Algorithmes classiques

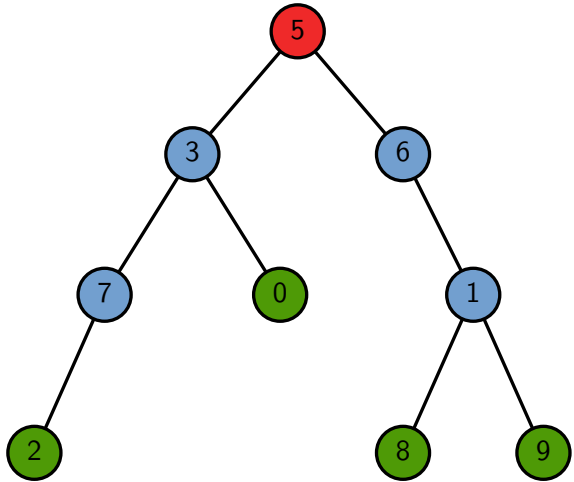
DIU Enseignement de l'informatique au lycée
Bloc 5 : Algorithmique avancée

Université de Montpellier
Juin 2021

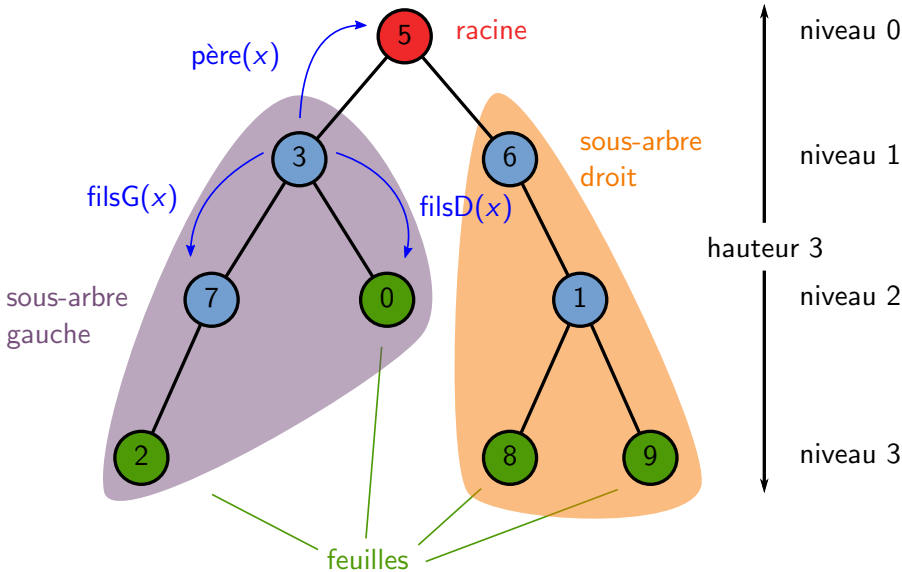
1. Arbres binaires

2. Graphes

Vocabulaire



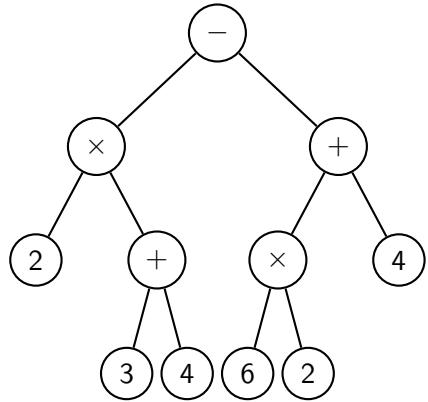
Vocabulaire



Utilité des arbres binaires

- Arbres binaires de recherche
- Tas

- Analyse syntaxique
- Bases de données
- Partition binaire de l'espace
- Tables de routage
- ...



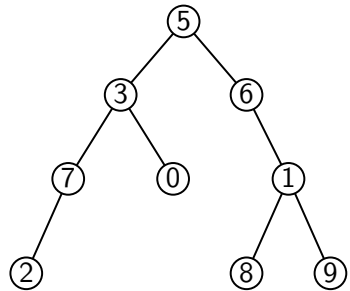
$2 \times (3 + 4) - (6 \times 2 + 4)$

Parcours en profondeur d'un arbre binaire

Algorithme : ParcoursInfixe(x)

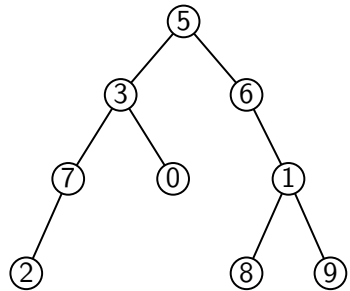
si $x \neq \emptyset$ **:**

- ParcoursInfixe(filsG(x))
- Afficher val(x)
- ParcoursInfixe(filsD(x))



Parcours en profondeur d'un arbre binaire

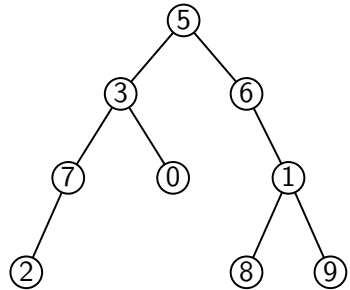
Algorithme : ParcoursInfixe(x)
si $x \neq \emptyset$ **:**
    ParcoursInfixe(filsG(x))
    Afficher val(x)
    ParcoursInfixe(filsD(x))



- Affichage : 2 7 3 0 5 6 8 1 9

Parcours en profondeur d'un arbre binaire

Algorithme : ParcoursInfixe(x)
si $x \neq \emptyset$ **:**
    ParcoursInfixe(filsG(x))
    Afficher val(x)
    ParcoursInfixe(filsD(x))

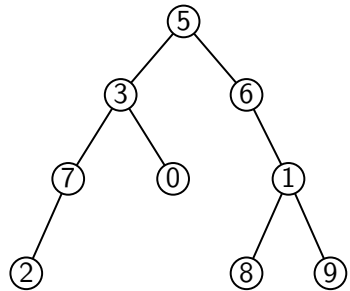


- Affichage : 2 7 3 0 5 6 8 1 9
- Complexité en $O(n(A))$

  Preuve $\mathcal{P}_n$: l'algo. effectue $2n$ appels à lui-même au total
  - $n = 0$: pas trop dur...
  - Supp. $\mathcal{P}_k$ pour tout $k < n(A)$ et soit n_G et n_D le nb de nœuds dans les sous-arbres gauche et droit. Dans les deux appels récursifs, $2n_G$ et $2n_D$ appels à ParcoursInfixe, donc au total $2n_G + 2n_D + 2$ appels. Or $n(A) = n_G + n_D + 1$, d'où le résultat.

Parcours en profondeur d'un arbre binaire

Algorithme : ParcoursInfixe(x)
si $x \neq \emptyset$ **:**
    ParcoursInfixe(filsG(x))
    Afficher val(x)
    ParcoursInfixe(filsD(x))

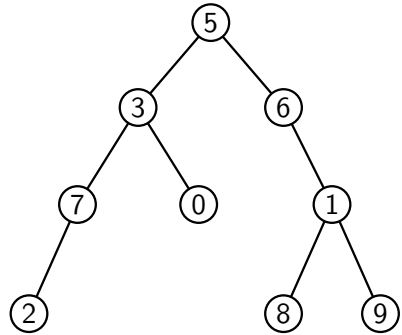


- Affichage : 2 7 3 0 5 6 8 1 9
- Complexité en $O(n(A))$
- Appel de la fonction : ParcoursInfixe(rac(A))
- Variantes :
  - ParcoursPrefixe : 5 3 7 2 0 6 1 8 9
  - ParcoursSuffixe : 2 7 0 3 8 9 1 6 5

Exemples d'algorithmes

Algorithme : MINIMUM(x)

$m \leftarrow +\infty$
si $x \neq \emptyset$ **:**
    $m_G \leftarrow$ MINIMUM(filsG(x))
    $m_D \leftarrow$ MINIMUM(filsD(x))
    $m \leftarrow \min(m_G, m_D, \text{val}(x))$
renvoyer m



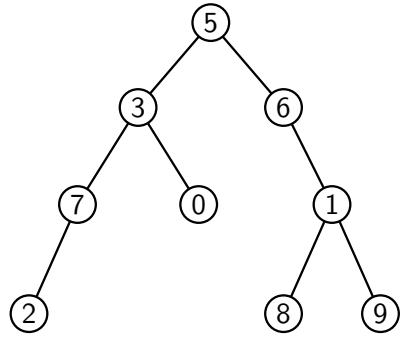
Exemples d'algorithmes

Algorithme : MINIMUM(x)

$m \leftarrow +\infty$
si $x \neq \emptyset$ **:**
    $m_G \leftarrow$ MINIMUM(filsG(x))
    $m_D \leftarrow$ MINIMUM(filsD(x))
    $m \leftarrow \min(m_G, m_D, \text{val}(x))$
renvoyer m

Algorithme : NBNŒUDS(x)

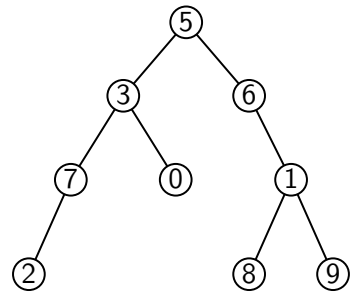
$n \leftarrow 0$
si $x \neq \emptyset$ **:**
    $n_G \leftarrow$ NBNŒUDS(filsG(x))
    $n_D \leftarrow$ NBNŒUDS(filsD(x))
    $n \leftarrow n_G + n_D + 1$
renvoyer n



Parcours en largeur d'un arbre binaire

Algorithme : PARCOURSLARGEUR(x)

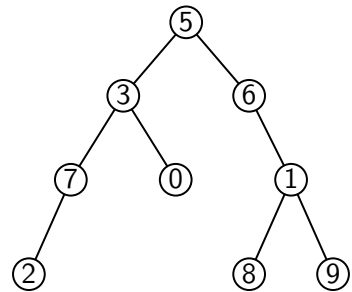
$F \leftarrow$ file vide
si $x \neq \emptyset$ **:** l'ajouter à F
tant que F *est non vide* **:**
    $y \leftarrow$ défiler un élément de F
    Afficher val(y)
    si filsG(y) $\neq \emptyset$ **:** l'ajouter à F
    si filsD(y) $\neq \emptyset$ **:** l'ajouter à F



Parcours en largeur d'un arbre binaire

Algorithme : PARCOURSLARGEUR(x)

$F \leftarrow$ file vide
si $x \neq \emptyset$ **:** l'ajouter à F
tant que F *est non vide* **:**
  $y \leftarrow$ défiler un élément de F
  Afficher val(y)
  si filsG(y) $\neq \emptyset$ **:** l'ajouter à F
  si filsD(y) $\neq \emptyset$ **:** l'ajouter à F

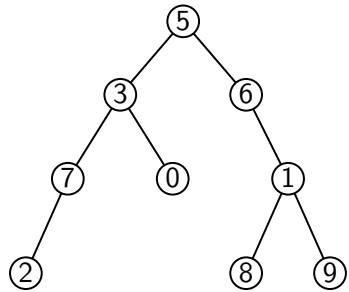


- Affichage : 5 3 6 7 0 1 2 8 9
  - niveau par niveau
  - de gauche à droite

Parcours en largeur d'un arbre binaire

Algorithme : ParcoursLargeur(x)

