Ch. 3 Parcours d'un graphe

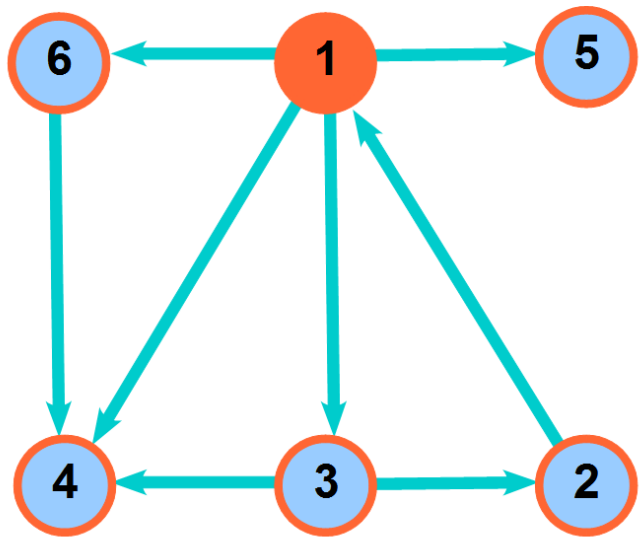
- Beaucoup de problèmes sur les graphes nécessitent un examen exhaustif des sommets ou des arcs du graphe : recherche de chemin, tri topologique, étude de la connexité, etc.
- Nous distinguerons deux types de parcours, le parcours en profondeur et le parcours en largeur.
- Terminologie : On parle indifféremment de parcours, d'exploration ou de visite de graphe, le terme anglo-saxon est "Graph Search".
- Le même algorithme de parcours pourra être utilisé avec les graphes orientés ou non orientés. Les différences apparaîtront au moment des applications du parcours.

Parcours en profondeur (Depth First Search ou DFS)

- Le parcours en profondeur consiste, à partir d'un sommet donné à suivre un chemin le plus loin possible, sans passer deux fois par le même sommet. Quand on rencontre un sommet déjà visité, on fait marche arrière pour revenir au sommet précédent et explorer les chemins ignorés précédemment.
- Cet algorithme se conçoit naturellement de manière récursive.
- Pour savoir si un sommet a déjà été visité, on utilise un vecteur de booléens *Visite* tel que $Visite[y] = vrai$ ssi le sommet y a déjà été visité.

```
initialiser Visite à faux;
Profond (G,x) :
{parcours en profondeur du graphe G à partir du sommet x}
Début
Visite[x]:=vrai;
{Traiter x en première visite}
Pour chaque voisin y de x faire
      Si Visite[y]=faux alors
         profond(G,y)
      {Sinon on détecte une revisite de y};
{Traiter x en dernière visite}
fin.
```

Vous trouverez sur Moodle des illustrations complètes du parcours en profondeur du graphe G_1 et du graphe G_3 suivant.



Parcours en profondeur à partir de 1

Ordre de parcours en première et dernière visite

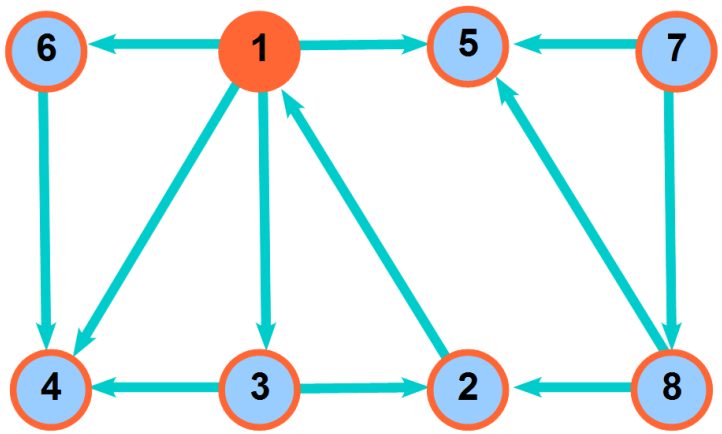
- Dans les applications on traitera les sommets x rencontrés soit en première visite (quand on commence le parcours en profondeur de x) soit en dernière visite (quand ce parcours est terminé et qu'on remonte au sommet qui a appelé récursivement $profond(G, x)$).
- L'ordre dans lesquels les sommets sont traités est alors respectivement appelé ordre de parcours en première ou en dernière visite.
- Le parcours en profondeur du graphe G_1 donne en première visite l'ordre $[1, 5, 2, 4, 3]$ et en dernière visite l'ordre $[3, 4, 2, 5, 1]$.
- Le parcours en profondeur du graphe G_1 donne en première visite l'ordre $[1, 3, 2, 4, 5, 6]$ et en dernière visite l'ordre $[2, 4, 3, 5, 6, 1]$.
- On constate sur cet exemple que l'ordre de parcours en dernière visite n'est pas l'ordre inverse de l'ordre de parcours en première visite.

