Breadth-first search

Parcourir un graphe : emprunter de façon systématique les arcs du graphe, pour en visiter les sommets.

- l'un des algorithmes les plus simples sur les graphes.
- fonctionne sur des graphes orientés ET non orientés.
- On se donne un sommet origine s.
- emprunte systématiquement les arc de G pour découvrir tous les sommets accessibles depuis s.
- calcule la distance (# d'arcs) entre s et tous les sommets accessibles.
- construit l'arborescence des plus courts chemins à partir de s.
- découvre d'abord tous les sommets situés à une distance k de s avant de découvrir tout sommet situé à une distance $k+1$.

Coloration des sommets en blanc, gris ou noir :

- au départ, tous les sommets sont blancs ;
- quand un sommet est découvert pour la première fois, il est colorié en gris.
- quand tous les sommets adjacents d'un sommet s ont été découverts, l'algorithme colorie s en noir

3 structures de données différentes :

- `couleur[u]` représente l'état de chaque sommet.
- `d[u]` représentera la distance entre le sommet origine et le sommet `u` ;
- `pere[u]` représentera le sommet qui a permis de *découvrir* le sommet `u`.

Parcours en largeur

Si on suppose que le graphe soit représenté par listes d'adjacence, l'algorithme s'écrit comme suit :

```
1  Parcours_Largeur(G=(S,A),s)
2  {
3     % INITIALISATION
4     for each node u of S\{s}:
5         couleur[u] = blanc;      % coloration
6         d[u] = infinity;         % distance
7         pere[u] = nil;           % pere
8     couleur[s] = gris;
9     d[s] = 0;
10    pere[s] = nil;
11
12    F = {s};                     % FIFO
13
14    tant que F != {} faire {
15      u = tete(F);
16      Pour chaque v, adjacent à u
17           si couleur[v] = blanc alors {
18               couleur[v] = gris;
19               d[v] = d[u] + 1;
20               pere[v] = u;
21               mettre_dans_la_file (F,v);
22           }
23      Retirer_de_la_file(F);
24      couleur[u] = noir;
25    }
26 }
```

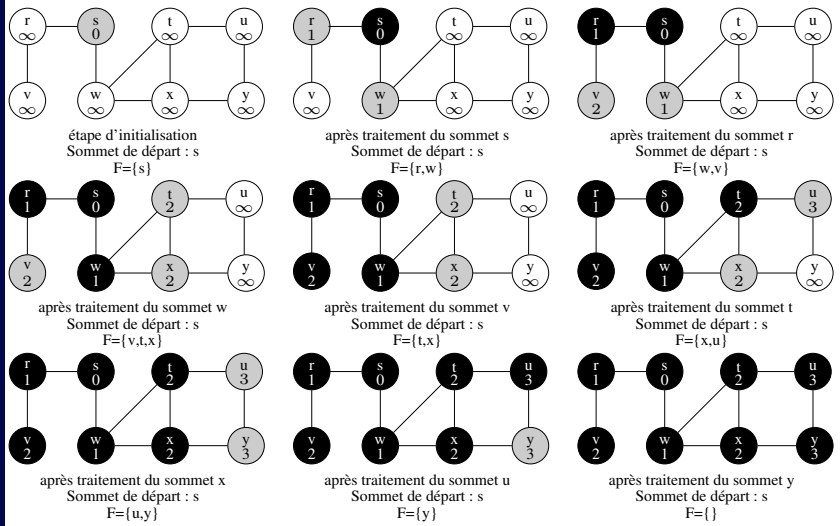

Parcours en largeur

- `F` est une file d'attente (FIFO) :
  Opérations : `mettre_dans_la_file(F,v)` et `Retirer_de_la_file(F)`
- initialisation :
  - colorer tous les sommets en blancs,
  - `d[t]`$= +\infty$ pour tous les sommets (sauf pour le sommet initial)
  - initialier les pères de chacun des nœuds à une valeur particulière `nil`.
- `F` représente la liste des sommets que l'on a déjà découverts mais qui n'ont pas encore été traités complètement (`F` soit initialisé à $\{s\}$).