$F \leftarrow$ file vide
si $x \neq \emptyset$ **:** l'ajouter à F
tant que F *est non vide* **:**
  $y \leftarrow$ défiler un élément de F
  Afficher val(y)
  si filsG(y) $\neq \emptyset$ **:** l'ajouter à F
  si filsD(y) $\neq \emptyset$ **:** l'ajouter à F



- Affichage : 5 3 6 7 0 1 2 8 9
  - niveau par niveau
  - de gauche à droite
- Complexité en $O(n(A))$

Arbre Binaires de Recherche : objectifs

Stocker un **ensemble ordonné** de n valeurs avec les opérations :

- Insérer et Supprimer
- Minimum et Maximum
- Rechercher
- Successeur et Prédécesseur

$\rightsquigarrow$ toutes ces opérations en $\ll$ bonne $\gg$ complexité

Arbre Binaires de Recherche : objectifs

Stocker un **ensemble ordonné** de n valeurs avec les opérations :

- Insérer et Supprimer
- Minimum et Maximum
- Rechercher
- Successeur et Prédécesseur

Liste chaînée triée : $O(1)$ pour max/min et succ/pred, $O(n)$ pour le reste

$\rightsquigarrow$ toutes ces opérations en $\ll$ bonne $\gg$ complexité

Arbre Binaires de Recherche : objectifs

Stocker un **ensemble ordonné** de n valeurs avec les opérations :

- Insérer et Supprimer
- Minimum et Maximum
- Rechercher
- Successeur et Prédécesseur

Liste chaînée triée : $O(1)$ pour max/min et succ/pred, $O(n)$ pour le reste

$\rightsquigarrow$ toutes ces opérations en $\ll$ bonne $\gg$ complexité

Utilisation

- Stockage de données dynamiques
- Base de données (valeurs = identifiant)
- Linux : ordonnancement, mémoire virtuelle, ...

Arbre Binaires de Recherche : objectifs

Stocker un **ensemble ordonné** de n valeurs avec les opérations :

- Insérer et Supprimer
- Minimum et Maximum
- Rechercher
- Successeur et Prédécesseur

Liste chaînée triée : $O(1)$ pour max/min et succ/pred, $O(n)$ pour le reste

$\rightsquigarrow$ toutes ces opérations en « bonne » complexité

Utilisation

- Stockage de données dynamiques
- Base de données (valeurs = identifiant)
- Linux : ordonnancement, mémoire virtuelle, ...

Les arbres binaires de recherche sont **une** structure de donnée remplissant ces objectifs, mais pas la seule !

Définition

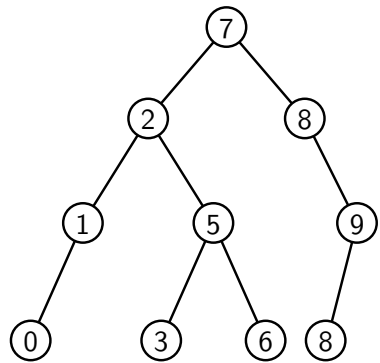
Un **arbre binaire de recherche** (ABR) est un arbre binaire tel que *pour tout nœud* x,

- tous les nœuds y du sous-arbre gauche de x vérifient $\text{val}(y) \leq \text{val}(x)$
- tous les nœuds z du sous-arbre droit de x vérifient $\text{val}(z) \geq \text{val}(x)$

Définition

Un **arbre binaire de recherche** (ABR) est un arbre binaire tel que *pour tout nœud* x,

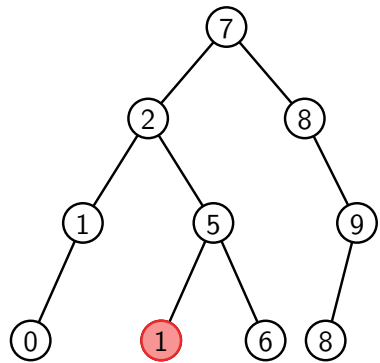
- tous les nœuds y du sous-arbre gauche de x vérifient $\text{val}(y) \leq \text{val}(x)$
- tous les nœuds z du sous-arbre droit de x vérifient $\text{val}(z) \geq \text{val}(x)$



Définition

Un **arbre binaire de recherche** (ABR) est un arbre binaire tel que *pour tout nœud* x,

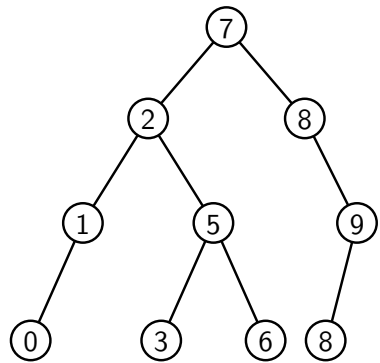
- tous les nœuds y du sous-arbre gauche de x vérifient $\text{val}(y) \leq \text{val}(x)$
- tous les nœuds z du sous-arbre droit de x vérifient $\text{val}(z) \geq \text{val}(x)$



Définition

Un **arbre binaire de recherche** (ABR) est un arbre binaire tel que *pour tout nœud* x,

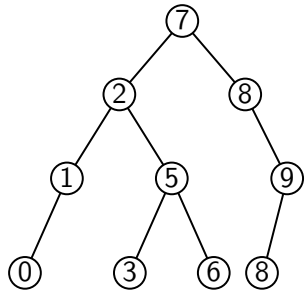
- tous les nœuds y du sous-arbre gauche de x vérifient $\text{val}(y) \leq \text{val}(x)$
- tous les nœuds z du sous-arbre droit de x vérifient $\text{val}(z) \geq \text{val}(x)$



Recherche dans un ABR

Algorithme : RECHERCHER(x, k)

si $x = \emptyset$ **: renvoyer** $\emptyset$
si $\text{val}(x) = k$ **: renvoyer** x
si $\text{val}(x) > k$ **:**
  renvoyer RECHERCHER($\text{filsG}(x), k$)
renvoyer RECHERCHER($\text{filsD}(x), k$)



Recherche dans un ABR

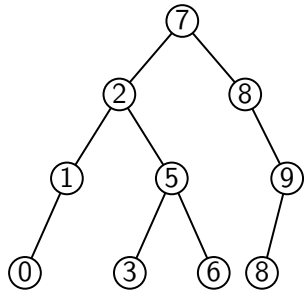
Algorithme : RECHERCHER(x, k)

tant que $x \neq \emptyset$ *et* $\text{val}(x) \neq k$ **:**

  si $k < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$

  sinon : $x \leftarrow \text{filsD}(x)$

renvoyer x



Recherche dans un ABR

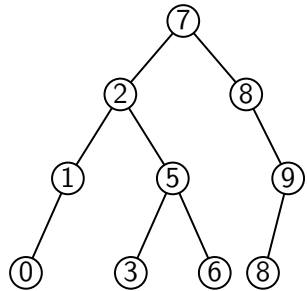
Algorithme : RECHERCHER(x, k)

tant que $x \neq \emptyset$ *et* $\text{val}(x) \neq k$ **:**

  si $k < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$

  sinon : $x \leftarrow \text{filsD}(x)$

renvoyer x



Lemme

RECHERCHER$(\text{rac}(A), k)$ *a une complexité* $O(h(A))$.

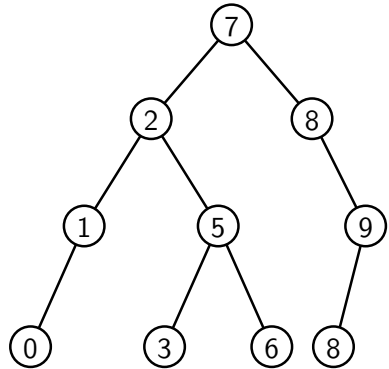
Preuve

- À chaque itération, la hauteur de x augmente de 1 : $\leq h(A)$ itérations
- Chaque itération coûte $O(1)$

Insertion d'un élément

Algorithme : INSÉRER(A, z)

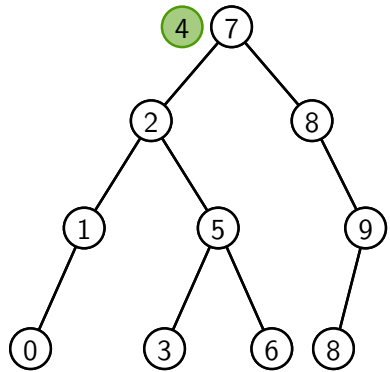
$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ **:**
    $p \leftarrow x$
    si $\text{val}(z) < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$
    sinon : $x \leftarrow \text{filsD}(x)$
père$(z) \leftarrow p$
si $p = \emptyset$ **:** $\text{rac}(A) \leftarrow z$
sinon si $\text{val}(z) < \text{val}(p)$ **:** $\text{filsG}(p) \leftarrow z$
sinon : $\text{filsD}(p) \leftarrow z$



Insertion d'un élément

Algorithme : INSÉRER(A, z)

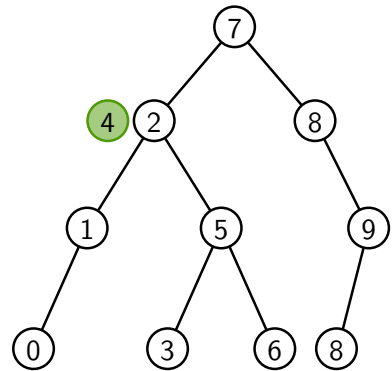
$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ **:**
    $p \leftarrow x$
    si $\text{val}(z) < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$
    sinon : $x \leftarrow \text{filsD}(x)$
père$(z) \leftarrow p$
si $p = \emptyset$ **:** $\text{rac}(A) \leftarrow z$
sinon si $\text{val}(z) < \text{val}(p)$ **:** $\text{filsG}(p) \leftarrow z$
sinon : $\text{filsD}(p) \leftarrow z$



Insertion d'un élément

Algorithme : INSÉRER(A, z)

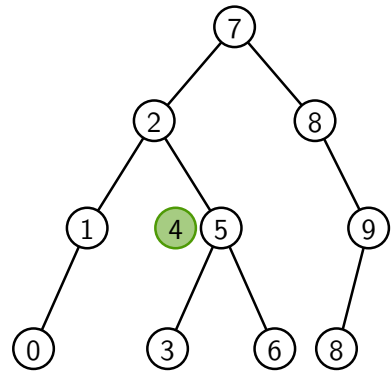
$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ **:**
    $p \leftarrow x$
    si $\text{val}(z) < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$
    sinon : $x \leftarrow \text{filsD}(x)$
père$(z) \leftarrow p$
si $p = \emptyset$ **:** $\text{rac}(A) \leftarrow z$
sinon si $\text{val}(z) < \text{val}(p)$ **:** $\text{filsG}(p) \leftarrow z$
sinon : $\text{filsD}(p) \leftarrow z$



Insertion d'un élément

Algorithme : INSÉRER(A, z)

