Algorithmique et Complexité
Cours 1/7 : Parcours de graphes

CentraleSupélec – Gif

ST2 – Gif

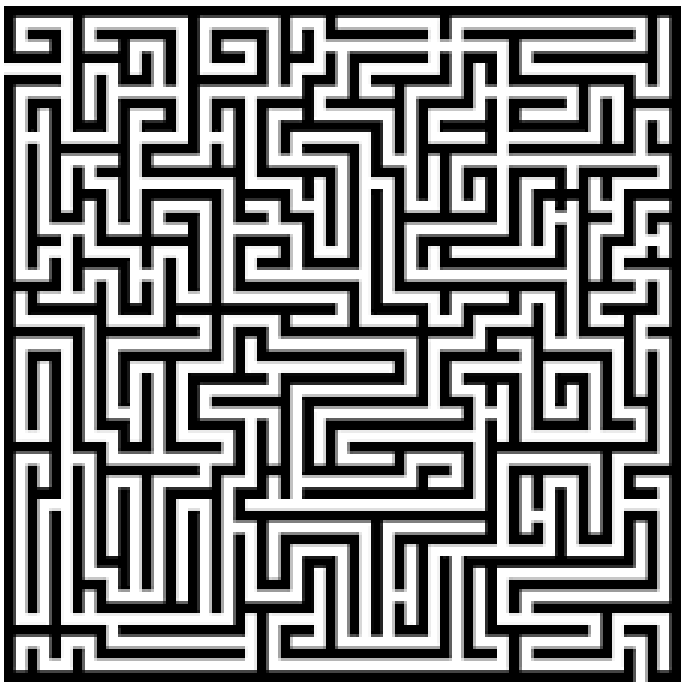
Plan

1. Problèmes de graphes
2. Parcours en profondeur
3. Parcours en largeur
4. Complexité
5. Connexité
6. Conclusion

Plan

1. Problèmes de graphes
   - Problèmes concrets
   - Modélisation par les graphes
   - Familles de problèmes
   - Algorithme de résolution
2. Parcours en profondeur
3. Parcours en largeur
4. Complexité
5. Connexité
6. Conclusion

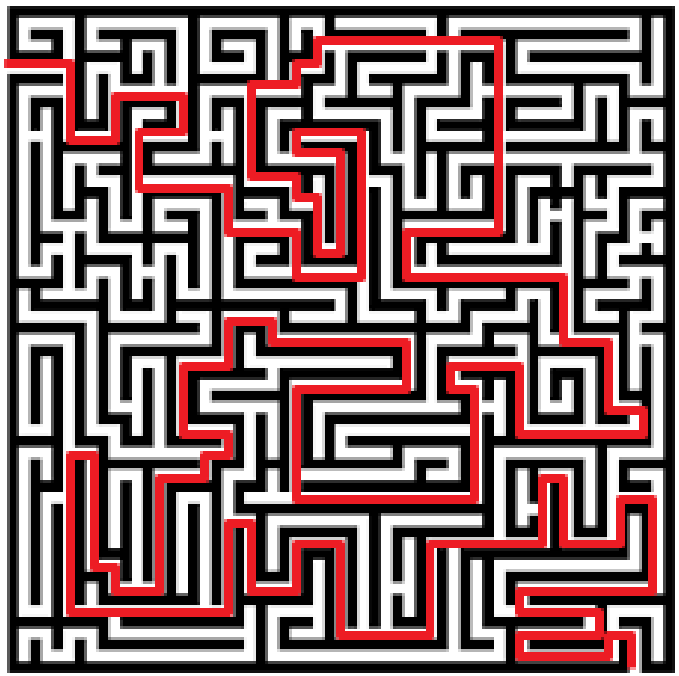
Trouver la sortie d'un labyrinthe



Problèmes

- Comment représenter informatiquement le problème du labyrinthe ?
- Qu'est-ce qui caractérise une solution à ce problème ?

Trouver la sortie d'un labyrinthe



Problèmes

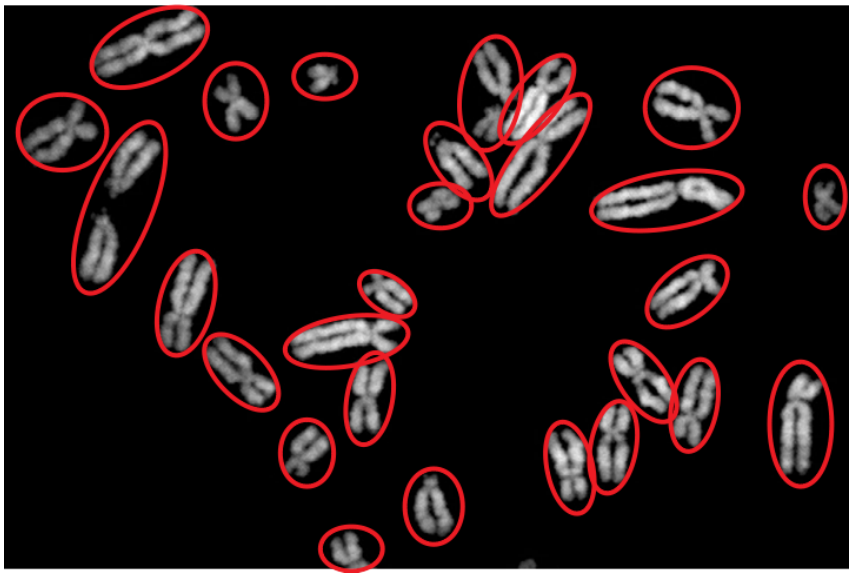
- Comment représenter informatiquement le problème du labyrinthe ?
- Qu'est-ce qui caractérise une solution à ce problème ?
- Comment calculer efficacement une solution ?

Identifier des éléments dans une image

Problèmes

- Comment représenter informatiquement cette image ?
- Qu'est-ce qui caractérise un chromosome ?

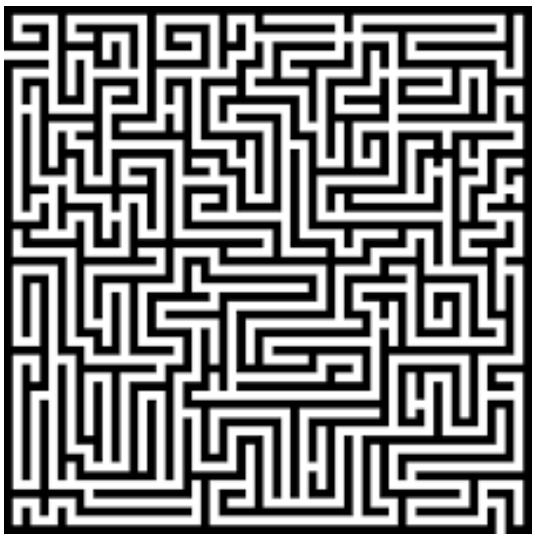
Identifier des éléments dans une image



Problèmes

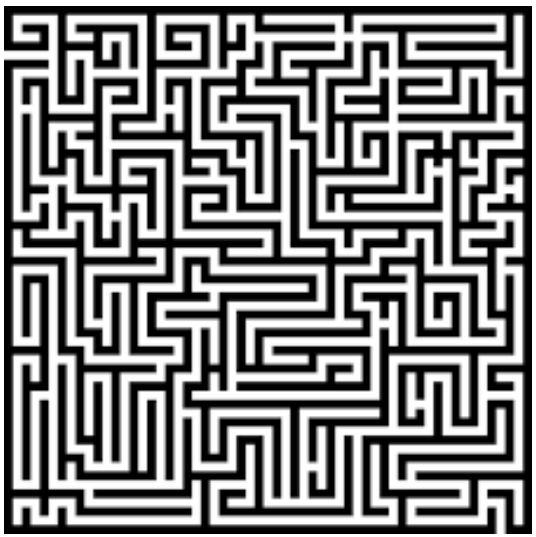
- Comment représenter informatiquement cette image ?
- Qu'est-ce qui caractérise un chromosome ?
- Comment calculer efficacement quelles zones de l'image représentent des chromosomes ?

Labyrinthes et chromosomes



Quel est le lien entre trouver son chemin dans un labyrinthe et compter des chromosomes ?

Labyrinthes et chromosomes



Quel est le lien entre trouver son chemin dans un labyrinthe et compter des chromosomes ?

➜ Les graphes, les parcours de graphes et la connexité !

Graphe

Structure de données

Vous avez vu en CPGE (ou en SG1) :

- ✓ Les variables (souvent associées à un type de représentation)
- ✓ Les tableaux à 1 ou plusieurs dimensions
- ✓ Les listes, les piles et les files
- ✗ Les objets
- ✗ Les dictionnaires

Graphe

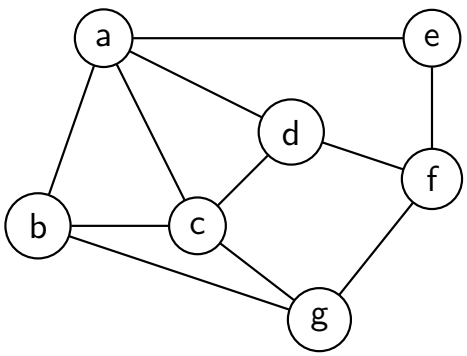
Les graphes sont une autre structure de données

➜ Probablement la plus utilisée en algorithmique !

Notion de graphe

Graphe

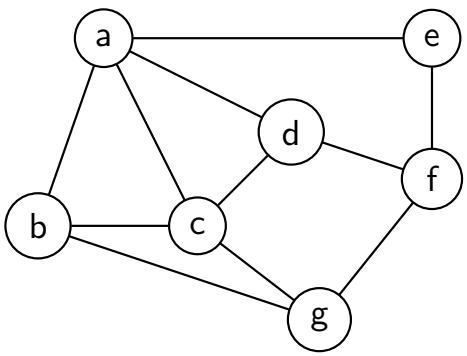
Structure mathématique utilisée pour représenter des relations entre des éléments



Notion de graphe

Graphe

Structure mathématique utilisée pour représenter des relations entre des éléments



Définitions

- Nœuds aussi appelés sommets

  En anglais : node, nodes, vertex, vertices

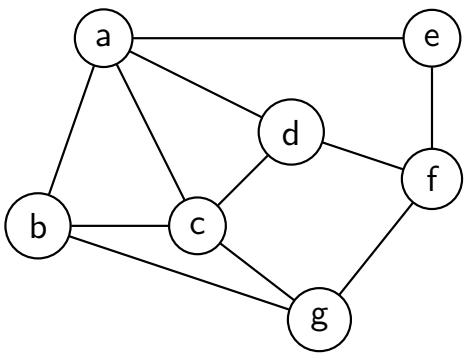
- Arêtes

  En anglais : edge, edges

Notion de graphe

Graphe

Structure mathématique utilisée pour représenter des relations entre des éléments



Définitions

- Nœuds aussi appelés sommets

  En anglais : node, nodes, vertex, vertices

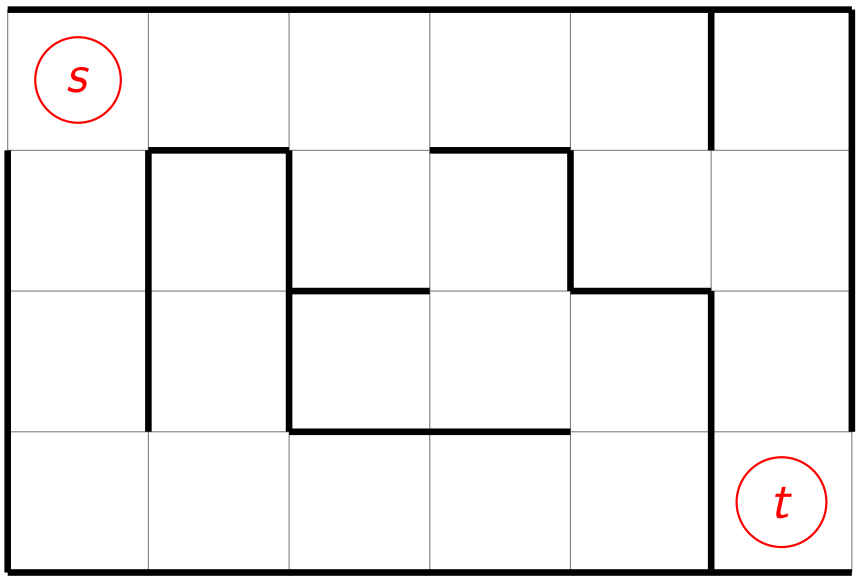
- Arêtes

  En anglais : edge, edges

Notation

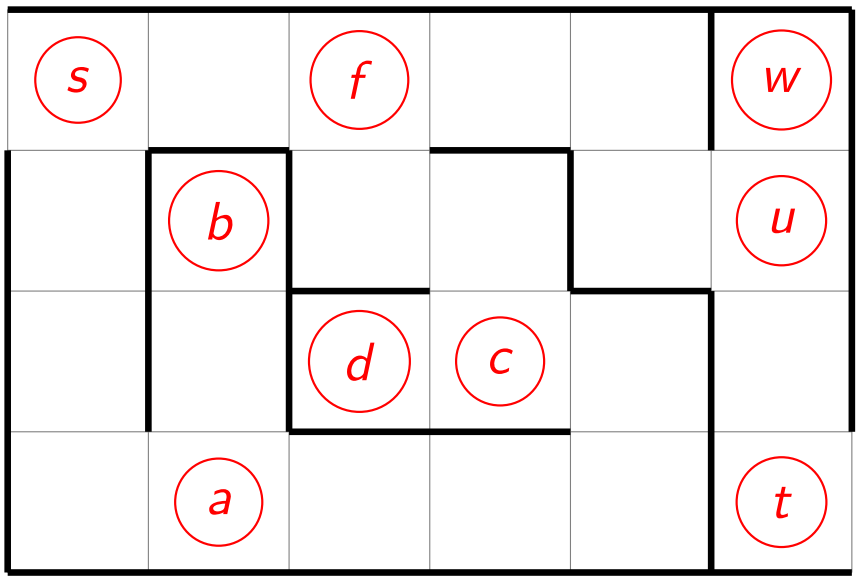
Graphe $G = (V, E)$
où V est l'ensemble des sommets et E l'ensemble des arêtes.

Représentation des labyrinthes



Un labyrinthe vu comme un graphe

Représentation des labyrinthes

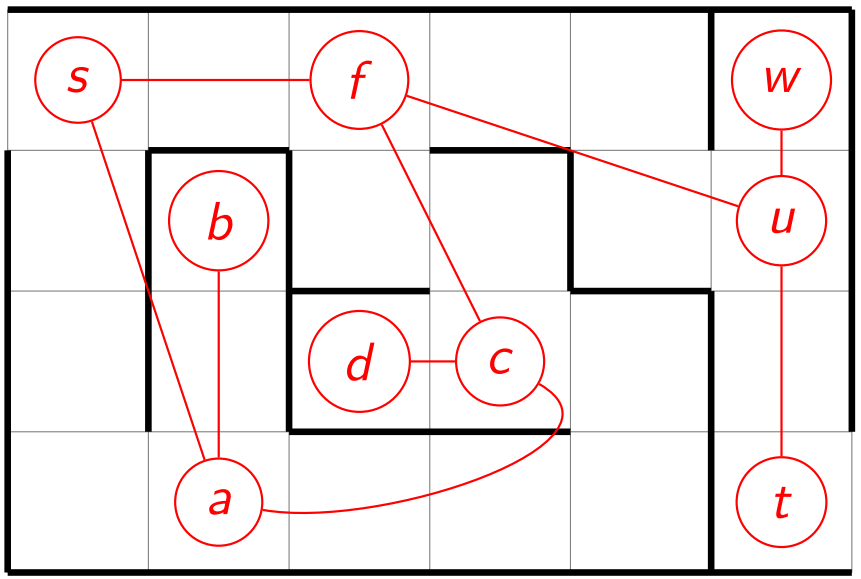


Un labyrinthe vu comme un graphe

- Les intersections et les culs-de-sac sont représentés par des sommets ;

  On peut associer à chaque sommet une étiquette.

Représentation des labyrinthes



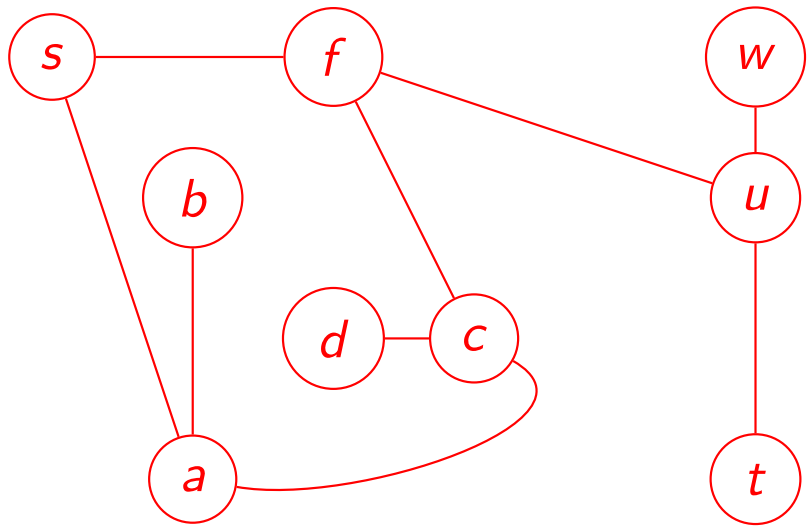
Un labyrinthe vu comme un graphe

- Les intersections et les culs-de-sac sont représentés par des sommets ;

  On peut associer à chaque sommet une étiquette.

- Chaque couloir est représenté par une arête.

Représentation des labyrinthes



Un labyrinthe vu comme un graphe

- Les intersections et les culs-de-sac sont représentés par des sommets ;

  On peut associer à chaque sommet une étiquette.

- Chaque couloir est représenté par une arête.

Remarques & définitions

Remarques

- Ce type de graphe est un graphe non-orienté.

  Nous verrons des graphes orientés au cours 2

- Chaque arête est caractérisée par ses deux sommets extrémités : $E \subseteq V \times V$

Remarques & définitions

Remarques

- Ce type de graphe est un graphe non-orienté.

  Nous verrons des graphes orientés au cours 2

- Chaque arête est caractérisée par ses deux sommets extrémités : $E \subseteq V \times V$

Définitions

- Une chaîne de x à y est une suite finie d'arêtes consécutives reliant x à y.
- On dit qu'un graphe est connexe lorsqu'il existe une chaîne entre toute paire de sommets.

Formalisation du problème dans les labyrinthes

À partir des mêmes données,

(ici un graphe $G = (V, E)$ et deux sommets s et t)

on peut construire plusieurs types de problèmes !

- Problème de décision
- Problème de construction
- Problème d'optimisation

Formalisation du problème dans les labyrinthes – Décision

Existence d'une chaîne

- Données : Étant donné un graphe $G = (V, E)$, un sommet de départ $s \in V$ et un sommet d'arrivée $t \in V$
- Question : Existe-t-il une chaîne de s à t ?

Problème de décision

✓ La réponse à la question posée ici est oui ou non.

Formalisation du problème dans les labyrinthes – Construction

Construction d'une chaîne

- Données : Étant donné un graphe $G = (V, E)$, un sommet de départ $s \in V$ et un sommet d'arrivée $t \in V$
- Question : Construire une chaîne s à t

Problème de construction

✓ La réponse est une solution au problème.

➜ Calculer une structure de données répondant aux exigences du problème.

- Il est possible qu'une telle structure n'existe pas !

➜ Le problème de décision associé répond non

Formalisation du problème dans les labyrinthes – Optimisation

Plus courte chaîne

- Données : Étant donné un graphe $G = (V, E)$, un sommet de départ $s \in V$ et un sommet d'arrivée $t \in V$
- Question : Quelle est la plus courte chaîne de s à t ?

Problème d'optimisation

✓ La réponse à la question posée ici est une solution.

✓ Il existe une fonction (ici la longueur de la chaîne) qu'on cherche à maximiser ou à minimiser.

Formalisation du problème dans les labyrinthes – Optimisation

Plus courte chaîne

- Données : Étant donné un graphe $G = (V, E)$, un sommet de départ $s \in V$ et un sommet d'arrivée $t \in V$
- Question : Quelle est la plus courte chaîne de s à t ?

Problème d'optimisation

✓ La réponse à la question posée ici est une solution.

✓ Il existe une fonction (ici la longueur de la chaîne) qu'on cherche à maximiser ou à minimiser.

Pour le cours 1, nous ne regarderons que les problèmes de décision. (existence d'une chaîne)

Instance de problème

Définition : instance

- Une instance est un ensemble de données d'entrée qui satisfont les contraintes imposées par un problème.
- La taille d'une instance correspond à la taille des données :
  - ➜ Dépend de la représentation ;
  - ➜ Niveau d'atomicité : *nombre d'éléments dans une liste, nombre de nœuds et d'arêtes dans un graphe...*

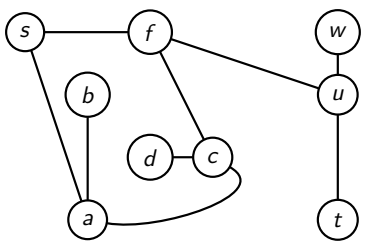
Instance de problème

Définition : instance

- Une instance est un ensemble de données d'entrée qui satisfont les contraintes imposées par un problème.
- La taille d'une instance correspond à la taille des données :
  - ➜ Dépend de la représentation ;
  - ➜ Niveau d'atomicité : *nombre d'éléments dans une liste, nombre de nœuds et d'arêtes dans un graphe...*

Exemple

- Un graphe $G = (V, E)$ et deux sommets s et $t \in V$.



- Taille de l'instance $= |V| + |E|$

Algorithme de résolution

Définition : algorithme

Un algorithme est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre un problème.

Exemple : existence d'une chaîne

1. En partant de s, choisir un sommet relié au sommet courant
2. Tant que le sommet n'est pas t, recommencer
3. Répondre "oui" quand on arrive à t

Algorithme de résolution

Définition : algorithme

Un algorithme est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre un problème.

Exemple : existence d'une chaîne

1. En partant de s, choisir un sommet relié au sommet courant
2. Tant que le sommet n'est pas t, recommencer
3. Répondre "oui" quand on arrive à t

Remarques

- Cet algorithme ne répond jamais non ;
- Cet algorithme peut ne pas se terminer.

  $\rightarrow$ On ne peut pas toujours dire si un algorithme se termine

Algorithme de résolution

Algorithme

En informatique, un algorithme est une suite finie d'instructions utilisant :

- des variables, des structures de données,
- des instructions de contrôle (boucles, instruction conditionnelle, appel de fonction, *etc*).

et qui peuvent être exécutées, pas à pas, par une machine déterministe.

Algorithme de résolution

Algorithme

En informatique, un algorithme est une suite finie d'instructions utilisant :

- des variables, des structures de données,
- des instructions de contrôle (boucles, instruction conditionnelle, appel de fonction, *etc*).

et qui peuvent être exécutées, pas à pas, par une machine déterministe.

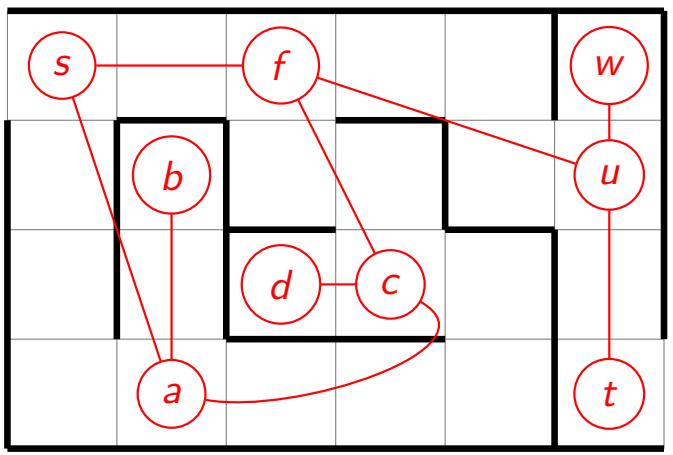
Peut-on écrire l'algorithme précédent en Python ?

