Algorithmique et Analyse d'Algorithmes

L3 Info
Cours 10 : Algorithmes gloutons
Coloration de graphe

Benjamin Wack

2025 - 2026

La dernière fois

- Codage
- Entropie
- Algorithme de Huffman

Aujourd'hui

- Problèmes d'optimisation
- Algorithmes gloutons
- Coloration de graphes

Plan

Problèmes d'optimisation
- Algorithmes gloutons
- Un exemple complet : le problème de choix des activités

Graphes
- Définitions, notations
- Manipulation algorithmique
- Complexité des algorithmes de graphes

Coloration de graphe

Définition

Un problème d'optimisation a les caractéristiques suivantes :

- Une solution est un **sous-ensemble** d'une des données du problème.
- Il existe en général plusieurs solutions **admissibles**.
- À chaque solution (admissible) est associée une **valeur** (en général un coût ou un gain).

Le problème d'optimisation consiste non seulement à trouver une solution admissible, mais à trouver une solution de valeur **minimale** (pour un coût) ou **maximale** (pour un gain).

Un exemple : rendu de monnaie

- **Problème** : On dispose de pièces de valeurs $v_1, v_2, \ldots$ (en nombre illimité).
  On cherche à fournir une somme d'argent S en monnaie.
- **Solution** : une suite de (valeurs de) pièces ayant pour total S
- **Meilleure solution** : celle utilisant le moins possible de pièces

Définition

Il s'agit d'*une stratégie particulière* pour résoudre un problème d'optimisation.

Algorithme glouton

Un algorithme glouton est un algorithme qui construit une telle solution :

- élément par élément **sans jamais revenir en arrière**
- en se basant sur des considérations **locales**.

On espère que la séquence construite est un optimum global ; ses sous-séquences sont des optimums des sous-problèmes.

Il n'existe pas toujours un algorithme glouton pour résoudre un problème d'optimisation.

Il s'agit d'une caractéristique de l'algorithme mais surtout de la structure du problème (notamment c'est impossible s'il existe des optima locaux).

Algorithme glouton pour le rendu de monnaie

Algorithme glouton

1. Tant que $S > 0$:
2. Choisir la pièce de plus grande valeur v inférieure à S
3. Recommencer avec $S - v$.

Exemple correct

$S = 16$ avec pièces de 10, 5, 2, 1 :

- $16 - 10 = 6$
- $6 - 5 = 1$
- $1 - 1 = 0$

3 pièces

Exemple incorrect

$S = 16$ avec pièces de 9, 8 et 1 :

- $16 - 9 = 7$
- $7 - 1 = 6$
- ...

8 pièces alors que $8 + 8 = 16$ en 2 pièces

Algorithme glouton générique

Initialiser un ensemble *Utilisables*
Initialiser $WIP := \emptyset$ `// ensemble vide ou suite vide ou ...`

tant que *WIP est incomplet et Utilisables n'est pas vide*
  Sélectionner x dans *Utilisables* `// selon critère glouton`

  si x *compatible avec WIP* `// dans certains problèmes`
   `// c'est toujours le cas`
    Insérer x dans *WIP*

  Mise à jour de *Utilisables* `// par exemple : retirer x`

Renvoyer *WIP*

Complexité

Le choix du prochain élément est en principe efficace car il ne considère qu'une partie des données :

- $\log n$ dans une FAP
- 1 pour les pièces de monnaie (si leurs valeurs sont préalablement triées)
- ...

La complexité de l'algorithme sera en général de la forme

$$\mathcal{O}(n \times f(n))$$

où f (le coût du choix) est une fonction sub-linéaire.

Preuve de correction

Schéma de preuve générique

Pour la boucle principale, on maintient l'invariant :

Il existe une solution optimale qui contient WIP.

La preuve de cet invariant se fait en utilisant les propriétés de l'opération **Sélectionner** et de la mise à jour de *Utilisables*.

Lorsqu'il y a plusieurs solutions optimales, il est souvent nécessaire de faire appel à une propriété « d'échange » :

Supposons que $WIP \subseteq Opt$ une solution optimale et que le choix glouton ajoute à WIP un élément $x \notin Opt$.

