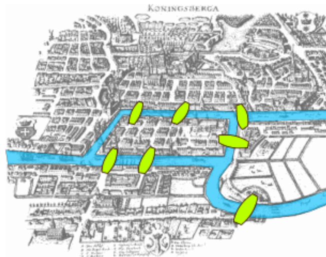
Informatique et Calcul Scientifique

Cours 11 : Algorithmes de graphes

Les sept ponts de Königsberg

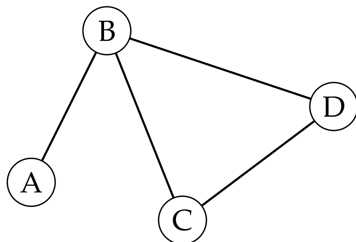


- Existe-t-il un chemin à travers la ville qui emprunte chaque pont exactement une fois ?
- En 1736, Euler prouva qu'il n'existe pas de tel chemin. Ce résultat est considéré comme le début de la théorie des graphes.

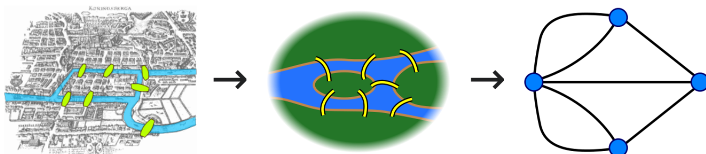
. https://fr.wikipedia.org/wiki/Problème_des_sept_ponts_de_Königsberg

Graphes

- Un graphe est une structure mathématique appropriée pour représenter des relations entre des objets.
- Un graphe consiste en un ensemble de **sommets** ou **nœuds** reliés par des **arêtes**.



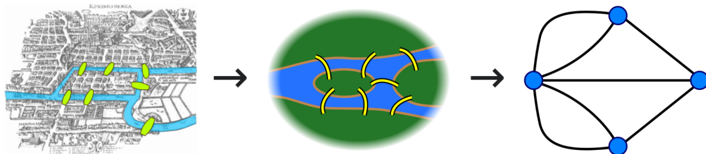
Les sept ponts de Königsberg - modélisation



- On peut représenter chaque masse de terre par un **sommet**, et relier deux sommets par une **arête** lorsque les deux masses de terre correspondantes sont reliées par un pont.
- Dans le **graphe** ainsi obtenu, la question devient : existe-t-il un chemin dans le graphe qui emprunte chaque arête exactement une fois ?

. Image source : Wikipedia

Les sept ponts de Königsberg - modélisation



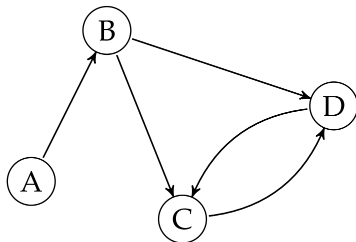
- La **modélisation** consiste à représenter mathématiquement un problème de manière à pouvoir le résoudre avec des outils et des méthodes mathématiques, en faisant abstraction des détails qui ne contribuent pas à la résolution du problème.

Graphe - définition

- Un graphe $G = (V, E)$ est défini par son ensemble V de sommets (*vertices* en anglais) et son ensemble E d'arêtes (*edges* en anglais).
- Une arête (u, v) représente une paire de sommets, ordonnés ou non.

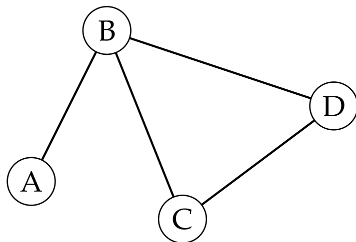
Graphe dirigé

- Si les arêtes d'un graphe G sont des paires **ordonnées** de sommets, le graphe est dit **dirigé**.
- Le graphe ci-dessous contient l'arête (A, B) mais pas l'arête (B, A).
  - On dira que B est **voisin** de A ou **adjacent** à A (mais A n'est pas voisin de B)
- Les graphes dirigés sont appropriés pour représenter des situations de flux ou de déplacement entre une source et une destination.



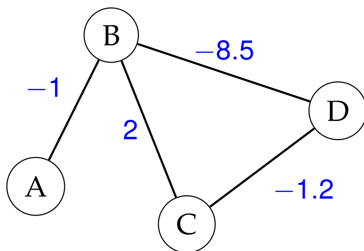
Graphe non dirigé

- Si les arêtes d'un graphe G sont des paires **non ordonnées** de sommets, le graphe est dit **non dirigé**.
- Dans le graphe ci-dessous, (A, B) et (B, A) dénotent la même arête.
  - A et B sont **voisins** l'un de l'autre ou **adjacents** l'un à l'autre.
- Les graphes non dirigés sont appropriés pour représenter des relations symétriques entre des objets.