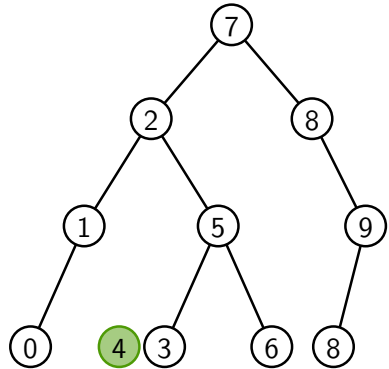
$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ **:**
    $p \leftarrow x$
    si $\text{val}(z) < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$
    sinon : $x \leftarrow \text{filsD}(x)$
$\text{père}(z) \leftarrow p$
si $p = \emptyset$ **:** $\text{rac}(A) \leftarrow z$
sinon si $\text{val}(z) < \text{val}(p)$ **:** $\text{filsG}(p) \leftarrow z$
sinon : $\text{filsD}(p) \leftarrow z$



Insertion d'un élément

Algorithme : INSÉRER(A, z)

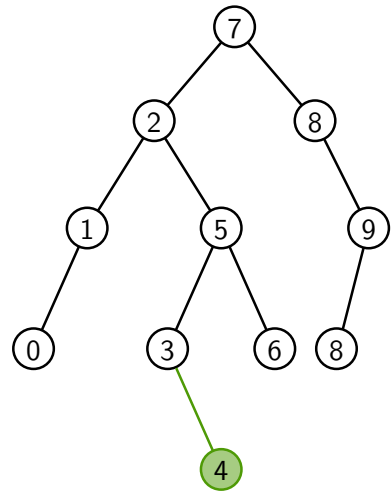
$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ **:**
    $p \leftarrow x$
    si $\text{val}(z) < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$
    sinon : $x \leftarrow \text{filsD}(x)$
père$(z) \leftarrow p$
si $p = \emptyset$ **:** $\text{rac}(A) \leftarrow z$
sinon si $\text{val}(z) < \text{val}(p)$ **:** $\text{filsG}(p) \leftarrow z$
sinon : $\text{filsD}(p) \leftarrow z$



Insertion d'un élément

Algorithme : INSÉRER(A, z)

$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ **:**
    $p \leftarrow x$
    si $\text{val}(z) < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$
    sinon : $x \leftarrow \text{filsD}(x)$
$\text{père}(z) \leftarrow p$
si $p = \emptyset$ **:** $\text{rac}(A) \leftarrow z$
sinon si $\text{val}(z) < \text{val}(p)$ **:** $\text{filsG}(p) \leftarrow z$
sinon : $\text{filsD}(p) \leftarrow z$



Importance de l'équilibrage

Bilan des complexités

- INSÉRER et SUPPRIMER[1] : $O(h)$
- RECHERCHER : $O(h)$
- MINIMUM[1] et MAXIMUM[1] : $O(h)$
- SUCCESSEUR[1] et PRÉDECESSEUR[1] : $O(h)$

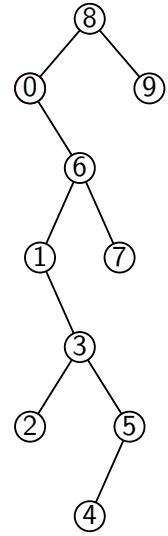
[1] Exercice !

Importance de l'équilibrage

Bilan des complexités

- INSÉRER et SUPPRIMER[1] : $O(h)$
- RECHERCHER : $O(h)$
- MINIMUM[1] et MAXIMUM[1] : $O(h)$
- SUCCESSEUR[1] et PRÉDECESSEUR[1] : $O(h)$

[1] Exercice !



Importance de l'équilibrage

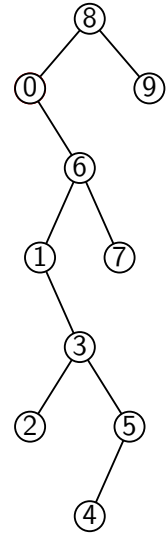
Bilan des complexités

- INSÉRER et SUPPRIMER[1] : $O(h)$
- RECHERCHER : $O(h)$
- MINIMUM[1] et MAXIMUM[1] : $O(h)$
- SUCCESSEUR[1] et PRÉDECESSEUR[1] : $O(h)$

[1] Exercice !

Lemme

$h < n < 2^{h+1}$, d'où $\lfloor \log n \rfloor \leq h < n$

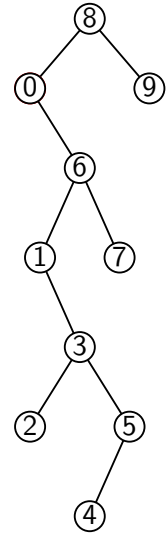


Importance de l'équilibrage

Bilan des complexités

- Insérer et Supprimer[1] : $O(h)$
- Rechercher : $O(h)$
- Minimum[1] et Maximum[1] : $O(h)$
- Successeur[1] et Prédecesseur[1] : $O(h)$

[1] Exercice !



Lemme

$h < n < 2^{h+1}$, d'où $\lfloor \log n \rfloor \leq h < n$

Preuve

- Nb de nœuds au niveau i : $1 \leq n_i \leq 2^i$ (récurrence)
- Nb total de nœuds : $h + 1 \leq \sum_{i=0}^{h} n_i \leq 2^{h+1} - 1$

Importance de l'équilibrage

Bilan des complexités

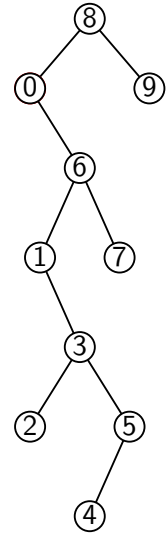
- Insérer et Supprimer[1] : $O(h)$
- Rechercher : $O(h)$
- Minimum[1] et Maximum[1] : $O(h)$
- Successeur[1] et Prédecesseur[1] : $O(h)$

[1] Exercice !

Lemme

$h < n < 2^{h+1}$, d'où $\lfloor \log n \rfloor \leq h < n$

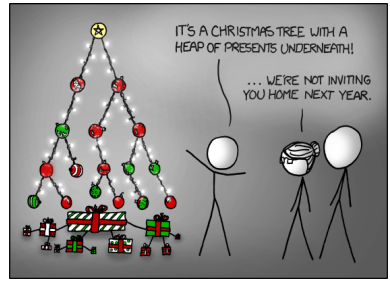
Un ABR est une structure de donnée efficace s'il est **équilibré**, c'est-à-dire si $h = O(\log n)$.



Conclusion

Les arbres binaires en informatique

- Représentation structurée de l'information
  - arbres binaires de recherche
  - tas
  - ...



https://xkcd.com/835/

Les arbres binaires en informatique

stackoverflow Search...

Home

PUBLIC

Stack Overflow

Tags

Users

Jobs

Teams
Q&A for work

Learn More

What are the applications of binary trees?

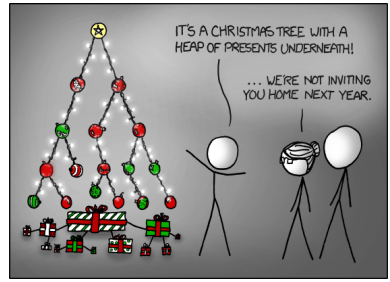
Applications of binary trees

380

- Binary Search Tree - Used in *many* search applications where data is constantly entering/leaving, such as the `map` and `set` objects in many languages' libraries.
- Binary Space Partition - Used in almost every 3D video game to determine what objects need to be rendered.
- Binary Tries - Used in almost every high-bandwidth router for storing router-tables.
- Hash Trees - used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.
- Heaps - Used in implementing efficient priority-queues, which in turn are used for scheduling processes in many operating systems, Quality-of-Service in routers, and A* *(path-finding algorithm used in AI applications, including robotics and video games)*. Also used in heap-sort.
- Huffman Coding Tree (Chip Uni) - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.
- GGM Trees - Used in cryptographic applications to generate a tree of pseudo-random numbers.

Les arbres binaires en informatique

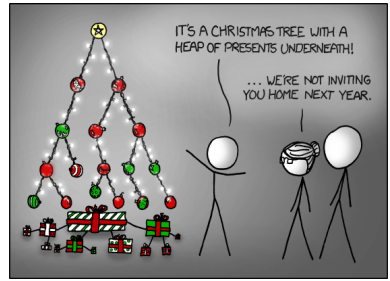
- Représentation structurée de l'information
  - arbres binaires de recherche
  - tas
  - ...
- Raisonnement informatique
  - Arbre de récursion (analyse des algorithmes récursifs)
  - Arbre de décision (borne inférieure sur le tri)
  - ...



https://xkcd.com/835/

Les arbres binaires en informatique

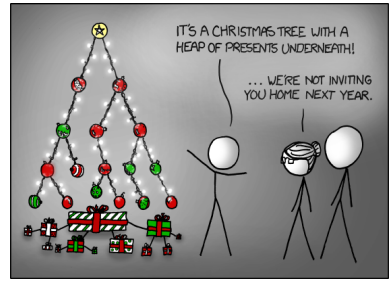
- Représentation structurée de l'information
  - arbres binaires de recherche
  - tas
  - ...
- Raisonnement informatique
  - Arbre de récursion (analyse des algorithmes récursifs)
  - Arbre de décision (borne inférieure sur le tri)
  - ...
- Pourquoi binaires ?
  - Arbres ternaires, ..., d-aires
  - Arbres avec nombre quelconque (non constant) de fils



https://xkcd.com/835/

Les arbres binaires en informatique

- Représentation structurée de l'information
  - arbres binaires de recherche
  - tas
  - ...
- Raisonnement informatique
  - Arbre de récursion (analyse des algorithmes récursifs)
  - Arbre de décision (borne inférieure sur le tri)
  - ...
- Pourquoi binaires ?
  - Arbres ternaires, ..., d-aires
  - Arbres avec nombre quelconque (non constant) de fils



https://xkcd.com/835/

Les arbres sont un des objets centraux de l'informatique !

