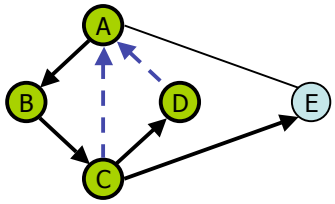
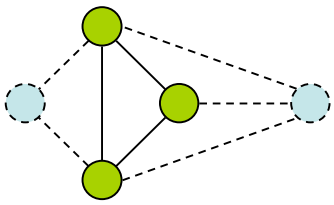
Traversées (Parcours) de graphes

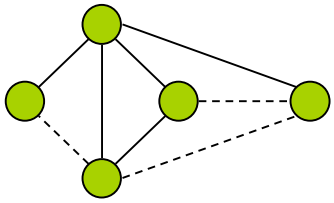


Sous-graphes

- **Un sous-graphe S d'un graphe G est un graphe tel que**
  - Les sommets de S forment un sous-ensemble des sommets de G
  - Les arêtes de S forment un sous-ensemble des arêtes de G
- **Un sous-graphe couvrant d'un graphe G est un sous-graphe contenant tous les sommets de G**



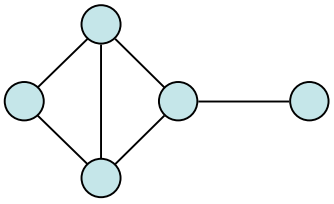
sous-graphe



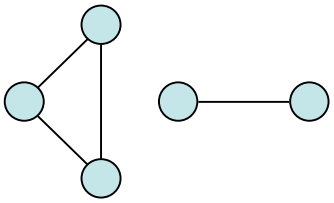
Un sous-graphe couvrant

Connexité

- Un graphe est **connexe** s'il existe un chemin entre chaque paire de sommets.
- Une **composante connexe** d'un graphe *G* est un sous-graphe *G'=(V',E')* connexe maximal (pour l'inclusion) : il n'est pas possible d'ajouter à *V'* d'autres sommets en conservant la connexité du sous-graphe.



graphe connexe

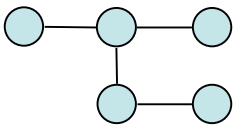


Un graphe non-connexe ayant deux composantes connexes

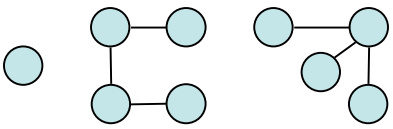
3

Arbres et forêts

- **Un arbre est un graphe non-orienté T tel que**
  - T est connexe
  - T est acyclique (sans cycles)
- **Une forêt est un graphe non-orienté acyclique (sans cycles)**
  - une collection des arbres
- **Les composantes connexes d'une forêt sont des arbres**



arbre

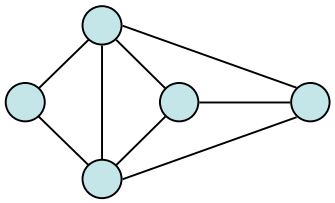


Forêt

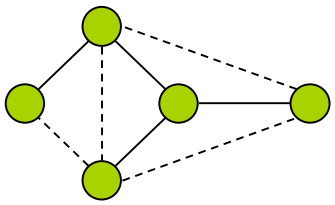
4

Arbres et forêts couvrants

- Un arbre couvrant d'un graphe connexe est un sous-graphe couvrant qui est un arbre
- Un arbre couvrant n'est pas unique sauf si le graphe est un arbre
- Arbres couvrant ont applications à design de réseaux informatiques
- Une forêt couvrante d'un graphe est un sous-graphe couvrant qui est une forêt



Graphe



Arbre couvrant

5

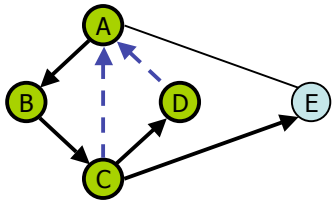
Traversées de graphes

Traversée d'un graphe G:

- Visite tous les sommets et toutes les arêtes de G
- Détermine si G est connexe ou non
- Trouve les composantes connexes de G
- Trouve un arbre/une forêt couvrante pour G

6

Parcours en profondeur:DFS (Depth-First Search)



7

Parcours en profondeur

Le parcours en profondeur est une technique pour traverser les graphes qui:

- Prends un temps en $O(n + m)$ pour un graphe avec n sommets et m arêtes
- peut être modifié pour résoudre d'autres problèmes sur les graphes
  - Trouver et retourner un chemin entre deux sommets donnés
  - Trouver un cycle dans le graphe

8

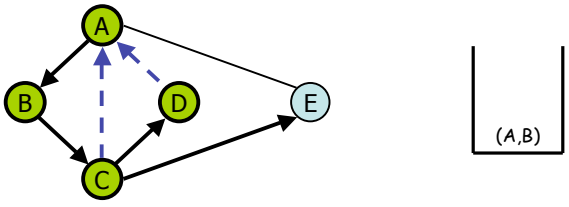
Parcours en profondeur

L'idée: recherche qui progresse à partir d'un sommet S en s'appelant récursivement pour chaque sommet voisin de S. Le nom d'algorithme en profondeur est dû au fait qu'il explore « à fond » les chemins un par un: pour chaque sommet, il prend le premier sommet voisin jusqu'à ce qu'un sommet n'aie plus de voisins (ou que tous ses voisins soient marqués), et revient alors au sommet père.

9

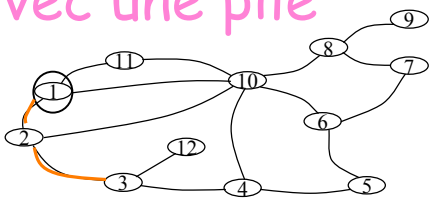
Algorithme DFS

avec une pile



10

Avec une pile



Visité : {1} T={}

à visiter:
(1,2)
(1,11)
(1,10)

POP → (1,2)
Visité : {1,2}

T={(1,2)}

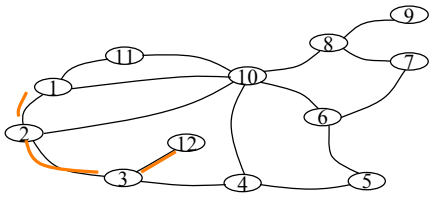
à visiter
(2,3)
(2,10)
(1,11)
(1,10)

POP → (2,3)
Visité : {1,2,3}

T={(1,2), (2,3)}

à visiter
(3,12)
(3,4)
(2,10)
(1,11)
(1,10)

11

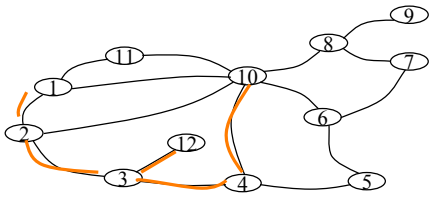


POP → (3,12)
Visité : {1,2,3,12}

T={(1,2), (2,3), (3,12)}

à visiter
(3,4)
(2,10)
(1,11)
(1,10)

12



POP → (3,4)

Visité : {1,2,3,12,4}

T={(1,2), (2,3), (3,12), (3,4)}

à visiter

(4,10)
(4,5)
(2,10)
(1,11)
(1,10)

POP → (4,10)

Visité : {1,2,3,12,4,10}

T={(1,2), (2,3), (3,12), (3,4), (4,10)}

à visiter

(10,8)
(10,6)
(4,5)
(2,10)
(1,11)
(1,10)

• • •

13

Complexité

Nombre de PUSH: $\sum_{v \in V} d(v) = 2m$

Nombre de POP: $\sum_{v \in V} d(v) = 2m$

Visite de noeuds: n

$O(n+m) = O(m)$

14

Algorithme DFS - récursif

Version simple

```
DFS(v)
Marque v visité
 ∀w ∈ Adj(v)
        Si w n'est pas visité
                T = T ∪ (v,w)
                DFS(w)
```