`def exists_chain(V,E,s,t): ...` (à continuer)

On suppose qu'il existe une fonction `neighbours(x,E)` qui renvoie la liste des sommets voisins

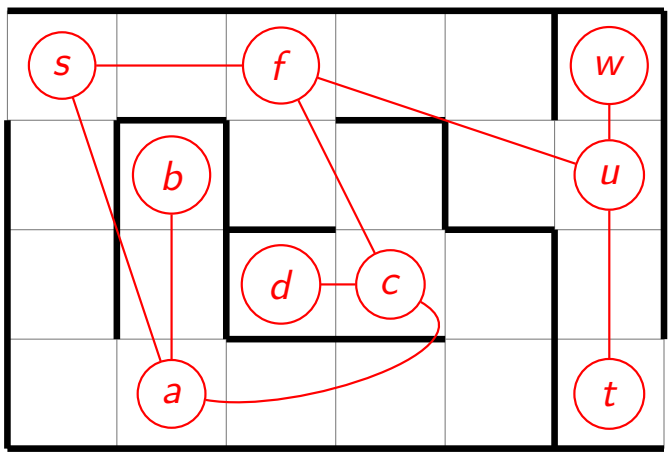
Revenons aux labyrinthes

Trouver la sortie = existence d'une chaîne !



Revenons aux labyrinthes

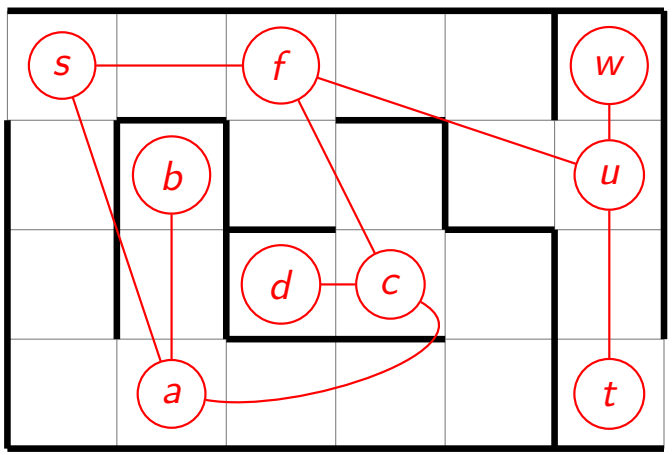
Trouver la sortie = existence d'une chaîne !



→ Algorithmes de parcours de graphe.

Revenons aux labyrinthes

Trouver la sortie = existence d'une chaîne !



→ Algorithmes de parcours de graphe.

Attention aux cycles !

➜ Noter les sommets déjà visités pour éviter de "boucler".
(contrairement à l'algorithme précédent...)

Plan

1 Problèmes de graphes

2 Parcours en profondeur
- Principe
- Implémentation récursive
- Implémentation itérative

3 Parcours en largeur

4 Complexité

5 Connexité

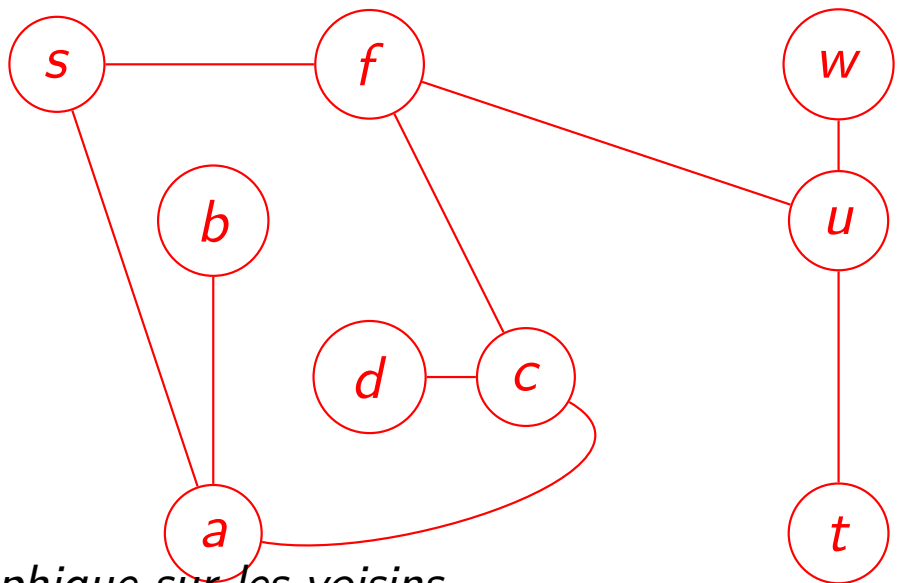
6 Conclusion

Parcours en profondeur

Idée générale de l'algorithme...

Prendre toujours le premier nœud non visité parmis les voisins du sommet courant *et revenir sur ses pas...*

⏩ Skip



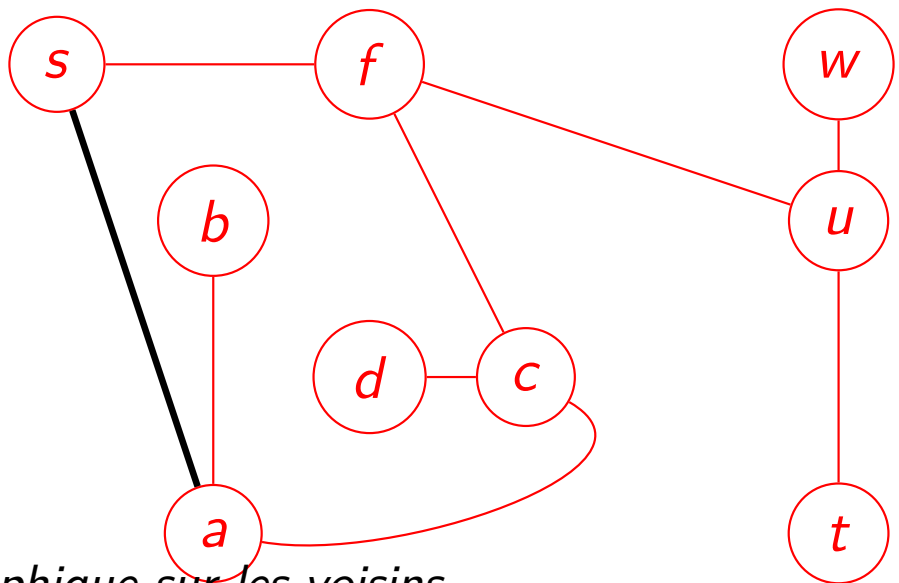
Ici, ordre lexicographique sur les voisins

Parcours en profondeur

Idée générale de l'algorithme...

Prendre toujours le premier nœud non visité parmis les voisins du sommet courant *et revenir sur ses pas...*

⏩ Skip



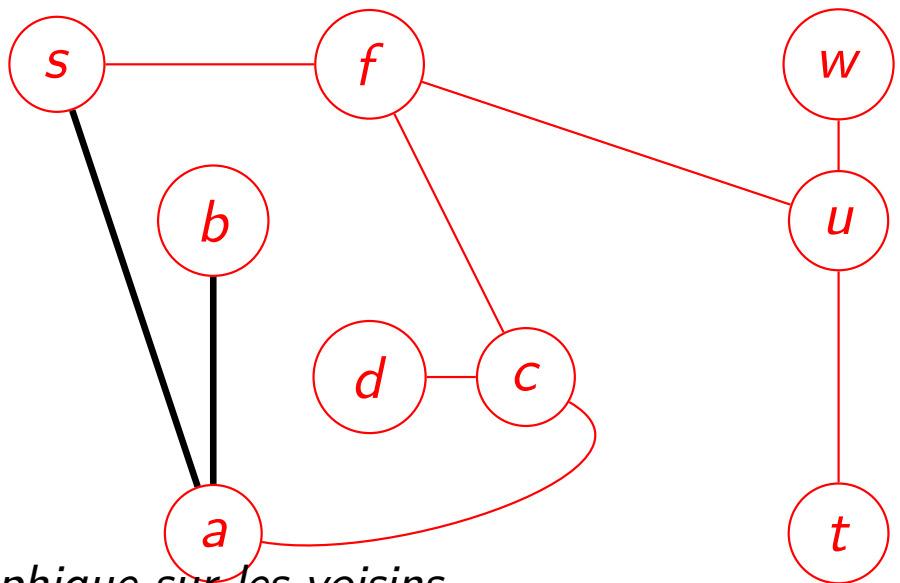
Ici, ordre lexicographique sur les voisins

Parcours en profondeur

Idée générale de l'algorithme...

Prendre toujours le premier nœud non visité parmis les voisins du sommet courant *et revenir sur ses pas...*

⏩ Skip



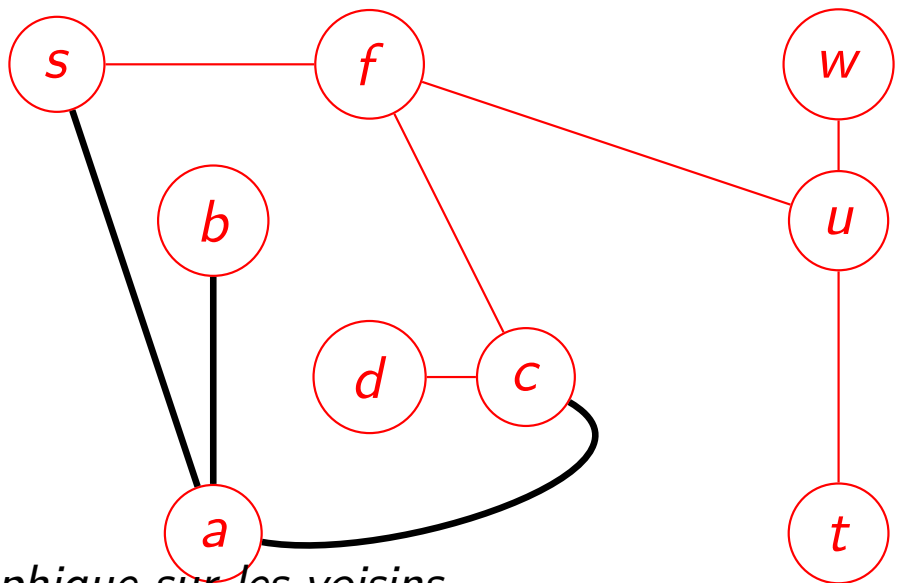
Ici, ordre lexicographique sur les voisins

Parcours en profondeur

Idée générale de l'algorithme...

Prendre toujours le premier nœud non visité parmis les voisins du sommet courant *et revenir sur ses pas...*

⏩ Skip



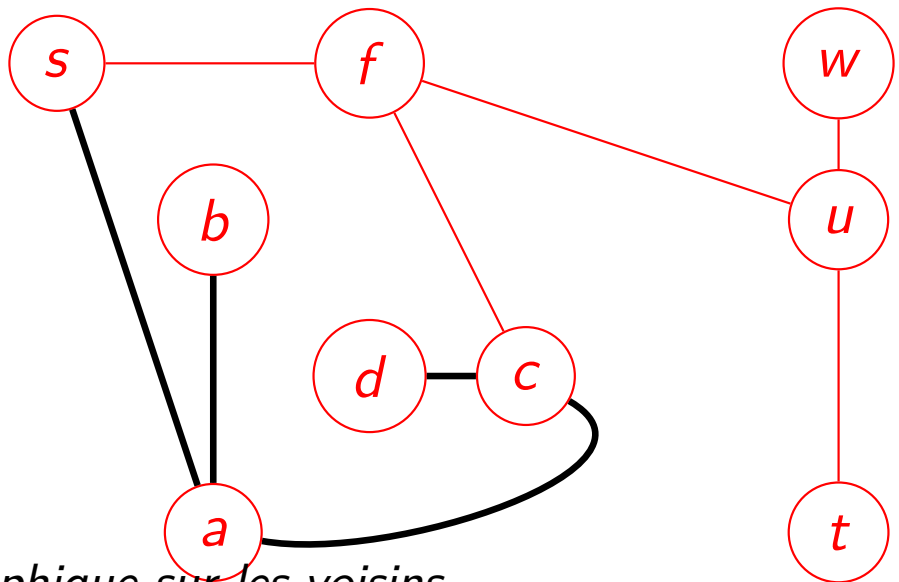
Ici, ordre lexicographique sur les voisins

Parcours en profondeur

Idée générale de l'algorithme...

Prendre toujours le premier nœud non visité parmis les voisins du sommet courant *et revenir sur ses pas...*

⏩ Skip



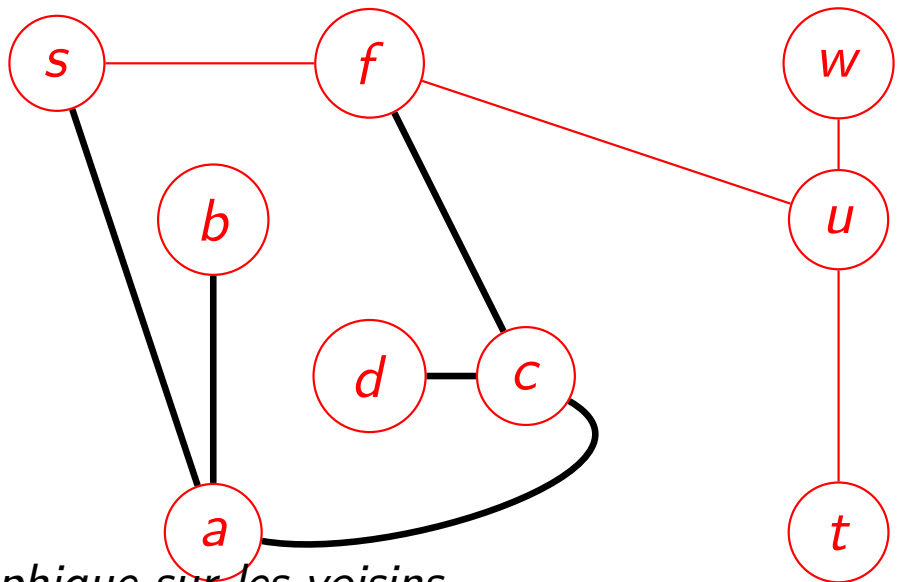
Ici, ordre lexicographique sur les voisins

Parcours en profondeur

Idée générale de l'algorithme...

Prendre toujours le premier nœud non visité parmis les voisins du sommet courant *et revenir sur ses pas...*

⏩ Skip



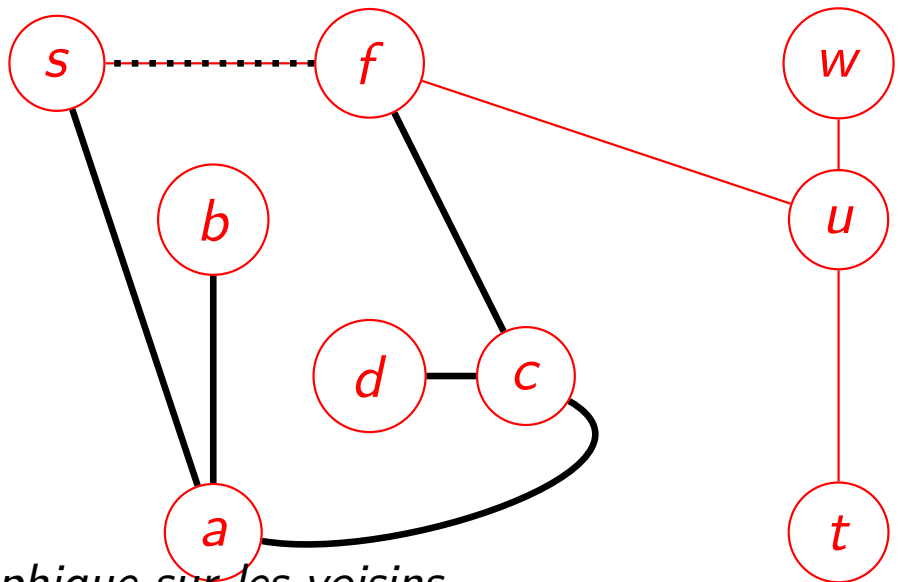
Ici, ordre lexicographique sur les voisins

Parcours en profondeur

Idée générale de l'algorithme...

Prendre toujours le premier nœud non visité parmis les voisins du sommet courant *et revenir sur ses pas...*

⏩ Skip



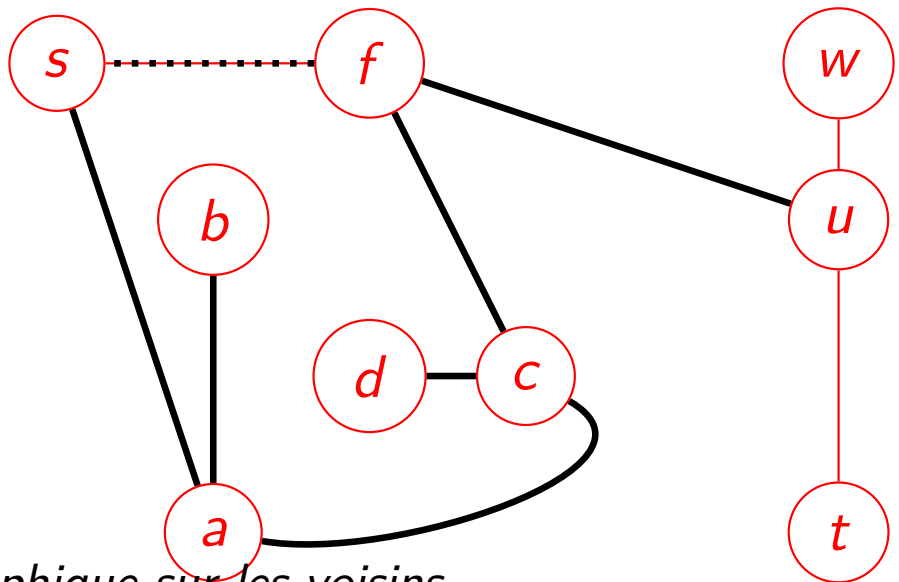
Ici, ordre lexicographique sur les voisins

Parcours en profondeur

Idée générale de l'algorithme...

Prendre toujours le premier nœud non visité parmis les voisins du sommet courant *et revenir sur ses pas...*

⏩ Skip

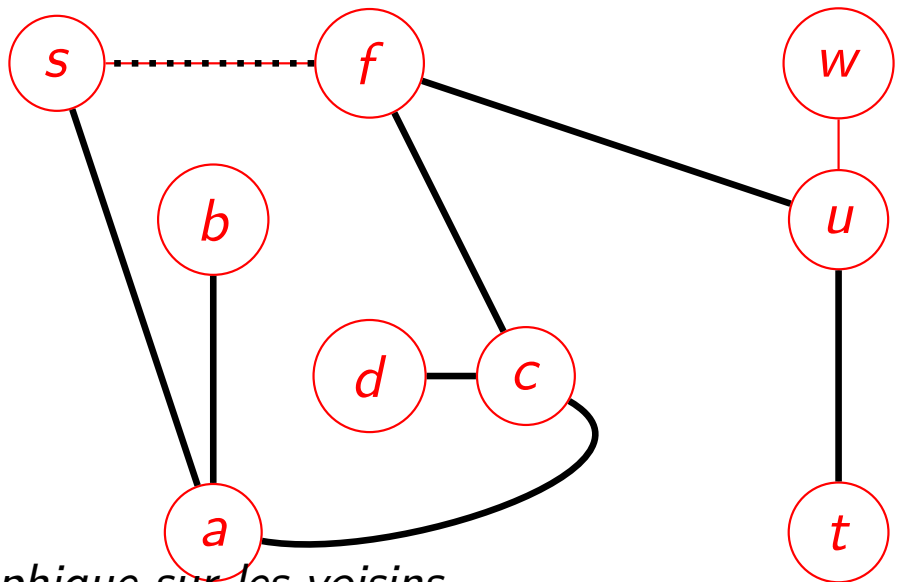


Ici, ordre lexicographique sur les voisins

Parcours en profondeur

Idée générale de l'algorithme...

Prendre toujours le premier nœud non visité parmis les voisins du sommet courant *et revenir sur ses pas...*



Ici, ordre lexicographique sur les voisins

On s'arrête lorsqu'on atteint t

Comment implémenter cette idée ?

i.e. la transformer en algorithme...

De quoi a-t-on besoin ?

1. De connaître les voisins d'un nœud (pour prendre le premier nœud parmi les voisins)
2. De savoir si un nœud a déjà été visité (pour éviter les cycles)
3. On choisit systématiquement le premier voisin non-visité
4. Si ce n'est pas t, on recommence à partir du nœud actuel

Comment implémenter cette idée ?

i.e. la transformer en algorithme...

De quoi a-t-on besoin ?

1. De connaître les voisins d'un nœud (pour prendre le premier nœud parmi les voisins)
   - ➜ Fonction `neighbours(x,E)` qui renvoie la liste des voisins
2. De savoir si un nœud a déjà été visité (pour éviter les cycles)
3. On choisit systématiquement le premier voisin non-visité
4. Si ce n'est pas t, on recommence à partir du nœud actuel

Comment implémenter cette idée ?

i.e. la transformer en algorithme...

De quoi a-t-on besoin ?

1. De connaître les voisins d'un nœud (pour prendre le premier nœud parmi les voisins)
   - ➜ Fonction `neighbours(x,E)` qui renvoie la liste des voisins
2. De savoir si un nœud a déjà été visité (pour éviter les cycles)
   - ➜ Une table `visited` associant les sommets à une valeur booléenne.
3. On choisit systématiquement le premier voisin non-visité
4. Si ce n'est pas t, on recommence à partir du nœud actuel

Comment implémenter cette idée ?

i.e. la transformer en algorithme...

De quoi a-t-on besoin ?

1. De connaître les voisins d'un nœud (pour prendre le premier nœud parmi les voisins)
   - ➜ Fonction `neighbours(x,E)` qui renvoie la liste des voisins
2. De savoir si un nœud a déjà été visité (pour éviter les cycles)
   - ➜ Une table `visited` associant les sommets à une valeur booléenne.
   - ➜ Dictionnaire Python (ou Set Python ici)
3. On choisit systématiquement le premier voisin non-visité
4. Si ce n'est pas t, on recommence à partir du nœud actuel

Comment implémenter cette idée ?

i.e. la transformer en algorithme...

De quoi a-t-on besoin ?

1. De connaître les voisins d'un nœud (pour prendre le premier nœud parmi les voisins)
   - ➜ Fonction `neighbours(x,E)` qui renvoie la liste des voisins
2. De savoir si un nœud a déjà été visité (pour éviter les cycles)
   - ➜ Une table `visited` associant les sommets à une valeur booléenne.
   - ➜ Dictionnaire Python (ou Set Python ici)
3. On choisit systématiquement le premier voisin non-visité
4. Si ce n'est pas t, on recommence à partir du nœud actuel
   - ➜ Fonction récursive

Rappel : dictionnaires Python

Structure de données (clef,valeur)

```
dico = { clef:valeur, ... }
```

- La structure associe une valeur à un nom (la clef) avec la notation : `dico[clef]=valeur`
- Les clefs sont souvent des chaînes de caractères
- On accède aux valeurs par : `dico[clef]`
- On peut parcourir les clefs :

```
for c in dico: ...
```

- On peut tester si une clef existe :

```
if c in dico: ...
```

Parcours en profondeur : implémentation récursive

1. Initialisation du dictionnaire : `visited = { }`

Parcours en profondeur : implémentation récursive

1. Initialisation du dictionnaire : `visited = { }`
2. Fonction récursive `DFS_rec(V,E,n,t)`

```
def DFS_rec(V,E,n,t):     # n : le noeud courant
  visited[n] = True
  if n==t: return True
  for v in neighbours(n,E):
    if not v in visited:
      if DFS_rec(V,E,v,t):
        return True
  return False
```

Parcours en profondeur : implémentation récursive

1. Initialisation du dictionnaire : `visited = { }`
2. Fonction récursive `DFS_rec(V,E,n,t)`

```
def DFS_rec(V,E,n,t):     # n : le noeud courant
  visited[n] = True
  if n==t: return True
  for v in neighbours(n,E):
    if not v in visited:
      if DFS_rec(V,E,v,t):
        return True
  return False
```

3. Appel de la fonction : `DFS_rec(V,E,s,t)`

Parcours en profondeur : implémentation récursive

1. Initialisation du dictionnaire : `visited = { }`
2. Fonction récursive `DFS_rec(V,E,n,t)`

```
def DFS_rec(V,E,n,t):     # n : le noeud courant
  visited[n] = True
  if n==t: return True
  for v in neighbours(n,E):
    if not v in visited:
      if DFS_rec(V,E,v,t):
        return True
  return False
```

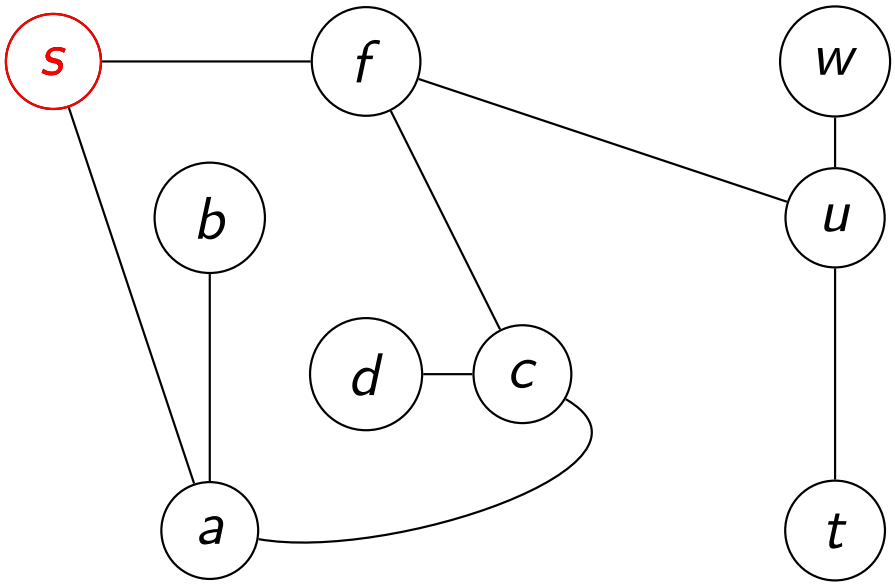
3. Appel de la fonction : `DFS_rec(V,E,s,t)`

Remarques

- La liste des nœuds V n'est pas utilisée dans `DFS_rec`
- Cet algorithme ne renvoie pas le chemin, seulement `True` ou `False` selon qu'il atteint t ou non ($\rightarrow$ problème de décision).