Parcours en profondeur généralisé à tout le graphe

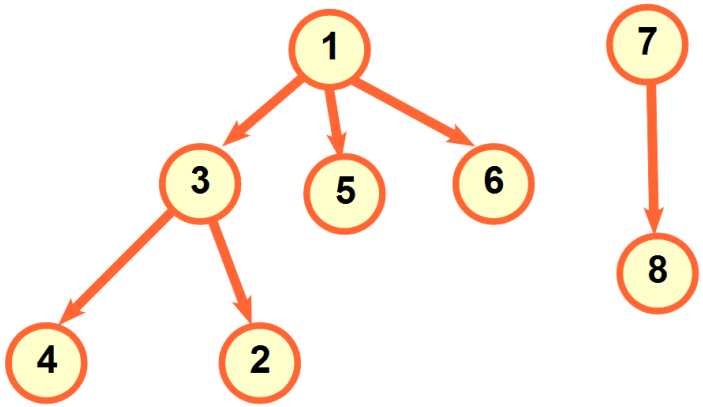
Pour parcourir tout les sommets du graphe G, on itère des parcours en profondeur à partir de x tant qu'il existe un sommet x non visité :

```
initialiser Visite à faux;
Pour i de 1 à n faire
  Si Visite[i]=faux alors
     profond(G,i);
```

Vous trouverez sur Moodle une illustration complète du parcours en profondeur généralisé du graphe G_4 suivant.



Parcours en profondeur à partir de 1



La forêt de parcours de G

Arborescence de parcours en profondeur

Le parcours en profondeur du graphe G à partir d'un sommet x permet de définir une arborescence de visite A.

Définition

L'arborescence de parcours en profondeur de G à partir du sommet x est un enraciné en x, orienté et tel qu'il existe un arc (x, y) dans A ssi l'appel à profond(G, x) a engendré récursivement un appel à profond(G, y).
Les sommets de l'arborescence de parcours sont tous les sommets de G accessibles à partir de x.

On associe alors au parcours complet du graphe G une forêt de visite contenant tous les sommets de G.

Parcours en largeur (Breath First Search ou BFS)

Le parcours en largeur est un parcours plus "prudent". Au lieu de s'aventurer le plus loin possible comme dans le parcours en profondeur, on va commencer par visiter d'abord tous les voisins de x, c'est-à-dire les sommets à une distance 1 de x, puis les voisins des voisins i.e. les sommets à une distance 2 etc. Pour cela, au fur et à mesure qu'on rencontre de nouveaux sommets (non encore visités,) on mémorise leurs voisins dans une file d'attente F pour une visite prochaine.
Visite est toujours un vecteur de booléens tel que $Visite[y] = vrai$ ssi le sommet y a déjà été visité.
F est la liste des sommets qu'on devra prochainement visiter.

```
initialiser Visite à Faux;
largeur (G,x) :
{parcours en largeur du graphe G à partir du sommet x}
F:=[x];Visite[x]:=vrai
Tant que  F n'est pas vide  faire
  Début
    considérer y la tête de F (et l'enlever de F)
    {Traiter y}
    pour chaque successeur z de y Faire
         si Visite[z]=faux alors
              Debut
                 Visite[z]:=vrai
                 ajouter z à la fin de la liste F
              Fin
 Fin
```

Vous trouverez sur Moodle une illustration complète du parcours en largeur des graphe G_1 et G_3 à partir du sommet 1.