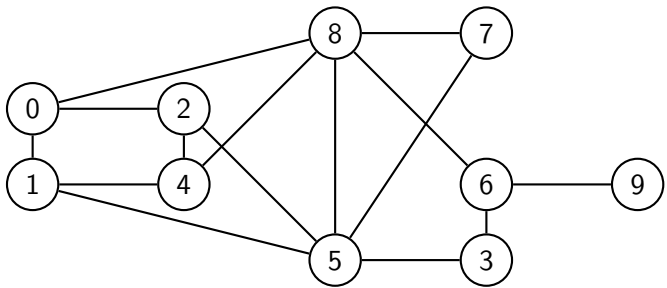
1. Arbres binaires

2. Graphes

Vocabulaire

Graphe **connexe** constitué de 10 **sommets** et 15 **arêtes** : $G = (\mathbf{S}, \mathbf{A})$

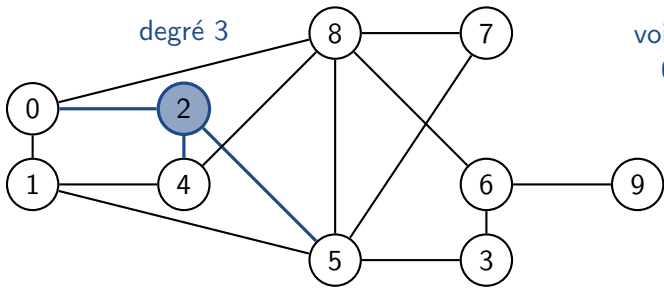
arête : ensemble de deux sommets $\{u, v\} \subset S$, notée souvent uv ou vu



Vocabulaire

Graphe **connexe** constitué de 10 **sommets** et 15 **arêtes** : $G = (\mathbf{S}, \mathbf{A})$

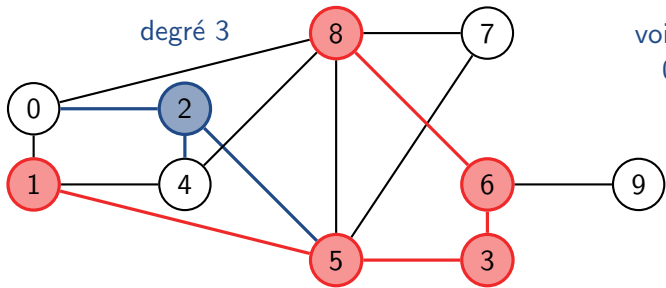
arête : ensemble de deux sommets $\{u, v\} \subset S$, notée souvent uv ou vu



Vocabulaire

Graphe **connexe** constitué de 10 **sommets** et 15 **arêtes** : $G = (\mathbf{S}, \mathbf{A})$

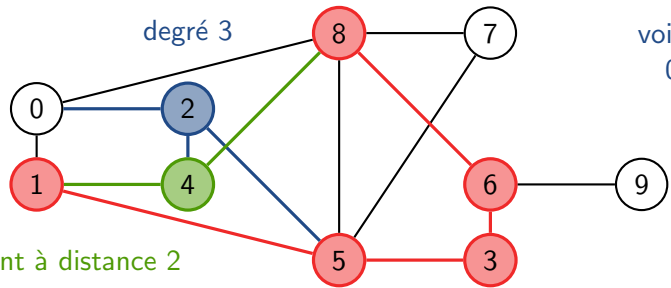
arête : ensemble de deux sommets $\{u, v\} \subset S$, notée souvent uv ou vu



Vocabulaire

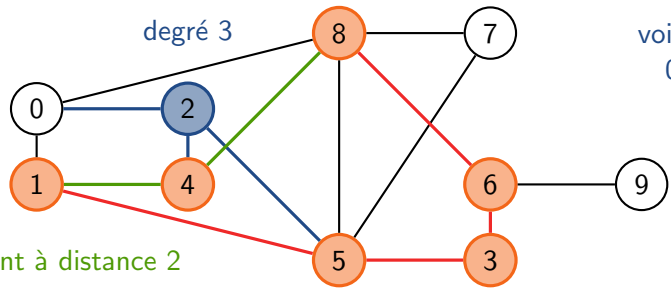
Graphe **connexe** constitué de 10 **sommets** et 15 **arêtes** : $G = (\mathbf{S}, \mathbf{A})$

arête : ensemble de deux sommets $\{u, v\} \subset S$, notée souvent uv ou vu



Vocabulaire

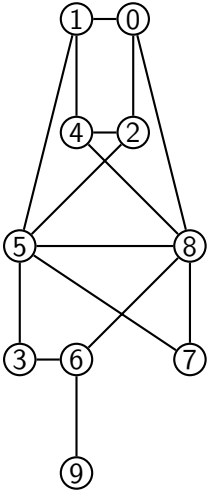
Graphe **connexe** constitué de 10 **sommets** et 15 **arêtes** : $G = (\mathbf{S}, \mathbf{A})$
arête : ensemble de deux sommets $\{u, v\} \subset S$, notée souvent uv ou vu



\+ chemin de longueur 4

= cycle de longueur 6

Représentations informatiques



Matrice d'adjacence

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Listes d'adjacence

$0 : 1 \to 2 \to 8$

$1 : 0 \to 4 \to 5$

$2 : 0 \to 4 \to 5$

$3 : 5 \to 6$

$4 : 1 \to 2 \to 8$

$5 : 1 \to 2 \to 7 \to 8$

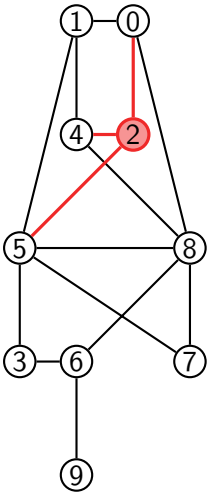
$6 : 3 \to 8 \to 9$

$7 : 5 \to 8$

$8 : 0 \to 4 \to 5 \to 6$

$9 : 6$

Représentations informatiques



Matrice d'adjacence

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

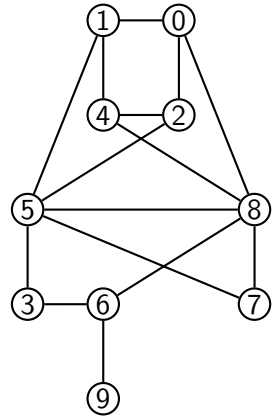
Listes d'adjacence

$$\begin{aligned} 0 &: 1 \to 2 \to 8 \\ 1 &: 0 \to 4 \to 5 \\ 2 &: 0 \to 4 \to 5 \\ 3 &: 5 \to 6 \\ 4 &: 1 \to 2 \to 8 \\ 5 &: 1 \to 2 \to 7 \to 8 \\ 6 &: 3 \to 8 \to 9 \\ 7 &: 5 \to 8 \\ 8 &: 0 \to 4 \to 5 \to 6 \\ 9 &: 6 \end{aligned}$$

Parcours en largeur

Algorithme : PARCOURSLARGEUR(x)

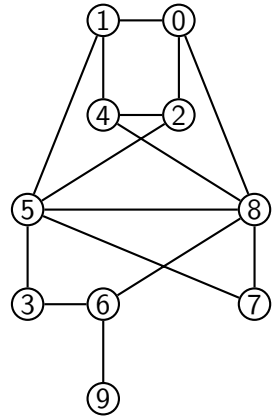
$F \leftarrow$ file vide
si $x \neq \emptyset$ **:** l'ajouter à F
tant que F *est non vide* **:**
    $y \leftarrow$ défiler un élément de F
    Afficher val(y)
    si filsG(y) $\neq \emptyset$ **:** l'ajouter à F
    si filsD(y) $\neq \emptyset$ **:** l'ajouter à F



Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

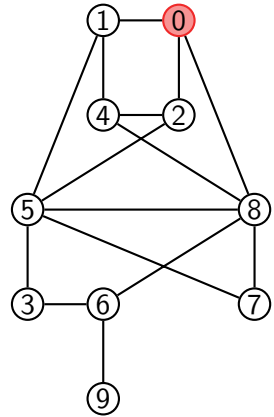
$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v



Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

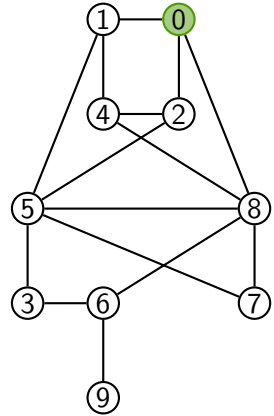


- File : 0
- Affichage :

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

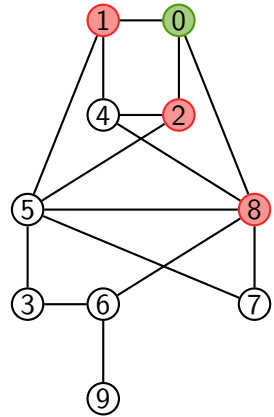


- File :
- Affichage : 0

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

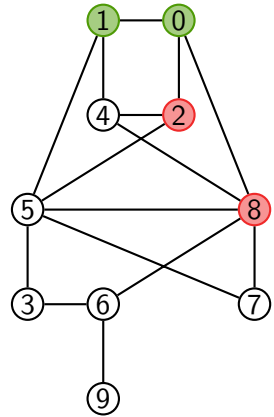


- File : 1 2 8
- Affichage : 0

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

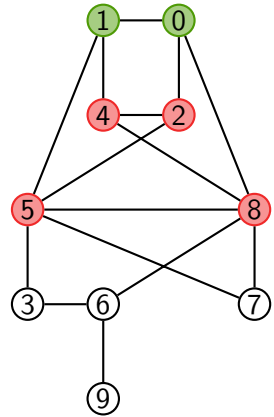


- File : 2 8
- Affichage : 0 1

Parcours en largeur

Algorithme : ParcoursLargeur(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

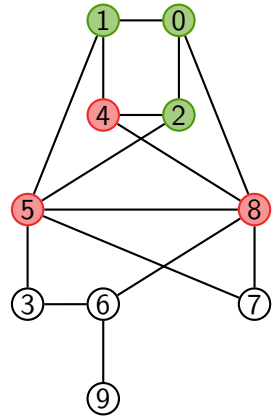


- File : 2 8 4 5
- Affichage : 0 1

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

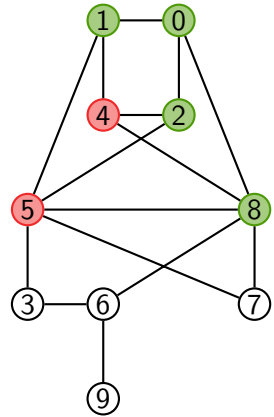


- File : 8 4 5
- Affichage : 0 1 2

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

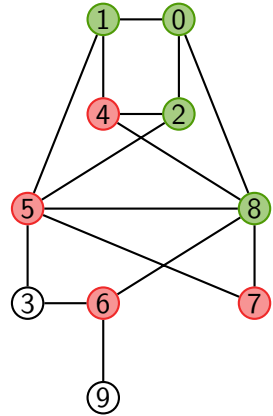


- File : 4 5
- Affichage : 0 1 2 8

Parcours en largeur

Algorithme : ParcoursLargeur(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

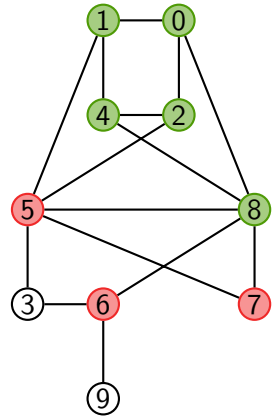


- File : 4 5 6 7
- Affichage : 0 1 2 8

Parcours en largeur

Algorithme : ParcoursLargeur(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