Implémentation : démo

⏩ Skip Demo



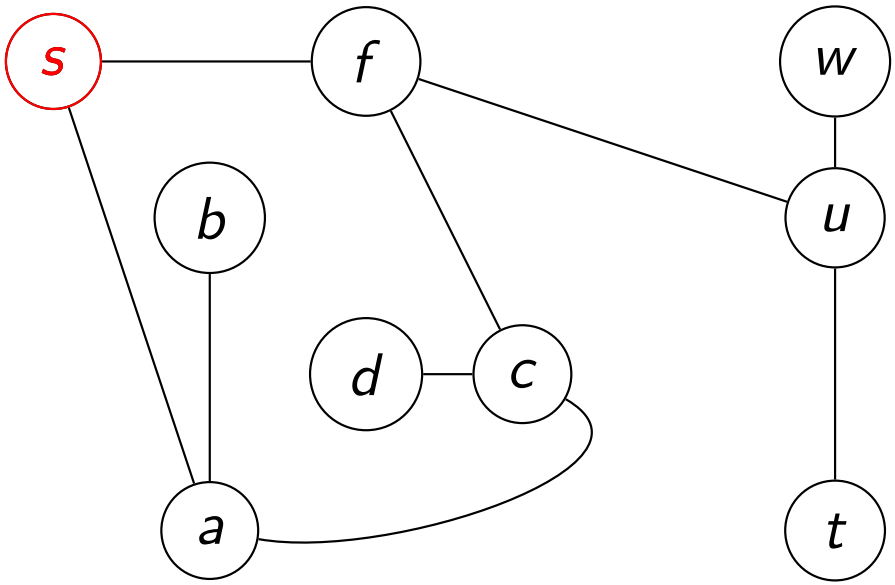
visited

a :
b :
c :
d :
f :
s :
t :
u :
w :

```
DFS_rec(V,E,'s','t')
```

Implémentation : démo

⏩ Skip Demo



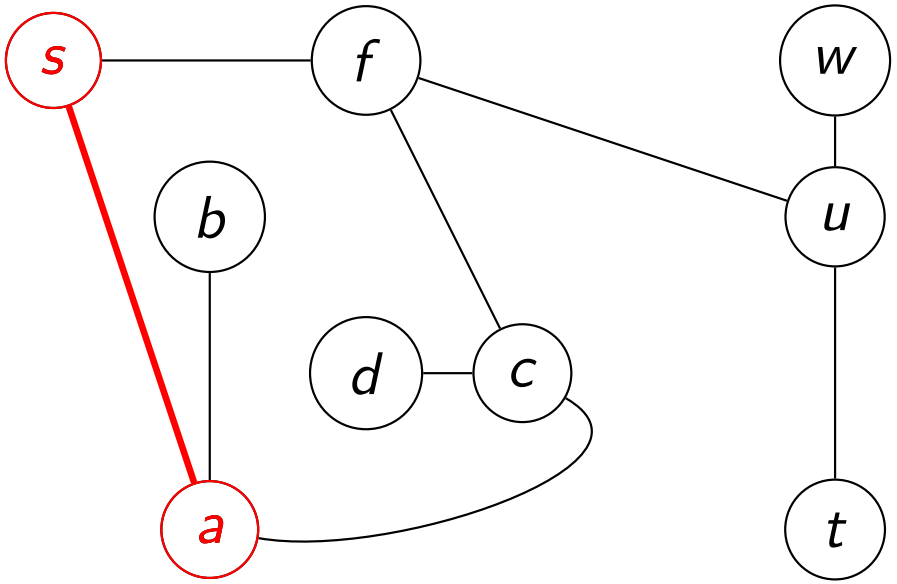
visited

a :
b :
c :
d :
f :
s : *True*
t :
u :
w :

```
DFS_rec(V,E,'s','t')
```

Implémentation : démo

⏩ Skip Demo



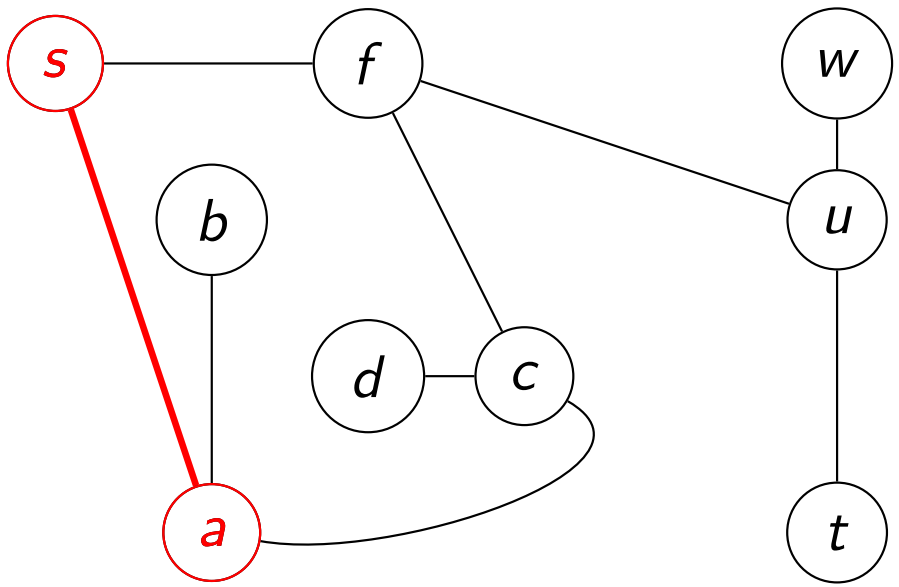
visited

a :
b :
c :
d :
f :
s : *True*
t :
u :
w :

```
DFS_rec(V,E,'s','t')
```

Implémentation : démo

⏩ Skip Demo



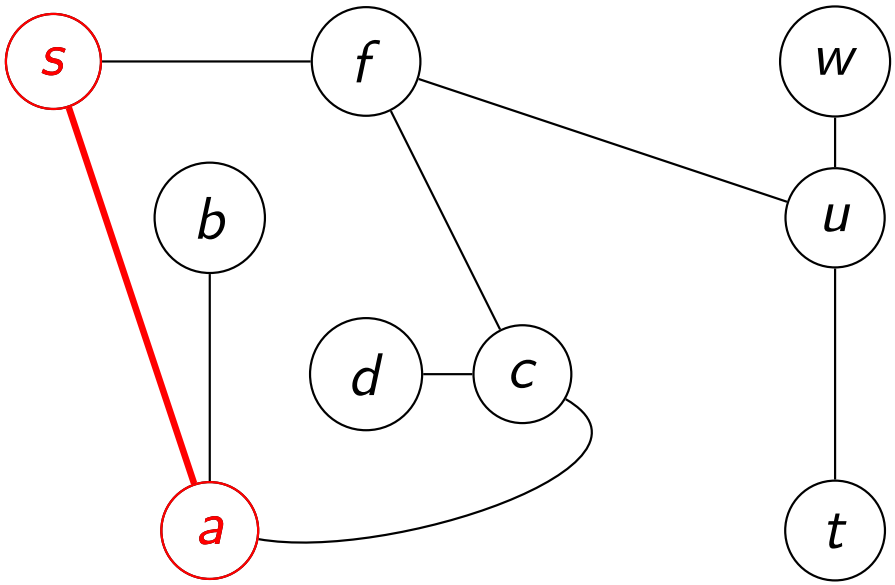
visited

a :
b :
c :
d :
f :
s : *True*
t :
u :
w :

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
```

Implémentation : démo

⏩ Skip Demo



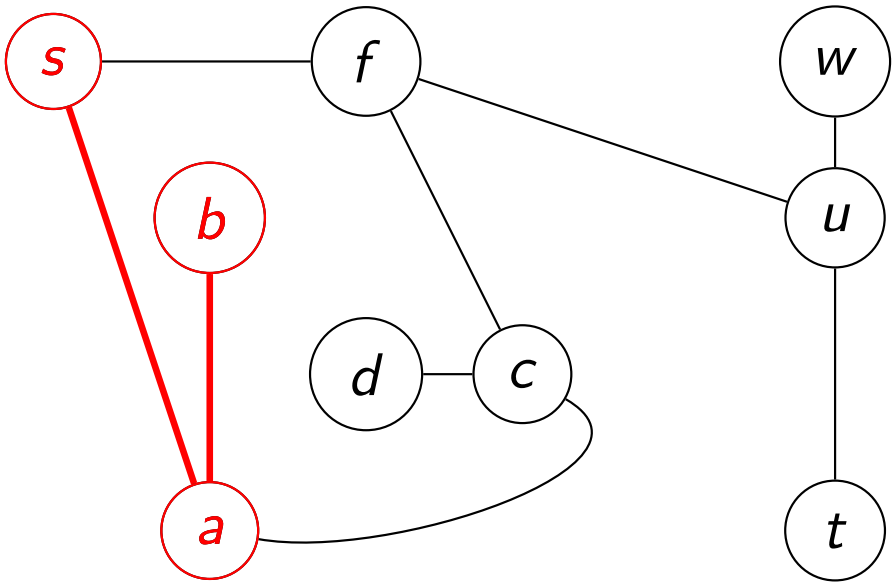
visited

a :	*True*
b :	
c :	
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
```

Implémentation : démo

⏩ Skip Demo



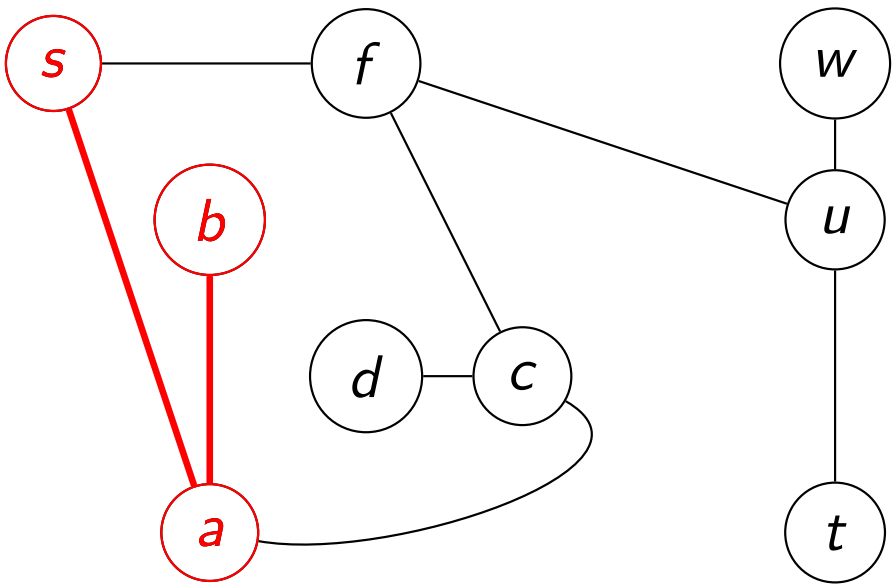
visited

a : *True*
b :
c :
d :
f :
s : *True*
t :
u :
w :

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
```

Implémentation : démo

⏩ Skip Demo



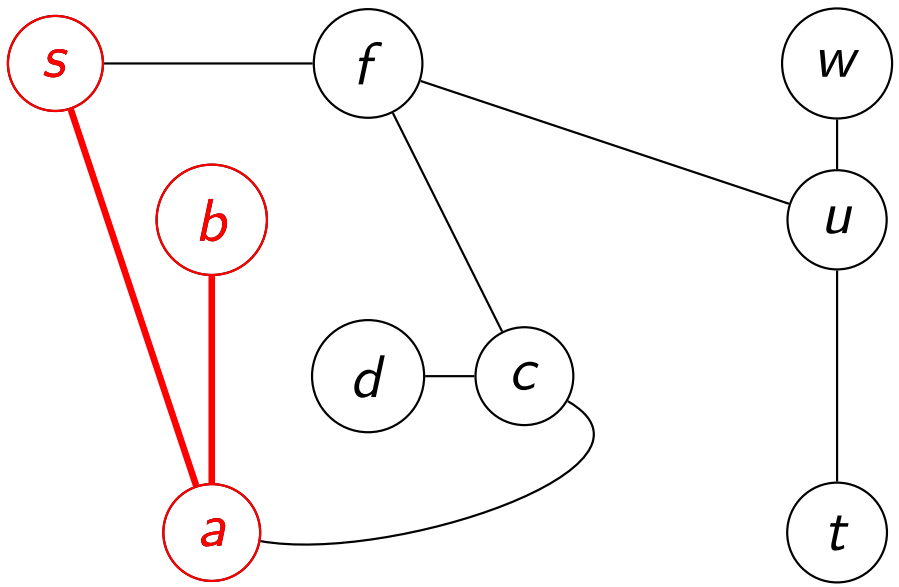
visited

a : *True*
b :
c :
d :
f :
s : *True*
t :
u :
w :

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'b','t')
```

Implémentation : démo

⏩ Skip Demo



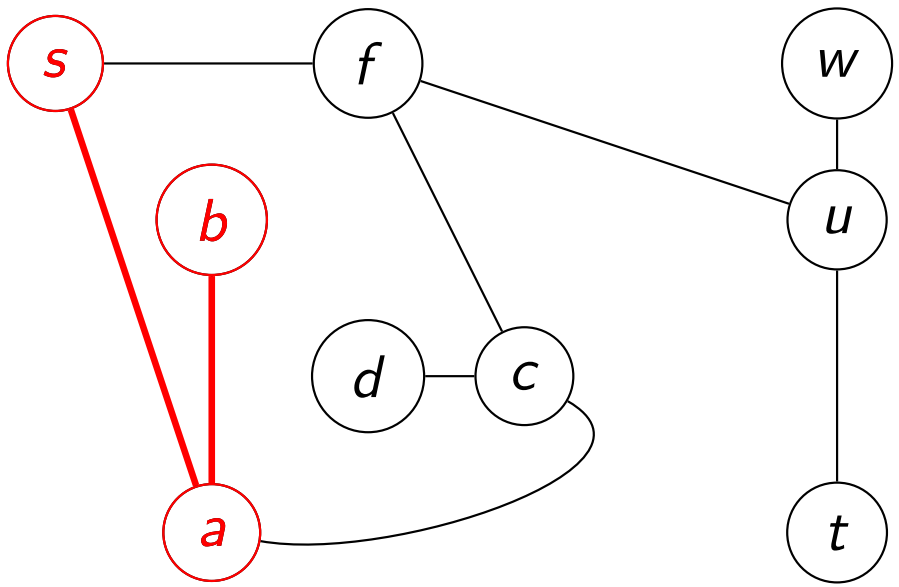
visited

a :	*True*
b :	*True*
c :	
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'b','t')
```

Implémentation : démo

⏩ Skip Demo



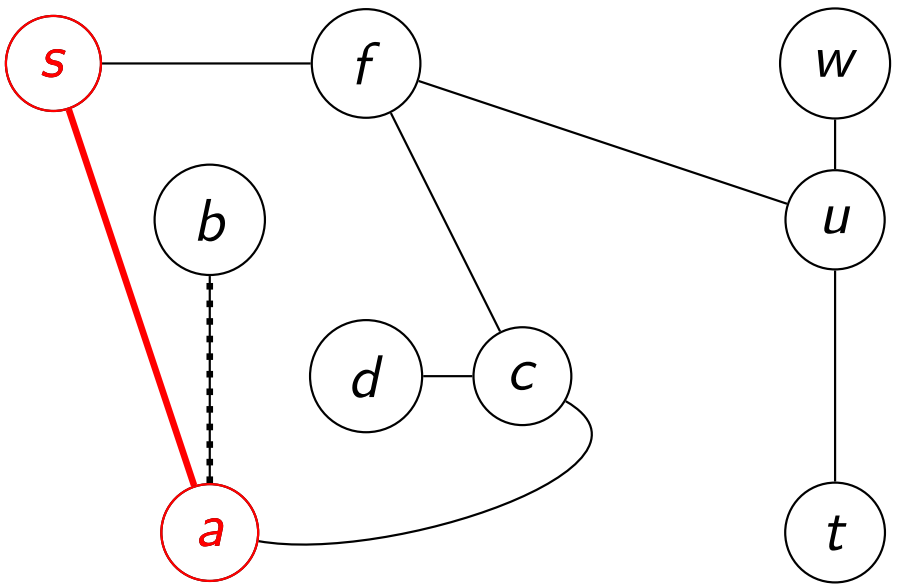
visited

a :	*True*
b :	*True*
c :	
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'b','t') : False
```

Implémentation : démo

⏩ Skip Demo



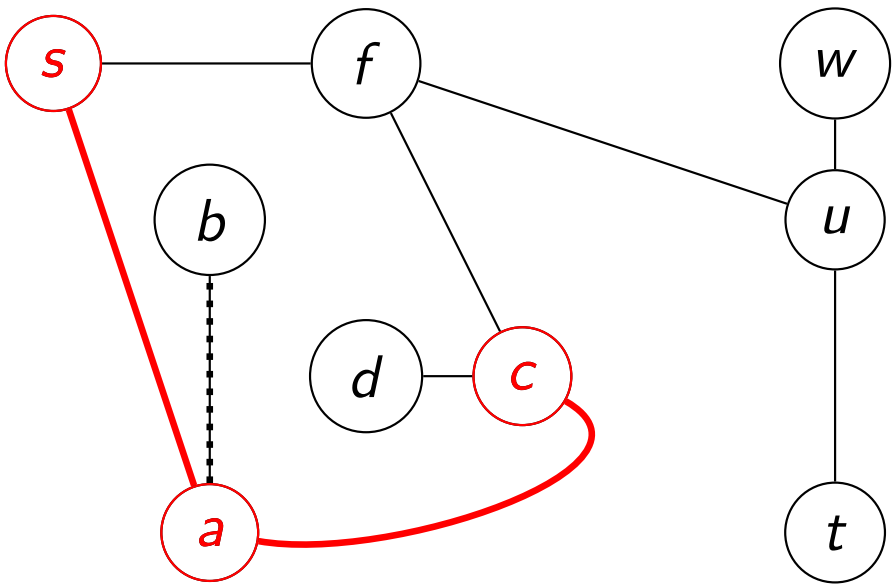
visited

a :	*True*
b :	*True*
c :	
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
```

Implémentation : démo

⏩ Skip Demo



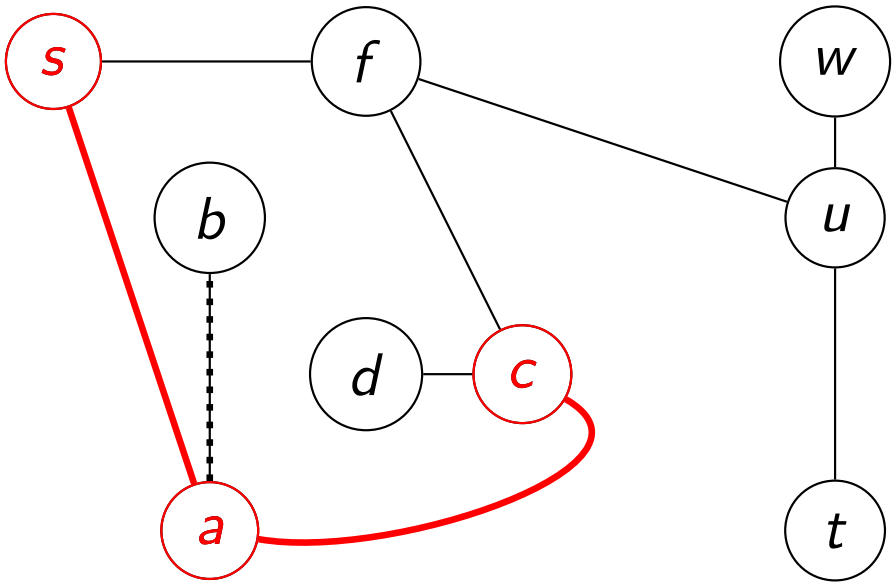
visited

a :	*True*
b :	*True*
c :	
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
```

Implémentation : démo

⏩ Skip Demo



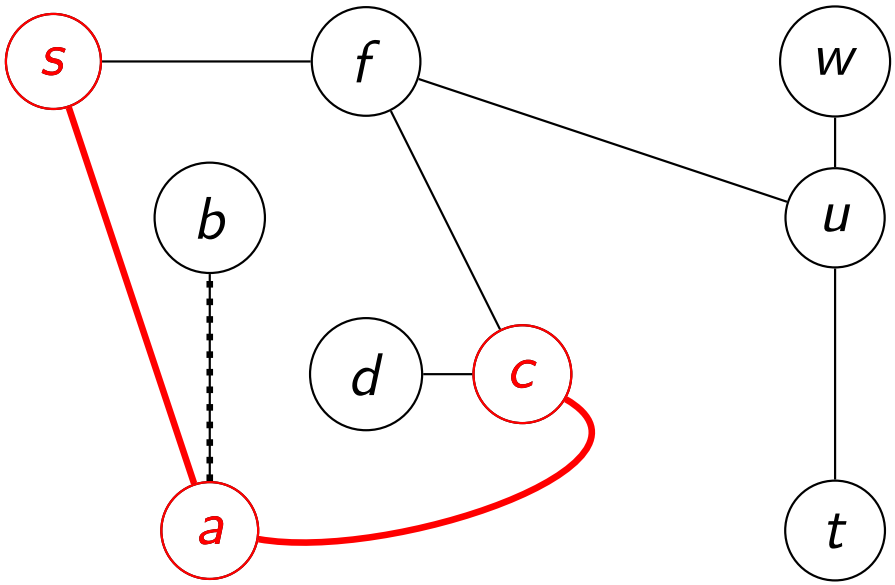
visited

a :	*True*
b :	*True*
c :	
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
```

Implémentation : démo

⏩ Skip Demo



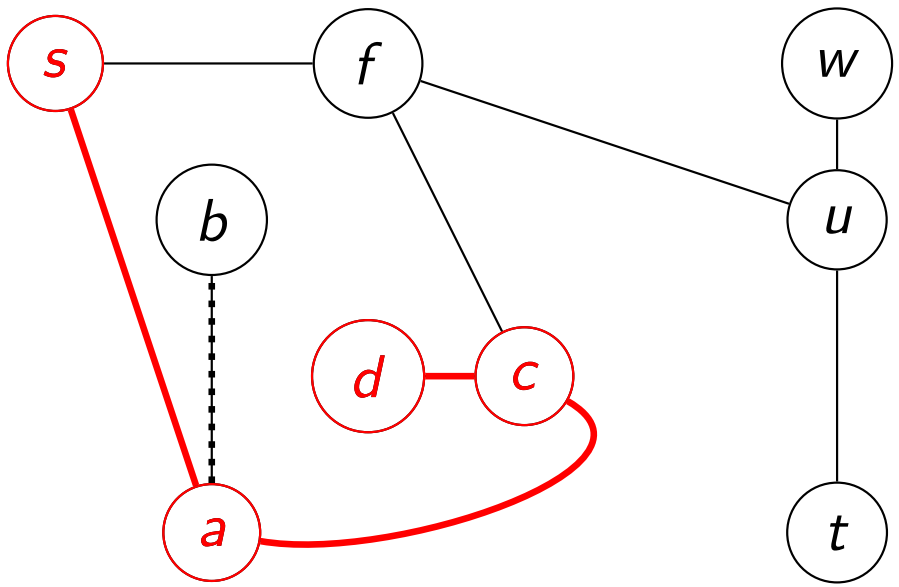
visited

a :	*True*
b :	*True*
c :	*True*
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
```