La boucle `tant que` va passer en revue tous les sommets découverts mais pas encore totalement traités dans l'ordre de découverte des sommets (grâce à la structure de file d'attente). Pour chacun de ces sommets :

- on détermine le prochain sommet à traiter, c'est la tête de la file d'attente
- pour ce sommet, on passe en revue tous les sommets adjacents `v` :
  - rien à faire si `v` avait déjà été découvert (si `couleur[v]`$\neq$`'blanc'`)
  - sinon : on colorie en gris ce nouveau sommet, on stocke le sommet qui a permis de découvrir ce sommet, et on rajoute ce nouveau somme en fin de liste d'attente.
- Lorsque tous les sommets adjacents de `u` ont été découvert : on supprime le sommet de la liste d'attente,
- on peut alors colorier ce sommet en noir

Déroulement de l'algorithme sur un exemple



Complexité

- Hypothèse : $\begin{cases} \text{complexité de } \texttt{mettre\_dans\_la\_file(F,v)} & = & \Theta(1) \\ \text{complexité de } \texttt{Retirer\_de\_la\_file(F)} & = & \Theta(1) \end{cases}$
- Initialisation : $\Theta(|S|)$
- On rentre dans la condition "couleur(v) = blanc" une et une seule fois pour chaque sommet.
  $\Longrightarrow$ Tous les ajouts dans la file se font en $\Theta(|S|)$
- Chaque sommet a été ajouté à la file une et une seule fois.
  $\Longrightarrow$ toutes les suppressions de la file se font en $\Theta(|S|)$
- On parcourt chacune des listes d'adjacence une seule fois (donc en $\Theta(|A|)$)
  $\Longrightarrow$ balayage de toutes les listes d'adjacence en $\Theta(|A|)$
- Complexité globale en $\Theta(|S| + |A|)$

On définit la distance du plus court chemin $\delta(s, v)$ de s à v comme le nombre minimum d'arcs dans un chemin quelconque reliant le sommet s au sommet v, ou $+\infty$ s'il n'existe pas de chemin allant de s à v.

Lemma

Soit $G = (S, A)$, et soit $s \in S$ un sommet arbitraire. Alors pour tout arc $(u, v) \in A$:

$$\delta(s, v) \leq \delta(s, u) + 1$$

La preuve est évidente et est laissée au lecteur.

Lemma

La valeur $d[v]$ calculée par `Parcours_Largeur` *vérifie :*

$$d[v] \geq \delta(s, v)$$

Preuve :

par induction sur le nombre total de sommets insérés dans `F`.

- hypothèse : $d[v] \geq \delta(s, v) \quad \forall v \in S$
- à l'initialisation : l'hypothèse est vérifiée
- induction : $d[v] = d[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$

□

Lemma

Si `F` contient $< v_1, v_2, ...v_r >$ où v_1 est la tête de `F` et v_r est la queue de `F`, alors

$$d[v_r] \leq d[v_1] + 1 \quad \text{et} \quad d[v_i] \leq d[v_{i+1}] \quad \text{pour} \quad i = 1, 2, ...r - 1$$

Interprétation de ces deux inégalités :

- 2^{de} inégalité : les nœuds v de la file sont classés $d[v]$ croissant,
- $1^{\text{ère}}$ inégalité : les distances $\{d[v], v \in F\}$ ne diffèrent au plus que d'un.

En d'autres termes, la file peut être coupée en deux :

- $1^{\text{ère}}$ partie : nœuds v tq $d[v] = d[\text{tête}(F)]$
- 2^{de} partie : nœuds v tq $d[v] = d[\text{tête}(F)] + 1$

Preuve (hors programme) :

Par induction sur le nombre d'opérations sur la file d'attente `F`.