Graphe pondéré

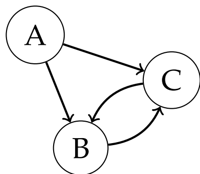
- Un graphe (dirigé ou non) est dit **pondéré** si **un poids** est associé à chaque arête du graphe.
- Les poids peuvent être des nombres entiers ou réels, positifs, négatifs ou nuls selon les applications.
- Les graphes pondérés sont appropriés par exemple pour représenter le coût d'emprunter un certain chemin ou de prendre une certaine décision.

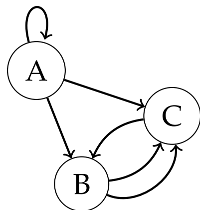


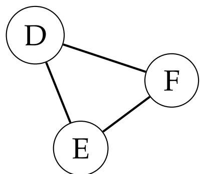
Graphe simple vs multigraphe

- Le graphe obtenu dans la modélisation du problème des ponts de Königsberg est en fait un **multigraphe**. Un multigraphe (dirigé ou non)
  - permet à plusieurs arêtes de relier la même paire de sommets (u, v) (dans le même ordre pour un multigraphe dirigé)
  - permet à une arête d'avoir la forme (u, u), c'est-à-dire de relier un sommet à lui-même.
- Un **graphe simple** ne permet ni les arêtes "parallèles", ni les arêtes qui relient un sommet à lui-même.
- Sauf si spécifié autrement, on considèrera toujours des graphes simples.

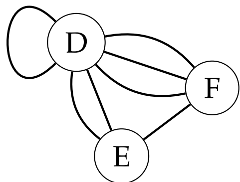
Graphe simple vs multigraphe







Deux graphes simples

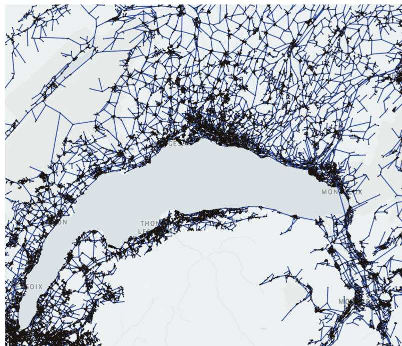


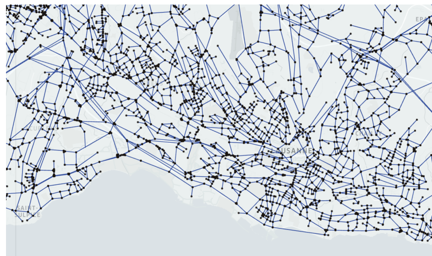
Deux multigraphes

Exemple - réseaux routiers

- On peut représenter un réseau routier par un graphe dirigé et pondéré :
  - Chaque intersection de routes est un sommet
  - Chaque segment de route entre deux intersections est une arête dirigée selon le sens de la circulation
  - Le poids d'une arête peut être la longueur du segment de route correspondant, ou une estimation du temps nécessaire pour parcourir ce segment de route
- On s'intéresse entre autres à des algorithmes qui calculent le plus court chemin entre deux points (problème d'optimisation).

Exemple - réseaux routiers





. Graphes obtenus à partir de Open Street Map

Exemple - réseaux sociaux

- On peut représenter les relations sur un réseau social comme Facebook par un graphe non dirigé :
  - Chaque personne est un sommet
  - Il existe une arête entre deux sommets si ces deux personnes sont amies
  - Le graphe est non dirigé puisque la relation d'amitié est symétrique.
- On s'intéresse entre autres à des questions du type "quels sont les amis d'amis de la personne A ?" ou bien "quelle est la distance entre les personnes A et B ?"
- Un réseau social où la relation de "suivi" est non symétrique, comme Twitter, Instagram ou Tiktok, sera modélisé par un graphe dirigé.

Exemple - réseaux de flux

Qu'est-ce que ces situations ont en commun ?

- Une station d'épuration produit de l'eau qui est transportée dans un réseau de tuyaux de diverses grosseurs jusqu'à nos robinets.
- Un champ d'éoliennes fournit de l'électricité qui est transportée dans un réseau électrique jusqu'aux utilisateurs.
- Dans le cadre d'un service de vidéo sur demande, des vidéos hébergées sur des serveurs sont streamées par des utilisateurs via Internet.
- ...

Elles peuvent toutes être modélisées par des **réseaux de flux** : des graphes dirigés, pondérés, où les sommets sont les "relais", les arêtes sont les "conduits" (qui ont chacun une certaine capacité), et un flux de matériel/d'information/.. chemine entre un sommet source et un sommet destination.

Représentation d'un graphe en mémoire

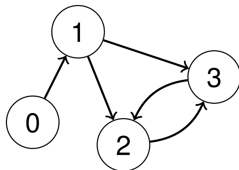
- Comment stocker les données relatives à un graphe en mémoire pour implémenter des algorithmes de graphe ?
- Le graphe est complètement décrit par l'ensemble de ses sommets et l'ensemble de ses arêtes. Comment représenter efficacement ces ensembles de manière à pouvoir
  - Accéder efficacement à un sommet donné/à une arête donnée
  - Parcourir efficacement les voisins d'un sommet
  - ...
- Que veut dire efficacement ?
  - Espace en mémoire
  - Temps de parcours des opérations
- Quelles sont les opérations qu'on veut pouvoir faire efficacement ?
  - Cela dépendra de l'application...