Implémentation : démo

⏩ Skip Demo



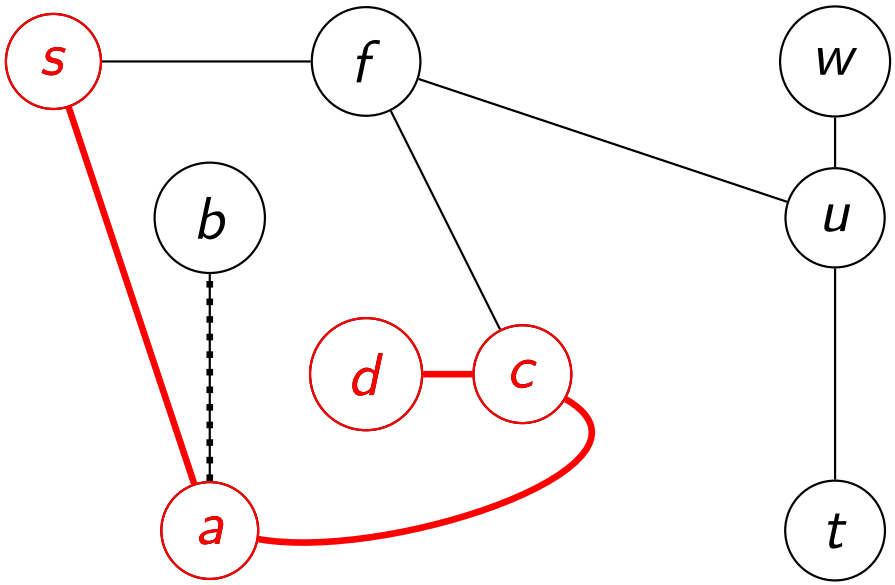
visited

a :	*True*
b :	*True*
c :	*True*
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
```

Implémentation : démo

Skip Demo



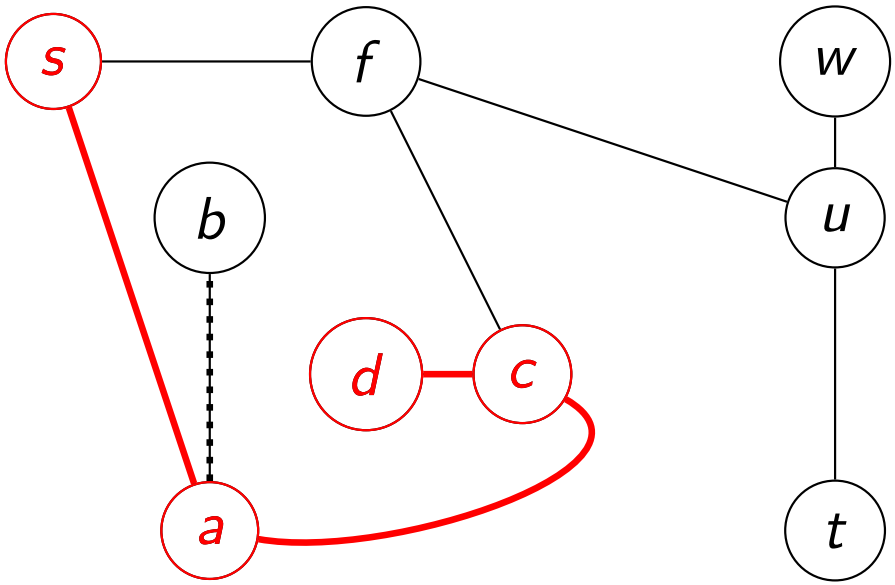
visited

a :	*True*
b :	*True*
c :	*True*
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'d','t')
```

Implémentation : démo

Skip Demo



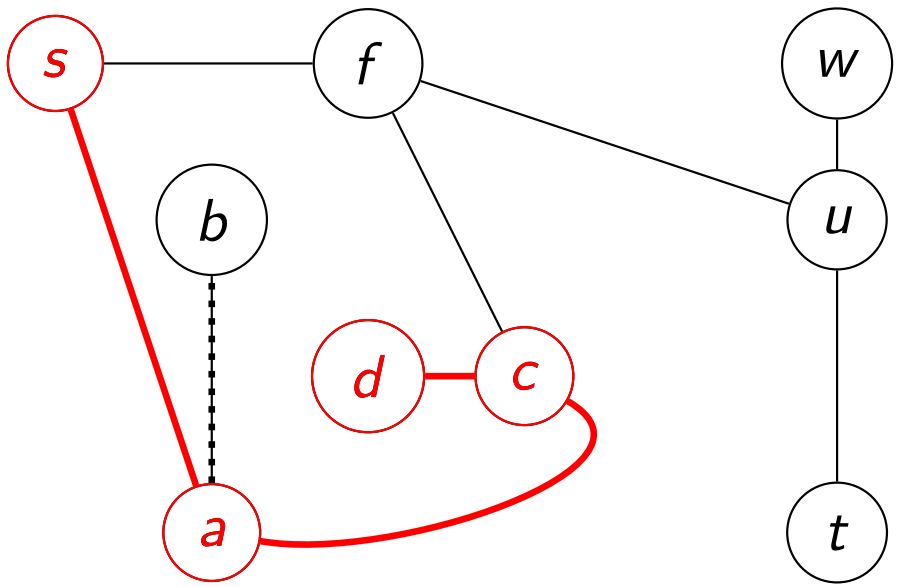
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'d','t')
```

Implémentation : démo

⏩ Skip Demo



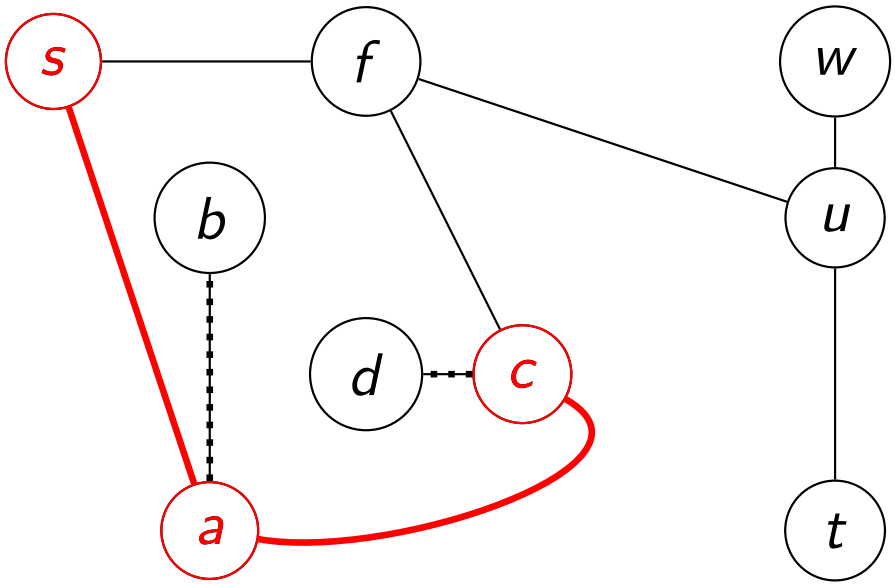
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'d','t') : False
```

Implémentation : démo

⏩ Skip Demo



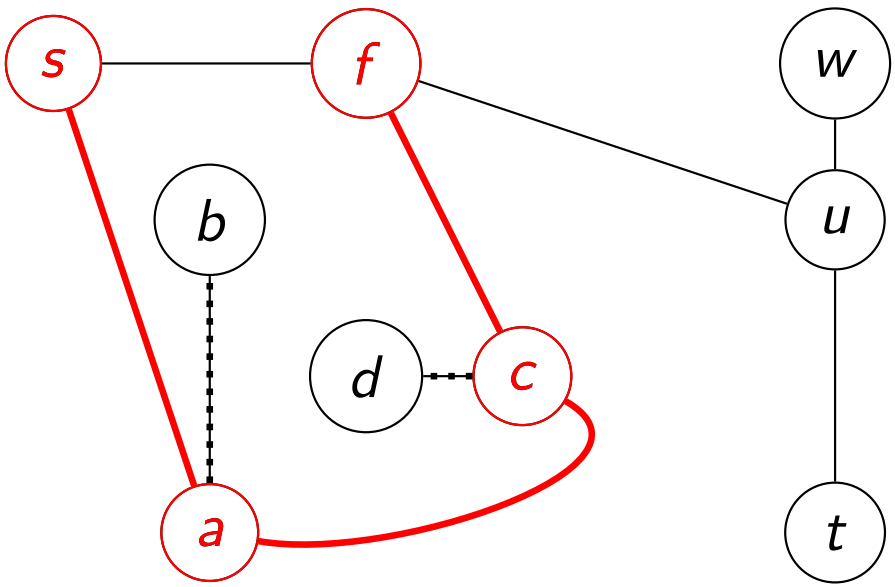
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
```

Implémentation : démo

⏩ Skip Demo



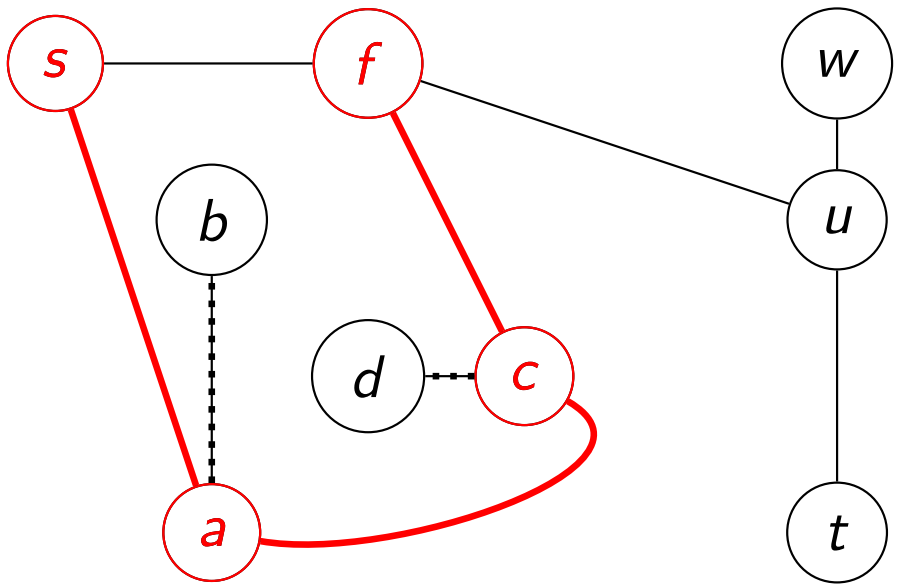
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
```

Implémentation : démo

⏩ Skip Demo



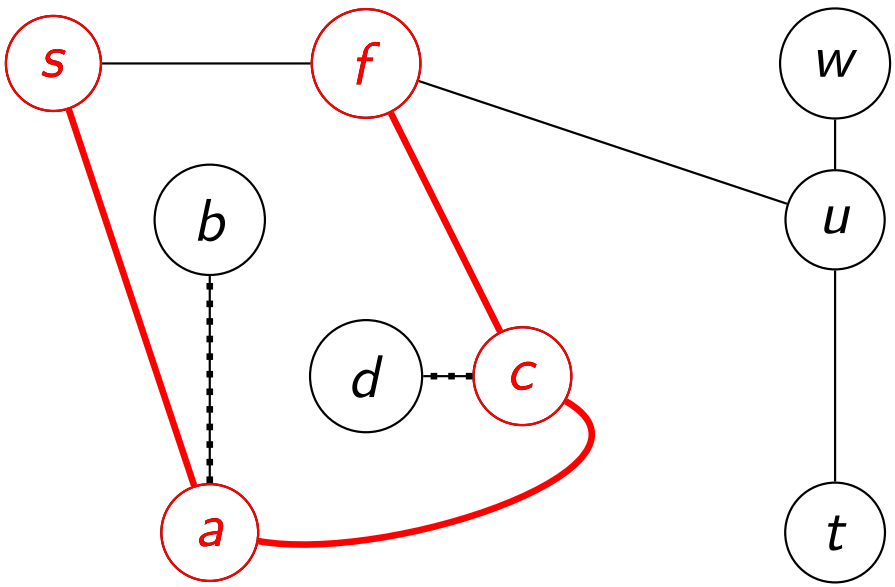
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
```

Implémentation : démo

⏩ Skip Demo



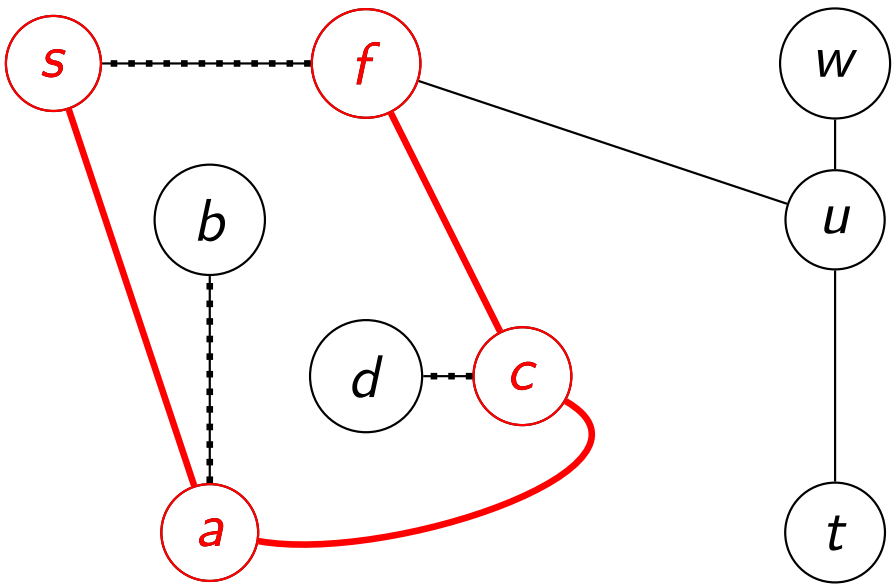
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
```

Implémentation : démo

Skip Demo



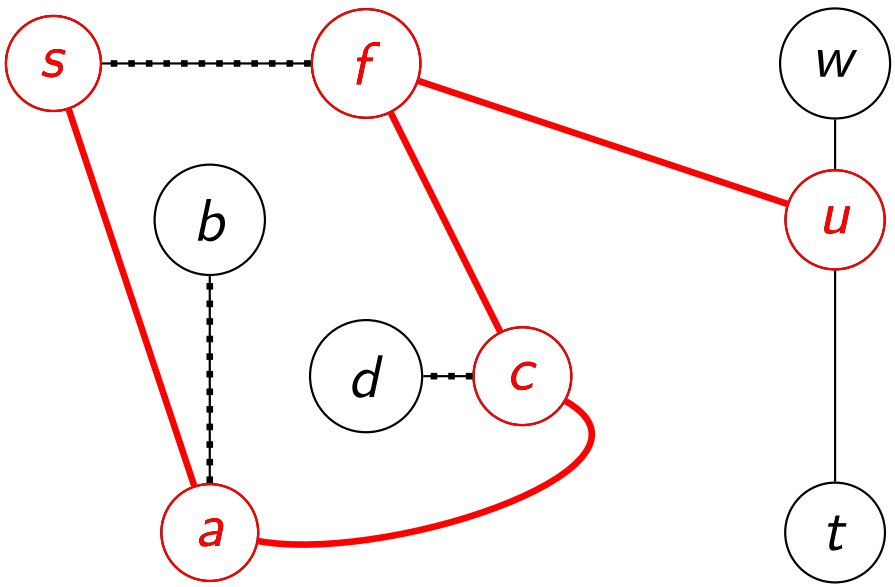
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
```

Implémentation : démo

Skip Demo



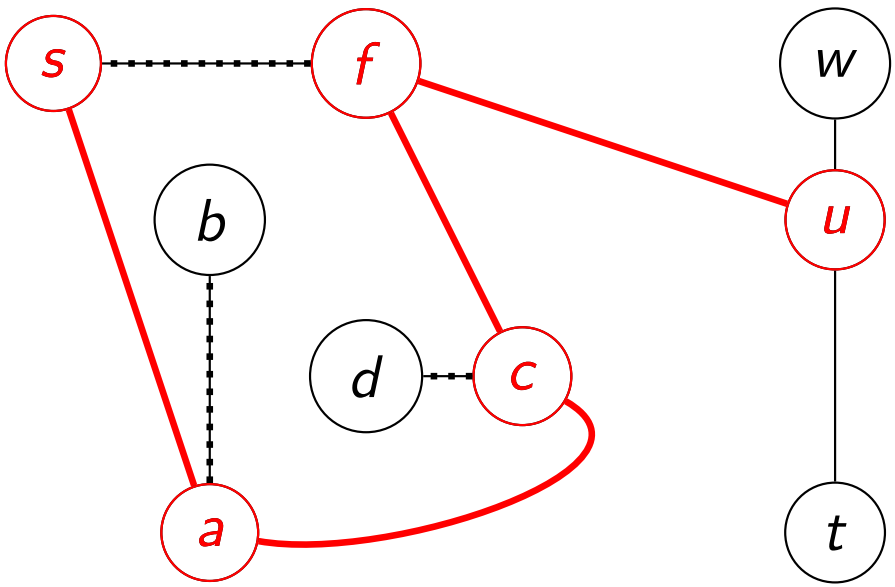
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
```

Implémentation : démo

⏩ Skip Demo



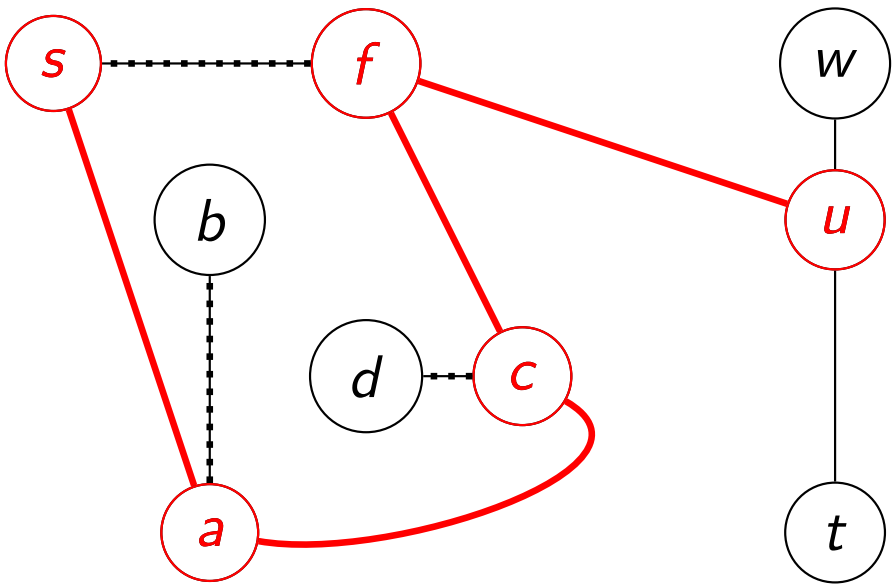
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
            ↪ DFS_rec(V,E,'u','t')
```

Implémentation : démo

Skip Demo



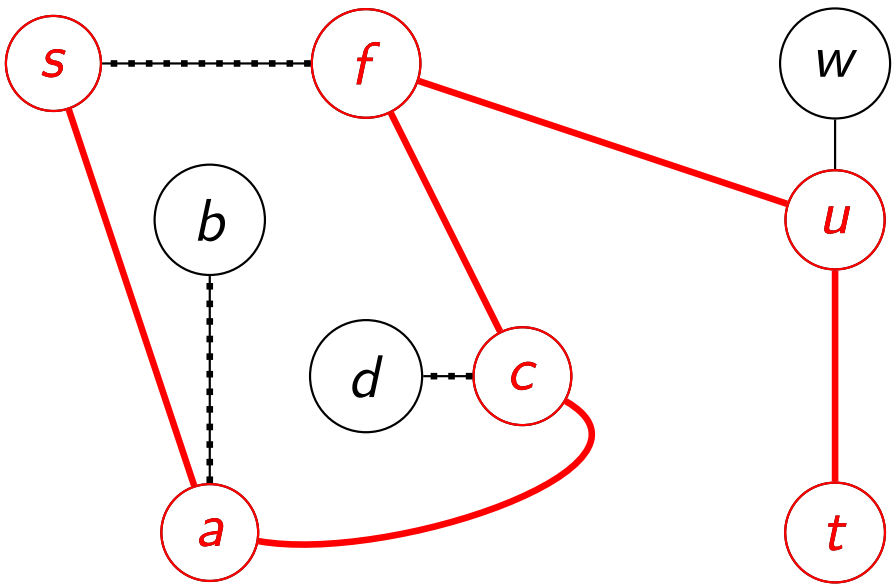
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	*True*
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
            ↪ DFS_rec(V,E,'u','t')
```

Implémentation : démo

⏩ Skip Demo



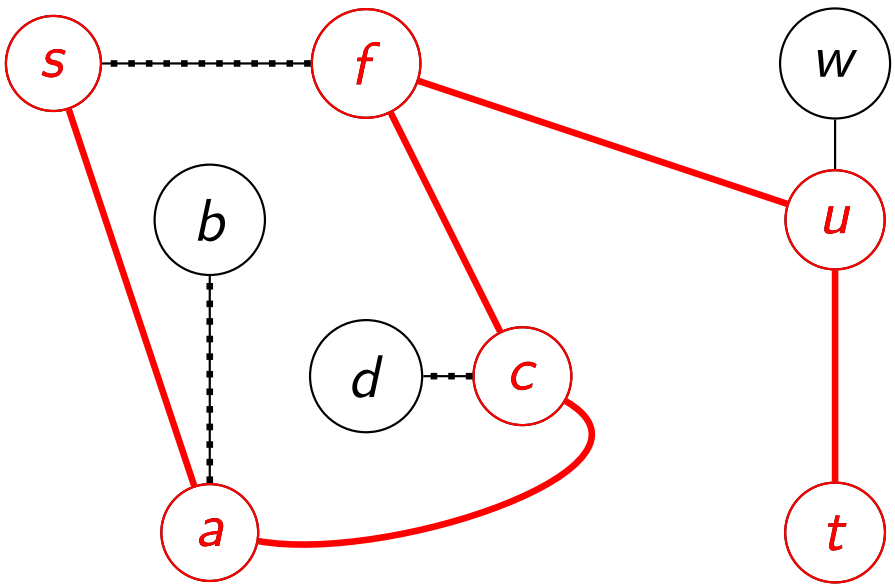
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	*True*
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
            ↪ DFS_rec(V,E,'u','t')
```

Implémentation : démo

Skip Demo



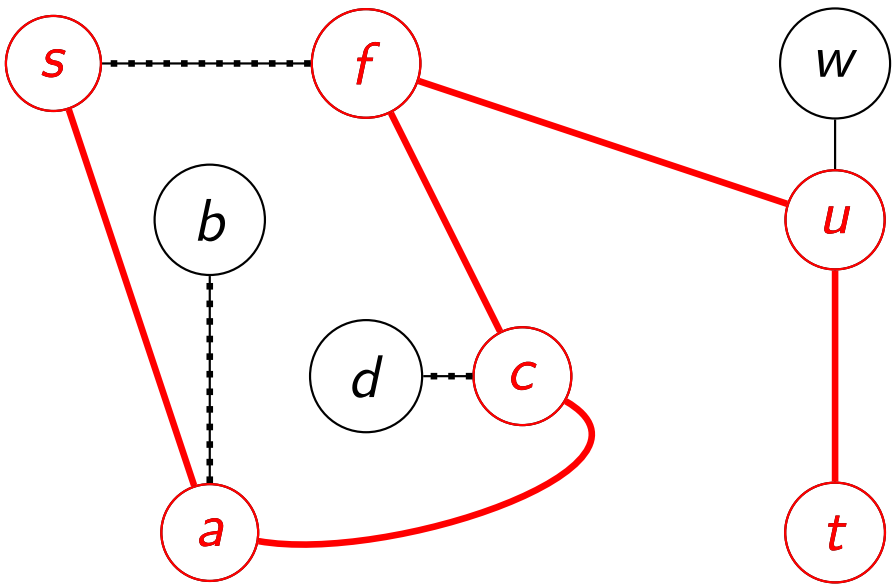
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	*True*
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
            ↪ DFS_rec(V,E,'u','t')
                ↪ DFS_rec(V,E,'t','t')
```

