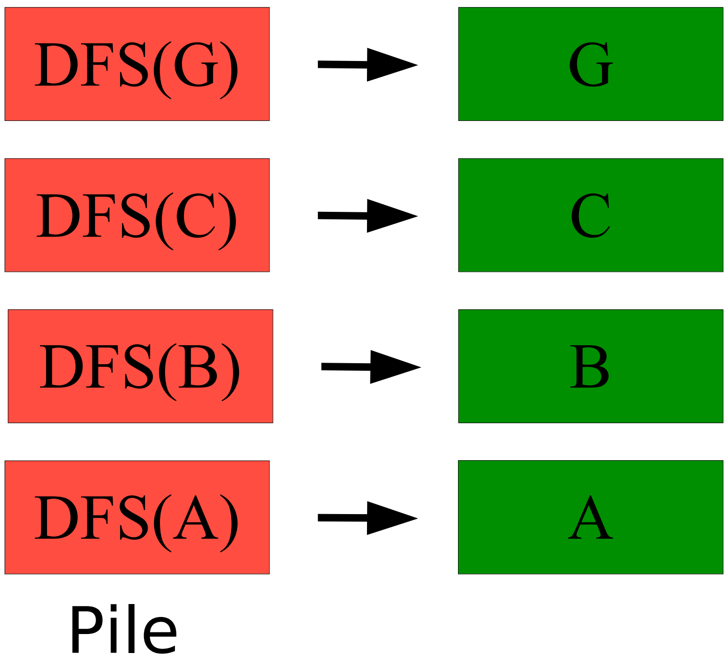


Algorithme de Tarjan : exemple de la version avec pile

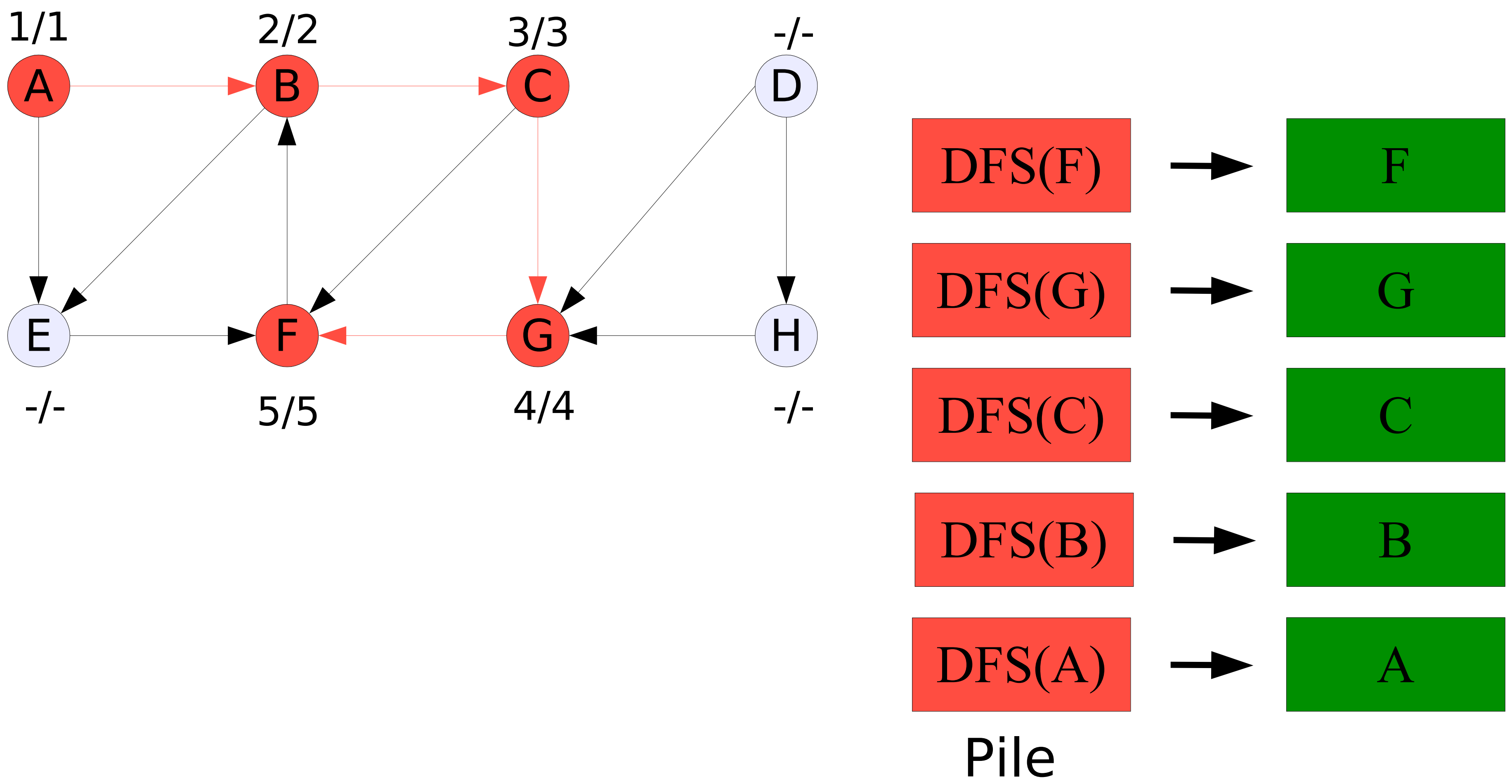