Représentation d'un graphe en mémoire

- Les deux manières les plus communes pour représenter un graphe en mémoire sont
  - Avec une matrice d'adjacence
  - Avec des listes d'adjacence.
- Dans ce qui suit, on va toujours considérer un graphe G avec n sommets $0, 1, \dots, n-1$, et m arêtes.
- Si G est dirigé, m peut prendre n'importe quelle valeur entre 0 (si G ne contient aucune arête) et $n(n-1)$ (si chaque paire de sommets est reliée par une arête.)
- Si G est non dirigé, m peut prendre n'importe quelle valeur entre 0 et $n(n-1)/2$.

Matrice d'adjacence

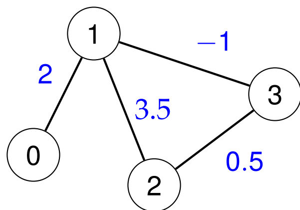
- Pour représenter G avec une matrice d'adjacence, on stocke une matrice dont les lignes et les colonnes sont indexées par les sommets de G. Elle contient à la ligne u et colonne v :
  - 1 si l'arête (u, v) existe dans le graphe (si v est **adjacent** à u ou **voisin** de u)
  - 0 sinon.
- En Python, on peut représenter une telle matrice avec une liste de taille n dont chaque élément est une liste de taille n.
  - Pour le graphe ci-dessous :
    `[[0,1,0,0],[0,0,1,1],[0,0,0,1],[0,0,1,0]]`



	0	1	2	3
0	0	1	0	0
1	0	0	1	1
2	0	0	0	1
3	0	0	1	0

Matrice d'adjacence

- Si G est non dirigé, la matrice d'adjacence sera symétrique.
- Si G est pondéré, il suffit de remplacer chaque 1 dans la matrice par le poids de l'arête correspondante.



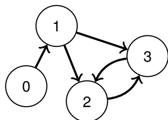
	0	1	2	3
0	0	2	0	0
1	2	0	3.5	-1
2	0	3.5	0	0.5
3	0	-1	0.5	0

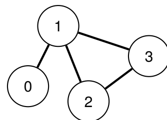
Liste d'adjacence

- Pour représenter G avec des listes d'adjacences, on stocke, pour chaque sommet de G, une liste de ses voisins.
- Quelle serait une bonne structure de données en Python pour stocker cette information ?

Liste d'adjacence

- En Python, on peut stocker ces données dans un dictionnaire dont les clés sont les sommets et les valeurs les listes des voisins des sommets.
  - Pour le graphe de gauche : `{0:[1], 1:[2,3], 2:[3], 3:[2]}`
  - Pour le graphe de droite : `{0:[1], 1:[0,2,3], 2:[1,3], 3:[1,2]}`
  - Pour un graphe pondéré, on peut stocker, dans la liste des voisins d'un sommet, des tuples représentant les paires (sommet, poids).





- On pourrait aussi utiliser une liste où l'élément `i` est la liste des voisins du sommet `i`, mais un dictionnaire nous permet d'avoir des sommets nommés autrement que $0, 1, \ldots, n-1$.

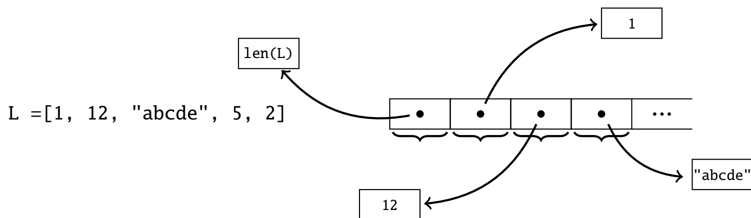
- Quelle est la meilleure manière de représenter des graphes en mémoire ?
  - Quel est l'espace requis en mémoire pour chacune de ces représentations ?
  - Quel est le temps requis pour effectuer diverses opérations pour chacune de ces représentations ?
    - Quelles sont les opérations qu'on désire effectuer ?

Structures de données

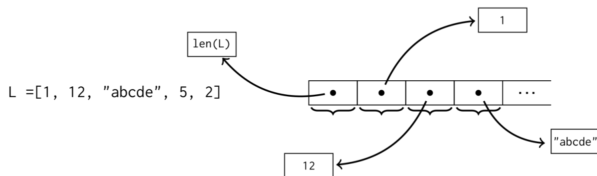
- Une partie fondamentale de l'algorithmique est l'étude des structures de données : la manière d'organiser les données et de les stocker en mémoire afin de permettre à certaines opérations de manipulation des données de se dérouler de manière efficace.
- Quelles opérations ?
  - L'insertion/la suppression
  - La lecture/l'écriture
  - La recherche
  - L'extraction du min/du max
  - ...
- En général, c'est le contexte de chaque algorithme qui déterminera quelles opérations doivent être optimisées et donc quelle structure de données doit être utilisée.