Implémentation : démo

Skip Demo



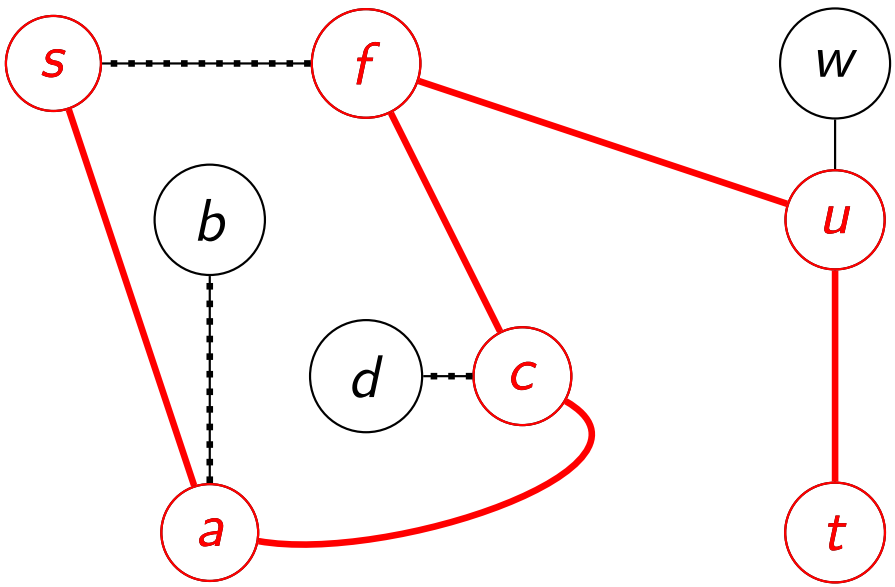
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	*True*
u :	*True*
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
            ↪ DFS_rec(V,E,'u','t')
                ↪ DFS_rec(V,E,'t','t')
```

Implémentation : démo

⏩ Skip Demo



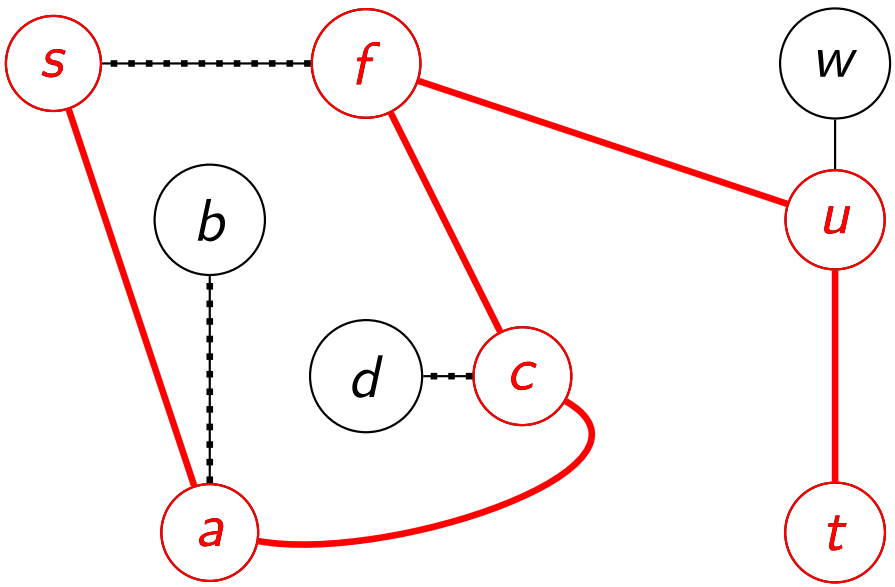
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	*True*
u :	*True*
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
            ↪ DFS_rec(V,E,'u','t')
                ↪ DFS_rec(V,E,'t','t') : True
```

Implémentation : démo

Skip Demo



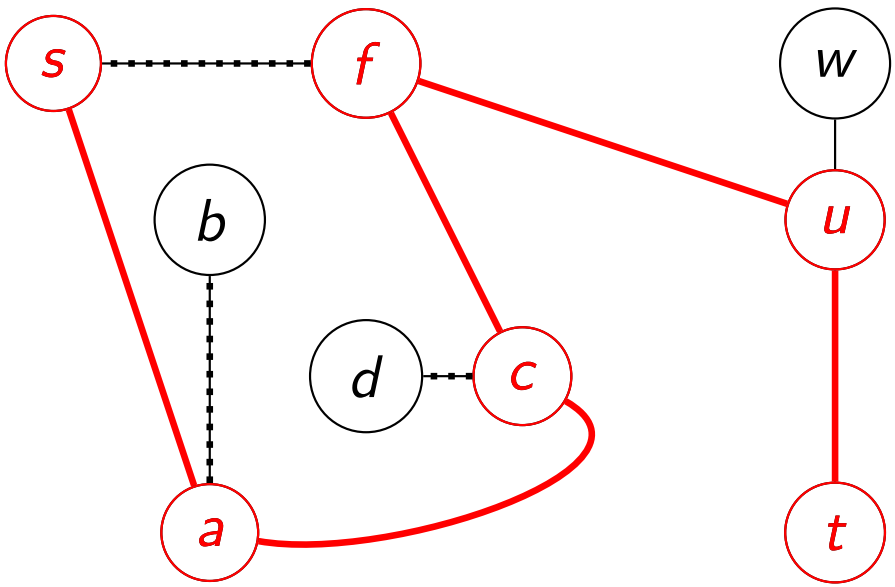
visited

a : *True*
b : *True*
c : *True*
d : *True*
f : *True*
s : *True*
t : *True*
u : *True*
w :

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
            ↪ DFS_rec(V,E,'u','t') : True
```

Implémentation : démo

⏩ Skip Demo



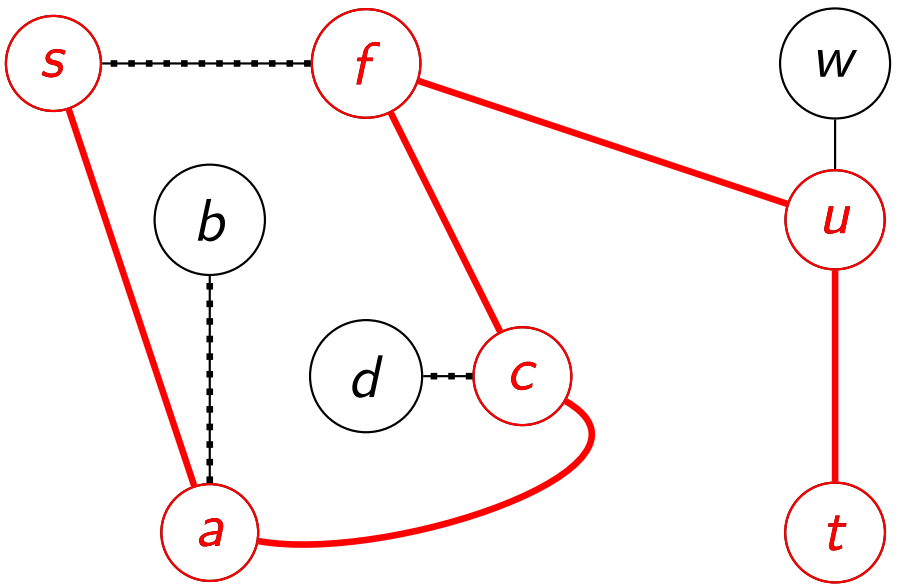
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	*True*
u :	*True*
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t') : True
```

Implémentation : démo

⏩ Skip Demo



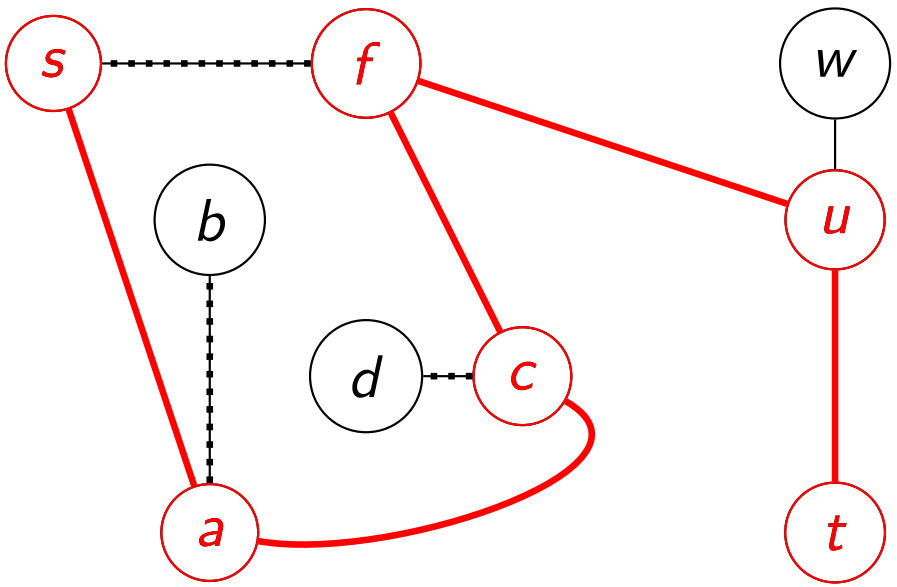
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	*True*
u :	*True*
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t') : True
```

Implémentation : démo

⏩ Skip Demo

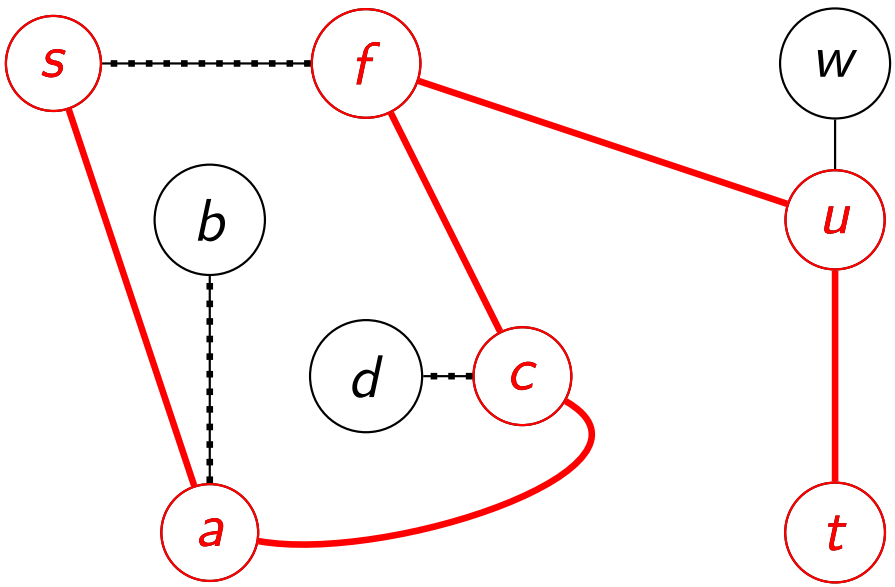


visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	*True*
u :	*True*
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t') : True
```

Implémentation : démo

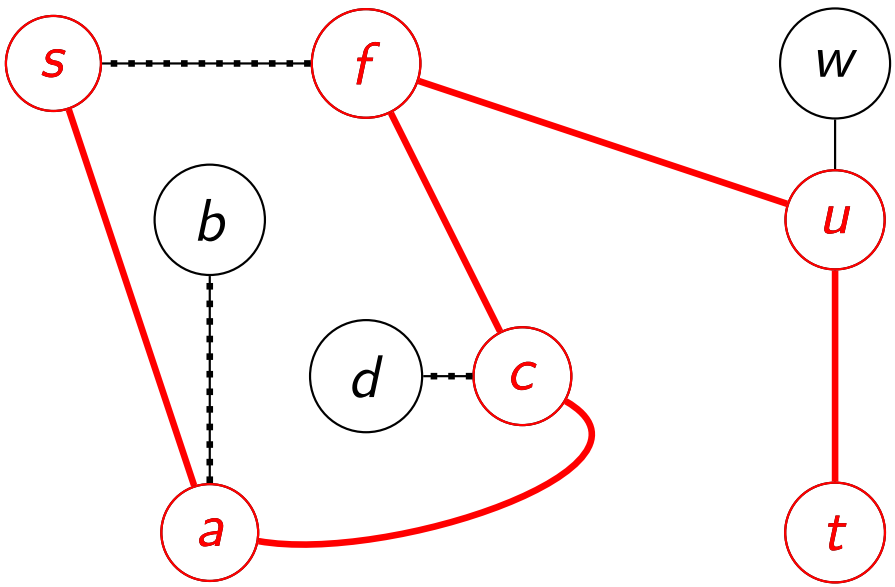


visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	*True*
u :	*True*
w :	

`DFS_rec(V,E,'s','t')` : True

Implémentation : démo



visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	*True*
u :	*True*
w :	

`DFS_rec(V,E,'s','t')` : True

Remarque

Tous les sommets marqués à `True` dans `visited` sont atteignables depuis s.

Nous y reviendrons. . .

Supprimer la récursion ?

Supprimer la récursion ?

Introduire une pile

✓ Transformer un algorithme récursif en algorithme itératif en utilisant une pile.

➜ Les prochains nœuds à visiter sont rangés dans la pile.

Supprimer la récursion ?

Introduire une pile

✓ Transformer un algorithme récursif en algorithme itératif en utilisant une pile.

➜ Les prochains nœuds à visiter sont rangés dans la pile.

Rappel : définition d'une pile

- Structure de données informatique.
- Séquence d'éléments dans laquelle on ajoute et on retire toujours du même côté.
- Les objets ressortent dans l'ordre inverse de l'entrée (Last In First Out).

Piles en Python

En algorithmique

Méthodes `push` et `pop` sur une *pile* (`stack` en anglais)

En Python

Le type `list` supporte les méthodes `append` et `pop`

Pour garder l'intuition

Nous utiliserons deux fonctions qui « cachent » l'implémentation :

```
def add_end(x,l):
    l.append(x)

def pop_end(l):
    return l.pop()
```

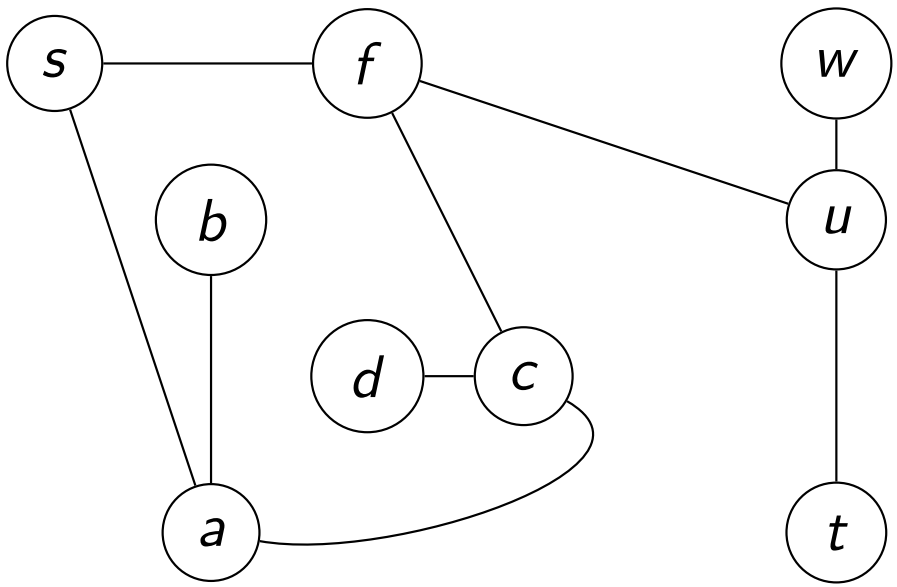
Parcours en profondeur : implémentation itérative

```
def DFS_iter(V,E,s,t):
    lnext = [s]                  # la pile
    reached = { s: True }        # eviter des ajouts multiples

    while len(lnext)>0:
      n = pop_end(lnext)

      if n==t:
        return True

      for v in neighbours(n,E):
        if not v in reached:
          reached[v] = True
          add_end(v,lnext)       # recursion -> ajout dans pile

    return False
```

Remarque : Ne renvoie pas le chemin-solution → voir TD1

Implémentation itérative : démo

⏩ Skip Demo



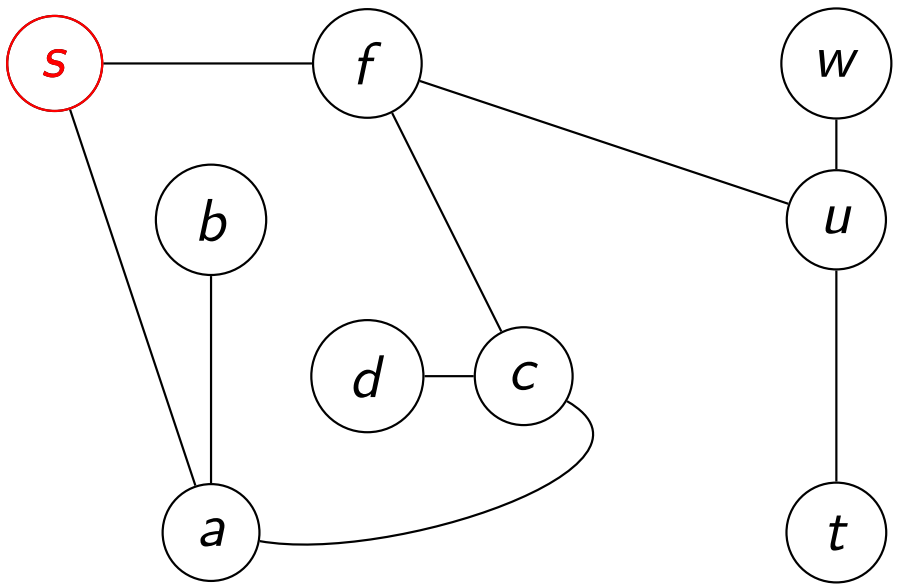
reached

a :
b :
c :
d :
f :
s : *True*
t :
u :
w :

```
lnext =  [ 's' ]
n =
```

Implémentation itérative : démo

Skip Demo



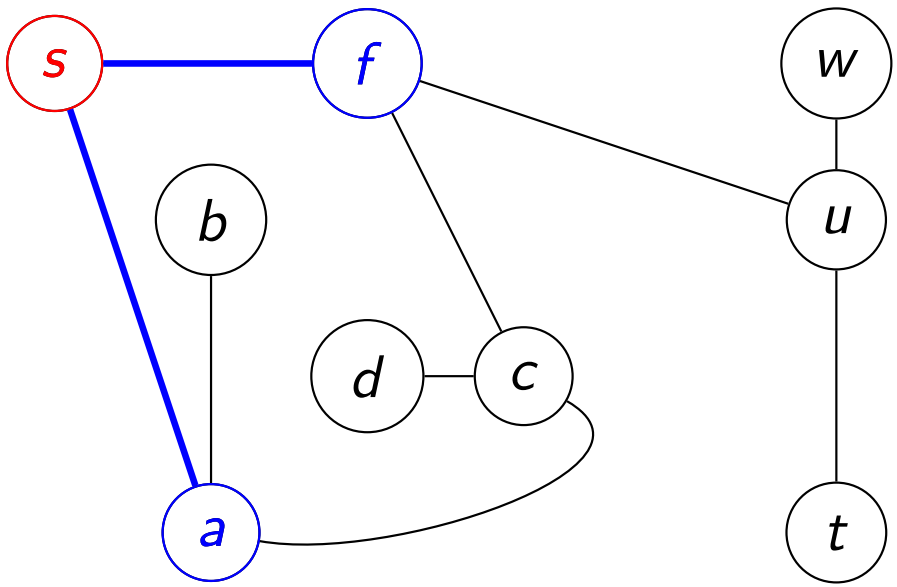
reached

a :
b :
c :
d :
f :
s : *True*
t :
u :
w :

```
lnext = [ ]
n = 's'
```

Implémentation itérative : démo

Skip Demo



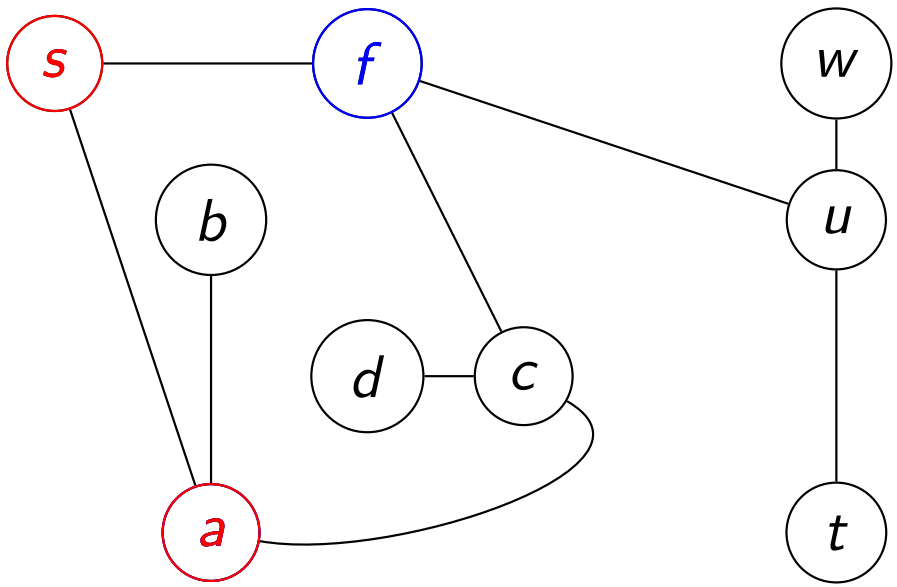
reached

a :	*True*
b :	
c :	
d :	
f :	*True*
s :	*True*
t :	
u :	
w :	

```
lnext =  [ 'f', 'a' ]
n =  's'
```

Implémentation itérative : démo

Skip Demo



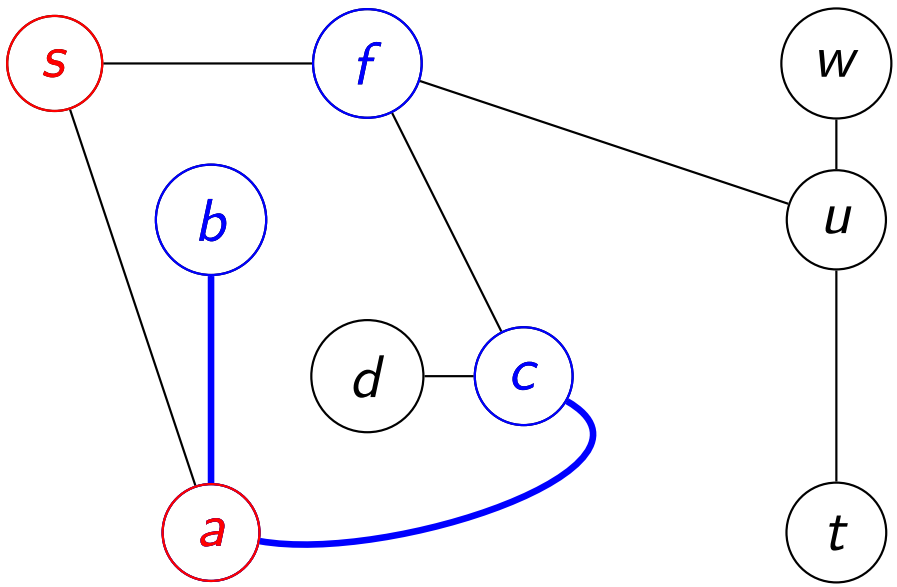
reached

a :	*True*
b :	
c :	
d :	
f :	*True*
s :	*True*
t :	
u :	
w :	

```
lnext =  [ 'f' ]
n =  'a'
```