Algorithme de Tarjan : exemple de la version avec pile

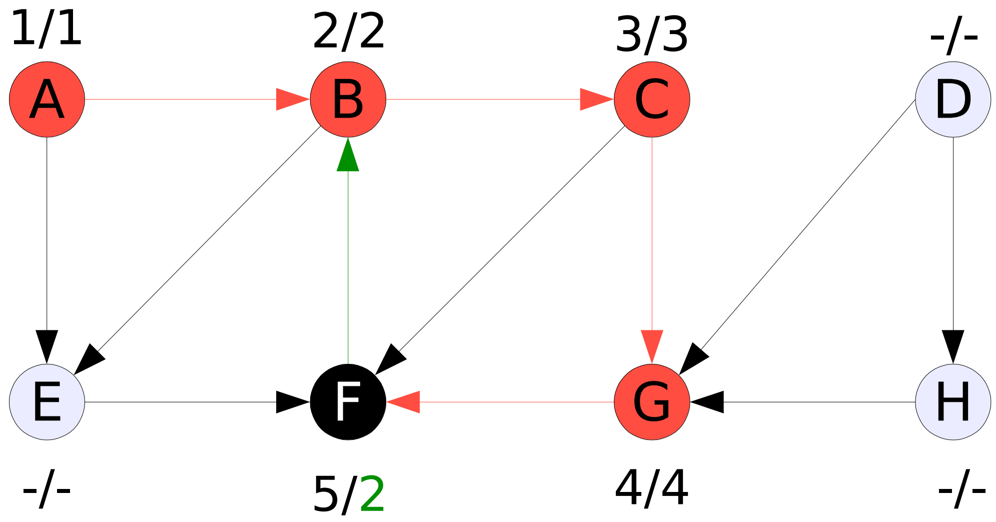


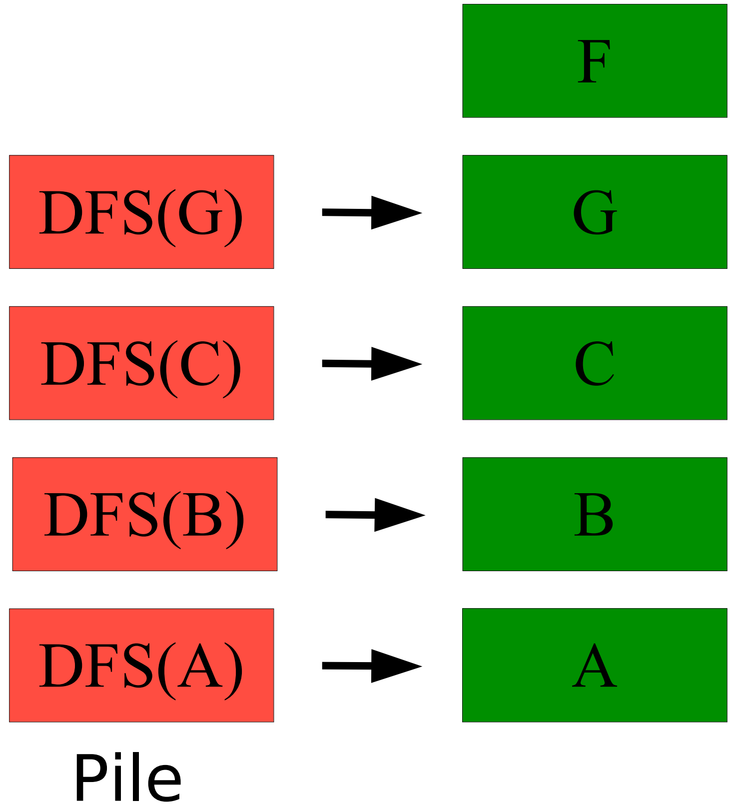