15

DFS encore - plus de détails…

- L'algorithme utilise un mécanisme pour placer et obtenir les "étiquettes" des sommets et arêtes

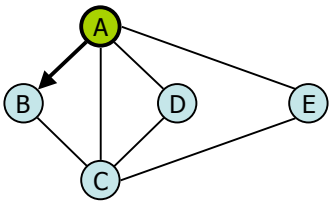
```
Algorithm DFS(G)
    Entrée graphe G
    Sortie étiquetage des arêtes de G
           comme arêtes découvertes et
           arêtes de retour
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        DFS(G, v)
```

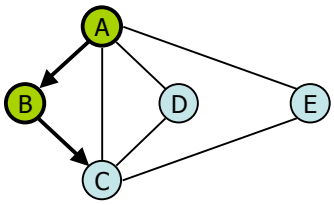
```
Algorithm DFS(G, v)
  Entrée graphe G et un sommet de
         début v de G
  Sortie étiquetage des arêtes de G dans
         le composant connexe de v
         comme arêtes découvertes et
         arêtes de retour
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        DFS(G, w)
      else
        setLabel(e, BACK)
```

16

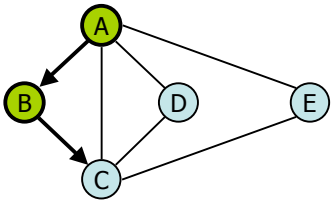
Exemple

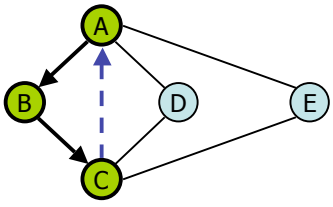

```
Algorithm DFS(G, v)
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        DFS(G, w)
      else
        setLabel(e, BACK)
```

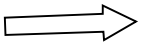


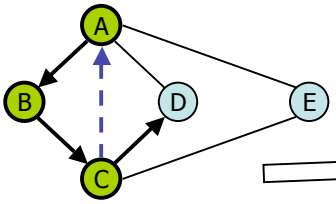


17



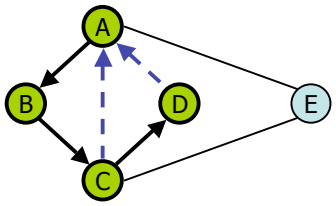




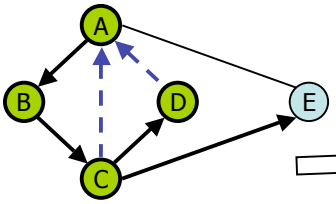


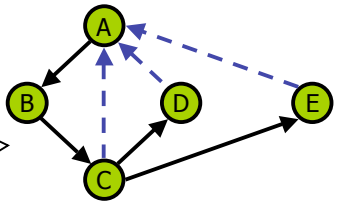

```
Algorithm DFS(G, v)
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        DFS(G, w)
      else
        setLabel(e, BACK)
```

18




```
Algorithm DFS(G, v)
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        DFS(G, w)
      else
        setLabel(e, BACK)
```

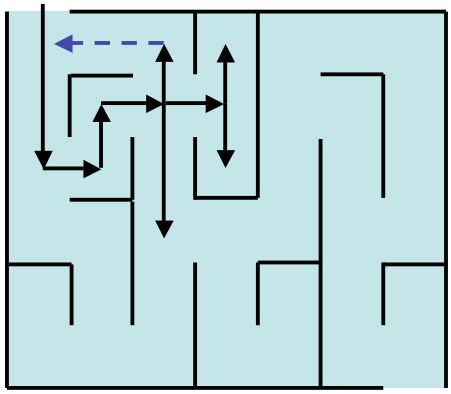




19

DFS et traversée de labyrinthe

- L'algorithme de DFS est similaire à une stratégie classique pour explorer un labyrinthe
  - Nous marquons chaque intersection, chaque coin et chaque cul de sac (sommet) visité
  - Nous marquons chaque couloir (arête) traversé
  - Nous prenons note du chemin de retour à l'entrée (le sommet du début) au moyen d'une corde (la pile de récursion)