Ordre de parcours en première et dernière visite

L'ordre dans lesquels les sommets sont traités est alors appelé ordre de parcours en largeur. Lors du parcours en largeur du graphe G_1 à partir du sommet 1, les sommets sont visités dans l'ordre :

$$[1, 5, 2, 4, 3].$$

Lors du parcours en largeur du graphe G_3 à partir du sommet 1, les sommets sont visités dans l'ordre :

$$[1, 3, 4, 5, 6, 2].$$

Parcours en profondeur généralisé à tout le graphe

Pour parcourir tout les sommets du graphe G, on itère là encore des parcours en largeur à partir de x tant qu'il existe un sommet x non marqué.

Arborescence de parcours en largeur

Le parcours en largeur du graphe G à partir d'un sommet x permet de définir une arborescence de visite A.

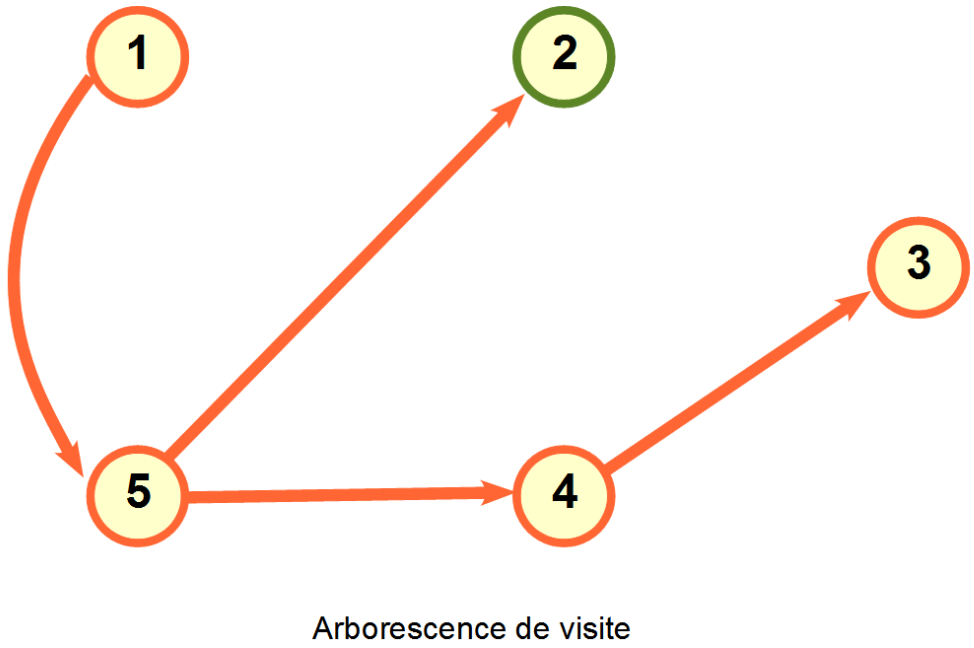


FIGURE: Arborescence de parcours du graphe G_1 en largeur à partir de 1

Définition

L'arborescence de parcours en largeur de G à partir du sommet x est un arbre enraciné en x, orienté et tel qu'il existe un arc (y, z) dans A ssi le traitement de y ajoute le sommet z dans la liste d'attente F.
Les sommets de l'arborescence de parcours sont tous les sommets de G accessibles à partir de x.

On associe alors au parcours complet du graphe G une forêt de visite contenant tous les sommets de G.