Alors il existe un élément $y \in Opt \setminus WIP$ tel que, en remplaçant y par x, on obtient une autre solution aussi bonne que Opt.

Exemple de preuve : rendu de monnaie avec 1,3,5,7

Rappel de l'invariant : *Il existe une solution optimale qui contient WIP.*

Initialisation

Avant la première itération, l'ensemble WIP est vide, donc inclus dans n'importe quelle solution optimale.

Maintien de l'invariant

Supposons que $WIP \subseteq Opt$ une solution optimale
et que le choix glouton soit une pièce $x \notin Opt \setminus WIP$.

Deux remarques :

- la plus grosse pièce de $Opt \setminus WIP$ est forcément $< x$
- et donc $Opt \setminus WIP$ contient au moins deux pièces.

Il reste à montrer qu'on peut échanger ces deux pièces pour utiliser x.

Preuve du rendu de monnaie 1,3,5,7 : les échanges

Si $p = \dots$	Si *Opt* commence par ...	alors on remplace par ...
7	5+5	7+3
7	5+3	7+1
7	~~5+1+1~~	*Opt* n'est pas optimal !
7	3+3+3	7+1+1
7	~~3+1~~	
7	~~1+1~~	
5	3+3	5+1
5	~~3+1~~	
5	~~1+1~~	
3	~~1+1~~	

On obtient ainsi systématiquement une solution

- qui commence par x
- qui contient autant de pièces que *Opt*.

Problème de choix des activités

Énoncé du problème

On dispose d'une salle pouvant être louée pour une durée variable.
On choisit parmi un ensemble de n demandes de location celles qui seront acceptées.

- Données : les dates de début d_i et de fin f_i de chaque demande i.
- Solution optimale = satisfaisant le plus de demandes.

CHOIX_ACTIVITES (demandes)
dispo := 0 `// Première date disponible`
Utilisables := toutes les demandes (triées par date de fin croissante)
WIP := $\emptyset$
tant que *il reste des demandes non étudiées*
  x := prochaine demande par **date de fin croissante**
  si $d_x \geq dispo$
    Insérer x dans *WIP*
    $dispo := f_x$

Preuve de l'algorithme glouton

On maintient l'invariant : *WIP est un préfixe d'une solution optimale.*
En début d'algorithme *WIP* est vide, c'est donc un préfixe de toute solution.

Supposons qu'au début d'une itération *WIP* est un préfixe d'une solution optimale *Opt*.

- Si l'itération n'ajoute pas de demande à *WIP*,
  alors il reste un préfixe de la même *Opt*.
- De même si elle ajoute la première demande de *Opt* à *WIP*.
- Sinon, on appelle x la demande ajoutée à *WIP*.

Preuve de l'algorithme glouton (2)

On distingue alors deux possibilités sur la *première demande* de $Opt \setminus WIP$:

- c'est une demande y qui finit avant x : impossible car dans ce cas, à une itération précédente, on a refusé la demande y qui était incompatible avec les autres demandes de WIP.
- c'est une demande y qui finit après x (ou en même temps) : on montre qu'on peut remplacer y par x.

Propriété d'échange

Si la première demande de $Opt \setminus WIP$ est une demande y telle que $f_y \geq f_x$, alors $Opt' = (Opt \setminus \{y\}) \cup \{x\}$ est également une solution optimale :

- les demandes de Opt antérieures à x sont dans WIP donc compatibles avec x.
- $f_y \geq f_x$ donc la demande x est compatible avec toutes les autres demandes de Opt (qui ont des dates postérieures à f_y).
- Opt' est de même taille que Opt.

Un autre algorithme glouton déjà connu

L'algorithme de Huffman est glouton :

C'est un problème d'optimisation (de la longueur moyenne du code).

Choix glouton : tout nœud construit est définitif

Le critère de choix du prochain nœud repose sur le poids minimal.
(ne nécessite pas de tous les reparcourir si la file à priorité est implémentée dans un tas)

Optimalité

À toute itération la forêt construite est incluse dans un arbre de codage optimal.