Listes Python

- En CPython, une liste est stockée en mémoire comme un bloc contigu de **références** aux éléments de la liste.
  - Une référence est l'adresse en mémoire de l'élément auquel elle se réfère.
- Chaque référence occupe un bloc de taille constante en mémoire.

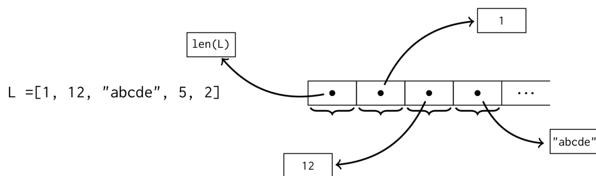


Listes Python - accéder à un élément d’index connu



- Pour accéder à un élément `L[i]`, il faut
  - calculer l’adresse de la référence à l’objet correspondant
  - accéder à l’élément en mémoire.
- Ces deux opérations se font en temps constant. On peut donc accéder à un élément `L[i]` (pour le lire ou le modifier) en temps $\Theta(1)$.

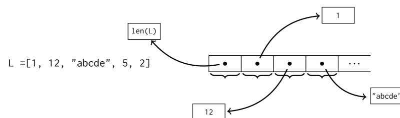
Listes Python - ajouter / effacer un élément en fin de liste



- Pour effectuer une opération `L.append()`, il faut
  - modifier la longueur de la liste
  - insérer une référence à un nouvel objet en fin de liste.
- On peut donc considérer que ces deux opérations se font en temps constant [1], et donc qu'on peut effectuer une opération `L.append()` en temps $\Theta(1)$.
- De même, on peut effectuer une opération `L.pop()` en temps constant.

1. En fait, elles se font en temps moyen constant.

Listes Python - insérer / effacer un élément arbitraire



- Pour insérer un élément à l'index `i` avec `L.insert(i, x)`, il faut décaler vers la "droite" tous les éléments à partir de l'indice `i`.
- De même, pour enlever un élément (ou une tranche d'éléments) avec `L.remove(x)`, `L.pop(i)` ou `del`, il faut décaler vers la "gauche" tous les éléments suivants.
- Ceci est nécessaire afin de maintenir la propriété de l'accès en temps constant.
- Pour une liste de taille n, les opérations d'insertion/d'effaçage à un index arbitraire d'une liste prennent donc temps $\Theta(n)$ au pire des cas.

Listes Python - temps de parcours

Soit `L` une liste de taille n.

Opération	Temps de parcours moyen
lecture/écriture `L[i]`	$\Theta(1)$
`L.append(x)`	$\Theta(1)$
`L.pop()`	$\Theta(1)$
`L.insert(i,x)`	$\Theta(n)$
`L.pop(i)`	$\Theta(n)$
`L.remove(x)`	$\Theta(n)$
`del L[i:j:k]`	$\Theta(n)$

. https://wiki.python.org/moin/TimeComplexity

Dictionnaires Python

- Les dictionnaires en Python sont implémentés comme des **tables de hachage** :
- Les clés sont transformées avec une **fonction de hachage**, et ce qui est stocké est l'image des clés par cette fonction.
- Une (bonne) fonction de hachage est conçue de telle sorte que la lecture, l'insertion et la suppression d'un élément se font en **temps moyen** constant.

Dictionnaires Python - temps de parcours

Soit `D` un dictionnaire de taille n (contenant n paires clé-valeur).

Opération	Temps de parcours moyen
lecture `D[key], D.get(key)`	$\Theta(1)$
insertion/écriture `D[key] = val`	$\Theta(1)$
suppression `D.pop(key), del D[key]`	$\Theta(1)$

Matrice vs listes d'adjacence - Espace

- La matrice d'adjacence prend un espace $\Theta(n^2)$ en mémoire indépendamment du nombre d'arêtes dans le graphe.
- Les listes d'adjacence prennent un espace $\Theta(n + m)$ en mémoire :
  - Le dictionnaire contient n éléments indépendamment du nombre d'arêtes.
  - Soit (u, v) une arête de G. Si G est dirigé, v apparaît dans la liste de u ; s'il est non dirigé, v apparaît dans la liste de u et u dans la liste de v. Chaque arête contribue donc 1 (si G dirigé) ou 2 (G non dirigé) à la taille totale des listes.
  - La taille totale de toutes les listes est donc m (G dirigé) ou $2m$ (G non dirigé).

```
G = {0:[1,52,6,8], 1:[2,3,4], 2:[5,6,13,17],...,n-1:[5,50,42]}
```

n
m
(ou 2m)

Matrice vs listes d’adjacence - Espace

- La matrice d’adjacence prend un espace $\Theta(n^2)$ en mémoire indépendamment du nombre d’arêtes dans le graphe.
- Les listes d’adjacence prennent un espace $\Theta(n + m)$ en mémoire. Par exemple,
  - Si $m = \mathcal{O}(n)$, alors une représentation par listes d’ajacence prend un espace $\Theta(n)$
  - Si $m = \Theta(n \log_2(n))$, alors cette représentation prend un espace $\Theta(n \log_2(n))$
  - Si $m = \Theta(n^2)$, alors cette représentation prend un espace $\Theta(n^2)$ en mémoire
- Sauf si le graphe a "beaucoup" d’arêtes ($m = \Theta(n^2)$), la matrice d’adjacence occupe beaucoup plus de place en mémoire que les listes d’adjacence.