Algorithme de Tarjan : exemple de la version avec pile

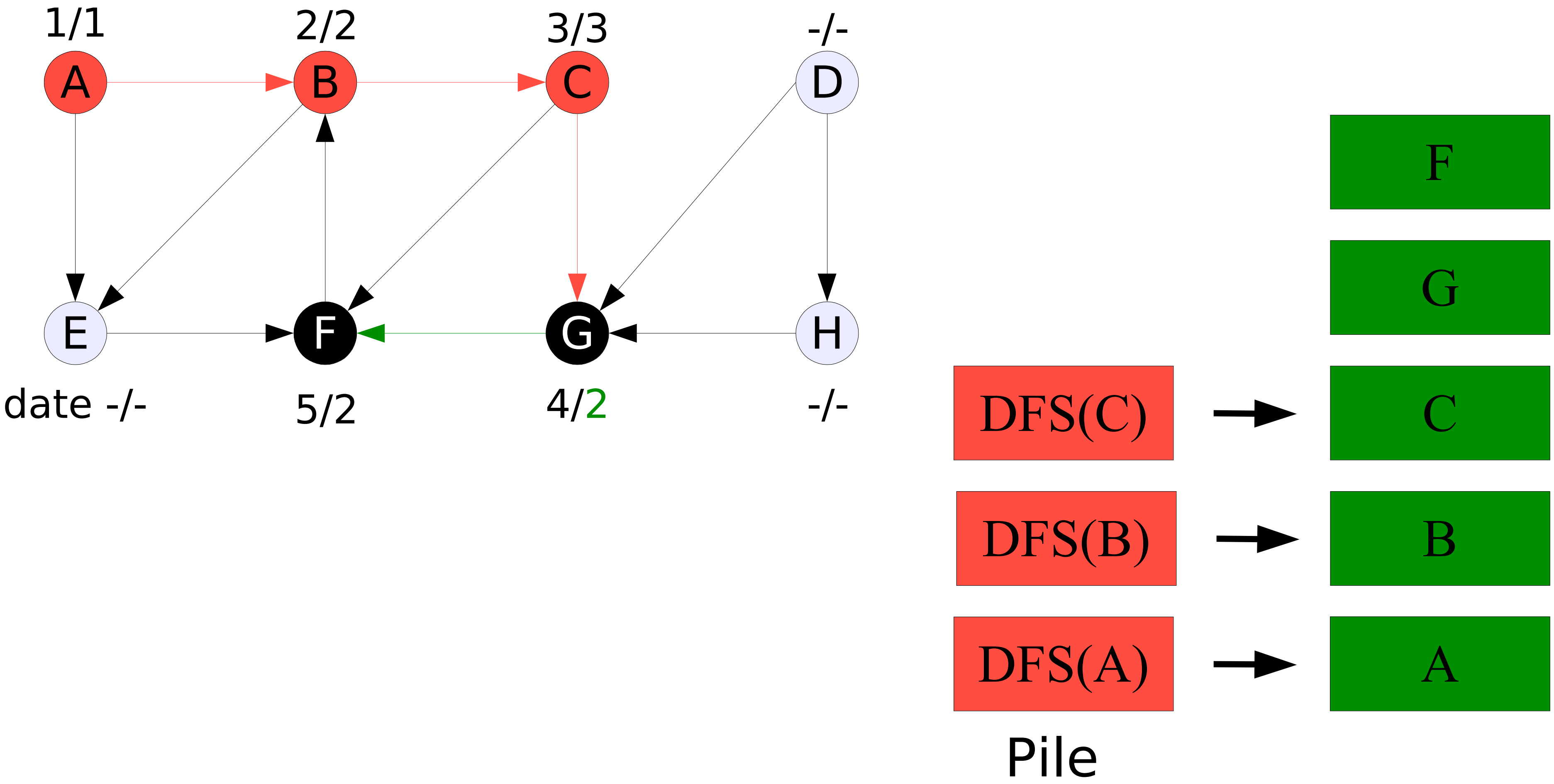


Algorithme de Tarjan : exemple de la version avec pile