- quand la queue ne contient que s, alors c'est vérifié
- Si la tête est enlevée : la nouvelle tête est v_2 (si $F = []$, c'est OK)
  - soit $d[v_2] = d[v_1]$ et les 2 inégalités restent vraies,
  - soit $d[v_2] = d[v_1] + 1$ et tous les éléments suivants sont à la même distance de s.

  Dans les 2 cas, on obtient les 2 inégalités après suppression de la tête de file.
- Si on ajoute un élément à la queue de la file,
  - v_1 est le sommet dont la liste d'adjacence est balayée
  - v_{r+1} est le nouveau sommet de la liste : $d[v_{r+1}] = d[v_1] + 1$

  Les 2 inégalités sont préservées lors de l'ajout d'un élément dans F. □

Theorem

Soit $G = (S, A)$ un graphe et soit $s \in S$ un sommet quelconque du graphe.

1. *Après l'exécution de* `Parcours_Largeur`, $d[v] = \delta(s, v) \quad \forall v \in S$.
2. *Pour tout sommet accessible à partir de s, l'un des plus courts chemins de s à v, est le plus court chemin de s à pere[v] complété par l'arc $(pere[v], v)$.*

Preuve (hors programme) :

1. Par l'absurde. Supposons qu'il existe des sommets v tq $d[v] > \delta(s, v)$.
   Prenons un des sommets v parmi ceux-là qui ont la valeur minimale $d[v]$.
   Par hypothèse , v a été découvert à partir de $u = pere[v]$ et on a $d[u] = \delta(s, u) = d[v] - 1$.
   Comme on a $d[v] > \delta(s, v)$, il existe un plus court chemin qui se termine par un arc $u' \to v$ avec $u' \neq u$.
   On en déduit $\delta(s, u') < \delta(s, u)$, car dans le cas contraire, le plus court chemin de s à v passant par u' ne serait pas strictement meilleur que celui passant par u.
   Par hypothèse, $\delta(s, u') = d[u']$. donc $d[u'] < d[u]$.
   Contradiction, car dans ce cas, u' a été traité avant u, et v aurait donc dû être découvert à partir de u' et on aurait dû avoir $d[v] = d[u'] + 1$.
2. Evident car pour tout sommet on a : $d[v] = \delta(s, v)$.

□

Parcours en profondeur

descendre le plus profondément dans le graphe chaque fois que c'est possible. Lorsqu'il n'est plus possible de descendre, alors l'algorithme revient en arrière pour emprunter un autre chemin pas encore exploité.
C'est une approche récursive :

1. `Parcours_Profondeur(G)` initialise les différentes structures de données :
   - la `couleur` de chaque sommet est initialisée à blanc,
   - le père de chaque sommet est initialisée à `NIL`
   - un compteur de temps ou d'étapes est initialisé à 0.

   Puis, on appelle à seconde fonction pour tout sommet blanc.

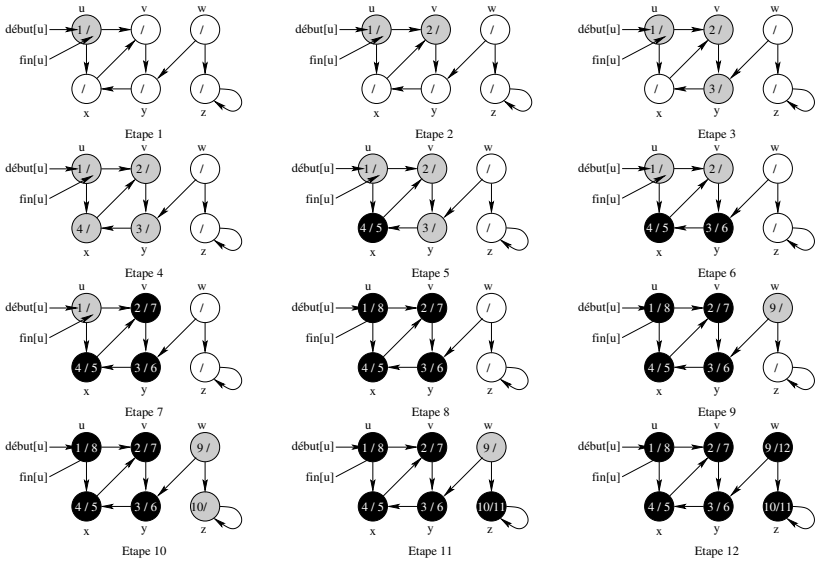
```
1  ParcoursProfondeur(G)
2       pour chaque sommet u de S(G) {
3            couleur[u] = blanc;
4            père[u] = NIL
5       }
6       temps = 0;
7       pour chaque sommet u de S {
8            si couleur[u] = blanc :
9              VisiterPP(u)
10      }
```