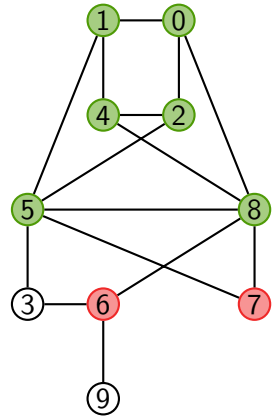


- File : 5 6 7
- Affichage : 0 1 2 8 4

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

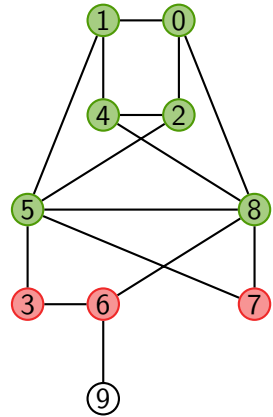


- File : 6 7
- Affichage : 0 1 2 8 4 5

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

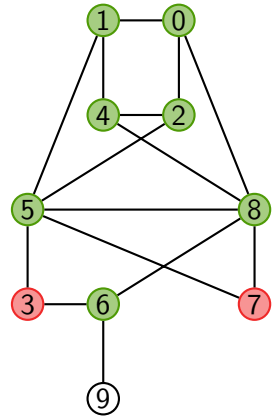


- File : 6 7 3
- Affichage : 0 1 2 8 4 5

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

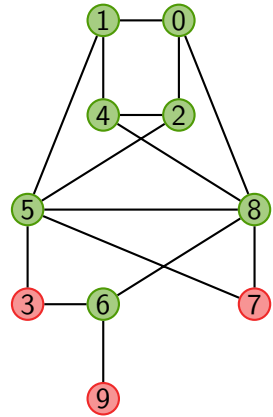


- File : 7 3
- Affichage : 0 1 2 8 4 5 6

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

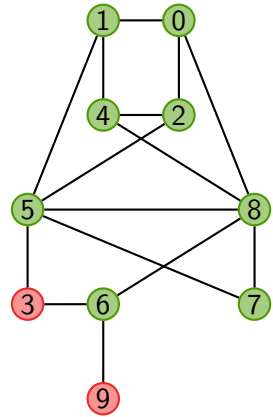


- File : 7 3 9
- Affichage : 0 1 2 8 4 5 6

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

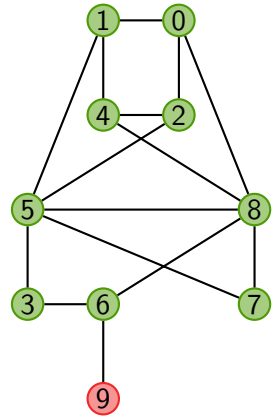


- File : 3 9
- Affichage : 0 1 2 8 4 5 6 7

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

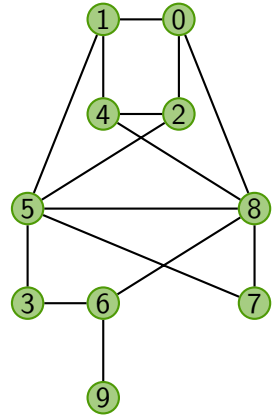


- File : 9
- Affichage : 0 1 2 8 4 5 6 7 3

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v



- File :
- Affichage : 0 1 2 8 4 5 6 7 3 9

Propriétés du parcours en largeur

Théorème

PARCOURSLARGEUR(G, s) *affiche une fois et une seule chaque sommet de la composante connexe de* s. *Sa complexité est*

- $O(n^2)$ *si le graphe est représenté par matrice d'adjacence*
- $O(m + n)$ *si le graphe est représenté par listes d'adjacence*

où n *est le nombre de sommets et* m *le nombre d'arêtes.*

Propriétés du parcours en largeur

Théorème

PARCOURSLARGEUR(G, s) *affiche une fois et une seule chaque sommet de la composante connexe de* s*. Sa complexité est*

- $O(n^2)$ *si le graphe est représenté par matrice d'adjacence*
- $O(m+n)$ *si le graphe est représenté par listes d'adjacence*

où n *est le nombre de sommets et* m *le nombre d'arêtes.*

Preuve de complexité :

- Matrice : pour chaque sommet, parcours de la ligne correspondante
- Liste : parcours de toutes les listes ; somme des longueurs $= 2m$

Propriétés du parcours en largeur

Théorème

PARCOURSLARGEUR(G, s) *affiche une fois et une seule chaque sommet de la composante connexe de* s*. Sa complexité est*

- $O(n^2)$ *si le graphe est représenté par matrice d'adjacence*
- $O(m + n)$ *si le graphe est représenté par listes d'adjacence*

où n *est le nombre de sommets et* m *le nombre d'arêtes.*

Preuve de correction :

- récurrence sur la distance à s ■

Calcul des distances

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide

Ajouter s à F et marquer s

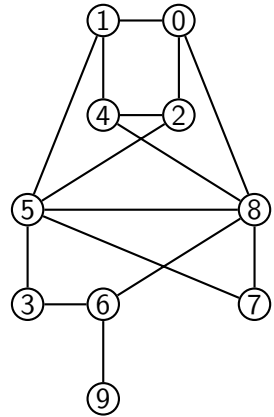
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  Afficher u

  pour *tout voisin non marqué* v *de* u **:**

    Ajouter v à F et marquer v



Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

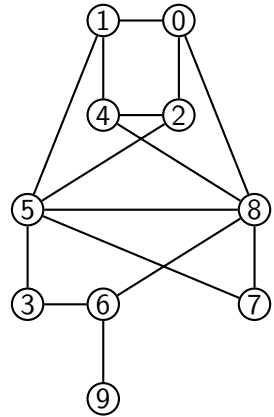
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

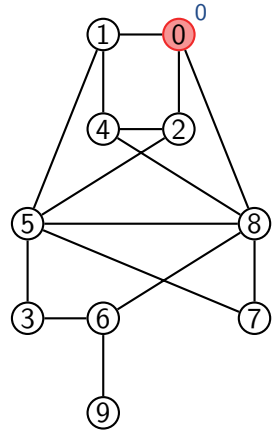
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File : 0

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

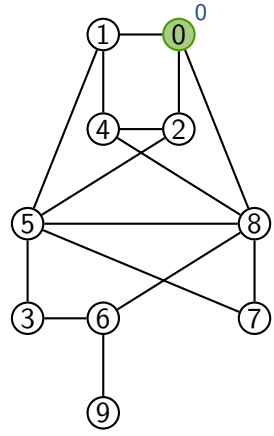
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File :

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

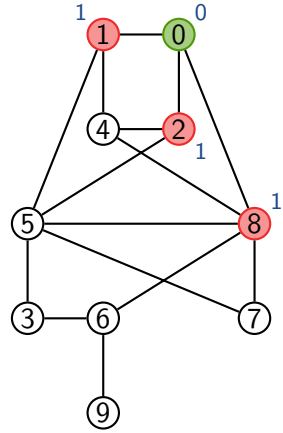
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File : 1 2 8

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

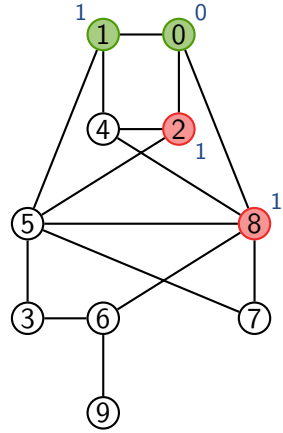
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File : 2 8

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

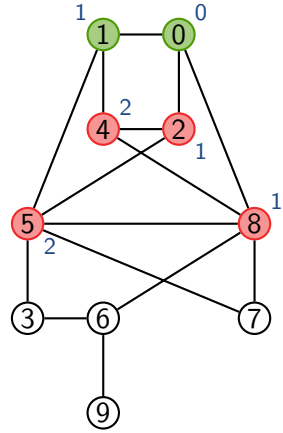
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File : 2 8 4 5

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

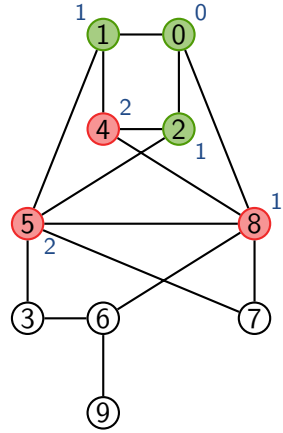
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File : 8 4 5

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

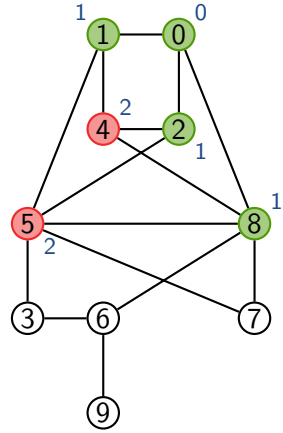
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File : 4 5

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

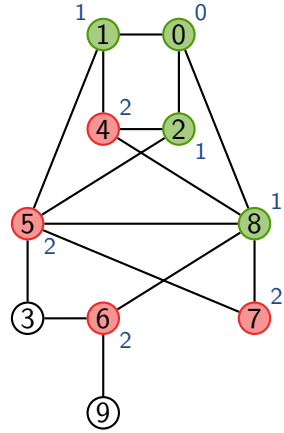
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File : 4 5 6 7

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

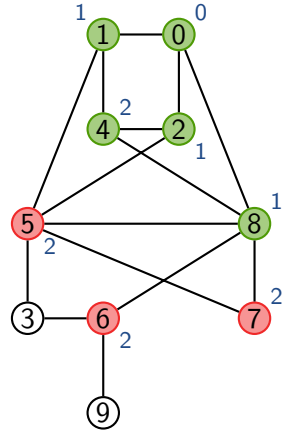
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File : 5 6 7

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

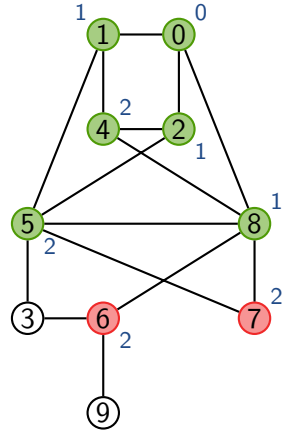
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File : 6 7

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

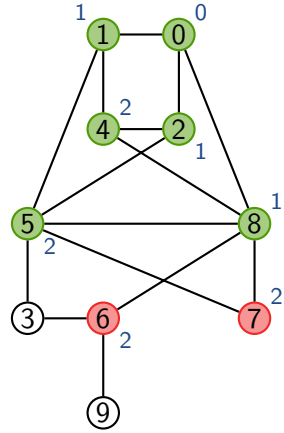
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File : 6 7 3

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

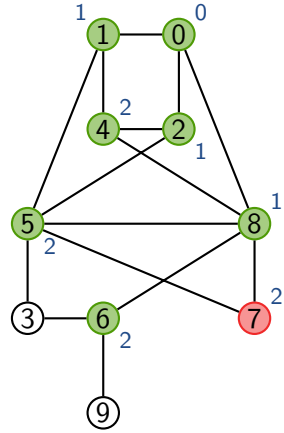
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File : 7 3

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

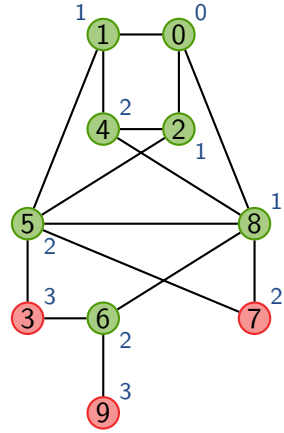
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File : 7 3 9

