Graphes #1

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

Graphe

Un *graphe* est un ensemble fini de *sommets* (ou *nœuds* ; en anglais : *vertex*/*vertices*) reliés par des *arcs* (généralement orientés) ou *arêtes* (généralement non orientées) (en anglais : *edges*)

Notation générale

Soit G un graphe, on note S (V) l'ensemble de ses sommets (généralement représentés par des entiers naturels) ; $A \subseteq S \times S$ (E) l'ensemble de ses arcs

Graphes : orientation

Graphe non-orienté

Un graphe est *non-orienté* si ses arcs ne sont pas orientés : on ne distingue pas un arc $u \to v$ d'un arc $v \to u$; un arc $u \leftrightarrow v$ apparaît comme (u, v) **et** (v, u) dans A

Graphe orienté

Un graphe est *orienté* si ses arcs sont orientés : on distingue un arc $u \to v$ d'un arc $v \to u$, et la présence de l'un n'implique pas celle de l'autre

Adjacence

Deux sommets reliés par un arc sont dits *adjacents* ou *voisins*

Exemples

Représentation graphique

On représente les sommets par des ronds, les arcs (orientés) par des traits (flèches) entre les sommets

Vocabulaire de base #1

Boucle

Une *boucle* est un arc $u \leftrightarrow u$ d'un sommet vers lui-même

- Sauf mention du contraire on considérera des graphes sans boucle
  - En particulier dans le cas non-orienté
- Mais l'on pourra aussi parfois supposer implicitement leur présence

Degré

Le *degré* d'un sommet est le nombre d'arcs impliquant ce sommet ; les boucles comptent double

- Graphe non-orienté : $d(u) := \#\{a \in A \mid a = (u, \_)\} + [(u, u) \in a]$
- Graphe orienté : on distingue degré *sortant* : $d_+(u) := \#\{a \in A \mid a = (u, \_)\}$ et degré *entrant* : $d_-(u) := \#\{a \in A \mid a = (\_, u)\}$; on a $d(u) := d_+(u) + d_-(u)$

On peut aussi définir le degré **max**, **min** d'un graphe (de façon évidente)

Vocabulaire de base #2 : chemins

Chemin

Un *chemin* de longueur ℓ entre deux sommets u, v est une suite finie d'arcs $a_1, \ldots, a_\ell$ telle que : $a_1 = (u, \_)$; $a_\ell = (\_, v)$; $\forall i \in [\![1, \ell - 1]\!]$, $a_i[1] = a_{i+1}[0]$

Notation *ad hoc* : $u \rightsquigarrow v = u \to w_1 \to w_2 \to \cdots \to w_{\ell-1} \to v$

- Chaîne : chemin dont les sommets sont deux à deux distincts
- Chemin *élémentaire* : chemin dont les sommets sont deux à deux distincts, sauf éventuellement le premier et le dernier
  - *fermé* : si le premier et le dernier sommet sont **égaux**
- Chemin *simple* : chemin dont les **arcs** sont deux à deux distincts

Cycle

Un *cycle* est un chemin simple élémentaire fermé non vide, qui n'est pas une boucle

On part d'un sommet et y revient en passant par d'autres sommets tous distincts, sans jamais prendre deux fois le même arc

Vocabulaire de base #3 : sous-graphes

Sous-graphe

Un *sous-graphe* d'un graphe $G = (S, A)$ est un graphe $G' = (S', A')$ avec $S' \subseteq S$, $A' \subseteq A$.

Sous-graphe induit

Le *sous-graphe* de $G = (S, A)$ induit par $S' \subset S$ est le graphe $G' = (S', A')$ avec $A' = (S' \times S') \cap A$

On garde un sous-ensemble de sommets de G, et uniquement les arcs impliquant ces sommets

Vocabulaire de base #4 : connexité

Connexité

Un graphe $G = (S, A)$ est *connexe* si pour tout $u, v \in S$ il existe un chemin $u \rightsquigarrow v$ ou $v \rightsquigarrow u$

- Dans un graphe non orienté, les deux conditions sont équivalentes
- Notion pas forcément très utile pour un graphe orienté

Forte connexité

Un graphe **orienté** est dit **fortement** *connexe* si pour tout $u, v \in S$, il existe un chemin $u \rightsquigarrow v$ **et** un chemin $v \rightsquigarrow u$

Faible connexité

Un graphe **orienté** est dit *faiblement connexe* si le graphe non-orienté obtenu en « oubliant » les orientations est connexe