Matrice vs listes d'adjacence - Temps de parcours

- Selon les opérations qu'on veut effectuer, il vaudra mieux adopter une représentation par matrice d'adjacence ou par listes d'adjacence.
- Déterminer s'il existe une arête entre deux sommets :
  - se fait en temps constant dans une représentation par matrice (il suffit de lire `G[u][v]`)
  - se fait en temps $\Theta(n)$ au pire des cas dans une représentation par liste (il faut parcourir toute la liste des voisins de u)

Matrice vs listes d'adjacence - Temps de parcours

- Selon les opérations qu'on veut effectuer, il vaudra mieux adopter une représentation par matrice d'adjacence ou par listes d'adjacence.
- Parcourir les voisins d'un sommet u :
  - se fait en temps $\Theta(n)$ dans une représentation par matrice (il faut parcourir toute la ligne `G[u]`)
  - se fait en temps proportionnel au nombre de voisins de u dans une représentation par liste ($\Theta(n)$ au pire des cas, mais souvent moins...)

Chemins, distance

Soit un graphe G et deux sommets u et v.

- Un **chemin de longueur** ℓ entre u et v est une suite de sommets

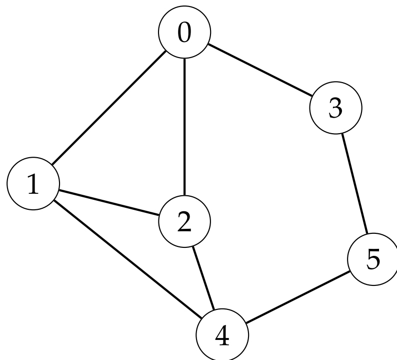
$$u_0, u_1, \ldots, u_\ell, \quad \ell \geq 1$$

  tous distincts, avec $u_0 = u$ et $u_\ell = v$, tels que u_i et u_{i+1} sont liés par une arête pour tout $0 \leq i \leq \ell - 1$.
- Un **plus court chemin** entre u et v est un chemin entre u et v de longueur minimale.
- La **distance** entre u et v est la longueur d'un plus court chemin entre u et v.
- S'il existe un chemin entre u et v, on dit que v est **atteignable** depuis u.

Exemple - graphe non dirigé

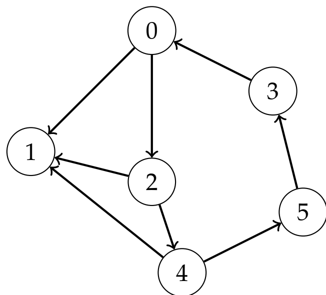
Il existe plusieurs chemins entre 0 et 4 :

- 0, 1, 4
- 0, 2, 4
- 0, 1, 2, 4
- 0, 2, 1, 4
- 0, 3, 5, 4



- Les chemins les plus courts sont 0, 1, 4 et 0, 2, 4.
- Les chemins inverses (par ex 4, 1, 0) sont des chemins entre 4 et 0.
- Les sommets 0 et 4 sont à distance 2 l'un de l'autre.

Exemple - graphe dirigé



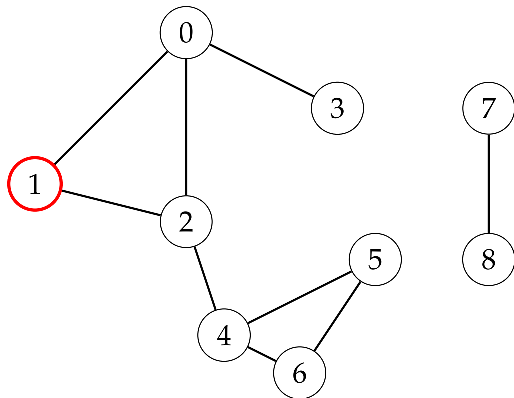
- Le seul chemin entre 0 et 4 est 0, 2, 4. Le sommet 4 est à distance 2 du sommet 0.
- Le seul chemin entre 4 et 0 est 4, 5, 3, 0. Le sommet 0 est à distance 3 du sommet 4.

Parcourir des graphes

- Etant donné un graphe G et un sommet s de G (la "source"), un **algorithme de parcours** permet d'**explorer** (ou de visiter), à partir de s, tous les sommets de G qui sont atteignables depuis s.
- Un grand nombre d'algorithmes de graphes sont basés sur des algorithmes de parcours.
- Pour l'instant, on va simplement afficher les sommets du graphe dans l'ordre dans lequel ils sont parcourus.
- Les algorithmes de parcours de graphe tombent en général dans deux catégories : le parcours en largeur et le parcours en profondeur.