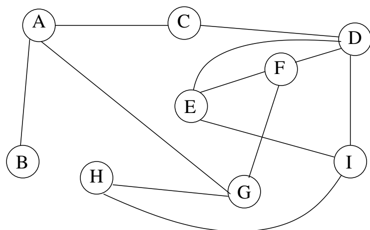
L'algorithme de Huffman produit un code optimal.

Notions de base

Graphe non orienté

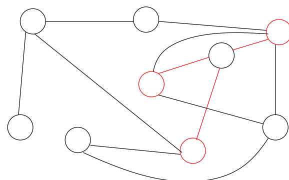
Un graphe non orienté est un couple $G = (X, R)$:

- X est l'ensemble des sommets
- R est une relation binaire **symétrique** sur X (ensemble de couples non ordonnés de X).



Sur l'exemple : $R = \{(B, A), (A, C), (C, D), (D, E), (D, F), (D, I), \ldots\}$

Voisinages



Voisin

On appelle voisin d'un sommet x tout sommet y tel que (x, y) (ou de façon équivalente (y, x)) est une arête du graphe.

Degré d'un sommet

Le degré d'un sommet x est le nombre d'arêtes *incidentes* à x, autrement dit le nombre de voisins de x.

Degré d'un graphe

Le degré d'un graphe est le maximum des degrés de ses sommets.

Étiquettes, pondération

Dans un graphe il est possible d'étiqueter :

- les sommets
- et / ou les arêtes

au moyen d'une fonction de X (respectivement de R) dans un ensemble d'étiquettes donné.

Si les étiquettes sont à valeur numérique (entier, réel...) elles peuvent représenter un poids (coût, valeur...) pour les sommets ou les arêtes : on parle alors de graphe pondéré.

Applications

- Réseau (routier, de communication)
  - Calculer un itinéraire
  - Identifier les goulots d'étranglement
- Conflits
  - Chaque sommet représente un processus, une activité...
  - Chaque arête représente une compétition pour une ressource
  - Déterminer le nombre de ressources nécessaires, ou les activités compatibles
- Dépendances (avec un graphe *orienté*)
  - Chaque sommet représente une tâche
  - L'origine de chaque arc doit être réalisée avant son extrémité
  - Détecter une incohérence, calculer un ordre approprié
- ...

De façon générale toute relation, orientée ou non, se traduit par un graphe qui permet ensuite de raisonner algorithmiquement sur cette relation.

Type abstrait

Opérations

GrapheVide	:	void $\to$ *Graphe*
AjouterSommet	:	*Etiquette* $\times$ *Graphe* $\to$ *Graphe*
AjouterArete	:	*Sommet* $\times$ *Sommet* $\times$ *Graphe* $\to$ *Graphe*
ensSommets	:	*Graphe* $\to$ *ensemble*(*Sommet*)
existeArete	:	*Sommet* $\times$ *Sommet* $\times$ *Graphe* $\to$ *bool*
OU ensVoisins	:	*Sommet* $\times$ *Graphe* $\to$ *ensemble*(*Sommet*)

$+$ si besoin : suppressions, étiquettes des arêtes...

Préconditions Pas de difficulté particulière

Axiomes AjouterArete(x,y,G) rend à la fois x voisin de y et inversement.

Parcours de graphe

Principes communs

On choisit un sommet (origine) par lequel commencer le parcours.
On traite chaque sommet **une seule fois** (attention aux cycles !).

Parcours en profondeur

On suit un chemin aussi « loin » que possible.
Si nécessaire on revient en arrière pour explorer d'autres chemins.

Parcours en largeur

On procède par « cercles concentriques » autour d'un sommet x : on traite **tous** ses voisins, puis les sommets situés à distance 2, etc.

Remarque

Parfois on ne parvient pas à visiter tout à partir de l'origine choisie.
Dans ce cas il faut « relancer » à partir d'une autre origine.

Notion de marquage

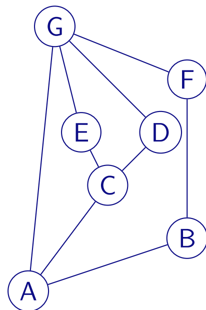
Les structures linéaires possèdent un ordre de parcours naturel.