Implémentation itérative : démo

⏩ Skip Demo



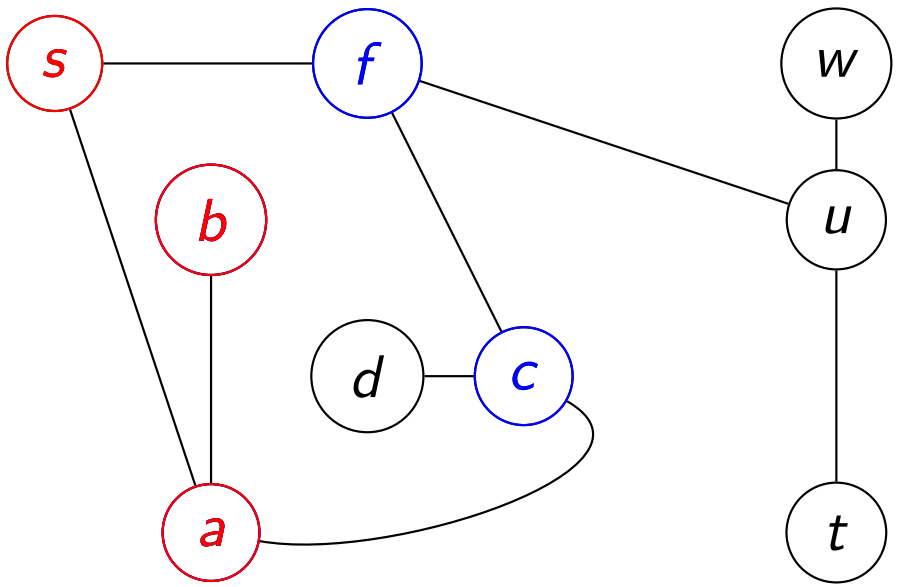
reached

a :	*True*
b :	*True*
c :	*True*
d :	
f :	*True*
s :	*True*
t :	
u :	
w :	

```
lnext =  [ 'f', 'c', 'b' ]
n =  'a'
```

Implémentation itérative : démo

⏩ Skip Demo



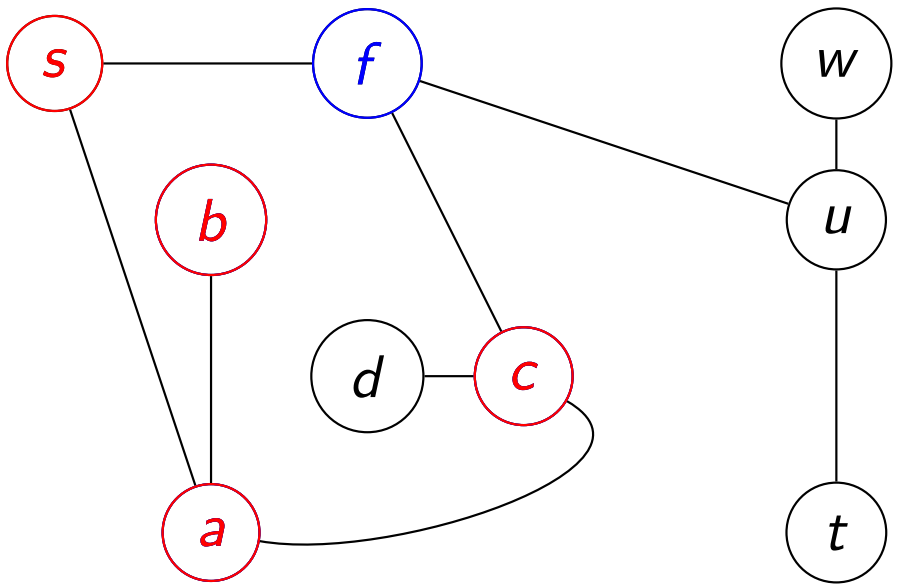
reached

a :	*True*
b :	*True*
c :	*True*
d :	
f :	*True*
s :	*True*
t :	
u :	
w :	

```
lnext =  [ 'f', 'c' ]
n =  'b'
```

Implémentation itérative : démo

⏩ Skip Demo



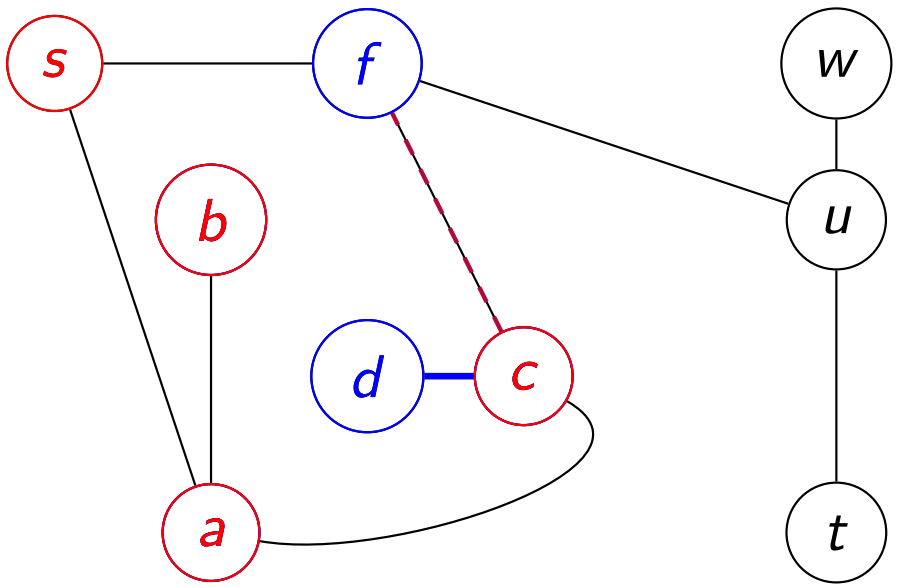
reached

a :	*True*
b :	*True*
c :	*True*
d :	
f :	*True*
s :	*True*
t :	
u :	
w :	

```
lnext =  [ 'f' ]
n =  'c'
```

Implémentation itérative : démo

⏩ Skip Demo



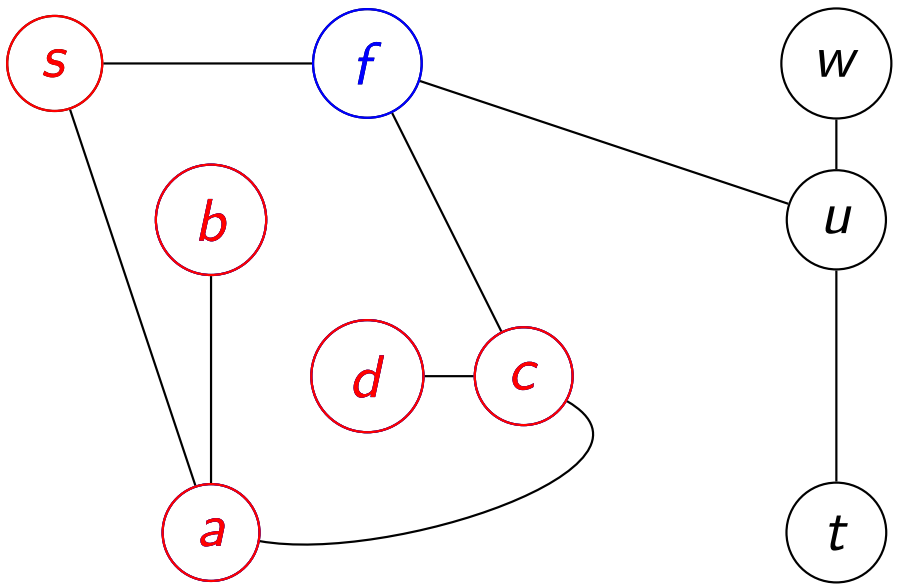
reached

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	
w :	

```
lnext =  [ 'f', 'd' ]
n =  'c'
```

Implémentation itérative : démo

⏩ Skip Demo



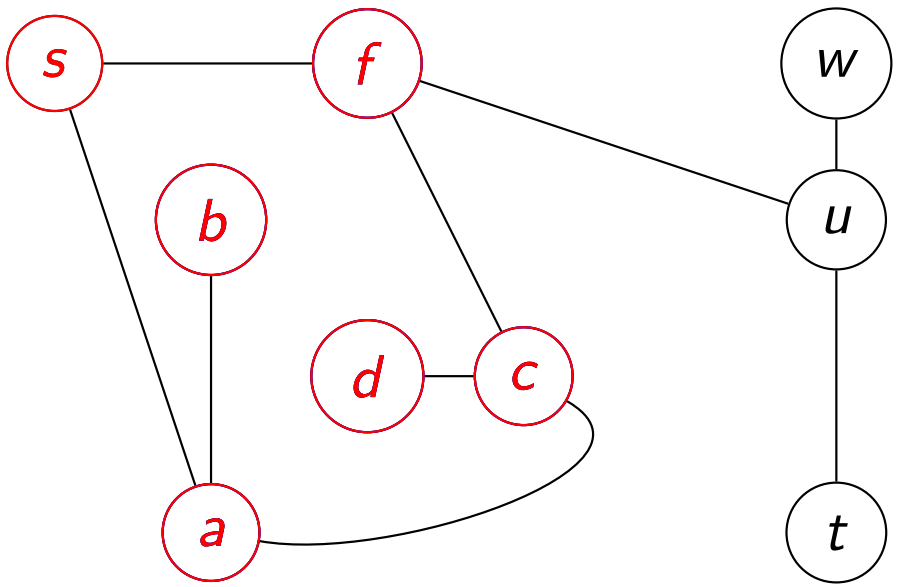
reached

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	
w :	

```
lnext =  [ 'f' ]
n =  'd'
```

Implémentation itérative : démo

⏩ Skip Demo



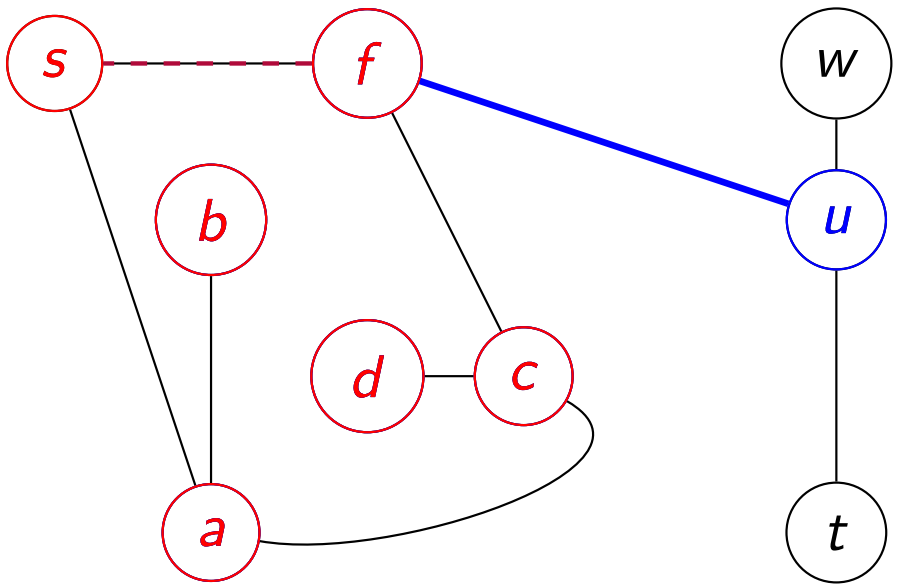
reached

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	
w :	

```
lnext = []
n = 'f'
```

Implémentation itérative : démo

⏩ Skip Demo



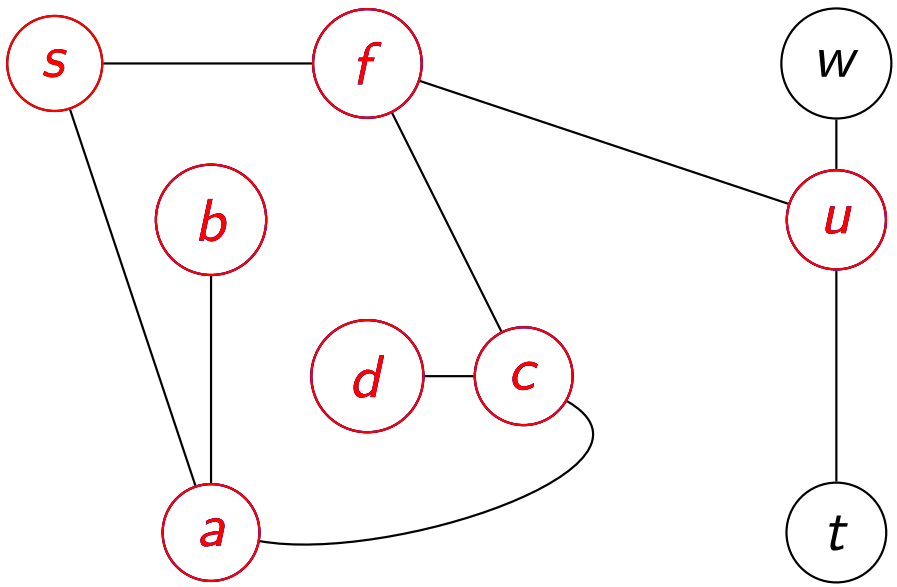
reached

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	*True*
w :	

```
lnext =  [ 'u' ]
n =  'f'
```

Implémentation itérative : démo

Skip Demo



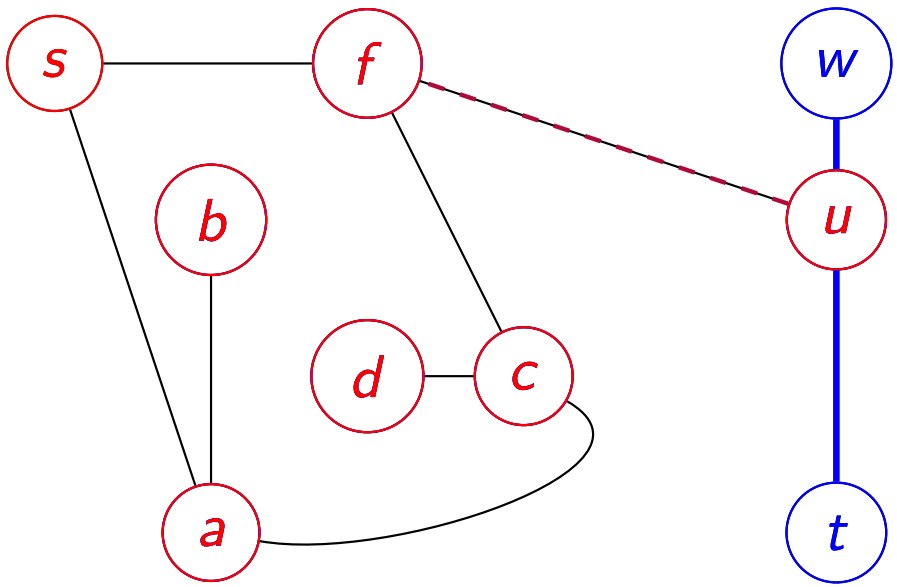
reached

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	*True*
w :	

```
lnext = []
n = 'u'
```

Implémentation itérative : démo

⏩ Skip Demo

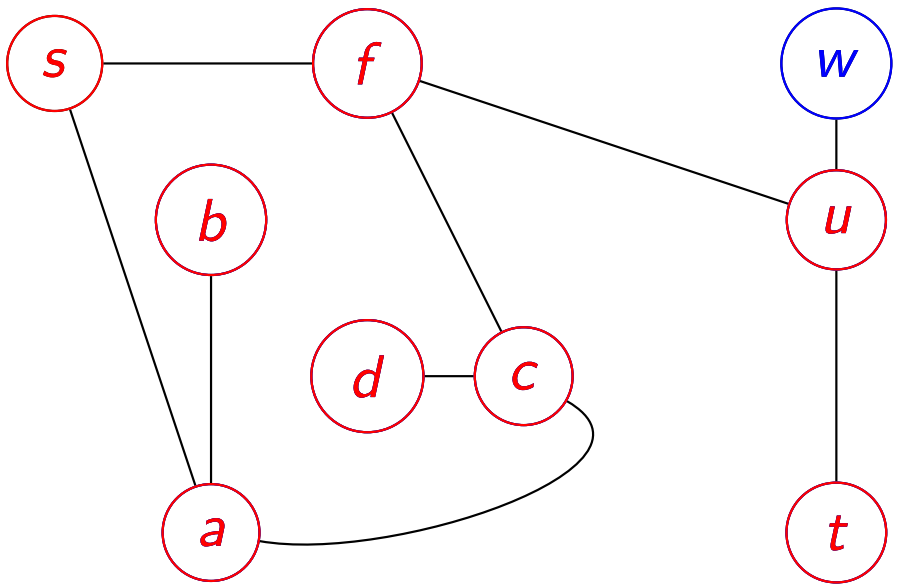


reached

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	*True*
u :	*True*
w :	*True*

```
lnext =  [ 'w', 't' ]
n =  'u'
```

Implémentation itérative : démo



reached

a : *True*
b : *True*
c : *True*
d : *True*
f : *True*
s : *True*
t : *True*
u : *True*
w : *True*

```
lnext =  [ 'w' ]
n =  't'
```

Implémentation itérative : remarques

Ordre des voisins

Pour obtenir le même parcours que précédemment, nous avons ajouté les voisins dans l'ordre lexical inverse !

Implémentation itérative : remarques

Ordre des voisins

Pour obtenir le même parcours que précédemment, nous avons ajouté les voisins dans l'ordre lexical inverse !

➜ **Exercice :** Exécuter l'algorithme en respectant l'ordre lexical des voisins

Implémentation itérative : remarques

Ordre des voisins

Pour obtenir le même parcours que précédemment, nous avons ajouté les voisins dans l'ordre lexical inverse !

➜ **Exercice :** Exécuter l'algorithme en respectant l'ordre lexical des voisins

Sommets visités/atteignables (la remarque de la version récursive est toujours valable)

Les sommets à `True` dans `reached` sont atteignables depuis s.

Plan

1 Problèmes de graphes

2 Parcours en profondeur

3 Parcours en largeur
- Principe
- Algorithme

4 Complexité

5 Connexité

6 Conclusion

Parcours en largeur

Principe

- Visiter les nœuds par ordre de proximité avec le sommet de départ.
- Implémentation en utilisant une file à la place de la pile.
- Difficilement adaptable de manière récursive.

Définition d'une file

- Structure de données informatique.
- Séquence d'éléments dans laquelle on ajoute d'un côté et on retire de l'autre.
- Les objets sortent dans l'ordre de l'entrée (First In First Out).

Files en Python

En informatique

Méthodes `enqueue` et `dequeue` sur une *file* (`queue` en anglais)

En Python

Le type `list` supporte les méthodes `append` et `pop(0)`

➜ en précisant de retirer au début !

En algo pour garder l'intuition

Deux fonctions pour « masquer » l'implémentation :

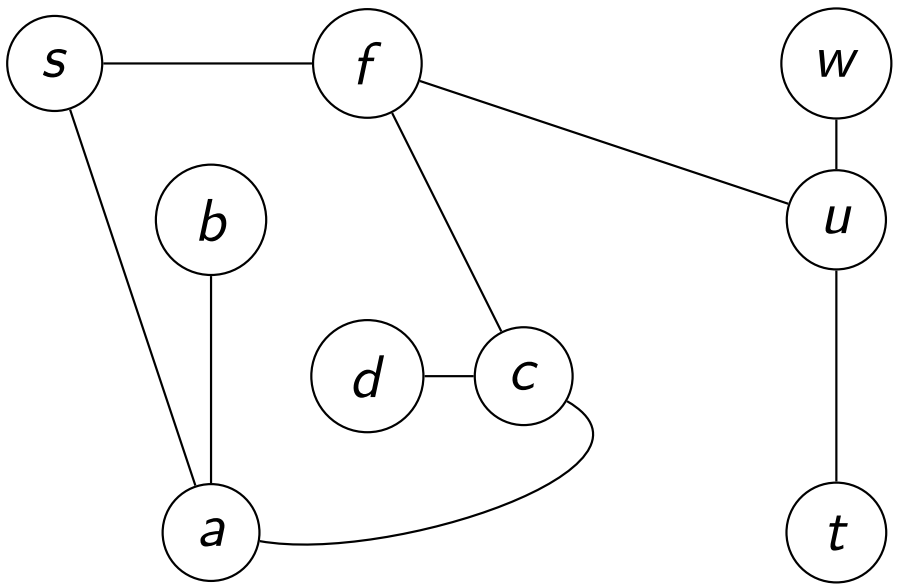
```
def add_end(x,l):
    l.append(x)

def pop_begin(l):
    return l.pop(0)
```

Exploration en largeur

```
def BFS(V,E,s,t):
    lnext = [s]                    # la file
    reached = { s : True }
    while len(lnext)>0:
      n = pop_begin(lnext)
      if n==t:
        return True
      for v in neighbours(n,E):
        if not v in reached:
          reached[v] = True
          add_end(v,lnext)
    return False
```

Implémentation itérative : démo

⏩ Skip Demo



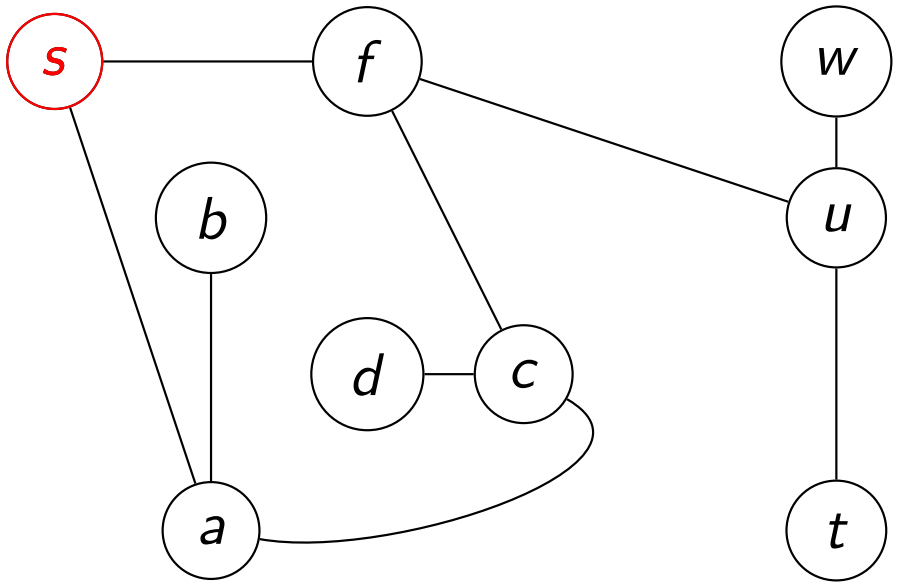
reached

a :
b :
c :
d :
f :
s : *True*
t :
u :
w :

```
lnext =  [ 's' ]
n =
```

Implémentation itérative : démo

⏩ Skip Demo



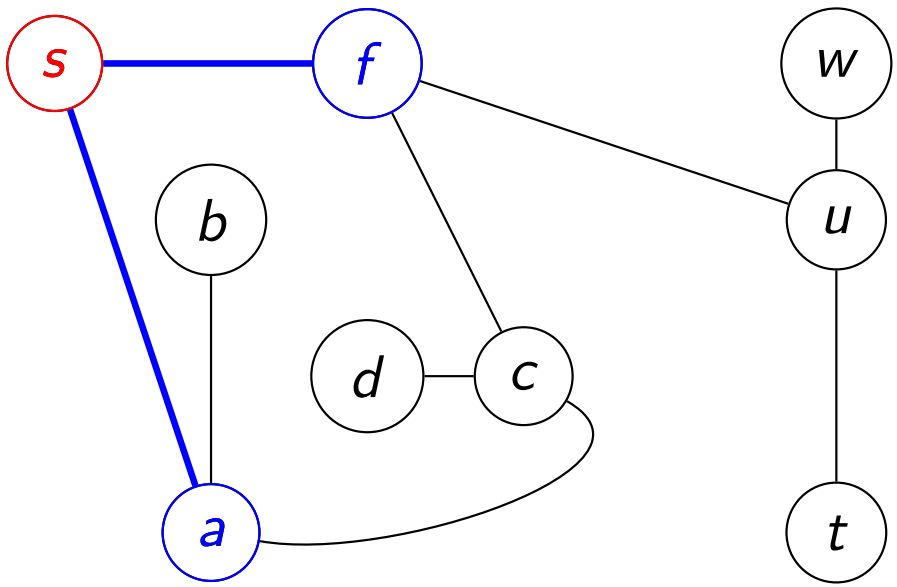
reached

a :
b :
c :
d :
f :
s : *True*
t :
u :
w :

```
lnext = [ ]
n = 's'
```

Implémentation itérative : démo

⏩ Skip Demo



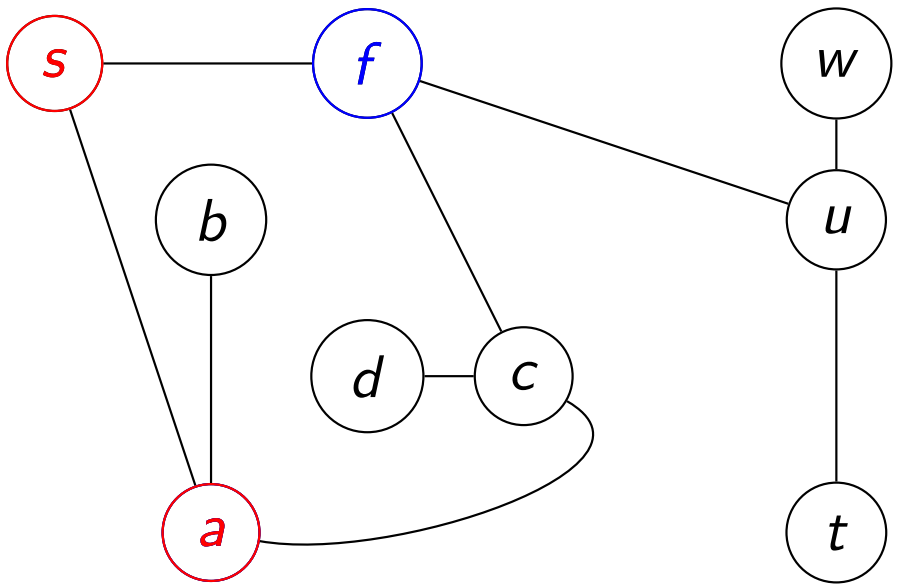
reached

a :	*True*
b :	
c :	
d :	
f :	*True*
s :	*True*
t :	
u :	
w :	

```
lnext =  [ 'a', 'f' ]
n =  's'
```