Complexité

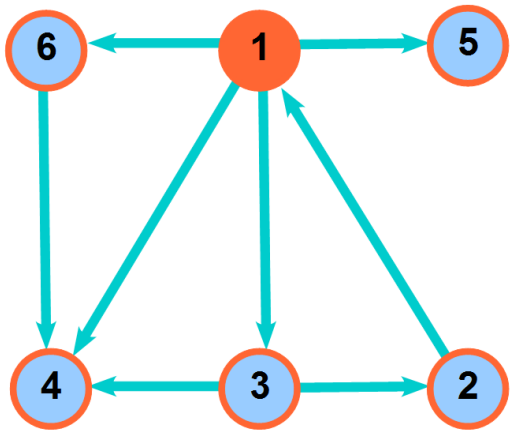
Le tableau *Visite* est initialisé en $O(n)$,
Le parcours est appelé exactement une fois pour chaque sommet x de G et a pour complexité $O(1)$ pour le traitement de x et $O(d(x))$ pour l'exploration des successeurs de x.
La complexité d'une visite complète de G est donc pour les deux parcours de

$$\begin{aligned} C(n) &= O(n) + \sum_{x=1}^{n} (d(x) + O(1)) \\ &= O(n) + \sum_{x=1}^{n} d(x) \\ &= O(n) + O(m) \\ &= O(max(n,m)). \end{aligned}$$

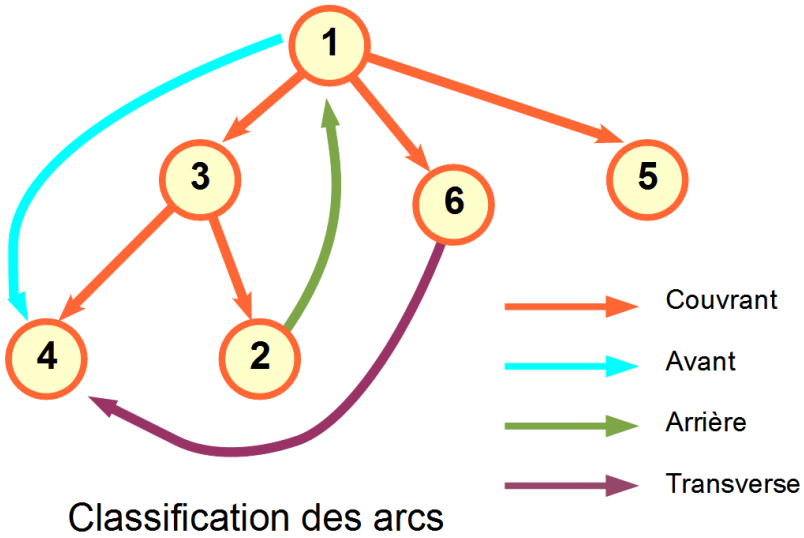
Classification des arcs

- Considérons le cas du parcours en profondeur d'un graphe G orienté. Si on ajoute à la forêt de visite tous les arcs du graphe G, on en déduit une classification des arcs :
  1. Un arc (x, y) de G est dit arc **couvrant** si (x, y) est encore un arc de la forêt de visite de G.
  2. Un arc (x, y) de G est dit arc **en avant** si il existe un chemin de x à y dans la forêt de visite de G.
  3. Un arc (x, y) de G est dit arc **en arrière** si il existe un chemin de y à x dans la forêt de visite de G.
  4. Tous les autres arcs de G sont dits arcs croisés ou **arcs traversiers**.

- Dans le cas du parcours en largeur d'un graphe orienté, il n'y a plus d'arcs en avant donc il ne reste que trois types d'arcs. En effet si l'arc (x, y) existe dans G et que lors de la visite de x, y est un voisin de x non visité alors x placera y dans F donc l'arc sera couvrant.
- Dans le cas du parcours en profondeur ou largeur d'un graphe non orienté, il n'y a plus d'arcs transverses et on ne distingue plus en avant ou en arrière, il ne reste donc plus que deux types d'arcs appelés couvrant et en avant.
- L'existence d'un cycle est caractérisé par la présence d'un arc en arrière.