Dans les arbres on va assez naturellement :

- de la racine vers les feuilles
- de la gauche vers la droite

Dans un graphe c'est plus compliqué :

- pas de « point d'entrée » (ou de sortie) unique
- risque de revenir à un endroit déjà visité



Marquage

Dans un algorithme qui parcourt un graphe, il est nécessaire de marquer les sommets déjà traités afin d'assurer la **terminaison** de l'algorithme.

- jamais vu
- marqué
- en cours de traitement
- déjà traité

Notion de marquage

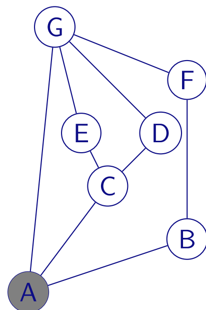
Les structures linéaires possèdent un ordre de parcours naturel.

Dans les arbres on va assez naturellement :

- de la racine vers les feuilles
- de la gauche vers la droite

Dans un graphe c'est plus compliqué :

- pas de « point d'entrée » (ou de sortie) unique
- risque de revenir à un endroit déjà visité



Marquage

Dans un algorithme qui parcourt un graphe, il est nécessaire de marquer les sommets déjà traités afin d'assurer la **terminaison** de l'algorithme.

- jamais vu
- marqué
- en cours de traitement
- déjà traité

Notion de marquage

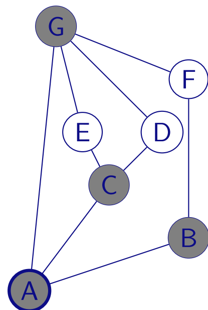
Les structures linéaires possèdent un ordre de parcours naturel.

Dans les arbres on va assez naturellement :

- de la racine vers les feuilles
- de la gauche vers la droite

Dans un graphe c'est plus compliqué :

- pas de « point d'entrée » (ou de sortie) unique
- risque de revenir à un endroit déjà visité



Marquage

Dans un algorithme qui parcourt un graphe, il est nécessaire de marquer les sommets déjà traités afin d'assurer la **terminaison** de l'algorithme.

- jamais vu
- marqué
- en cours de traitement
- déjà traité

Notion de marquage

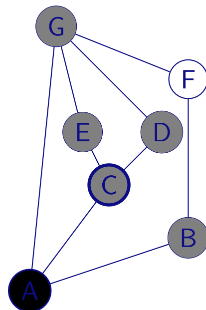
Les structures linéaires possèdent un ordre de parcours naturel.

Dans les arbres on va assez naturellement :

- de la racine vers les feuilles
- de la gauche vers la droite

Dans un graphe c'est plus compliqué :

- pas de « point d'entrée » (ou de sortie) unique
- risque de revenir à un endroit déjà visité



Marquage

Dans un algorithme qui parcourt un graphe, il est nécessaire de marquer les sommets déjà traités afin d'assurer la **terminaison** de l'algorithme.

- jamais vu
- marqué
- en cours de traitement
- déjà traité

Notion de marquage

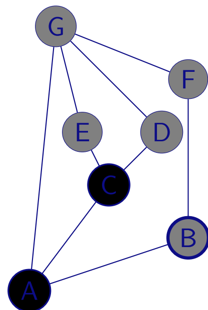
Les structures linéaires possèdent un ordre de parcours naturel.

Dans les arbres on va assez naturellement :

- de la racine vers les feuilles
- de la gauche vers la droite

Dans un graphe c'est plus compliqué :

- pas de « point d'entrée » (ou de sortie) unique
- risque de revenir à un endroit déjà visité



Marquage

Dans un algorithme qui parcourt un graphe, il est nécessaire de marquer les sommets déjà traités afin d'assurer la **terminaison** de l'algorithme.

- jamais vu
- marqué
- en cours de traitement
- déjà traité

Notion de marquage

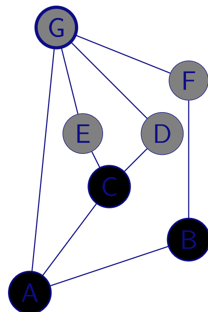
Les structures linéaires possèdent un ordre de parcours naturel.