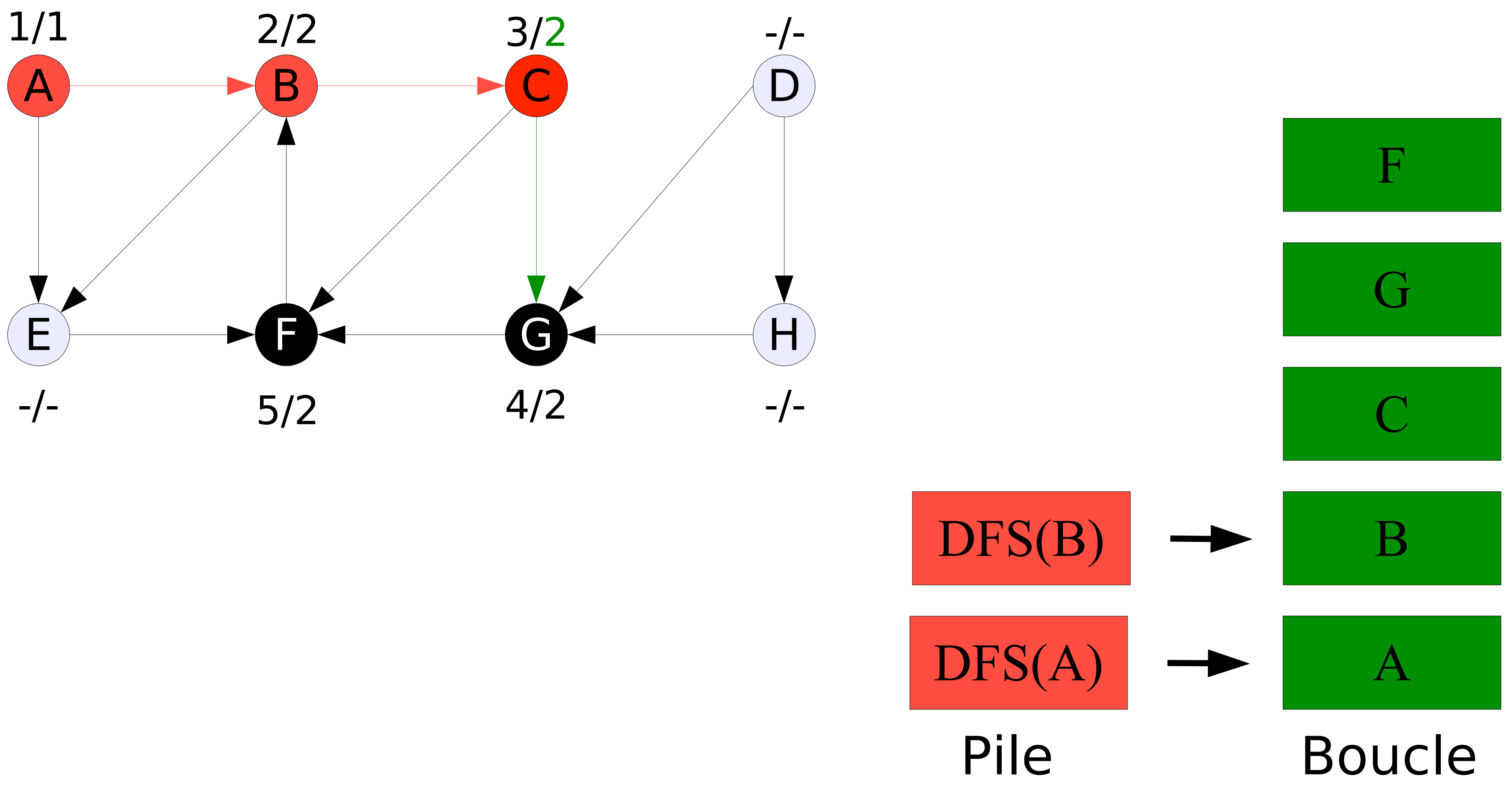




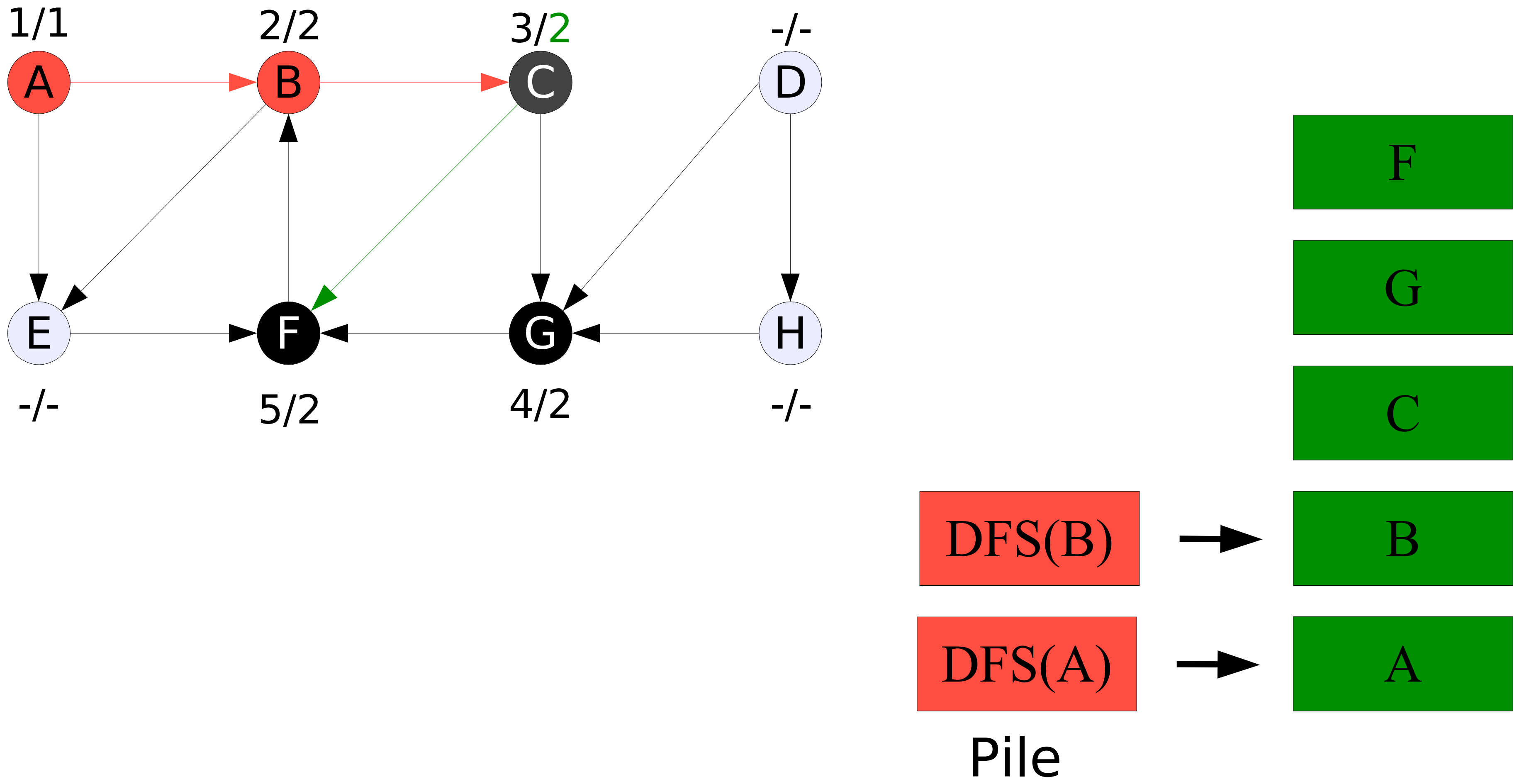
Algorithme de Tarjan : exemple de la version avec pile