Implémentation itérative : démo

⏩ Skip Demo



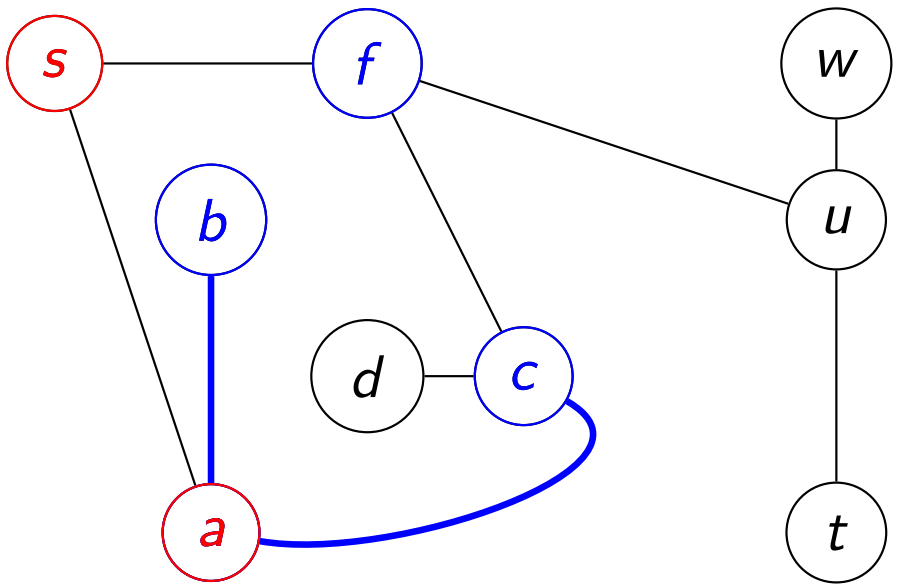
reached

a : *True*
b :
c :
d :
f : *True*
s : *True*
t :
u :
w :

```
lnext =  [ 'f' ]
n =  'a'
```

Implémentation itérative : démo

⏩ Skip Demo



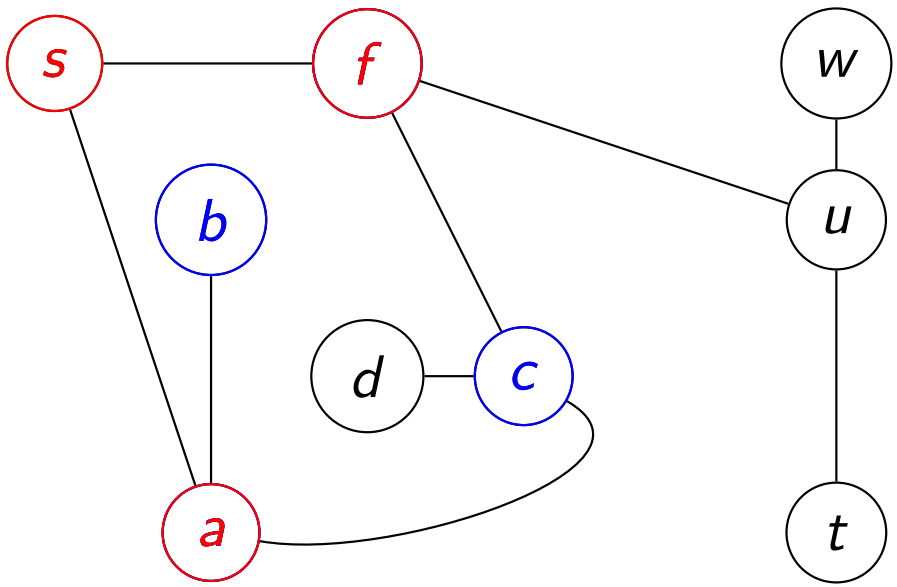
reached

a : *True*
b : *True*
c : *True*
d :
f : *True*
s : *True*
t :
u :
w :

```
lnext =  [ 'f', 'b', 'c' ]
n =  'a'
```

Implémentation itérative : démo

⏩ Skip Demo



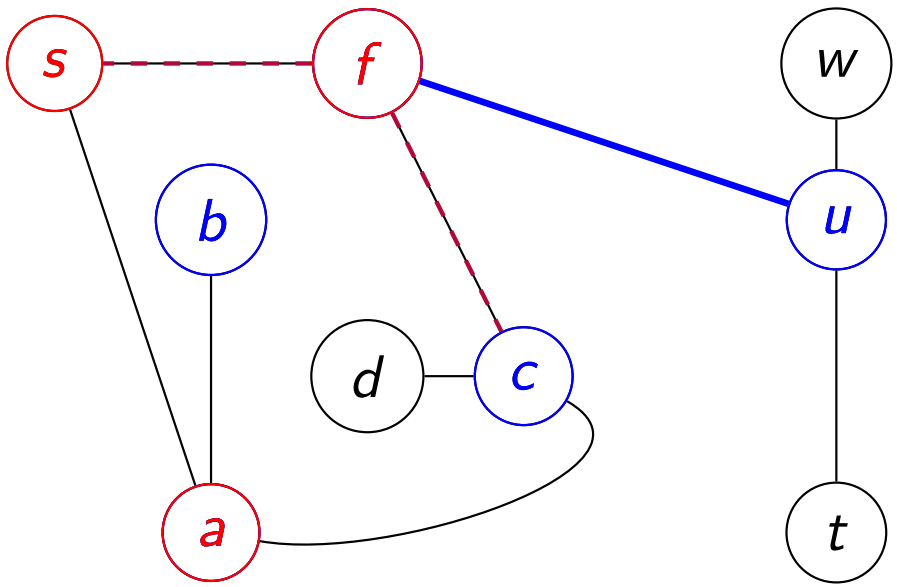
reached

a :	*True*
b :	*True*
c :	*True*
d :	
f :	*True*
s :	*True*
t :	
u :	
w :	

```
lnext =  [ 'b', 'c' ]
n =  'f'
```

Implémentation itérative : démo

⏩ Skip Demo



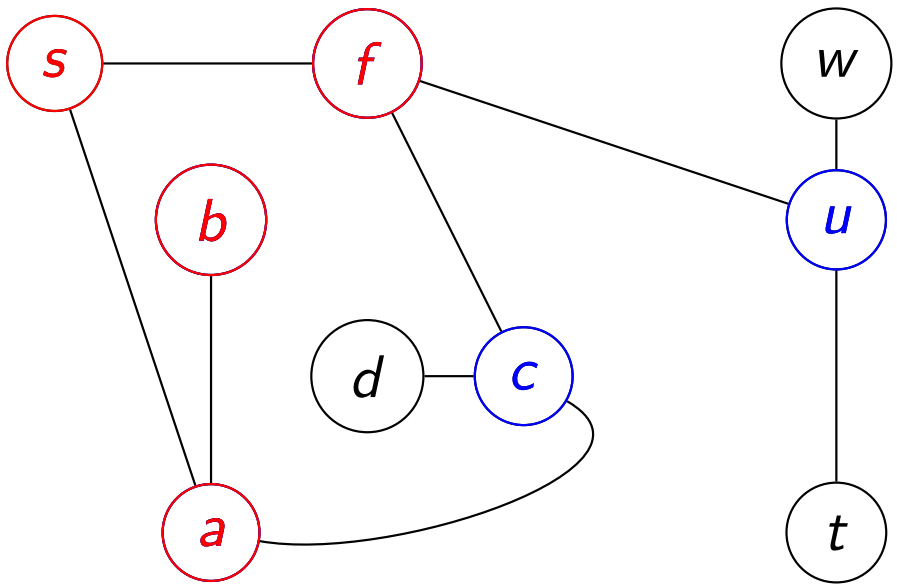
reached

a :	*True*
b :	*True*
c :	*True*
d :	
f :	*True*
s :	*True*
t :	
u :	*True*
w :	

```
lnext =  [ 'b', 'c', 'u' ]
n =  'f'
```

Implémentation itérative : démo

Skip Demo



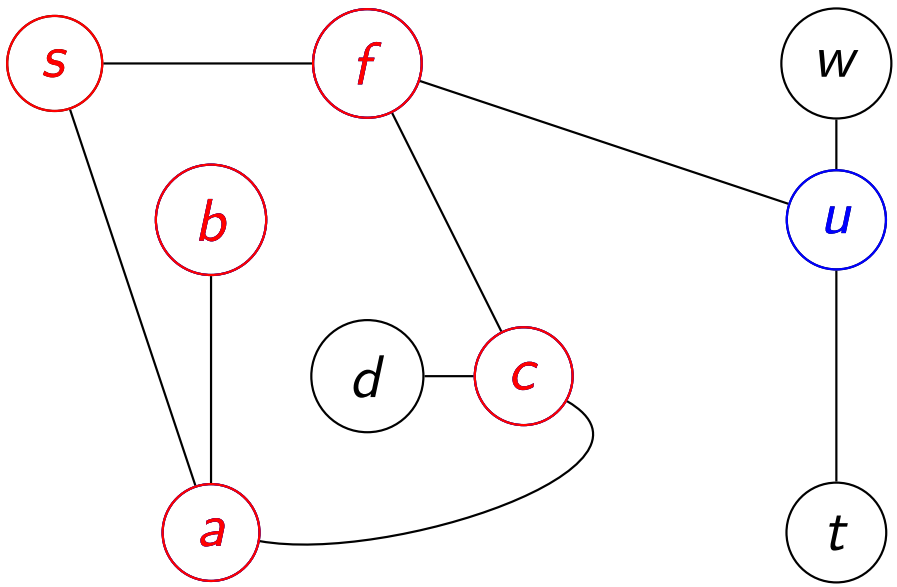
reached

a :	*True*
b :	*True*
c :	*True*
d :	
f :	*True*
s :	*True*
t :	
u :	*True*
w :	

```
lnext =  [ 'c', 'u' ]
n =  'b'
```

Implémentation itérative : démo

Skip Demo



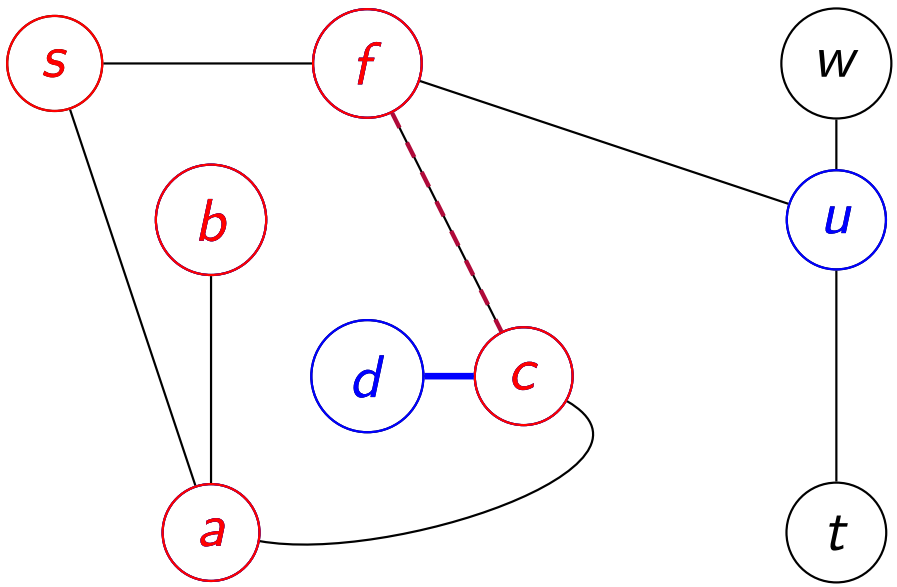
reached

a :	*True*
b :	*True*
c :	*True*
d :	
f :	*True*
s :	*True*
t :	
u :	*True*
w :	

```
lnext =  [ 'u' ]
n =  'c'
```

Implémentation itérative : démo

Skip Demo



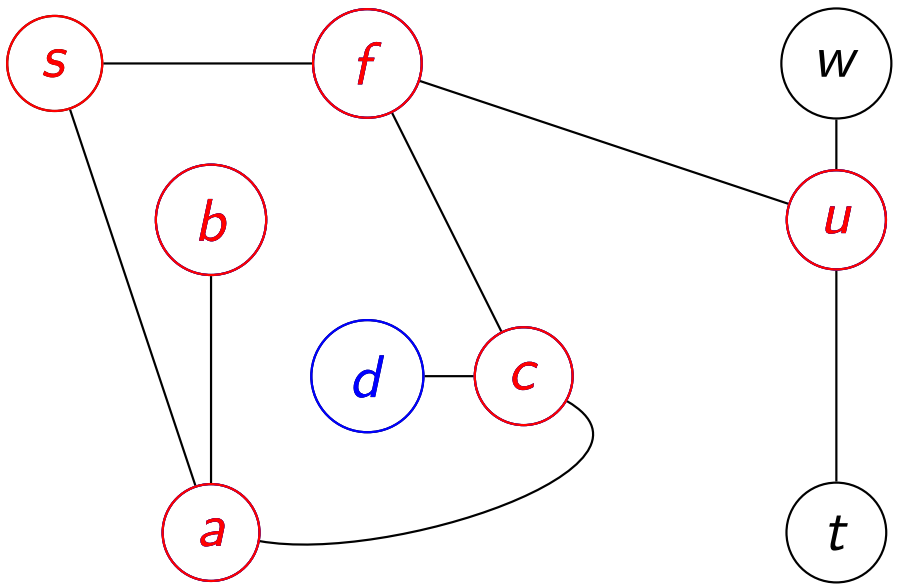
reached

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	*True*
w :	

```
lnext =  [ 'u', 'd' ]
n =  'c'
```

Implémentation itérative : démo

⏩ Skip Demo



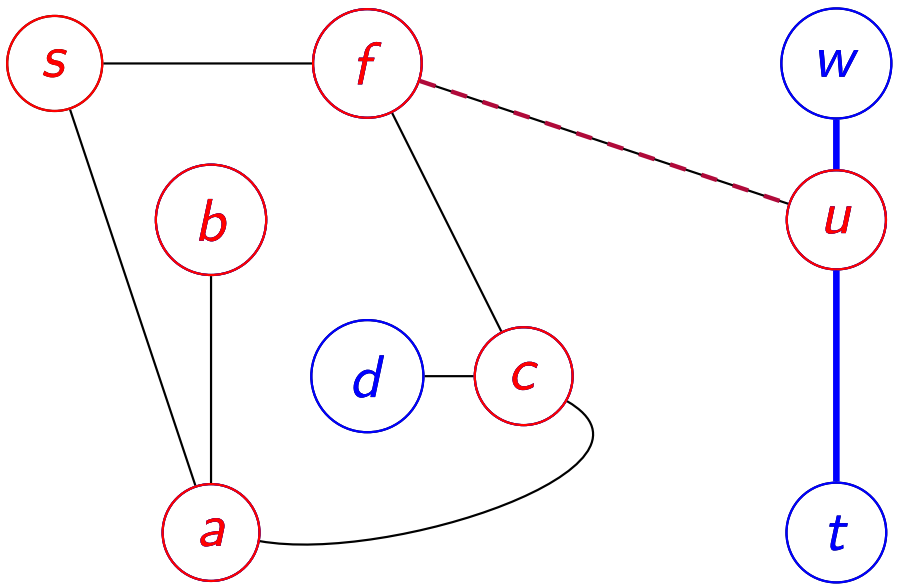
reached

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	*True*
w :	

```
lnext =  [ 'd' ]
n =  'u'
```

Implémentation itérative : démo

Skip Demo



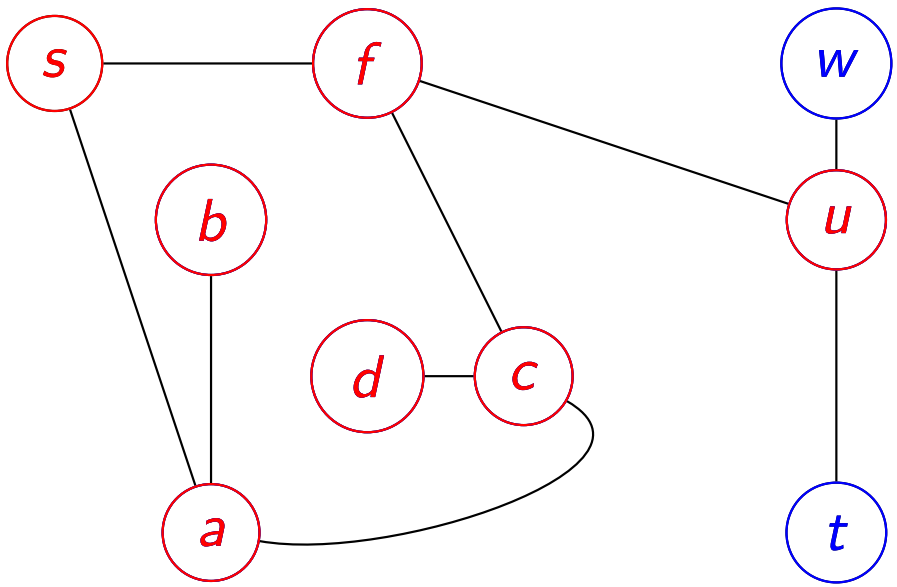
reached

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	*True*
u :	*True*
w :	*True*

```
lnext =  [ 'd', 't', 'w' ]
n =  'u'
```

Implémentation itérative : démo

Skip Demo

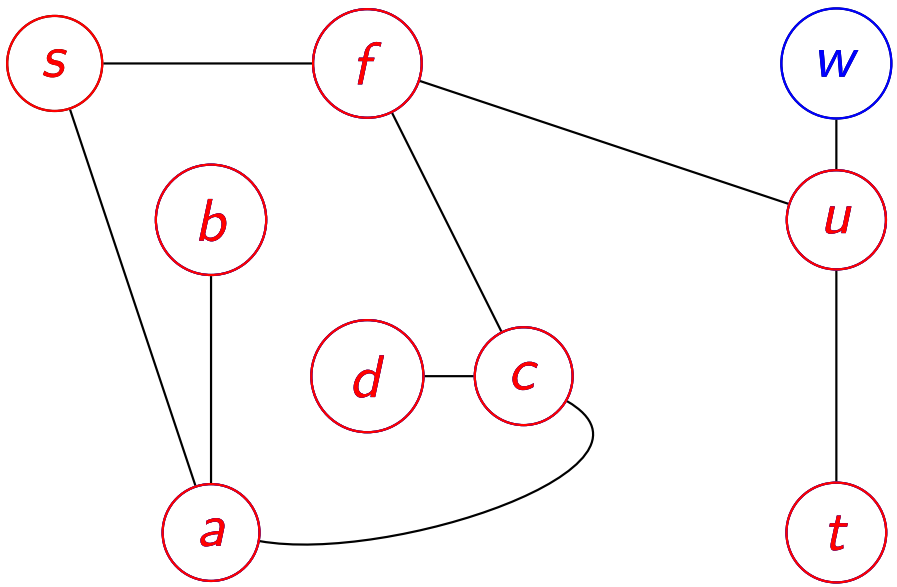


reached

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	*True*
u :	*True*
w :	*True*

```
lnext =  [ 't', 'w' ]
n =  'd'
```

Implémentation itérative : démo



reached

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	*True*
u :	*True*
w :	*True*

```
lnext =  [ 'w' ]
n =  't'
```

Plan

1 Problèmes de graphes

2 Parcours en profondeur

3 Parcours en largeur

4 Complexité
- Principe
- Complexité de parcours itératif
- Complexité avec structure de données

5 Connexité

6 Conclusion

Comment évaluer la performance de l'algorithme ?

Analyse de complexité

L'analyse de la complexité d'un algorithme consiste en l'étude de la quantité de ressources (temps et espace) nécessaire à l'exécution de cet algorithme.

Attention

Ne pas confondre avec *la théorie de la complexité*, qui elle étudie la difficulté intrinsèque des problèmes (vue plus tard en cours).

Comment évaluer la performance de l'algorithme ?

Analyse de complexité

L'analyse de la complexité d'un algorithme consiste en l'étude de la quantité de ressources (temps et espace) nécessaire à l'exécution de cet algorithme.

Attention

Ne pas confondre avec *la théorie de la complexité*, qui elle étudie la difficulté intrinsèque des problèmes (vue plus tard en cours).

Utilisation

Comparer des algorithmes sans dépendre de l'implémentation, du processeur, de la mémoire, du langage...

Calcul de la complexité

- La complexité d'un algorithme est calculée en fonction de la taille d'une instance.

  Nombre d'éléments dans une liste, de nœuds et d'arêtes dans un graphe. . .

Calcul de la complexité

- La complexité d'un algorithme est calculée en fonction de la taille d'une instance.

  Nombre d'éléments dans une liste, de nœuds et d'arêtes dans un graphe...

- On comptabilise le nombre d'opérations élémentaires, *i.e.* dont le coût **ne** dépend **pas** de la taille de l'instance.

```
def contains(T,x):
    i = 0
    while i<len(T) and T[i]!=x:
        i = i + 1
    return i<len(T)
```

$\Rightarrow$ au minimum 0 additions et 2 comparaisons (si `T` est vide)

au maximum n add et $2n+2$ cmp avec `n=len(T)`

Calcul de la complexité

- La complexité d'un algorithme est calculée en fonction de la taille d'une instance.

  Nombre d'éléments dans une liste, de nœuds et d'arêtes dans un graphe...