Parcours en profondeur

2. `Visiter_PP(u)` assure le parcours en profondeur :
   - déclarer le sommet choisi `u` comme *découvert* (gris),
   - incrémenter le compteur d'étapes et on stocke le # d'étape de la découverte de `u`,
   - passer en revue tous les sommets adjacents à `u` et pour chaque sommet `v` adjacent à `u`, s'il est encore à découvrir (blanc), on remplit la structure de données `père[v]` avec le sommet choisi `u` et on rappelle de manière récursive le parcours en profondeur pour ce sommet `v` adjacent à `u`.
   - lorsque tous les sommets adjacents de `u` ont été visités, déclarer le sommet `u` comme totalement traité (noir) et on incrémente le compteur de temps et on mémorise le temps de fin de traitement du sommet `u` dans la structure de données `fin[u]`.

```
1  Visiter_PP(u)
2      couleur[u] = gris;
3      début[u] = temps +1;
4      temps = temps + 1;
5      pour chaque v, adjacent de u {
6            si couleur[v] = blanc alors
7            père[v] = u;
8            Visiter_PP(v);
9      }
10     couleur[u] = noir;
11     fin[u] = temps = temps+1;
```


Parcours en profondeur : exemple



Parcours en profondeur

Il existe plusieurs applications du parcours en profondeur :

- la recherche de tous les sommets atteignables à partir d'un sommet donné
- le tri topologique (Cf ...)
- la recherche des composantes fortement connexes (Cf ...)

Complexité du parcours en profondeur :

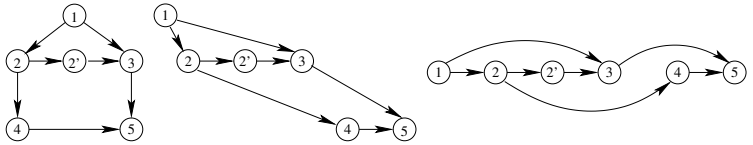
- initialisation : $\Theta(|S|)$
- boucle de `Parcours_Profondeur(G)` : $\Theta(|S|)$ hormis `Visiter_PP`.
- La procédure `Visiter_PP` est appelée exactement 1 fois par sommet. La boucle de `Visiter_PP` est effectuée $|Adj[v]|$. Comme $\sum |Adj[v]| = \Theta(|A|)$, le coût total de la boucle de `Visiter_PP` est $\Theta(|A|)$.
- $\Longrightarrow$ Complexité globale : $\Theta(|S| + |A|)$

Tri topologique d'un graphe orienté acyclique

Trier topologiquement un graphe :

- ordonner linéairement tous les sommets de sorte que si G contient un arc (u, v), u apparaisse avant v.
- aligner les sommets sur une ligne de telle manière que tous les arcs soient orientés de gauche à droite.

Si le graphe n'est pas acyclique, aucun ordre n'est possible.