20

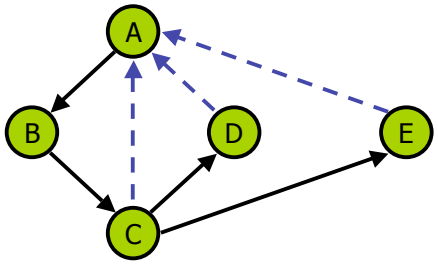
Propriétés du DFS

Propriété 1

DFS(G, v) visite tous les sommets et les arêtes dans le composant connexe de *v*

Propriété 2

Les arêtes découvertes par *DFS(G, v)* forment un arbre couvrant des composantes connexes de *v*



21

Complexité de DFS récursif + etiquetage

- Chaque sommet est étiquetté deux fois
  - Une fois "non-exploré"
  - Une fois "visité"

  2n
- Chaque arête est étiquettée deux fois
  - Une fois "non-explorée"
  - Une fois "découverte" ou "retour"

  2m
- L'opération *incidentEdges* est appelée une fois pour chaque sommet

$$\sum_{v \in V} d(v) = 2m$$

si ...

22

- La complexité total de DFS est donc $O(n + m)$ si le graphe est représenté avec une liste d'adjacence

$$O(n + m) = O(m)$$

Conclusion

Avec Liste d'adjacence:
O(m)

CAS PIRE: m = $O(n^2)$, when …

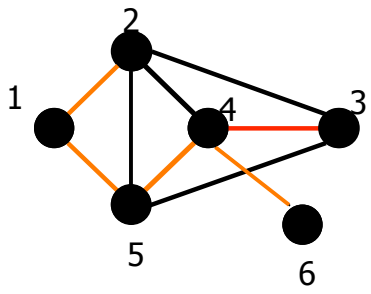
Question: Avec matrice d'adjacence ???

23

Chemin entre deux sommets

- Nous pouvons spécialiser l'algorithme de DFS pour trouver un chemin entre deux sommets donnés ***u*** et ***z***
- Nous appelons $DFS(G, u)$ avec ***u*** comme le sommet de début
- Nous utilisons une pile S pour prendre note du chemin entre le sommet de début et le sommet actuel
- Aussitôt que le sommet de destination z est rencontré, nous retournons le chemin comme éléments stockés dans la pile

24



2 -- 6

2 1 5 4 ~~3~~ 6

```
Algorithm pathDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  if v = z
    return S.elements()
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        S.push(e)
        pathDFS(G, w, z)
        S.pop(e)
      else
        setLabel(e, BACK)
  S.pop(v)
```

25

Trouver un cycle dans un graphe

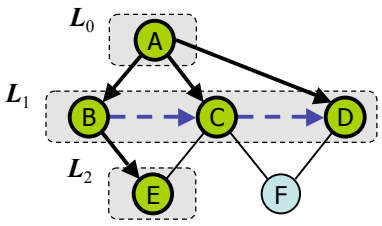
- Nous pouvons spécialiser l'algorithme de DFS pour trouver un cycle
- Nous utilisons une pile S pour prendre note du chemin entre le sommet de début *v* et le sommet actuel
- Aussitôt qu'une arête retour (*v*, *w*) est rencontrée, nous retournons le cycle comme la portion des éléments de la pile du sommet (top) au sommet *w*

26

```
Algorithm cycleDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      S.push(e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        pathDFS(G, w, z)
        S.pop(e)
      else
        T ← new empty stack
        repeat
          o ← S.pop()
          T.push(o)
        until o = w
        return T.elements()
  S.pop(v)
```

27

Parcours en largeur :BFS (Breadth-First Search)



28

Parcours en largeur

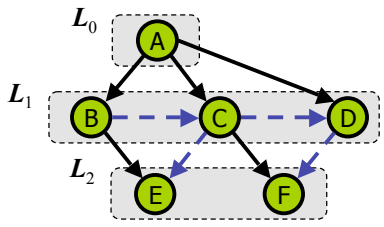
Le parcours en largeur est une technique pour traverser les graphes qui:

- prends temps $O(n + m)$ sur un graphe avec n sommets et m arêtes
- peut être modifié pour résoudre d'autres problèmes sur les graphes
  - Découvrir et retourner d'un chemin entre deux sommets donnés avec le minimum d'arêtes
  - Trouve un cycle simple dans le graphe s'il y en a un

29

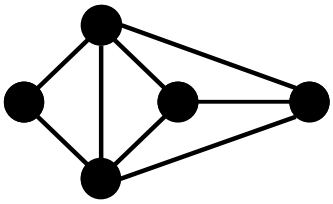
Parcours en largeur

L'idée: Cet algorithme liste d'abord les voisins de S pour ensuite les explorer un par un.



niveau par niveau

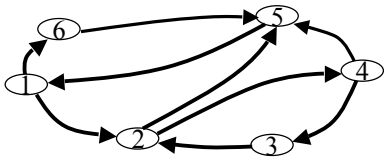
30



On pourrait utiliser une file
dans laquelle on prend le premier
sommet et place en dernier ses voisins encore
non-explorés.

31

Parcours en largeur avec une file



visité : {1}
T = ϕ

à visiter : {(1,2), (1,6)}

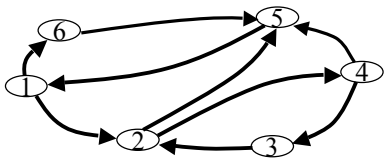
(1,2) – 2 est visité?
visité : {1,2}
T = {(1,2)}

à visiter : {(1,6), (2,4), (2,5)}

(1,6) - 6 est visité?
visité : {1,2,6}
T = {(1,2), (1,6)}

à visiter : {(2,4), (2,5), (6,5)}

32



(2,4) - 4 est visité?
visité : {1,2,6,4}
T = {(1,2), (1,6), (2,4)}
à visiter : {(2,5), (6,5), (4,5), (4,3)}

(2,5) -5 est visité?
visité : {1,2,6,4,5}
T = {(1,2), (1,6), (2,4), (2,5)}
à visiter : {(6,5), (4,5), (4,3)}

(6,5) - 5? déjà visité!

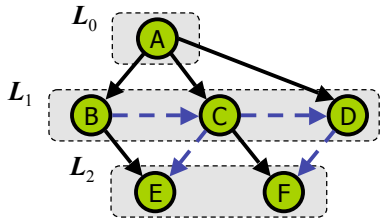
(4,5) - 5? déjà visité!

(4,3) - 3?