- Notion pas forcément très utile

Composantes connexes

Les *composantes connexes* de $G = (S, A)$ sont ses sous-graphes induits connexes **maximaux pour l'inclusion** ; elles forment une partition de G

Illustrations

Quelques graphes particuliers

Graphe complet

« Le » *graphe complet* à n sommets K_n est un graphe non-orienté tel que toute paire de sommets sont adjacents

Graphe cycle

« Le » *graphe cycle* à $n \geq 3$ sommets C_n est un graphe non-orienté constitué d'un unique cycle de longueur n

Graphe biparti

Les sommets sont l'union disjointe de L et R, avec aucun arc entre deux sommets de L ou deux sommets de R

Arbre

Un graphe non-orienté connexe sans cycle est un *arbre* (au sens des graphes)

Forêt

Un graphe non-orienté sans cycle est une *forêt* (pareil)

Ajout d'information

On peut enrichir les graphes d'information supplémentaire, généralement à des fins de modélisation

Pondération

Un graphe *pondéré* est un graphe $G = (S, A)$ où chaque arc $a \in A$ est pondéré par un *poids* w_a (pour nous : généralement $\in \mathbb{N}$)

- Représente souvent un coût à minimiser (ou plus généralement, une contrainte à optimiser)

Étiquettes

On peut ajouter des *étiquettes* aux sommets ou arcs d'un graphe pour indiquer ce qu'ils représentent (*cf.* exemples de modélisation plus bas)

Table des matières

Représentation #1

Soit $G = (S, A)$, on veut *représenter* S, A & d'éventuelles informations ajoutées avec les types de données habituels, pour pouvoir algorithmiquement manipuler des graphes

Sommets

La plupart du temps : $S \hookrightarrow [\![\#S]\!]$

- Avec éventuellement des étiquettes en plus

Arcs : deux grandes approches classiques (mais pas les seules) :

Listes d'adjacence

- Pour **chaque sommet** u, on liste les sommets v tels qu'il existe un arc (u, v)
  - généralement dans un ordre quelconque
  - avec les éventuelles informations ajoutées (poids, étiquette…)
- On collecte les listes pour chaque sommet, par ex. dans une `list`, un `dict`…

Représentation #2

Matrice d'adjacence

Pour $S \hookrightarrow [\![\#S]\!]$, on représente A par une matrice booléenne carrée a de dimension $\#S$ t.q. `a[i][j]` vaut *vrai* (resp. *faux*) s'il y a (resp. n'y a pas) un arc (i, j) dans A

- On peut remplacer les valeurs booléennes par des poids, des étiquettes…

Quelques comparaisons des représentations

- Taille (avec poids etc. de taille $O(1)$) :
  - Liste de listes d'adjacence : $O(\#S + \#A)$
  - Matrice d'adjacence : $O(\#S^2)$
- Coût pour déterminer si $(u, v) \in A$
  - LA : $O(d(u)) = O(\#S)$
  - MA : $O(1)$
- Coût pour visiter les voisins de u
  - LA : $O(d(u))$
  - MA : $O(\#S)$

Matrice d'adjacence : bien adapté aux graphes *denses* (où $\#A \approx \#S^2$)

Exercice / exemple

Table des matières

Modélisation

Les graphes modélisent « naturellement » beaucoup de systèmes. Par exemple :

- Des réseaux (télécom, électrique, de transport de personnes…)
  - Exemple de poids : longueur d'une route
- Des relations entre entités (souhaits d'intégration v. propositions d'admission)
  - Exemple de poids : ordre de préférence
- Des circuits/programmes sans branchement
  - Exemple d'étiquette : opération ; littéral
- Les états d'un *automate*, d'un puzzle à résoudre

On peut alors formuler des problèmes sur de tels systèmes en terme des graphes qui les modélisent, pour les résoudre algorithmiquement

- Beaucoup de problèmes *d'optimisation* (« quelle est la meilleure façon de… »), mais aussi de *décision* (« est-ce qu'il est vrai que… ») ou de recherche (« trouver quelque chose qui… »)

Représentation #4 : représentations implicites

- Il peut arriver qu'un graphe ne soit pas représenté explicitement, notamment quand il est très grand
- Une représentation *implicite* se base à la place sur une description de S et une fonction décidant/listant l'adjacence

Exemples

- Graphe des parties d'un jeu (d'échecs, go…)
- Graphe fonctionnel d'une fonction
- Graphe d'un grand réseau

Exemples de problèmes de graphes #1

Quelques problèmes « structurels »

Ignorent les éventuelles informations auxiliaires (poids etc.)