Tri topologique : algorithme

Theorem

Le parcours en profondeur permet de trier topologiquement un graphe.

```
1 | Tri_topologique (G)
2 |    - appeler Parcours_profondeur(G) pour calculer
3 |      les temps de fin de traitement
4 |    - chaque fois que le traitement d un sommet s achève,
5 |      l insérer au début d une liste chainée.
6 |    - retourner la liste chainée.
```

$\Longrightarrow$ la complexité est la même que celle du parcours en profondeur

$\Longrightarrow$ la complexité est en $\Theta(|S| + |A|)$

Preuve :

Il suffit de montrer que s'il existe un arc de u à v alors $fin[v] < fin[u]$.

- On considère un arc (u, v) exploré pour la première fois par `PP(G)`. Lorsque cet arc est exploré pour la première fois, v ne peut pas être gris car dans ce cas v serait un ancêtre de u. Donc, v est soit blanc, soit noir.
- si v est blanc, il est un descendant de u, et $fin[v] < fin[u]$
- si v est noir, $fin[v] < fin[u]$

Donc $fin[v] < fin[u]$ dans tous les cas, ce qui prouve la validité de l'algorithme. □

Strongly related components

Definition

les ***composantes fortement connexes*** de $G = (S, A)$ sont les ensembles maximaux de sommets $R \in S$ tels que :

$$\forall (u, v) \in R^2, \quad (u \longrightarrow v) \quad et \quad (v \longrightarrow u)$$

Dans chacune des composantes fortement connexes, les sommets sont mutuellement accessibles.

Soit ${}^tG = (S, {}^tA)$ avec ${}^tA = \{(u, v)/(v, u) \in A\}$

Deux remarques préliminaires :

- G et ${}^tG$ ont les mêmes composantes fortement connexes
- La construction de ${}^tG$ se fait en $O(|S| + |A|)$ avec une représentation par liste d'adjacence.

Algorithme en 3 étapes :

1. Appeler `Parcours_Profondeur(G)` pour calculer les dates de fin de traitement $f[u]$ pour chaque sommet u.
2. Calcul de ${}^tG$
3. Appeler `Parcours_Profondeur(`${}^tG$`)` avec un ordre sur les sommets : par $f[u]$ décroissant.

$\Rightarrow$ chaque arborescence de la forêt est une Composante Fortement Connexe.

Lemma (hors programme)

si 2 sommets appartiennent à la même Composante Fortement Connexe, aucun chemin entre eux ne sort de la Composante Fortement Connexe.

Preuve (hors programme) :

Soient u et v 2 sommets mutuellement accessibles. Soit w un sommet tel que $u \longrightarrow w \longrightarrow v$.
Alors, u est accessible à partir de w : $w \longrightarrow v \longrightarrow u$

Donc u et w appartiennent à la même Composante Fortement Connexe. □

CÔTE D'AZUR

Algorithmics for Biology
Jean-Paul Comet
Complexity
Pat. Matching
Graphs
Intro
Definitions
Adjacency List
Adjacency Matrix
BFS
DFS
Topological sorting
strongly. connect.
Shortest path
Dijkstra
Bellman–Ford
Dyn.Prog.
Sequences

Theorem (hors programme)

Lors d'un parcours en profondeur quelconque, tous les sommets appartenant à la même Composante Fortement Connexe se trouvent dans la même arborescence en profondeur.

Preuve (hors programme) :

- soit r le premier sommet découvert de la CFC. Comme r est le premier, les autres sommets de la CFC sont blancs au moment de la découverte de r.
- Il existe un chemin de r à tous les autres sommets de la CFC. Comme les chemins de sortent jamais de la CFC, tous les sommets qui la composent sont blancs.
  Soit ch un chemin de r à v. Supposons que v ne devienne pas un descendant de r. On peut, sans perte de généralité, supposer que tous les autres sommets du chemin deviennent descendants de r.
  Soit w le père de v dans ce chemin. w est descendant de r.
  On a donc $f[w] \leq f[r]$ (évident)
  v doit donc être découvert après r, mais avant $f[w]$.
  donc $d[r] < d[v] < f[w] \leq f[r]$
  donc l'intervalle $[d[v], f[v]]$ est inclus dans $[d[r], f[r]]$, et donc v est un descendant de r □