Dans les arbres on va assez naturellement :

- de la racine vers les feuilles
- de la gauche vers la droite

Dans un graphe c'est plus compliqué :

- pas de « point d'entrée » (ou de sortie) unique
- risque de revenir à un endroit déjà visité



Marquage

Dans un algorithme qui parcourt un graphe, il est nécessaire de marquer les sommets déjà traités afin d'assurer la **terminaison** de l'algorithme.

- jamais vu
- marqué
- en cours de traitement
- déjà traité

Notion de marquage

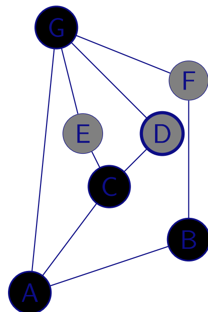
Les structures linéaires possèdent un ordre de parcours naturel.

Dans les arbres on va assez naturellement :

- de la racine vers les feuilles
- de la gauche vers la droite

Dans un graphe c'est plus compliqué :

- pas de « point d'entrée » (ou de sortie) unique
- risque de revenir à un endroit déjà visité



Marquage

Dans un algorithme qui parcourt un graphe, il est nécessaire de marquer les sommets déjà traités afin d'assurer la **terminaison** de l'algorithme.

- jamais vu
- marqué
- en cours de traitement
- déjà traité

Notion de marquage

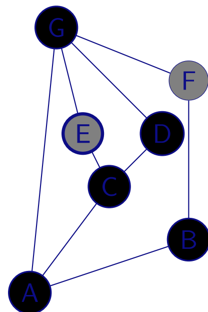
Les structures linéaires possèdent un ordre de parcours naturel.

Dans les arbres on va assez naturellement :

- de la racine vers les feuilles
- de la gauche vers la droite

Dans un graphe c'est plus compliqué :

- pas de « point d'entrée » (ou de sortie) unique
- risque de revenir à un endroit déjà visité



Marquage

Dans un algorithme qui parcourt un graphe, il est nécessaire de marquer les sommets déjà traités afin d'assurer la **terminaison** de l'algorithme.

- jamais vu
- marqué
- en cours de traitement
- déjà traité

Notion de marquage

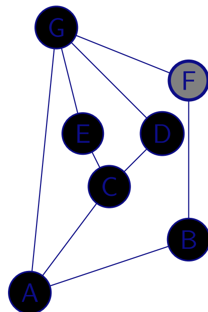
Les structures linéaires possèdent un ordre de parcours naturel.

Dans les arbres on va assez naturellement :

- de la racine vers les feuilles
- de la gauche vers la droite

Dans un graphe c'est plus compliqué :

- pas de « point d'entrée » (ou de sortie) unique
- risque de revenir à un endroit déjà visité



Marquage

Dans un algorithme qui parcourt un graphe, il est nécessaire de marquer les sommets déjà traités afin d'assurer la **terminaison** de l'algorithme.

- jamais vu
- marqué
- en cours de traitement
- déjà traité

Notion de marquage

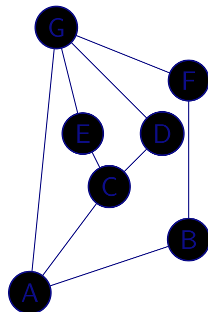
Les structures linéaires possèdent un ordre de parcours naturel.

Dans les arbres on va assez naturellement :

- de la racine vers les feuilles
- de la gauche vers la droite

Dans un graphe c'est plus compliqué :

- pas de « point d'entrée » (ou de sortie) unique
- risque de revenir à un endroit déjà visité



Marquage

Dans un algorithme qui parcourt un graphe, il est nécessaire de marquer les sommets déjà traités afin d'assurer la **terminaison** de l'algorithme.

- jamais vu
- marqué
- en cours de traitement
- déjà traité

Algorithme de parcours générique

```
PARCOURS_GRAPHE( Graphe g, Sommet origine )
L = ListeVide()
Marquer( origine )
Insérer( origine, L )
tant que ¬ EstVide(L)
    x = Premier(L)
    L = ExtrairePremier(L)
    Traiter( x )
    foreach y ∈ ensVoisins(x, g) do
        si y non marqué
            Marquer( y )
            Insérer( y, L )
```

Si L est une pile

Parcours en profondeur

Si L est une file

Parcours en largeur

+ si besoin mémoriser comment on a accédé à y : ancêtre, distance...