- On comptabilise le nombre d'opérations élémentaires, *i.e.* dont le coût **ne** dépend **pas** de la taille de l'instance.

```
def contains(T,x):
    i = 0
    while i<len(T) and T[i]!=x:
        i = i + 1
    return i<len(T)
```

  $\Rightarrow$ au minimum 0 additions et 2 comparaisons (si `T` est vide)

  au maximum n add et $2n+2$ cmp avec `n=len(T)`

- C'est une mesure asymptotique, le plus souvent une majoration, ici $\mathcal{O}(n)$.

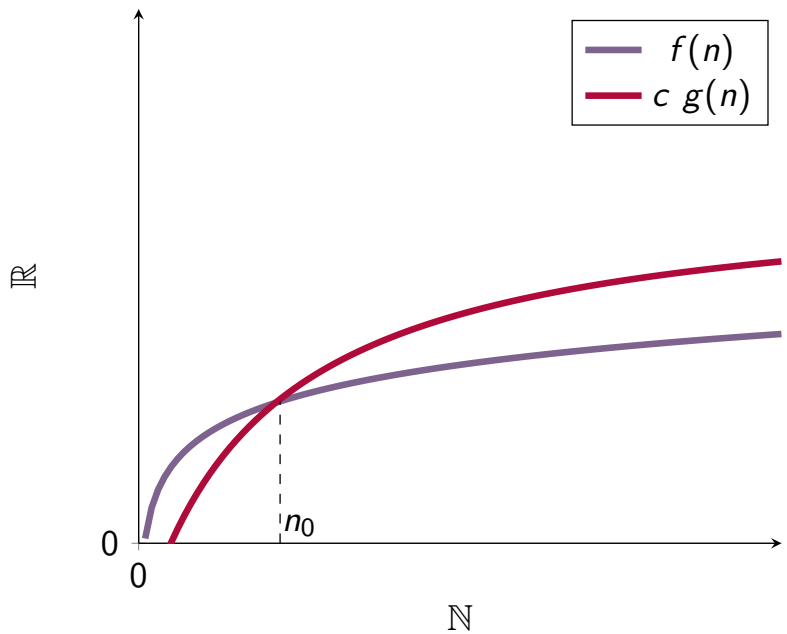
Majoration asymptotique

Définition : majoration asymptotique

Une fonction $f : \mathbb{N} \to \mathbb{R}$ est majorée asymptotiquement par une fonction $g : \mathbb{N} \to \mathbb{R}$ si et seulement si :

- $\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n > n_0 \qquad f(n) \leq c\ g(n)$

On note : $f \in \mathcal{O}(g)$



Complexité au pire de DFS_iter ou BFS_iter

```
def DFS_iter(V,E,s,t):
    lnext = [s]                                  O(1)
    reached = { s: True }                        O(1)
    while len(lnext)>0:                          ×?
      n = pop_...(lnext)                         O(1)
      if n==t:                                   O(1)
        return True
      for v in neighbours(n,E):                  ×?
        if not v in reached:                     O(1)
          reached[v] = True                      O(1)
          add_end(v,lnext)                       O(1)
    return False
```

- On sait faire `add_end`, `pop_begin` et `pop_end` en $\mathcal{O}(1)$
- L'existence/écriture dans un dictionnaire est aussi en $\mathcal{O}(1)$

Complexité au pire de DFS_iter ou BFS_iter

```
def DFS_iter(V,E,s,t):
    lnext = [s]                                  O(1)
    reached = { s: True }                        O(1)
    while len(lnext)>0:                          ×a
      n = pop_...(lnext)                         O(1)×a
      if n==t:                                   O(1)×a
        return True
      for v in neighbours(n,E):                  ×b
        if not v in reached:                     O(1)×b
          reached[v] = True                      O(1)×b
          add_end(v,lnext)                       O(1)×b
    return False
```

- On sait faire `add_end`, `pop_begin` et `pop_end` en $\mathcal{O}(1)$
- L'existence/écriture dans un dictionnaire est aussi en $\mathcal{O}(1)$
- Combien de tours de boucle : a ?
  Combien de passages sur chaque opération de l'intérieur : b ?

Complexité au pire de DFS_iter ou BFS_iter

```
def DFS_iter(V,E,s,t):
    lnext = [s]                                  O(1)
    reached = { s: True }                        O(1)
    while len(lnext)>0:                          ×|V|
      n = pop_...(lnext)                         O(1)×|V|
      if n==t:                                   O(1)×|V|
        return True
      for v in neighbours(n,E):                  ×b
        if not v in reached:                     O(1)×b
          reached[v] = True                      O(1)×b
          add_end(v,lnext)                       O(1)×b
    return False
```

- On sait faire `add_end`, `pop_begin` et `pop_end` en $\mathcal{O}(1)$
- L'existence/écriture dans un dictionnaire est aussi en $\mathcal{O}(1)$
- Combien de tours de boucle : `a` ?
  Combien de passages sur chaque opération de l'intérieur : `b` ?
  $a \rightarrow$ au pire $|V|$ fois si tous les sommets sont ajoutés dans `lnext`

Complexité au pire de DFS_iter ou BFS_iter

```
def DFS_iter(V,E,s,t):
    lnext = [s]                                  O(1)
    reached = { s: True }                        O(1)
    while len(lnext)>0:                          ×|V|
      n = pop_...(lnext)                         O(1)×|V|
      if n==t:                                   O(1)×|V|
        return True
      for v in neighbours(n,E):                  ×|E|
        if not v in reached:                     O(1)×|E|
          reached[v] = True                      O(1)×|E|
          add_end(v,lnext)                       O(1)×|E|
    return False
```

- On sait faire `add_end`, `pop_begin` et `pop_end` en $\mathcal{O}(1)$
- L'existence/écriture dans un dictionnaire est aussi en $\mathcal{O}(1)$
- Combien de tours de boucle : `a` ?
  Combien de passages sur chaque opération de l'intérieur : `b` ?
  $a \rightarrow$ au pire $|V|$ fois si tous les sommets sont ajoutés dans `lnext`
  $b \rightarrow$ autant d'ajouts que d'arêtes ! comme on n'ajoute que les voisins.

Complexité au pire de DFS_iter ou BFS_iter

```
def DFS_iter(V,E,s,t):
    lnext = [s]                                  O(1)
    reached = { s: True }                        O(1)
    while len(lnext)>0:                          ×|V|
      n = pop_...(lnext)                         O(1)×|V|
      if n==t:                                   O(1)×|V|
        return True
      for v in neighbours(n,E):                  ×|E|
        if not v in reached:                     O(1)×|E|
          reached[v] = True                      O(1)×|E|
          add_end(v,lnext)                       O(1)×|E|
    return False
```

- On sait faire `add_end`, `pop_begin` et `pop_end` en $\mathcal{O}(1)$
- L'existence/écriture dans un dictionnaire est aussi en $\mathcal{O}(1)$
- Combien de tours de boucle : a ?
  Combien de passages sur chaque opération de l'intérieur : b ?
  $a \rightarrow$ au pire $|V|$ fois si tous les sommets sont ajoutés dans `lnext`
  $b \rightarrow$ autant d'ajouts que d'arêtes ! comme on n'ajoute que les voisins.

➜ Complexité de l'algorithme en $\mathcal{O}(|V| + |E|) \approx \mathcal{O}(|E|)$ (si G suffisamment dense)

Structures de données

Attention

Cela dépend de l'implémentation !

Exemple : *Vérifier si l'élément est dans la liste*

```
if not v in reached
```

- Avec une liste : $\mathcal{O}(|liste|)$
- Avec un dictionnaire : $\mathcal{O}(1)$

  ➜ *Voir TP semaine prochaine...*

Structures de données

Attention

Cela dépend de l'implémentation !

Exemple : *Vérifier si l'élément est dans la liste*

```
if not v in reached
```

- Avec une liste : $\mathcal{O}(|liste|)$
- Avec un dictionnaire : $\mathcal{O}(1)$

  ➜ *Voir TP semaine prochaine...*

Tout est important !

✓ Liste pour les successeurs

✓ Dictionnaire pour les sommets visités (`reached`)

➜ Et pour le graphe (arêtes E) ?

Quelle(s) implémentation(s) pour les graphes ?

Structure de données

Un graphe est une structure de donnée abstraite.

➜ Comment s'implémente-t-elle ?

- Quelle représentation pour les sommets et les arêtes ?
- Quelle structure de données regroupe les sommets et les arêtes ?

➜ Quelles conséquences sur la complexité des algorithmes ?

Quelle(s) implémentation(s) pour les graphes ?

Structure de données

Un graphe est une structure de donnée abstraite.

➜ Comment s'implémente-t-elle ?

- Quelle représentation pour les sommets et les arêtes ?
- Quelle structure de données regroupe les sommets et les arêtes ?

➜ Quelles conséquences sur la complexité des algorithmes ?

Plusieurs implémentations possibles

1. Listes d'adjacence
2. Matrices d'adjacence
3. Matrices d'incidence (que nous ne verrons pas)

Représentation 1 : liste d'adjacence

Principe

Représenter en mémoire la fonction de voisinage (souvent notée Γ) :

$$\Gamma : V \to \mathcal{P}(V), \qquad x \mapsto \Gamma(x) = \{y \in V \mid (x, y) \in E\}$$

sous la forme d'une table d'associations (sommet,voisins).

Représentation 1 : liste d'adjacence

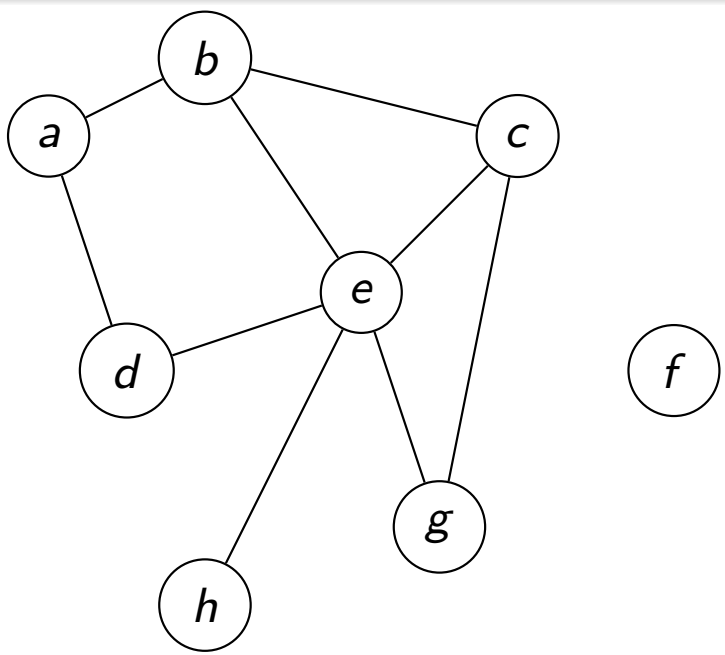
Principe

Représenter en mémoire la fonction de voisinage (souvent notée Γ) :

$$\Gamma : V \to \mathcal{P}(V), \qquad x \mapsto \Gamma(x) = \{y \in V \mid (x, y) \in E\}$$

sous la forme d'une table d'associations (sommet,voisins).

x	$\Gamma(x)$
a	$\{b, d\}$
b	$\{a, c, e\}$
c	$\{b, e, g\}$
d	$\{a, e\}$
e	$\{b, c, d, g, h\}$
f	$\{\}$
g	$\{c, e\}$
h	$\{e\}$



Représentation 1 : liste d'adjacence

- Encombrement mémoire en $\mathcal{O}(|E| + |V|)$
- Parcours des voisins d'un sommet u en $\mathcal{O}(deg(u))$ [1]

  Utile, pour un parcours BFS/DFS, Dijkstra (Cours 2), Prim (Cours 3)...
- Facilité de stocker les arêtes (existence représentée par valeur 1)

  `{a:{b:1,c:1},...}`
- Vérifier l'existence d'une arête (u, v) en $\mathcal{O}(1)$ [2]
- Ajout d'une arête en $\mathcal{O}(1)$ [2]
- Suppression d'une arête en $\mathcal{O}(1)$ [2]

1. Degré d'un nœud = nombre d'arêtes adjacentes
   pire cas : $deg(u) = |V| - 1$ si u est connecté à tous les autres.
2. Voir TP sur les dictionnaires

Représentation 2 : Matrice d'adjacence

Principe

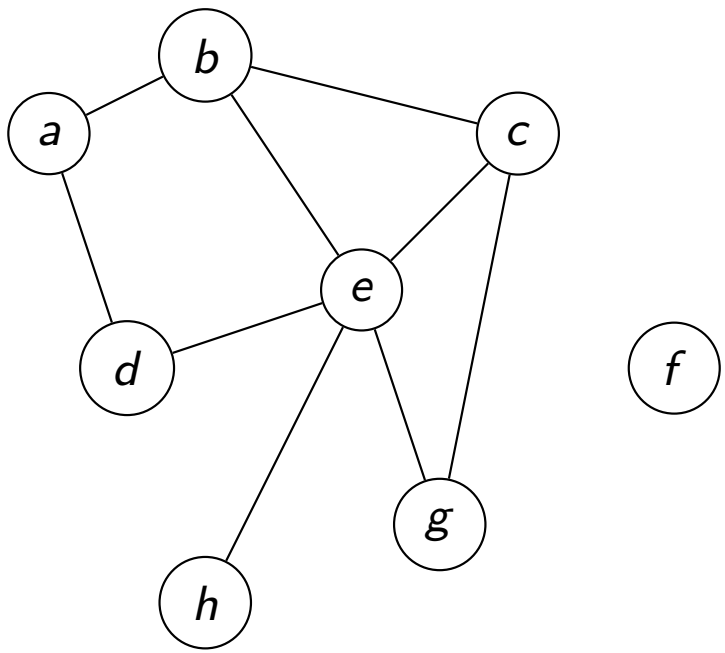
- Tableau à deux dimensions indexé sur l'ensemble des sommets $|V|$;
- $tab[i, j] = 1$ si les sommets sont reliés par une arête, 0 sinon.

Représentation 2 : Matrice d'adjacence

Principe

- Tableau à deux dimensions indexé sur l'ensemble des sommets $|V|$;
- $tab[i, j] = 1$ si les sommets sont reliés par une arête, 0 sinon.

	a	b	c	d	e	f	g	h
a	0	1	0	1	0	0	0	0
b	1	0	1	0	1	0	0	0
c	0	1	0	0	1	0	1	0
d	1	0	0	0	1	0	0	0
e	0	1	1	1	0	0	1	1
f	0	0	0	0	0	0	0	0
g	0	0	1	0	1	0	0	0
h	0	0	0	0	1	0	0	0



Représentation 2 : Matrice d'adjacence

- ✗ Encombrement mémoire en $\mathcal{O}(|V| \times |V|)$
- ✗ Parcours des voisins d'un sommet u en $\mathcal{O}(|V|)$

  il faut parcourir toute une ligne de la matrice...
- ✓ Vérifier l'existence d'une arête (u, v) en $\mathcal{O}(1)$

  Et cela s'écrit simplement `tab[i][j]` en Python !
- ✓ Ajout d'une arête en $\mathcal{O}(1)$
- ✓ Suppression d'une arête en $\mathcal{O}(1)$

Un exemple concret

Algorithme BFS avec deux représentations différentes

➜ seul la fonction de voisinage change !

```
def BFS(g,s, t, neighbours):
  lnext = [s]                  # la file
  reached = { s : True }
  while len(lnext)>0:
    n = pop_begin(lnext)
    if n==t:
      return True
    for v in neighbours(n,E):
      if not v in reached:
        reached[v] = True
        add_end(v,lnext)
  return False
```

```
def neighbours_mat(i,g):
  l=[]
  for j in range(len(g[i])):
    if g[i][j]:
      l.append(j)
  return l

def neighbours_list(i,g):
  return g[i]
```

sommets et indices sont confondus...

➜ Comparer le temps d'exécution de `BFS(mat,0,neighbours_mat)` et de `BFS(list,0,neighbours_list)` sur un graphe pas trop petit...

Complexité des algorithmes de parcours

Comment on itère sur les voisins ?

➜ La complexité de l'algorithme de parcours est sensible à l'implémentation du graphe !

Complexité des algorithmes de parcours

Comment on itère sur les voisins ?

➜ La complexité de l'algorithme de parcours est sensible à l'implémentation du graphe !

Matrice d'adjacence

Itérer sur les voisins nécessite $\mathcal{O}(|V|)$

➜ Complexité de l'algorithme en $\mathcal{O}(|V|^2)$

Complexité des algorithmes de parcours

Comment on itère sur les voisins ?

➜ La complexité de l'algorithme de parcours est sensible à l'implémentation du graphe !

Matrice d'adjacence

Itérer sur les voisins nécessite $\mathcal{O}(|V|)$

➜ Complexité de l'algorithme en $\mathcal{O}(|V|^2)$

Liste d'adjacence

Itérer sur les voisins de u nécessite $\mathcal{O}(deg(u))$

➜ Complexité de l'algorithme en $\mathcal{O}(|E|)$ car $2|E| = \sum_{u \in V} deg(u)$

À retenir sur l'exploration en profondeur et en largeur

- Deux algorithmes très similaires
- Permettent de déterminer s'il existe une chaîne entre deux sommets (décision)

  et d'en construire une le cas échéant (construction) → notebook TD1 !
- Permet de détecter des cycles *(lorsqu'un voisin est déjà marqué).*
- L'exploration en profondeur s'implémente de manière récursive ou de façon itérative avec une pile.
- L'exploration en largeur s'implémente de manière itérative avec une file.
- Complexité en temps de $\mathcal{O}(|E|)$ en théorie

  ($\mathcal{O}(|V|^2)$ avec matrice d'adjacence et $\mathcal{O}(|E|)$ avec liste d'adjacence).

À retenir sur l'exploration en profondeur et en largeur

- Deux algorithmes très similaires
- Permettent de déterminer s'il existe une chaîne entre deux sommets (décision)

  et d'en construire une le cas échéant (construction) → notebook TD1 !
- Permet de détecter des cycles *(lorsqu'un voisin est déjà marqué).*
- L'exploration en profondeur s'implémente de manière récursive ou de façon itérative avec une pile.
- L'exploration en largeur s'implémente de manière itérative avec une file.
- Complexité en temps de $\mathcal{O}(|E|)$ en théorie

  ($\mathcal{O}(|V|^2)$ avec matrice d'adjacence et $\mathcal{O}(|E|)$ avec liste d'adjacence).

Composante connexe

Les deux algorithmes (DFS et BFS) permettent aussi de calculer une composante connexe...

Plan

1 Problèmes de graphes

2 Parcours en profondeur

3 Parcours en largeur

4 Complexité

5 Connexité
- Composante connexe
- Algorithme
- Application

6 Conclusion

Composante connexe

Remarque

Si t n'est pas atteignable depuis s, l'algorithme renvoie la valeur False.

Composante connexe

Remarque

Si t n'est pas atteignable depuis s, l'algorithme renvoie la valeur False.

➜ La table `reached` permet alors de connaître l'ensemble des nœuds atteignables depuis s. On parle de composante connexe.

Définition : composante connexe

Dans un graphe $G = (V, E)$, un sous-ensemble maximal $V' \subseteq V$ de sommets forme une composante connexe de G s'il existe une chaîne entre tout couple de sommets de V'.

Composante connexe

Remarque

Si t n'est pas atteignable depuis s, l'algorithme renvoie la valeur False.

➜ La table `reached` permet alors de connaître l'ensemble des nœuds atteignables depuis s. On parle de composante connexe.

Définition : composante connexe

Dans un graphe $G = (V, E)$, un sous-ensemble maximal $V' \subseteq V$ de sommets forme une composante connexe de G s'il existe une chaîne entre tout couple de sommets de V'.

Une idée ?

➜ Pourrait-on modifier BFS/DFS pour calculer une composante connexe

Composante connexe – Construction

Définition du problème

- Données : Un graphe $G = (V, E)$ et un sommet de départ $s \in V$
- Question : Construire la composante connexe contenant s

Composante connexe – Construction

Définition du problème

- Données : Un graphe $G = (V, E)$ et un sommet de départ $s \in V$
- Question : Construire la composante connexe contenant s

Solution

Il suffit de reprendre notre algorithme de parcours (en largeur ou en profondeur) sans s'arrêter sur un sommet donné mais jusqu'à épuisement de la file/pile.

Recherche de composante connexe : BFS

```
def BFS_connex(V,E,s):
    lnext = [s]
    reached = {s}
    while len(lnext)>0:
        n = pop_begin(lnext)
        # on retire le test n==t
        for m in neighbours(n,E):
            if m not in reached:
                reached.add(m)
                lnext.append(m)
    # on retourne la liste des noeuds atteignables
    return reached
```

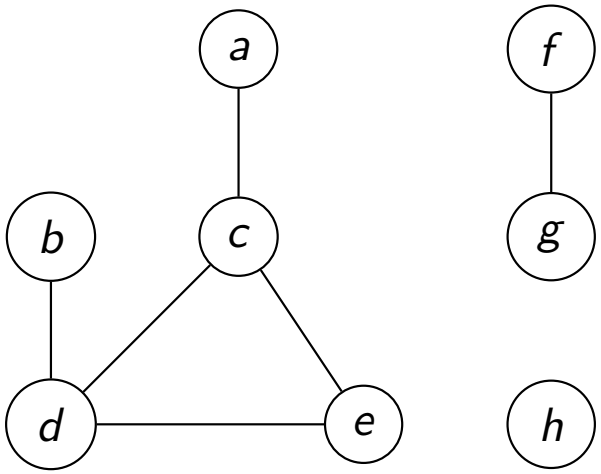
Recherche de composante connexe : BFS

```
def BFS_connex(V,E,s):
    lnext = [s]
    reached = {s}
    while len(lnext)>0:
        n = pop_begin(lnext)
        # on retire le test n==t
        for m in neighbours(n,E):
            if m not in reached:
                reached.add(m)
                lnext.append(m)
    # on retourne la liste des noeuds atteignables
    return reached
```

Exercice

Modifiez les algorithmes DFS et DFS_iter vus précédemment pour calculer la composante connexe d'un graphe.

Identifier toutes les composantes connexes d'un graphe



Définition du problème (construction)

- Données : Étant donnée un graphe $G = (V, E)$
- Question : Construire une structure de données qui associe chaque sommet de V à un entier tel que deux sommets différents s et t sont associés à la même valeur si et seulement si ils sont dans la même composante connexe.

Algorithme pour identifier les composantes

```
def ident_CC(V,E):
    res={v:-1 for v in V}
    i=0
    for v in res:
        if res[v]==-1:
            cc=BFS_connex(V,E,v)
            for c in cc:
                res[c]=i
            i = i+1
    return res
```

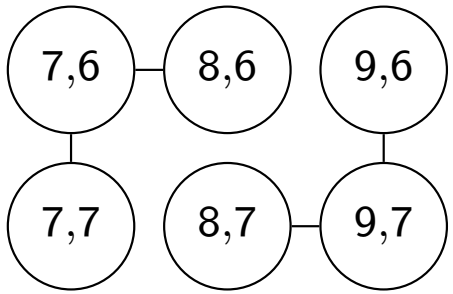
Algorithme pour identifier les composantes

```
def ident_CC(V,E):
    res={v:-1 for v in V}
    i=0
    for v in res:
        if res[v]==-1:
            cc=BFS_connex(V,E,v)
            for c in cc:
                res[c]=i
            i = i+1
    return res
```

La complexité en temps de cet algorithme est aussi $\mathcal{O}(|V| + |E|)$

➜ Chaque composante connexe n'est visitée qu'une fois !

Comptages des chromosomes



Algorithme d'identification des chromosomes

1. Charger une image en niveau de gris.
2. Appliquer un seuil pour obtenir une image en noir et blanc.
3. Transformer l'image en un graphe ou chaque pixel blanc correspond à un sommet et où il existe une arrête entre deux sommets si ces derniers sont adjacents et tous les deux blancs.
4. Utiliser l'algorithme d'identification

Plan

1 Problèmes de graphes

2 Parcours en profondeur

3 Parcours en largeur

4 Complexité

5 Connexité

6 Conclusion

Ce qu'il faut retenir

- Définition d'un graphe, notation
- Problème de décision, de construction et d'optimisation
- Instance de problème
- Définition d'un algorithme
- Parcours en largeur et en profondeur
  - Algorithme général
  - Implémentation avec une file ou une pile
  - Propriétés
- Complexité
  - Complexité général
  - Complexité avec liste d'adjacence
  - Complexité avec matrice d'adjacence
- Application à un problème pratique