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

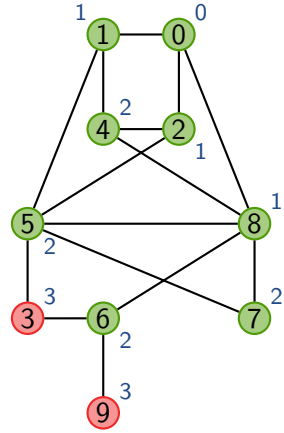
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File : 3 9

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

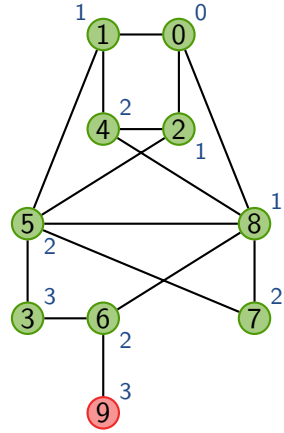
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File : 9

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

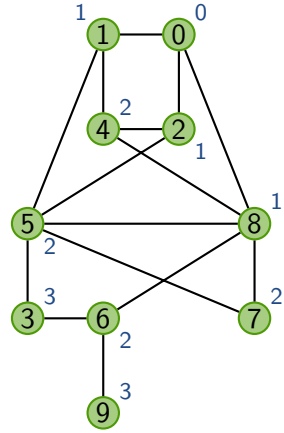
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

renvoyer D



- File :

Calcul des distances

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide ; $D_{[u]} \leftarrow +\infty$ pour tout u

Ajouter s à F ; $D_{[s]} \leftarrow 0$

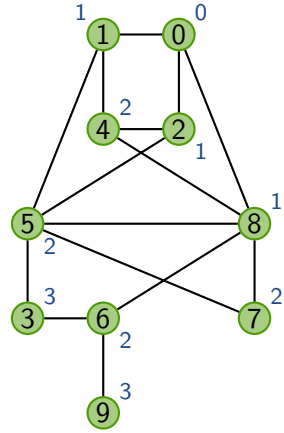
tant que F *est non vide* **:**

  $u \leftarrow$ défiler un élément de F

  pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**

    Ajouter v à F ; $D_{[v]} \leftarrow D_{[u]} + 1$

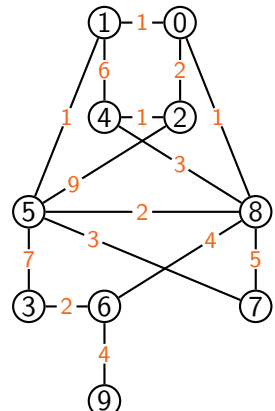
renvoyer D



- File :
- DISTANCES calcule les distances entre s et tous les autres sommets

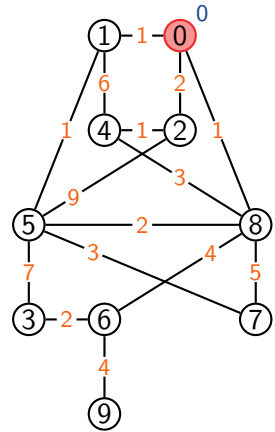
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



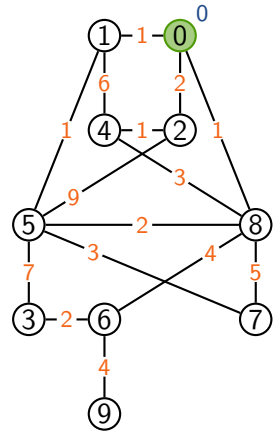
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



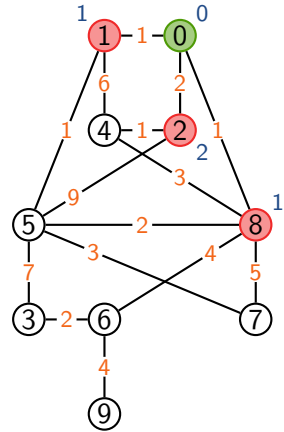
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



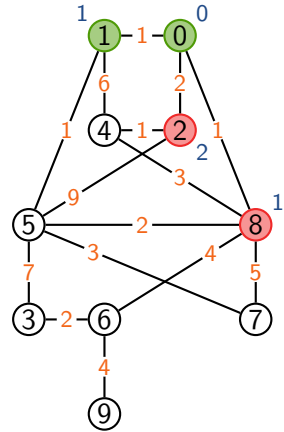
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



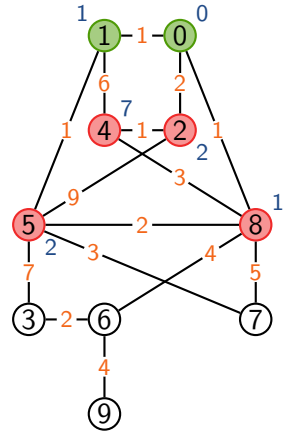
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



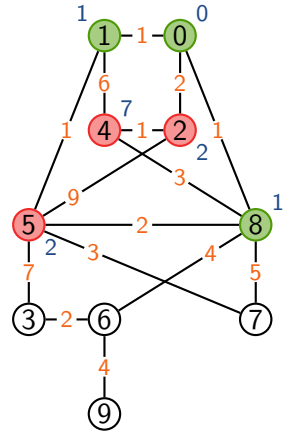
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



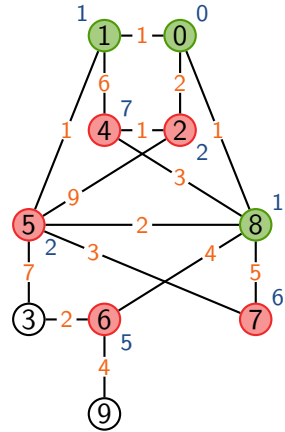
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



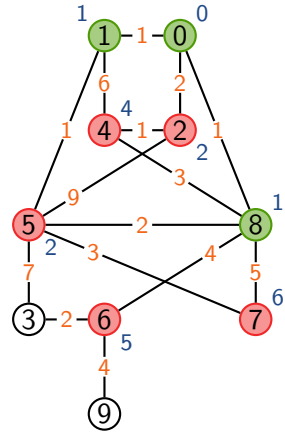
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



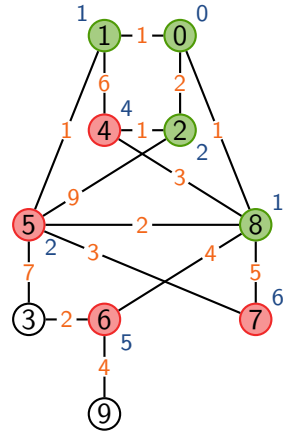
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



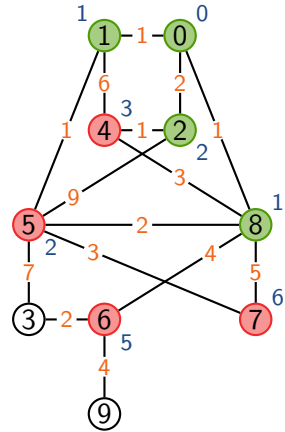
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



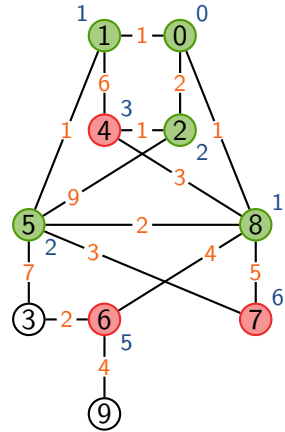
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



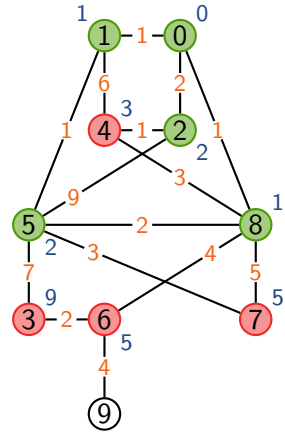
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



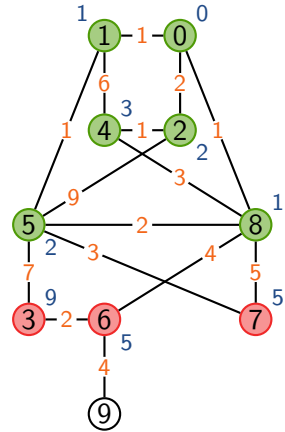
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



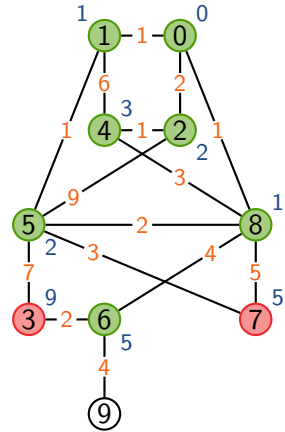
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



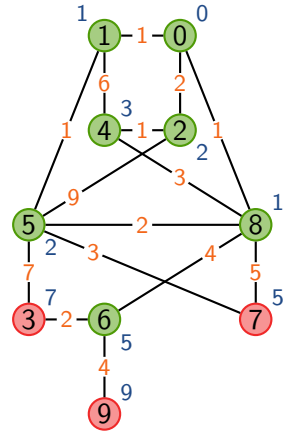
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



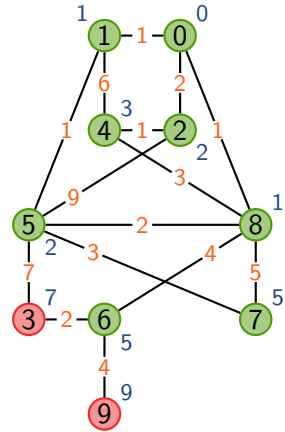
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



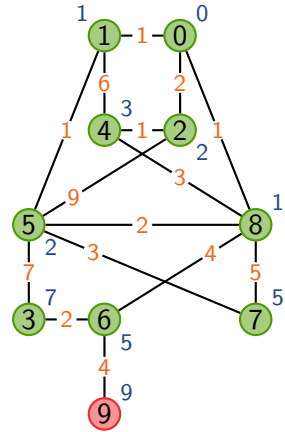
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



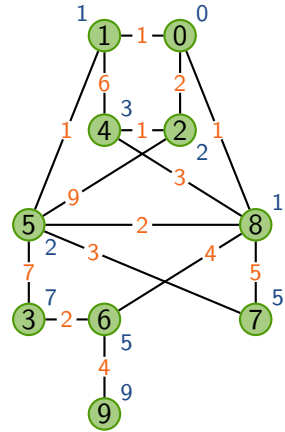
Algorithme de Dijkstra

Et si le graphe est *pondéré* ?



Algorithme de Dijkstra

Et si le graphe est *pondéré* ?

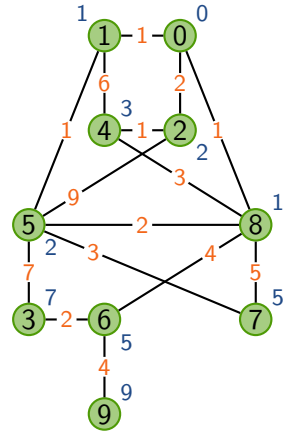


Algorithme de Dijkstra

Et si le graphe est *pondéré* ?