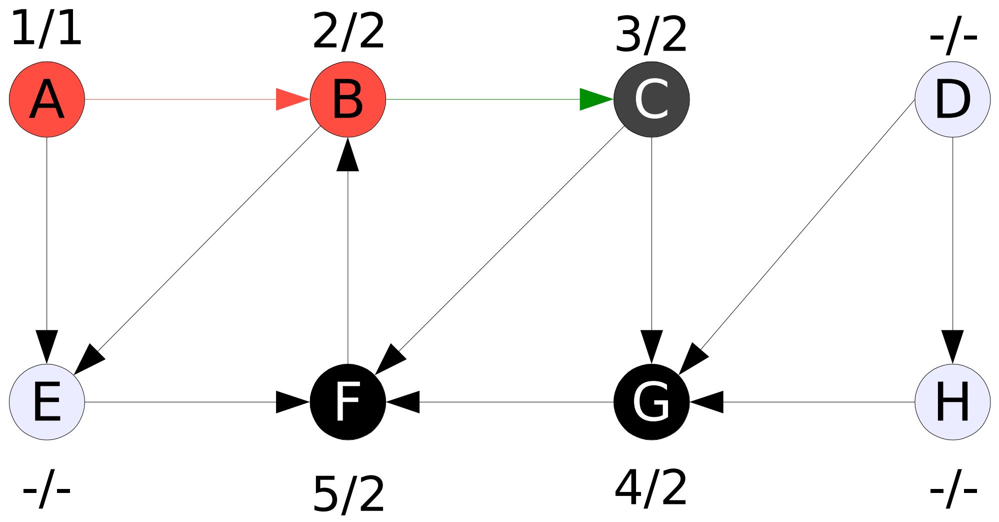
Algorithme de Tarjan : exemple de la version avec pile