Soit un graphe $G = (S, A)$:

- Soit $u, v \in S$, existe-t'il un chemin entre deux sommets ?
- Quelles sont les composantes (fortement) connexes de G ?
  - De plus si G orienté, quelle est sa *fermeture transitive* ?
- G possède-t'il des cycles ?
- G possède-t'il un chemin (ou cycle) « hamiltonien » passant exactement une fois par chaque sommet ?
  - Coûte cher
- — « eulérien » — arc ?
  - Ne coûte pas cher
- Quel est un *coloriage* optimal de G ?
- Soit un autre graphe G', est-il isomorphe à G (« identique à renommage près »)

Exemples de problèmes de graphes #2

Problèmes pas uniquement structurels

Font intervenir la structure du graphe, mais aussi les éventuelles informations auxiliaires
Soit un graphe $G = (S, A)$:

- Soit $u, v \in S$ reliés par un chemin, quel est un *plus court chemin* entre eux ; un chemin de *capacité maximum* ?
- G possède-t'il un cycle de poids négatif ?
- Si G possède un cycle hamiltonien, quel en est un de poids minimum ?

Table des matières

Parcours : définition

Un parcours de graphe depuis un sommet u est un algorithme *visitant* exactement les sommets v t.qu'il existe un chemin $u \rightsquigarrow v$, *uniquement en suivant les arcs du graphe*

- *visiter* : appliquer un traitement (par ex. ajouter à une liste des sommets visités)
- Même chose qu'un parcours d'arbre, mais pour un graphe…

Contre-exemple

- Un algorithme qui pour un graphe non-orienté garanti connexe tire un ordre aléatoire sur les sommets et les visite dans cet ordre

Parcours : utilité

- Permet de parcourir (…) un graphe même quand il est représenté de façon implicite, et de calculer la relation d'accessibilité entre sommets
  - Un parcours depuis un sommet visite exactement le sous-graphe accessible depuis lui-même (sa composante connexe), mais on peut évt. recommencer
- Respecte la « topologie » du graphe
  - Permet d'apprendre des choses sur la connexité, les cycles, les distances…

Préliminaires : piles & files en Python

Piles : *too easy*

Correspondance immédiate & efficace avec des opérations sur les `list`

- `def make(): return []`
- `def isEmpty(s): return len(s) == 0`
- `def push(v, s): s.append(v)`
- `def peek(s): return s[len(s) - 1]`
- `def pop(s): return s.pop()`

(Aussi : avec des `dict` et `popitem()` 🙈)

Files : ?

Surtout pas avec `q.pop(0)`

- Hors programme
- Très inefficace

Quelques approches :

- Avec un compteur et une « fuite en avant » (*cf. next*)
- Avec un module Python dédié
- Avec deux piles

Files : réalisation en Python

Version « fuite en avant »

Idée : on utilise des `list`, `.append` pour *push* ; jamais `.pop` ; et un pointeur vers le prochain élément à retirer

- `def make(): return [1]`
- `def isEmpty(q): return len(q) == q[0]`
- `def push(v, q): q.append(v)`
- `def peek(q): return q[q[0]]`
- `def pop(q): q[0] = q[0] + 1; return q[q[0] - 1]`

On ne supprime jamais d'éléments dans la `list` : efficace (seulement) si on n'ajoute pas *trop* d'éléments

- Pas un problème pour l'usage qui sera fait en parcours de graphe

Piles, Files avec `collections.deque`

Double-ended queue

Structure de données abstraite combinant file et pile :

- *push* : un « à gauche », un « à droite »
- *pop* : un « à gauche » (file), un « à droite » (pile)

Réalisation Python

Implémenté par `collections.deque`

```
from collections import deque
```

Files : version deque

- `def make(): return deque()`
- `def isEmpty(q): return len(q) == 0`
- `def push(v, q): q.append(v)`
- `def peek(q): return q[0]`
- `def pop(q): return q.popleft()`

Retour aux parcours

- Pour un graphe représenté par liste de listes d'adjacences

Principe générique

On part d'un sommet, on l'ajoute dans une structure (pile, file, …) de sommets à traiter, puis tant qu'il y a des sommets à traiter on en choisit un « u », on découvre ses voisins et ajoute les non traités aux sommets à traiter, on marque u comme ayant été traité

- En fonction de la structure utilisée, on obtient différents *types* de parcours
- Attention : l'ordre des opérations à un impact ; ici : le marquage des sommets est **tardif** (quand on les *extrait* de la structure, pas quand on les insère)