Algorithme : DISTANCES(G, s)

$F \leftarrow$ file vide
$D_{[u]} \leftarrow +\infty$ pour tout u
Ajouter s à F ; $D_{[s]} \leftarrow 0$
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    pour *tout voisin* v *de* u *tq* $D_{[v]} = +\infty$ **:**
        Ajouter v à F
        $D_{[v]} \leftarrow D_{[u]} + 1$
renvoyer D

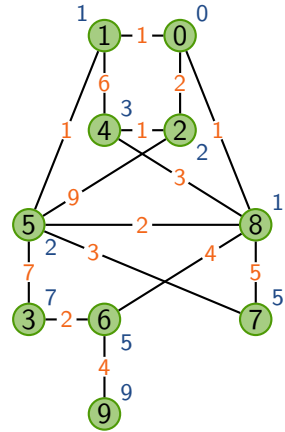


Algorithme de Dijkstra

Et si le graphe est *pondéré* ?

Algorithme : DIJKSTRA(G, s, p)

$F \leftarrow$ file de priorité vide
$D_{[u]} \leftarrow +\infty$ pour tout u
Ajouter s à F ; $D_{[s]} \leftarrow 0$
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F de distance minimale
    pour *tout voisin* v *de* u **:**
        Ajouter v à F **si** $D_{[v]} = +\infty$
        $D_{[v]} \leftarrow \min(D_{[v]}, D_{[u]} + p(u, v))$
renvoyer D



L'algorithme en détail

Algorithme : DIJKSTRA(G, s, p)

$F \leftarrow$ file de priorité vide
$D \leftarrow$ tableau de n entiers, initialisé à $+\infty$ `// Tableau des distances`
pour *chaque sommet u de G* **:** AJOUTER$(F, u, +\infty)$ `// Insertion dans la file`
CHANGERPRIORITÉ$(F, s, 0)$; $D_{[s]} \leftarrow 0$ `// Distance 0 pour s`

tant que *F est non vide* **:**
    $u \leftarrow$ EXTRAIREMIN(F) `// Élément de distance minimale`
    pour *tout voisin v de u* **:**
        si $D_{[u]} + p(u, v) < D_{[v]}$ **:**
            $D_{[v]} \leftarrow D_{[u]} + p(u, v)$ `// Mise à jour de la distance`
            CHANGERPRIORITÉ$(F, v, D_{[v]})$ `// Mise à jour de la file`

renvoyer D

Propriétés de l'algorithme de Dijkstra

Théorème

Si G est connexe, DIJKSTRA(G, s, p) *calcule les longueurs des plus courts chemins de s à chaque sommet de G. Sa complexité est*

- $O(m \log n)$ *avec des listes d'adjacence*
- $O(n^2)$ *avec une matrice d'adjacence*

où n est le nombre de sommets, m le nombre d'arêtes.

Propriétés de l'algorithme de Dijkstra

Théorème

Si G est connexe, DIJKSTRA(G, s, p) *calcule les longueurs des plus courts chemins de s à chaque sommet de G. Sa complexité est*

- $O(m \log n)$ *avec des listes d'adjacence*
- $O(n^2)$ *avec une matrice d'adjacence*

où n est le nombre de sommets, m le nombre d'arêtes.

Preuve de complexité

- Listes d'adjacence :
  - Liste de priorité : AJOUTER/EXTRAIREMIN/CHANGERPRIORITÉ en $O(\log n)$
  - Chaque sommet est extrait une fois $\rightsquigarrow O(n \log n)$
  - Chaque arête *peut* déclencher un CHANGERPRIORITÉ $\rightsquigarrow O(m \log n)$

Propriétés de l'algorithme de Dijkstra

Théorème

Si G est connexe, DIJKSTRA(G, s, p) *calcule les longueurs des plus courts chemins de s à chaque sommet de G. Sa complexité est*

- $O(m \log n)$ *avec des listes d'adjacence*
- $O(n^2)$ *avec une matrice d'adjacence*

où n est le nombre de sommets, m le nombre d'arêtes.

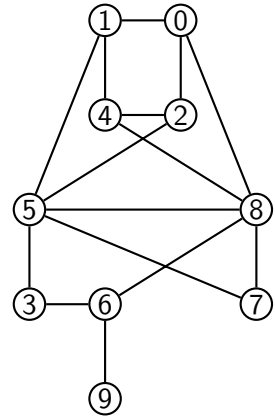
Preuve de complexité

- Listes d'adjacence :
  - Liste de priorité : AJOUTER/EXTRAIREMIN/CHANGERPRIORITÉ en $O(\log n)$
  - Chaque sommet est extrait une fois $\rightsquigarrow O(n \log n)$
  - Chaque arête *peut* déclencher un CHANGERPRIORITÉ $\rightsquigarrow O(m \log n)$
- Matrices d'adjacence
  - Pas de liste de priorité : parcours de tous les sommets à chaque fois
  - $\ll$ pour tout voisin v de $u \gg \rightsquigarrow n \times O(n)$

Parcours en profondeur

Algorithme : PARCOURSLARGEUR(G, s)

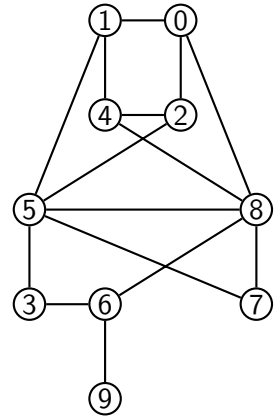
$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v



Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

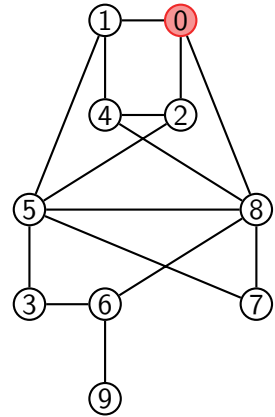
$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v



Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

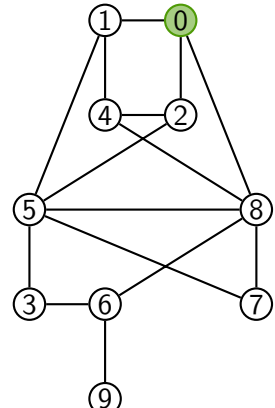


- Pile : 0
- Affichage :

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

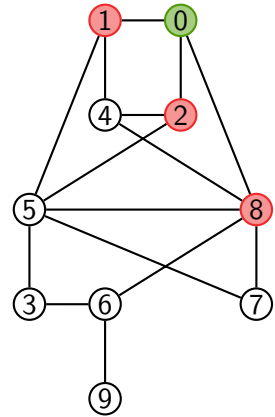


- Pile :
- Affichage : 0

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

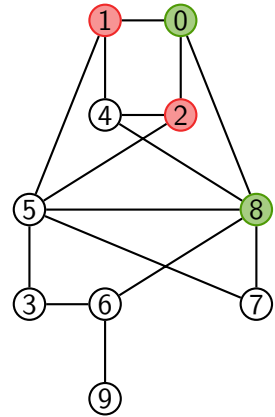


- Pile : 1 2 8
- Affichage : 0

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

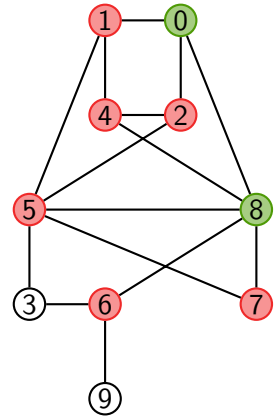


- Pile : 1 2
- Affichage : 0 8

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

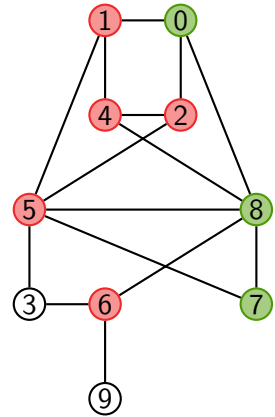


- Pile : 1 2 4 5 6 7
- Affichage : 0 8

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

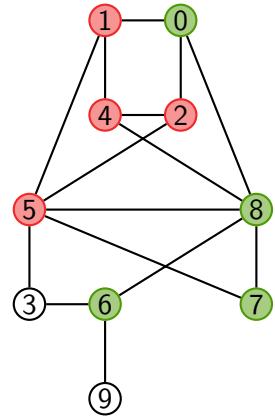


- Pile : 1 2 4 5 6
- Affichage : 0 8 7

Parcours en profondeur

Algorithme : ParcoursProfondeur(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

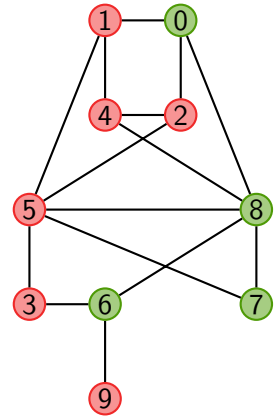


- Pile : 1 2 4 5
- Affichage : 0 8 7 6

Parcours en profondeur

Algorithme : ParcoursProfondeur(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

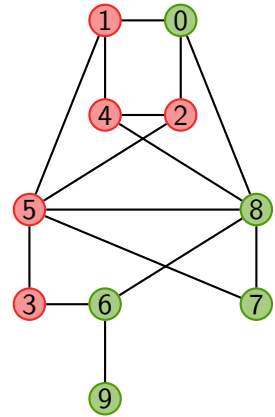


- Pile : 1 2 4 5 3 9
- Affichage : 0 8 7 6

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

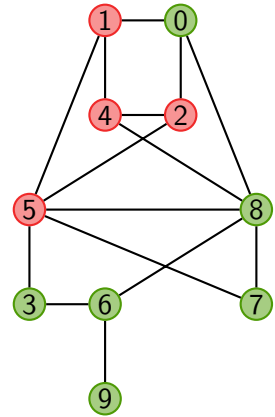


- Pile : 1 2 4 5 3
- Affichage : 0 8 7 6 9

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

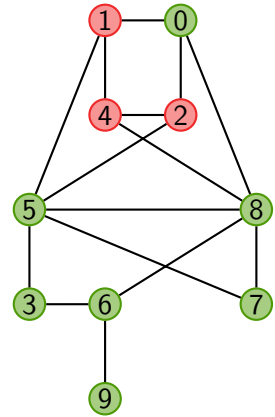


- Pile : 1 2 4 5
- Affichage : 0 8 7 6 9 3

Parcours en profondeur

Algorithme : ParcoursProfondeur(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

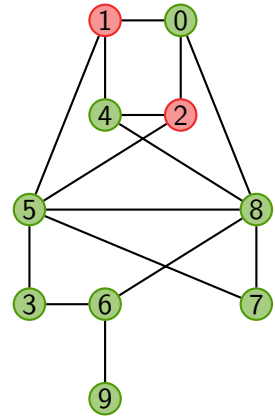


- Pile : 1 2 4
- Affichage : 0 8 7 6 9 3 5

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

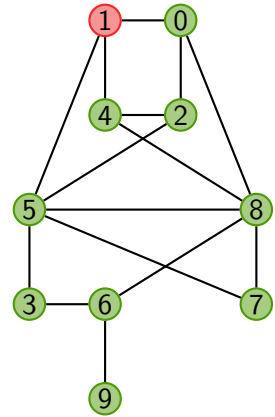


- Pile : 1 2
- Affichage : 0 8 7 6 9 3 5 4

Parcours en profondeur

Algorithme : ParcoursProfondeur(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