33

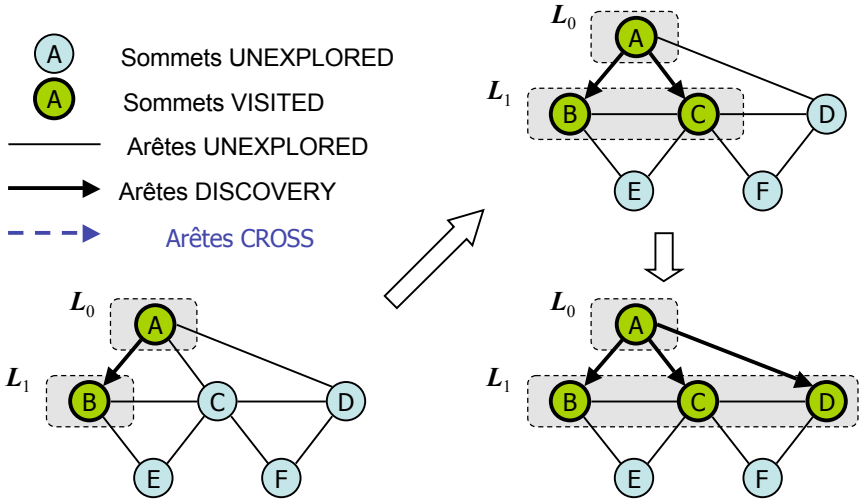
BFS avec etiquettes

En utilisant une sequence pour chaque niveau



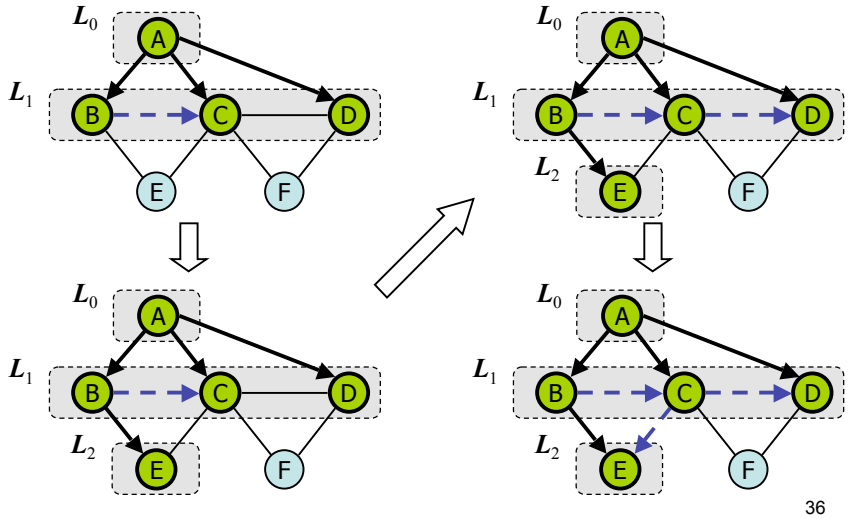
34

Exemple



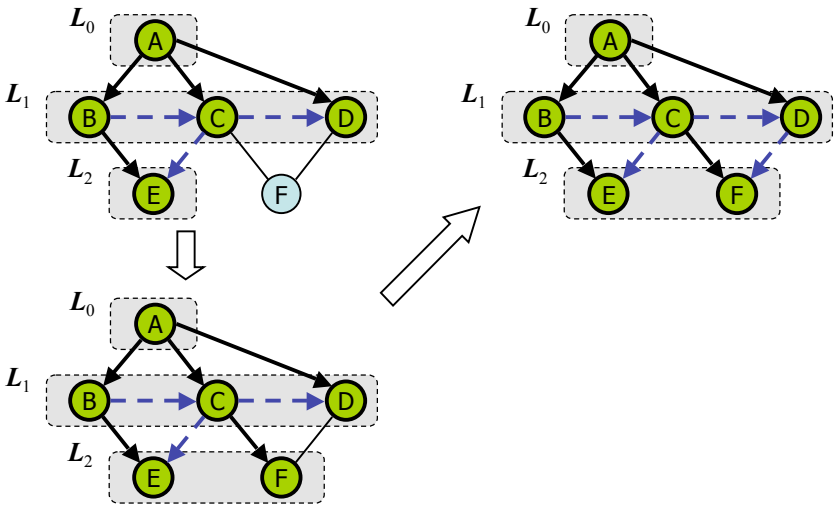
35

Exemple (suite)



36

Exemple (suite)



37

BFS plus de details

- L'algorithme utilise un mécanisme pour placer et obtenir les "étiquettes" des sommets et arêtes

Algorithm *BFS*(*G*)
Entrée graphe *G*
Sortie étiquetage des arêtes et partition des sommets de *G*
for all *u* ∈ *G.vertices*()
setLabel(*u*, *UNEXPLORED*)
for all *e* ∈ *G.edges*()
setLabel(*e*, *UNEXPLORED*)
for all *v* ∈ *G.vertices*()
if *getLabel*(*v*) = *UNEXPLORED*
BFS(*G*, *v*)