Parcours : algorithme générique (“whatever-first search’ ’, Erickson)

```
def wfs(G, u, make, push, pop, isEmpty):
    vis = {} # pour stocker les sommets visités
    bag = make()
    push((u, None), bag)
    while not isEmpty(bag):
        v, p = pop(bag) # magic!
        if v not in vis: # pas encore traité
            vis[v] = p # on a visité v depuis p
            for w in G[v]:
                if w not in vis: # à traiter, si pas déjà fait
                    push((w, v), bag)
    return vis
```

À éventuellement répéter pour chaque sommet non encore visité…

Algorithme générique (2)

Détails des arguments

- `G` : liste ou dictionnaire t.q. `G[v]` est une liste d'adjacence du sommet v
- `u` : origine (point de départ) du parcours
- `make`, `push`, `pop`, `isEmpty` : fonctions correspondante d'une structure abstraite (pile, file…)

Reconstruction d'un chemin

Ici, `vis[v]` est le sommet depuis lequel v a été (effectivement) découvert

- Permet de reconstruire un « chemin » de v à l'origine du parcours (attention dans le cas d'un graphe orienté !)

```
def buildPath(vis, v):
    pt = [v]
    while vis[v] != None:
        v = vis[v]
        pt.append(v)
    return pt
```

Il peut parfois être utile de stocker d'autres informations lors des visites

Algorithme générique : coût

Coût

Pour un parcours **depuis un sommet** u dans un graphe de $\#S$ sommets ($\#S_u$ accessibles depuis u), $\#A_u$ arcs accessibles depuis u :

- Basique : $O(\#A_u + \#S)$ temps et espace
- Basique + structure efficace pour visite : $O(\#A_u)$ temps et espace *(exemple précédent)*
- Basique + structure efficace pour visite + test bag (si possible) : $O(\#A_u)$ temps, $O(\#S_u)$ espace

Dans tous les cas : linéaire en la taille de l'entrée (« efficace ») !

Parcours en largeur, profondeur

Parcours en largeur

Les sommets sont visités par distance (non pondérée) croissante à l'origine

- Implémentation possible : parcours générique avec une file pour *bag*

Exemples d'applications

- Donne directement un algorithme de plus-court-chemins non pondérés
- Base utile pour des recherches de distance plus sophistiquées, par ex. pour des plus-court-chemins pondérés (*cf.* prochain cours)

Parcours en profondeur

Les temps de pré- et post-visite sont bien parenthésés pour toute paire de sommet 

- Implémentation possible : parcours générique avec une pile pour *bag*
- Alternative : récursivement

Exemples d'applications 

- Détection de cycles dans un graphe orienté
- Tri topologique d'un graphe orienté acyclique

Table des matières

Test de connexité dans un graphe non-orienté

Immédiat

- Comment ?

Détection de cycle dans un graphe non-orienté

Lemme

Soit $G = (S, A)$ un graphe non-orienté, C ses composantes connexes, alors il est sans cycle ssi. $\#A = \#S - \#C$

Esquisse de preuve

On construit une suite de sous-graphes de G avec $G_0 = (S, \varnothing)$ et G_{i+1} obtenu en ajoutant une arête quelconque de A à G_i. On s'arrête quand on atteint G.
On montre l'invariant sur les $G_i = (S, A_i)$: $\#A_i = \#S - \#C_i$ ou (exclusif)
($\#A_i > \#S - \#C_i$ et G_i possède un cycle).

- Cas de base G_0 : $\#S$ composantes connexes, 0 arêtes et pas de cycles : $0 = \#S - \#S$ est vérifié
- Récurrence : l'arête ajoutée peut :
  - relier deux sommets de composantes connexe différentes, ce qui ne crée pas de nouveau cycle et diminue de 1 le nombre de composantes connexes dans G_{i+1}
  - relier deux sommets de la même composante connexe, ce qui crée un cycle et n'augmente pas le nombre de composantes connexes dans G_{i+1}

Détection de cycle dans un graphe non-orienté (bis)

(Maintenant) (relativement) immédiat

- On compte le nombre de composantes connexes
- On conclut en comparant au nombre d'arêtes et de sommets
- Coût (pour une représentation par listes d'adjacence) : somme des coûts des parcours de chaque composante connexe, chacun linéaire en leur taille (si suffisamment malin) : total linéaire en la taille de G : $O(\#S + \#A)$

Optimisation

- Montrez qu'on peut détecter la présence d'un cycle en coût $O(\#S)$ (indépendant de $\#A$) (pour une représentation par listes d'adjacence)

Bicoloriage d'un graphe biparti