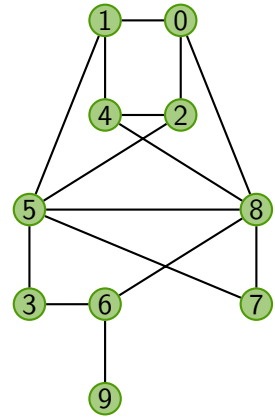


- Pile : 1
- Affichage : 0 8 7 6 9 3 5 4 2

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v



- Pile :
- Affichage : 0 8 7 6 9 3 5 4 2 1

Propriétés du parcours en profondeur

Théorème

PARCOURSPROFONDEUR(G, s) *affiche une fois et une seule chaque sommet de la composante connexe de* s*. Sa complexité est*

- $O(n^2)$ *si le graphe est représenté par matrice d'adjacence*
- $O(m + n)$ *si le graphe est représenté par listes d'adjacence*

où n *est le nombre de sommets et* m *le nombre d'arêtes.*

Propriétés du parcours en profondeur

Théorème

PARCOURSPROFONDEUR(G, s) *affiche une fois et une seule chaque sommet de la composante connexe de* s*. Sa complexité est*

- $O(n^2)$ *si le graphe est représenté par matrice d'adjacence*
- $O(m + n)$ *si le graphe est représenté par listes d'adjacence*

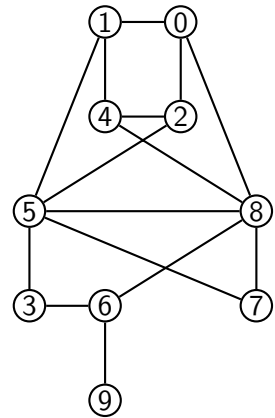
où n *est le nombre de sommets et* m *le nombre d'arêtes.*

Preuve : Identique au cas du parcours en largeur !

Détection de cycles

Algorithme : PARCOURSPROFONDEUR(G, s)

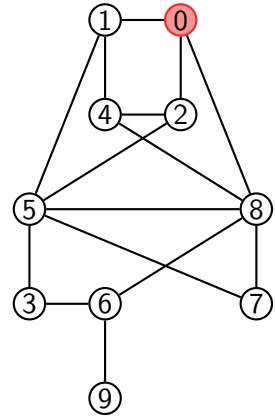
$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v



Détection de cycles

Algorithme : PARCOURSPROFONDEUR(G, s)

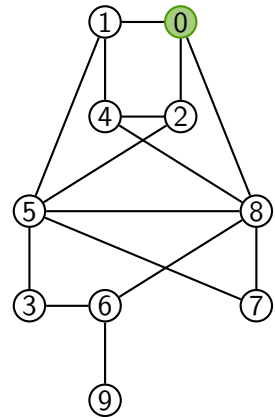
$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v



Détection de cycles

Algorithme : PARCOURSPROFONDEUR(G, s)

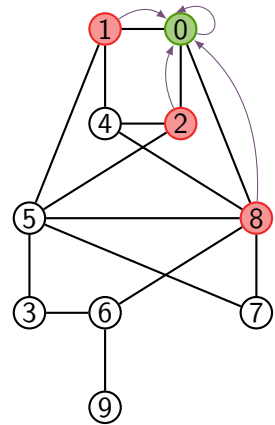
$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v



Détection de cycles

Algorithme : PARCOURSPROFONDEUR(G, s)

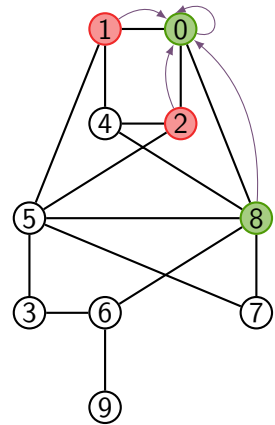
$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v



Détection de cycles

Algorithme : ParcoursProfondeur(G, s)

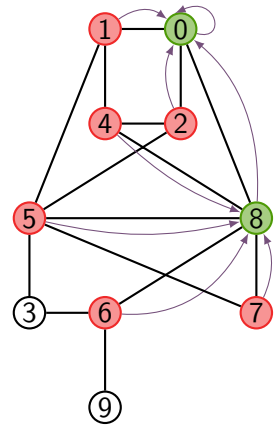
$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v



Détection de cycles

Algorithme : ParcoursProfondeur(G, s)

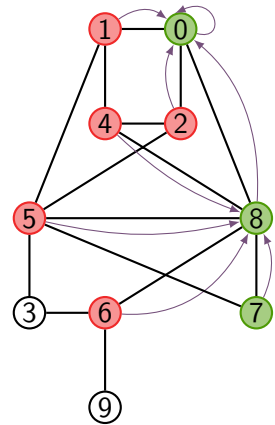
$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v



Détection de cycles

Algorithme : PARCOURSPROFONDEUR(G, s)

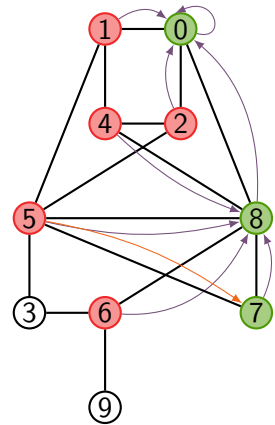
$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v



Détection de cycles

Algorithme : ParcoursProfondeur(G, s)

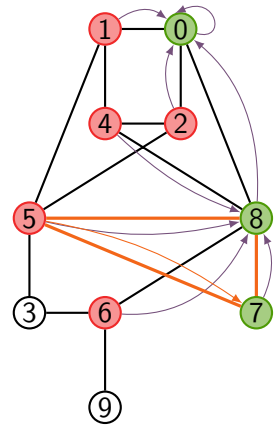
$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v



Détection de cycles

Algorithme : ParcoursProfondeur(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v



Détection de cycles

Algorithme : DÉTECTIONCYCLES(G)

$P \leftarrow$ pile vide ; $\text{Pred}_u \leftarrow -1$ pour tout u

Ajouter 0 à P ; $\text{Pred}_0 \leftarrow 0$

tant que P *est non vide* **:**

  $u \leftarrow$ dépiler un élément de P

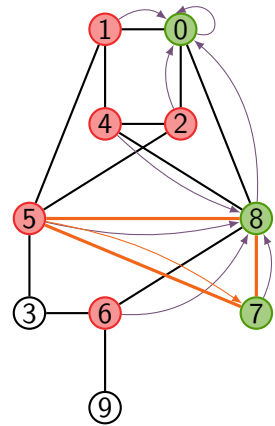
  pour *tout voisin* v *de* u **:**

    si $\text{Pred}_v = -1$ **:**

      Ajouter v à P ; $\text{Pred}_v \leftarrow u$

    sinon si $\text{Pred}_u \neq v$ **: Renvoyer** VRAI

Renvoyer FAUX



Propriétés de la détection de cycles

Théorème

Si G est connexe, DétectionCycles(G) *renvoie* Vrai *si et seulement s'il existe un cycle. Sa complexité est*

- $O(n^2)$ *si le graphe est représenté par matrice d'adjacence*
- $O(n)$ *si le graphe est représenté par listes d'adjacence*

où n est le nombre de sommets.

Propriétés de la détection de cycles

Théorème

Si G est connexe, DétectionCycles(G) *renvoie* Vrai *si et seulement s'il existe un cycle. Sa complexité est*

- $O(n^2)$ *si le graphe est représenté par matrice d'adjacence*
- $O(n)$ *si le graphe est représenté par listes d'adjacence*

où n est le nombre de sommets.

Preuve de complexité : différent des parcours !

On ne peut pas visiter plus de n arêtes, car un graphe à n sommets ayant $\geq n$ arêtes possède forcément un cycle.

Propriétés de la détection de cycles

Théorème

Si G est connexe, DétectionCycles(G) *renvoie* Vrai *si et seulement s'il existe un cycle. Sa complexité est*

- $O(n^2)$ *si le graphe est représenté par matrice d'adjacence*
- $O(n)$ *si le graphe est représenté par listes d'adjacence*

où n est le nombre de sommets.

Preuve de correction :

Conclusion

Les parcours de graphes servent à tout !

Conclusion

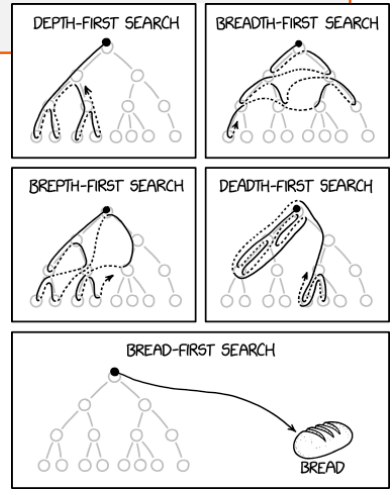
Les parcours de graphes servent à tout !

- Trois types parcours :
  - en largeur – avec une file
  - en profondeur – avec une pile
  - *par priorité* – avec une file de priorité DIJKSTRA

Conclusion

Les parcours de graphes servent à tout !

- Trois types parcours :
  - en largeur – avec une file
  - en profondeur – avec une pile
  - *par priorité* – avec une file de priorité



https://xkcd.com/2407

Conclusion

Les parcours de graphes servent à tout !

- Trois types parcours :
  - en largeur – avec une file
  - en profondeur – avec une pile
  - *par priorité* – avec une file de priorité DIJKSTRA
- Complexité similaires :
  - $O(m+n)$ si listes d'adjacence
    (sauf $O((m+n)\log n)$ pour le parcours par priorité)
  - $O(n^2)$ si matrice d'adjacence

Conclusion

Les parcours de graphes servent à tout !

- Trois types parcours :
  - en largeur – avec une file
  - en profondeur – avec une pile
  - *par priorité* – avec une file de priorité DIJKSTRA
- Complexité similaires :
  - $O(m+n)$ si listes d'adjacence
    (sauf $O((m+n)\log n)$ pour le parcours par priorité)
  - $O(n^2)$ si matrice d'adjacence
- Nombreuses applications :
  - calculs de distances et de chemins
  - détection de cycle, arbre couvrant
  - composantes (fortement) connexes

Conclusion

Les parcours de graphes servent à tout !

- Trois types parcours :
  - en largeur – avec une file
  - en profondeur – avec une pile
  - *par priorité* – avec une file de priorité DIJKSTRA
- Complexité similaires :
  - $O(m+n)$ si listes d'adjacence
    (sauf $O((m+n)\log n)$ pour le parcours par priorité)
  - $O(n^2)$ si matrice d'adjacence
- Nombreuses applications :
  - calculs de distances et de chemins
  - détection de cycle, arbre couvrant
  - composantes (fortement) connexes
- Pour aller plus loin : algorithmes de flots et coupes