Parcours en profondeur à partir de 1



Classification des arcs

FIGURE: Le graphe G_3 et la classification de ses arcs dans un parcours en profondeur à partir de 1.

Existence de chemin et connexité

Il existe de nombreuses applications des parcours en profondeur. Tous les algorithmes qui suivent seront mis en oeuvre en TP en copiant un des parcours et en l'adaptant (en général il suffit d'ajouter une ou deux structures de données et d'écrire leur initialisation et leur mise à jour...)

1. On a vu que les deux parcours d'un graphe à partir du sommet x permettent de visiter tous les sommets y accessibles à partir de x. Il est donc facile d'adapter ces parcours pour déterminer la liste de tous les sommets accessibles ou pour **tester l'existence d'un chemin entre deux sommets** x **et** y.
2. Dans le cas d'un graphe non orienté, le graphe est **connexe** ssi le parcours à partir du sommet 1 visite les n sommets. Si le graphe n'est pas connexe, un parcours à partir du sommet x permet de déterminer la composante connexe de x.

Plus courts chemins

Il s'agit ici de déterminer le plus court chemin en "nombre de sauts" entre deux sommets x et y et non du plus court chemin dans un graphe valué que nous étudierons plus loin.
Le parcours en profondeur, trop "aventurier" n'assure pas la minimalité du chemin obtenu. L'avantage d'un parcours en largeur est que du fait de l'ordre de visite des sommets, (les voisins de x, les sommets à un distance 2, puis 3 etc.) la première fois que l'on rencontrera y ce sera par un chemin de longueur minimale.
On va en un seul parcours calculer le plus court chemin de x vers tous les sommets y accessibles à partir de y. On utilise un vecteur *Dist* tel que $Dist[y]$ est la longueur du plus court chemin de x vers y (et ∞ si y n'est pas accessibles). Afin de retrouver le chemin, on utilise un tableau de pères tel que Pere[y] est le prédécesseur de y dans le plus court chemin de x vers y.

```
  trouvé:=faux;
initialiser Visite à Faux;
initialiser Dist à l'infini;
F:=[x];Visite[x]:=vrai;
Dist[x]=0;
Pere[x]=x;
tant que F<>[]  faire
     Début
       y:=enleveTete(F);
       Pour chaque successeur z de y Faire
           si Visite[z]=faux alors
              Debut
              Visite[z]:=vrai;
              ajouterFin(z,F);
              Dist[z]=Dist[y]+1;
              Pere[z]=y
              Fin
     Fin
```

Existence de cycles

On a vu que l'existence de cycles est caractérisée par la présence d'arcs en arrière.

- Dans un graphe non orienté, on détecte un arc en arrière (et donc un cycle) en cas de revisite d'un sommet déjà visité (et différent de son père).
- Dans un graphe orienté, les arcs en avant et les arcs transverses provoque également des revisites. On détecte un arc en arrière (et donc un cycle) en cas de revisite d'un sommet dont le parcours en profondeur n'est pas encore terminé (on a fait la première visite mais pas la dernière). On peut pour cela utiliser le vecteur de visite et considérer que $Visite[x]$ vaut 0 s'il n'a pas été visité, 1 si on a effectué la première visite et 2 quand on a effectué la dernière visite.

Test d'arbre

On sait qu'un arbre est un graphe non orienté, connexe et sans cycle, les méthodes vues précédemment pour tester l'existence d'un cycle et la connexité d'un graphe non orienté permettent donc de vérifier si un graphe est bien un arbre.

Test de bipartisme

Il existe un critère simple pour vérifier qu'un graphe est biparti :

Propriété

Un graphe non orienté est biparti ssi il n'admet pas de cycle de longueur impaire.

Démonstration .
L'algorithme de recherche de cycle peut être adapté en test de bipartisme. Il suffit pour cela de numéroter alternativement à l'aide de 0 et 1 les sommets visités. Un cycle de longueur impaire est détecté lorsque x revisite un sommet y de même numéro que lui.