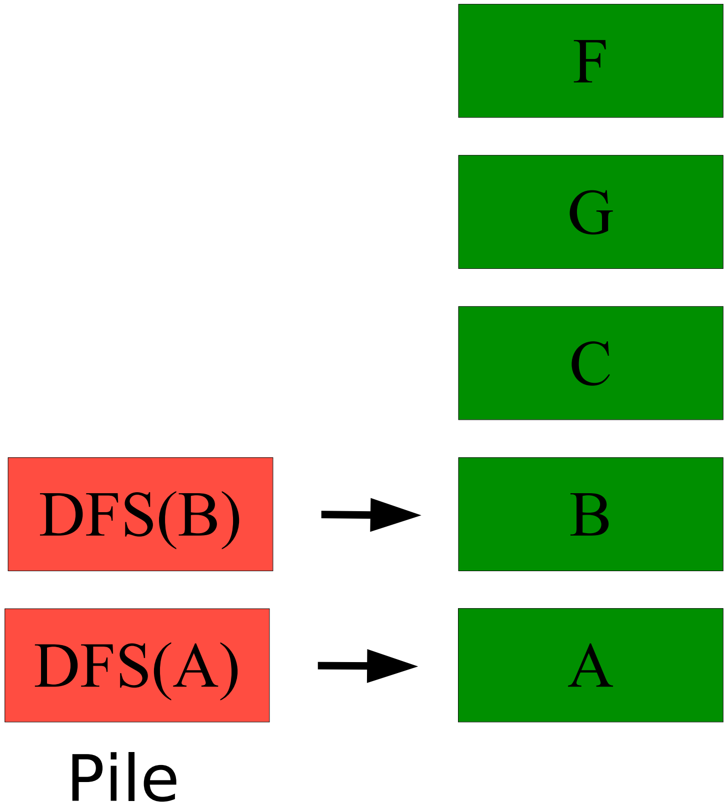


Algorithme de Tarjan : exemple de la version avec pile

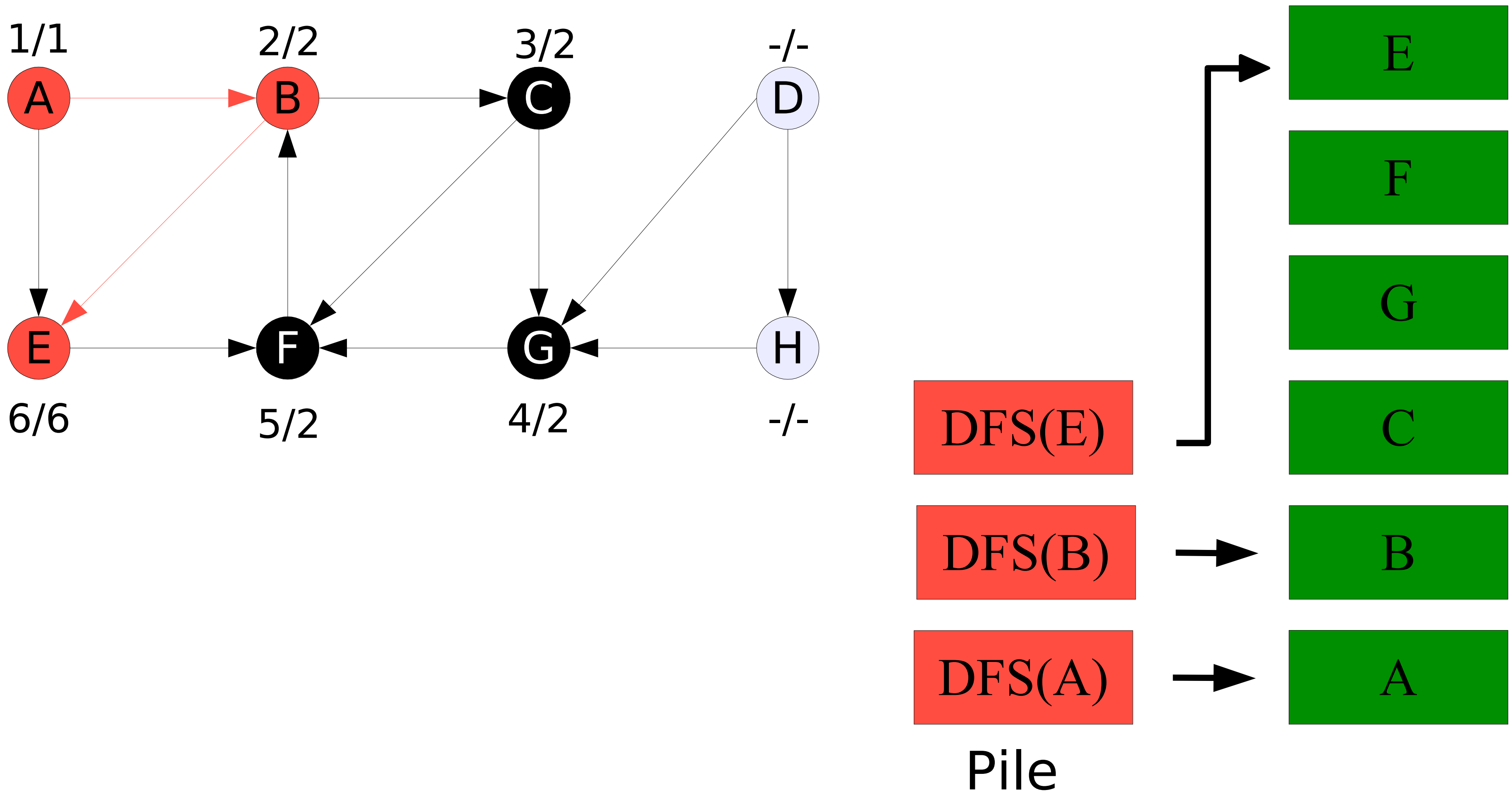


Algorithme de Tarjan : exemple de la version avec pile

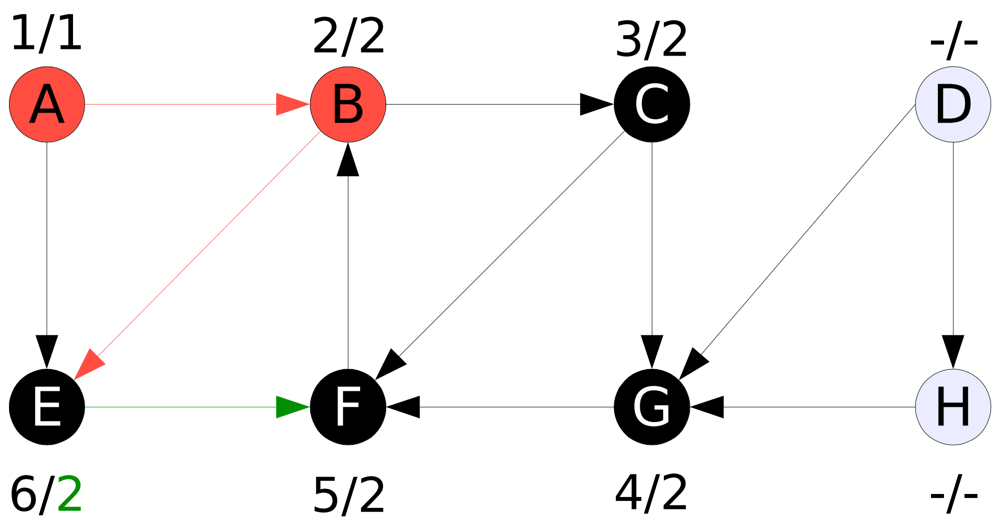


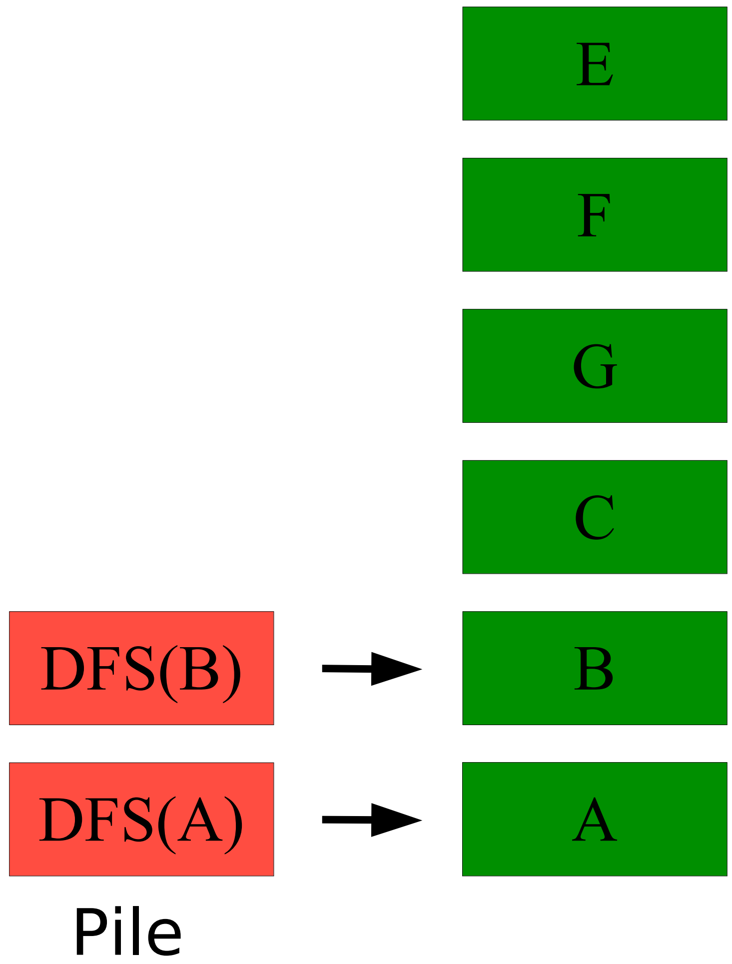


Algorithme de Tarjan :
exemple de la version avec pile

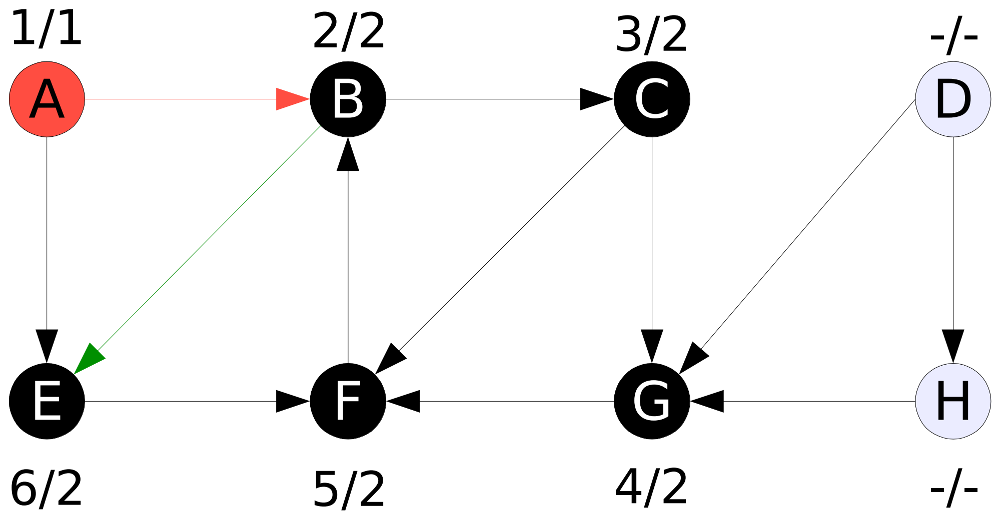


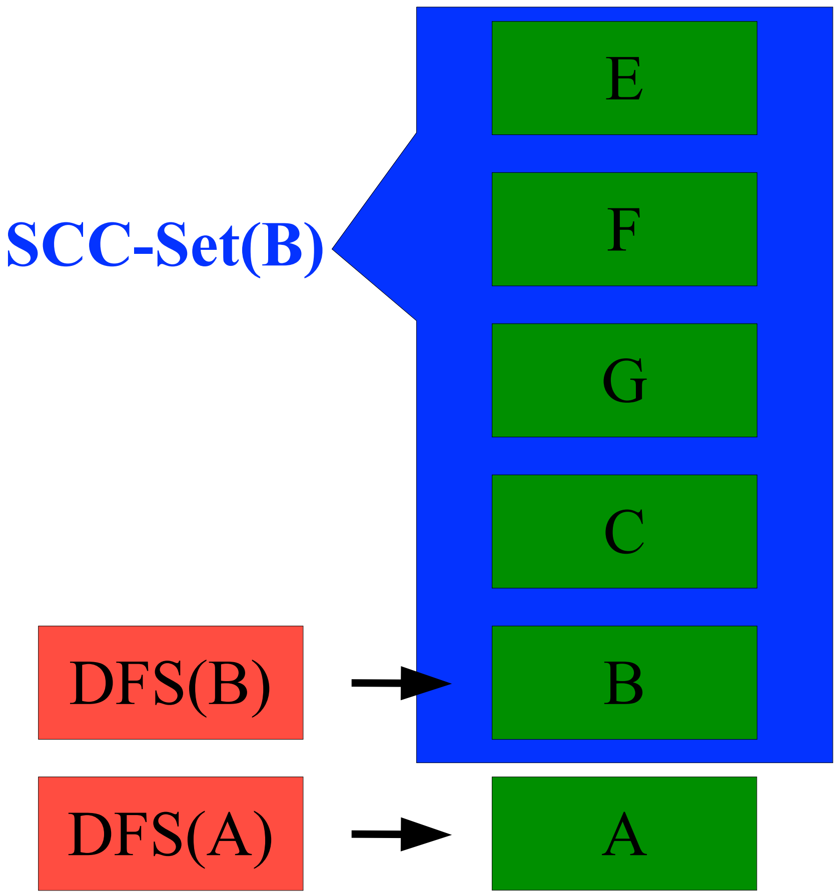
Algorithme de Tarjan : exemple de la version avec pile





Algorithme de Tarjan : exemple de la version avec pile





Le parcours de DFS-Tree(B) est totalement terminé et on a B.lowlink = B.num

Algorithme de Tarjan

```
Variables globales
    num = 0
    sccs = ϕ
    pile = ϕ

Tarjan-SCC (graphe <S, E>)

    POUR CHAQUE v ∈ S  FAIRE
        v.num = v.lowlink = -
    FIN POUR

    POUR v ∈ S  FAIRE
        SI v.num == -  ALORS
            SCC(<S,E>, v)
        FIN SI
    FIN POUR
```

```
SCC (graphe <S, E>, sommet v )
    v.num = v.lowlink = ++num
    pile.empiler(v)

    POUR CHAQUE (v, w) ∈ E  FAIRE
        SI w.num == -  ALORS
            SCC(<S,E>, w)
            v.lowlink = min(v.lowlink, w.lowlink)
        SINON SI w ∈ pile ALORS
            v.lowlink = min(v.lowlink, w.num)
        FIN SI
    FIN POUR

    SI v.lowlink == v.num ALORS
        Faire
            w = pile.depiler()
            scc = scc ∪ {w}
        Tanque  w != v
        sccs =  sccs ∪ scc
    FIN SI
```