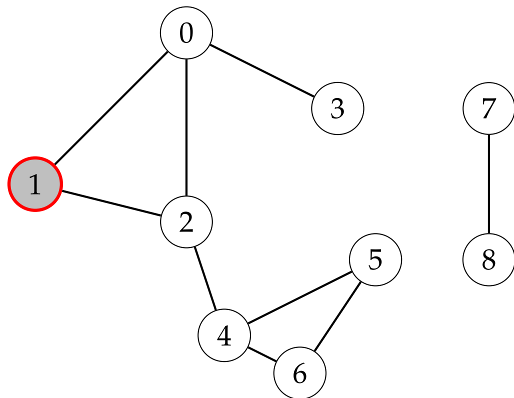
Parcours en largeur (BFS)

- Le parcours de G en largeur (Breadth-First Search) à partir d'un sommet source s explore les sommets de G en commençant par s, puis par les sommets à distance 1 de s, puis les sommets à distance 2,...



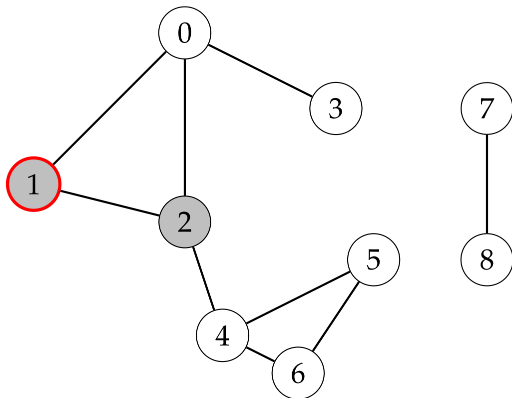
Parcours en largeur (BFS)

- Le parcours de G en largeur (Breadth-First Search) à partir d'un sommet source s explore les sommets de G en commençant par s, puis par les sommets à distance 1 de s, puis les sommets à distance 2,...



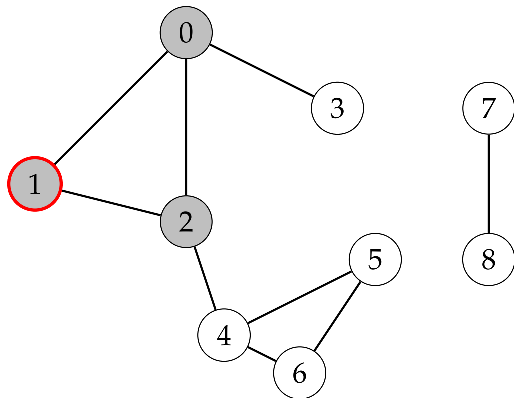
Parcours en largeur (BFS)

- Le parcours de G en largeur (Breadth-First Search) à partir d'un sommet source s explore les sommets de G en commençant par s, puis par les sommets à distance 1 de s, puis les sommets à distance 2,...



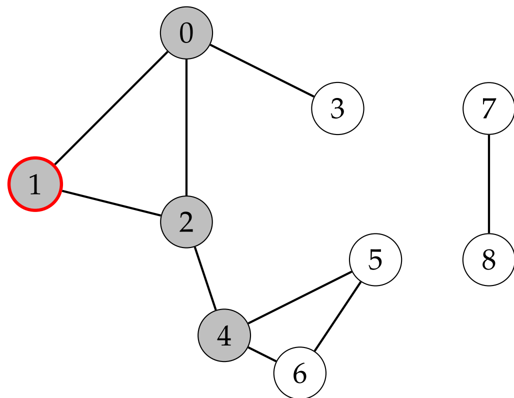
Parcours en largeur (BFS)

- Le parcours de G en largeur (Breadth-First Search) à partir d'un sommet source s explore les sommets de G en commençant par s, puis par les sommets à distance 1 de s, puis les sommets à distance 2,...



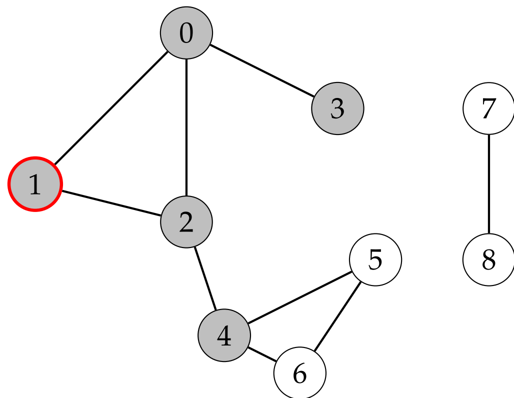
Parcours en largeur (BFS)

- Le parcours de G en largeur (Breadth-First Search) à partir d'un sommet source s explore les sommets de G en commençant par s, puis par les sommets à distance 1 de s, puis les sommets à distance 2,...