Algorithm *BFS*(*G*, *s*)
L_0 ← nouvelle séquence vide
L_0.*insertLast*(*s*)
setLabel(*s*, *VISITED*)
i ← 0
while ! L_i.*isEmpty*()
L_{i+1} ← nouvelle séquence vide
for all *v* ∈ L_i.*elements*()
for all *e* ∈ *G.incidentEdges*(*v*)
if *getLabel*(*e*) = *UNEXPLORED*
w ← *opposite*(*v*,*e*)
if *getLabel*(*w*) = *UNEXPLORED*
setLabel(*e*, *DISCOVERY*)
setLabel(*w*, *VISITED*)
L_{i+1}.*insertLast*(*w*)
else
setLabel(*e*, *CROSS*)
i ← *i* +1

38

Propriétés

Notation

G_s: composante connexe de s

Propriété 1

$BFS(G, s)$ visite tous les sommets et toutes les arêtes de G_s

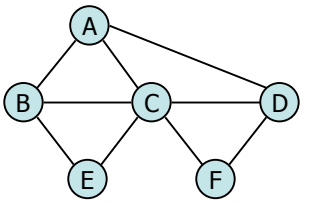
Propriété 2

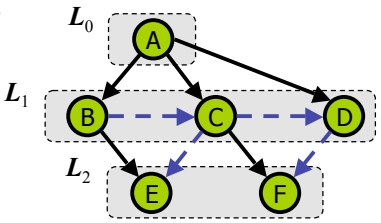
Les arêtes découvertes par $BFS(G, s)$ forment un arbre couvrant T_s de G_s

Propriété 3

Pour chaque sommet v dans L_i

- Le chemin de T_s à partir de s à v a i arêtes
- Chaque chemin de s à v dans G_s a au moins i arêtes





39

Analyse

- Chaque sommet est étiquetté deux fois
  - Une fois "non-exploré"
  - Une fois "visité"
- Chaque arête est étiquettée deux fois
  - Une fois "non-explorée"
  - Une fois "découverte" ou "de traversée"
- Chaque sommet est inséré une fois dans une séquence L_i
- L'opération *incidentEdges (s)* est appelée pour chaque sommet

- La complexité en temps du parcours en profondeur est en $O(n+m)$ si on utilise la structure de données "liste d'adjacence"
  - Rappel que $\sum_v \deg(v) = 2m$

40

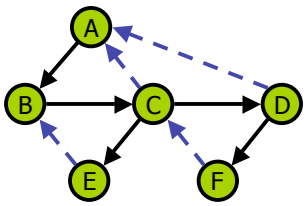
Applications

- Nous pouvons spécialiser l'algorithme de BFS pour résoudre les problèmes suivants en $O(n + m)$
  - Calculer les composantes connexes de G
  - Calculer une forêt couvrante de G
  - Chercher un chemin simple dans G, ou signaler que G est une forêt
  - Avec deux sommets de G donnés, chercher un chemin dans G entre eux avec le minimum d'arêtes ou signaler qu'aucun tel chemin existe

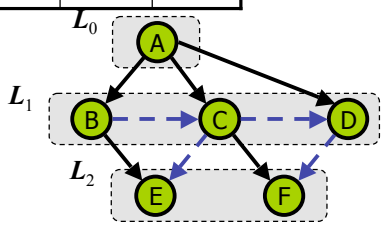
41

DFS versus BFS

Applications	DFS	BFS
Forêt couvrante, composantes connexes, chemins, cycles	√	√
Plus court chemin		√
Composantes biconnexes	√	



DFS



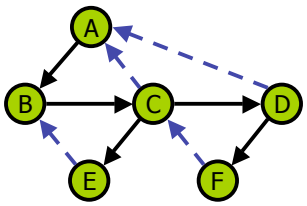
BFS

42

DFS versus BFS (suite)

Arêtes de retour (v,w)

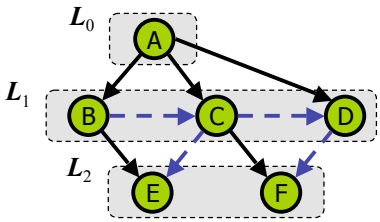
- w est un ancêtre de v dans l'arbre des arêtes découvertes



DFS

Arête de traverse (v,w)

- w est sur le même niveau que v ou dans le prochain niveau dans l'arbre des arêtes découvertes



BFS

43