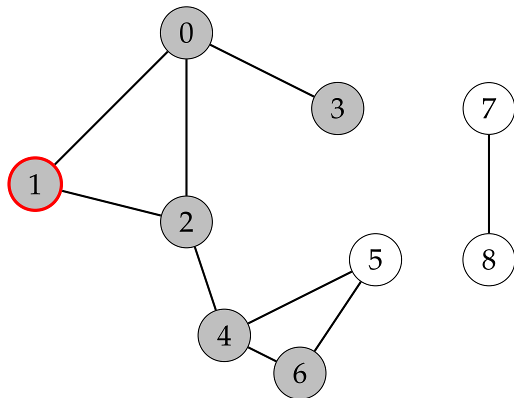
Parcours en largeur (BFS)

- Le parcours de G en largeur (Breadth-First Search) à partir d'un sommet source s explore les sommets de G en commençant par s, puis par les sommets à distance 1 de s, puis les sommets à distance 2,...

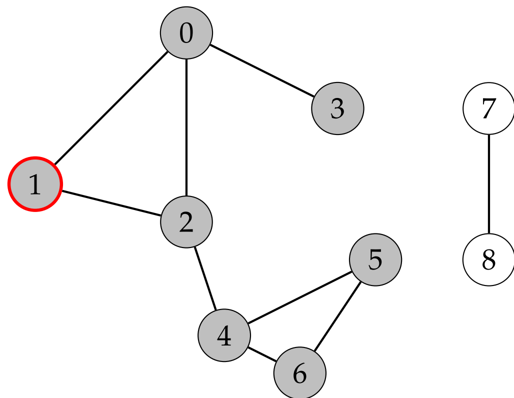


Parcours en largeur (BFS)

- Le parcours de G en largeur (Breadth-First Search) à partir d'un sommet source s explore les sommets de G en commençant par s, puis par les sommets à distance 1 de s, puis les sommets à distance 2,...

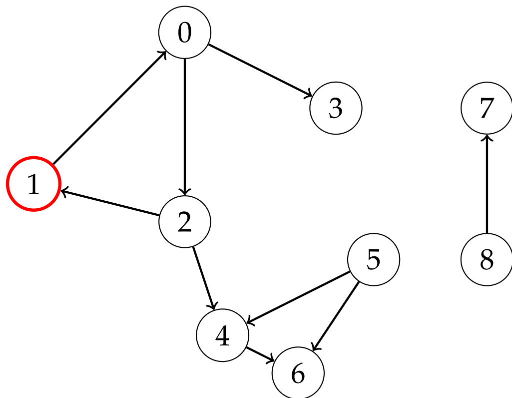


- Le parcours de G en largeur (Breadth-First Search) à partir d'un sommet source s explore les sommets de G en commençant par s, puis par les sommets à distance 1 de s, puis les sommets à distance 2,...



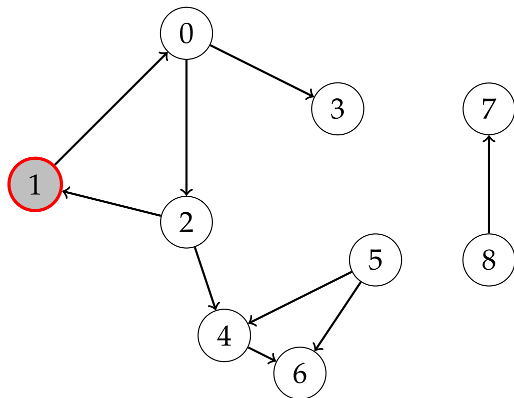
Parcours en profondeur (DFS)

- Le parcours de G en profondeur (Depth-First Search) à partir d'un sommet source s explore les sommets de G en allant le plus loin possible avant de revenir en arrière.



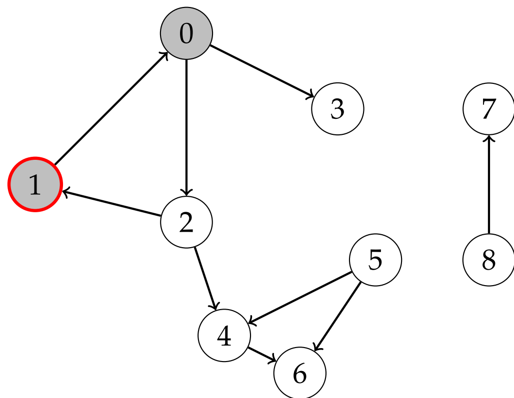
Parcours en profondeur (DFS)

- Le parcours de G en profondeur (Depth-First Search) à partir d'un sommet source s explore les sommets de G en allant le plus loin possible avant de revenir en arrière.



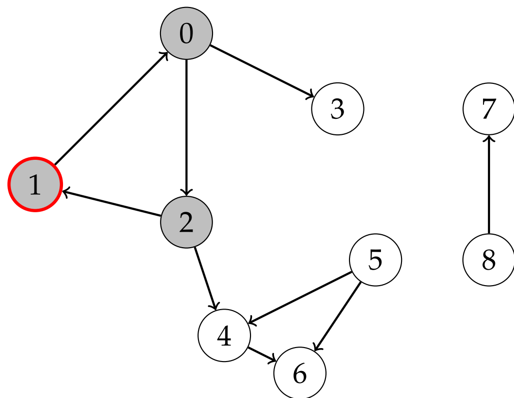
Parcours en profondeur (DFS)

- Le parcours de G en profondeur (Depth-First Search) à partir d'un sommet source s explore les sommets de G en allant le plus loin possible avant de revenir en arrière.



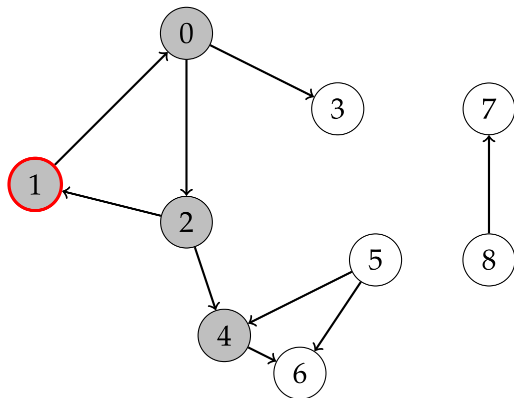
Parcours en profondeur (DFS)

- Le parcours de G en profondeur (Depth-First Search) à partir d'un sommet source s explore les sommets de G en allant le plus loin possible avant de revenir en arrière.



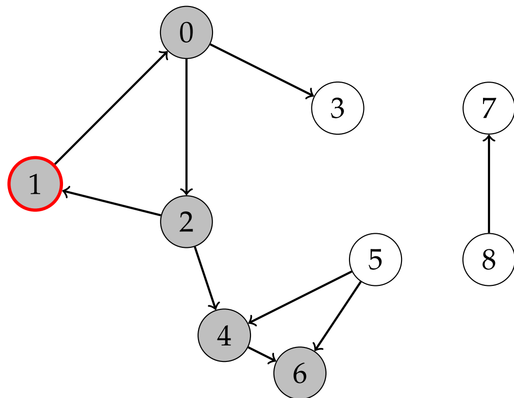
Parcours en profondeur (DFS)

- Le parcours de G en profondeur (Depth-First Search) à partir d'un sommet source s explore les sommets de G en allant le plus loin possible avant de revenir en arrière.



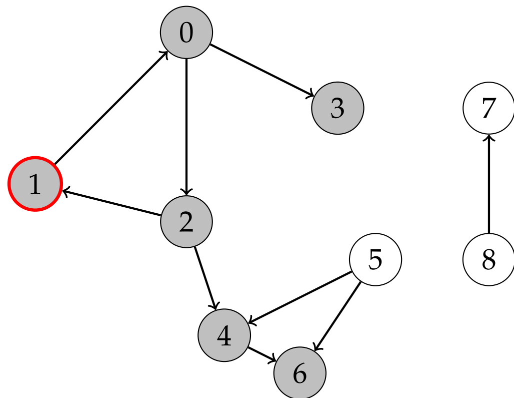
Parcours en profondeur (DFS)

- Le parcours de G en profondeur (Depth-First Search) à partir d'un sommet source s explore les sommets de G en allant le plus loin possible avant de revenir en arrière.



Parcours en profondeur (DFS)

- Le parcours de G en profondeur (Depth-First Search) à partir d'un sommet source s explore les sommets de G en allant le plus loin possible avant de revenir en arrière.



Parcours de graphe - BFS

- L'algorithme BFS prend en entrée une représentation par listes d'adjacence d'un graphe G, et un sommet s de G.
- Il affiche les sommets de G atteignables depuis s, en garantissant que l'ordre d'affichage correspond à un parcours en largeur du G : si la distance entre s et u est plus petite que la distance entre s et v, u apparaîtra avant v dans le parcours.
- L'algorithme maintient un ensemble de sommets à parcourir et choisit de cet ensemble un prochain sommet à parcourir à chaque étape.
- Le choix de la structure de données qui contient cet ensemble de sommets est crucial au bon fonctionnement et à l'efficacité de l'algorithme.
- On présente aujourd'hui une version de BFS qui fonctionne correctement mais qui n'a pas un temps de parcours optimal.

BFS

```python
def BFS(G,s):
    '''
    Entree: graphe G en dict de listes d'adj, s sommet
    Parcourt G en largeur, affiche les sommets parcourus
    '''
    n = len(G)
    a_parcourir = [s]
    vu = [0 for i in range(n)]
    vu[s] = 1

    while a_parcourir:
        sommet = a_parcourir.pop(0)
        for u in G[sommet]:
            if not vu[u]:
                a_parcourir.append(u)
                vu[u] = 1
        print(sommet)
```