Mesure de complexité

Rappel

La complexité d'un algorithme s'exprime en fonction de la taille de la donnée.

La taille d'un graphe dépend de deux paramètres :

- le nombre de sommets V,
- le nombre d'arêtes E,

qui ne sont pas liés linéairement : en général $0 \leq E \leq V^2$.

On précisera donc toujours la complexité d'un algorithme en fonction de E et/ou de V.

Parcours de graphe

L'algorithme de parcours générique vu plus haut est en $\mathcal{O}(E + V)$.
(ce qui revient à écrire $\mathcal{O}(max(E, V))$)

Incidence de la représentation concrète

Selon la représentation choisie, la complexité des opérations élémentaires peut varier :

- Déterminer si deux sommets sont voisins peut être en temps :
  - constant
  - ou proportionnel à V.
- Parcourir la liste des voisins d'un sommet peut être en temps :
  - proportionnel à V
  - ou proportionnel à son degré.

L'empreinte en mémoire d'une structure de graphe peut elle-même varier entre $E + V$ et V^2.

Il est donc parfois délicat de quantifier la complexité d'un algorithme de graphe en se basant uniquement sur le type abstrait : on énonce ce qui est **possible sous réserve d'une représentation adéquate**.

(cf TD2 pour les deux représentations les plus utilisées)

Optimisation dans les graphes

Dans le contexte des graphes, de nombreux problèmes d'optimisation consistent à déterminer un ensemble optimal de sommets et/ou d'arêtes.

- Réseau
  - Itinéraire le plus court, le plus rapide
    (ensemble d'arêtes de poids total minimal formant un chemin)
  - Tolérance aux pannes
    (ensemble de sommets de cardinal minimal brisant la connexité du graphe)
- Conflits
  - Maximiser le nombre d'activités simultanées
    (ensemble de sommets de valeur maximale sans arête interne)
  - Minimiser le nombre de ressources nécessaires
    (coloration des sommets compatible avec les arêtes utilisant un minimum de couleurs)
- ...

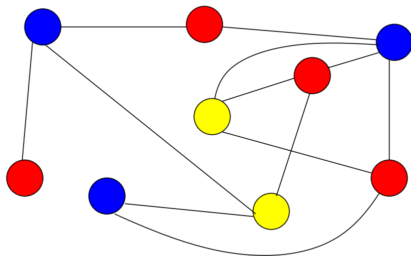
Coloration de graphe

Énoncé du problème

Soit un graphe $G = (X, R)$.
Une coloration de G est une fonction $c : X \to \mathbb{N}$

- qui à chaque sommet x associe la couleur numéro $c(x)$
- telle que si x et y sont voisins alors $c(x) \neq c(y)$.

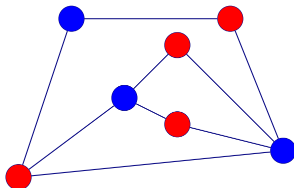
On cherche de plus à utiliser le moins possible de couleurs.

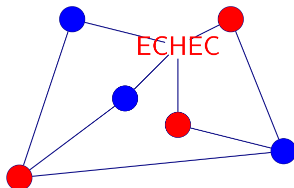


Un graphe G est-il colorable avec 2 couleurs seulement ?

Ce problème admet un algorithme glouton :

- Colorer l'origine (couleur 1)
- Parcourir le graphe (en largeur ou en profondeur, peu importe)
- Pour chaque voisin y du sommet x courant :
  - si y est déjà coloré comme x, échec
  - si y est déjà coloré différemment de x, rien à faire
  - sinon (y n'est pas encore coloré) :
    colorer y différemment de x (et le marquer pour le visiter plus tard)



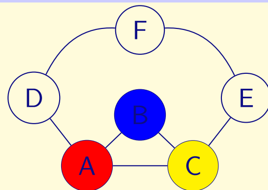


Un graphe G est-il 3-colorable ?

Ce problème n'est pas résoluble par un algorithme glouton.
Preuve : supposons qu'un tel algorithme existe et construisons un **adversaire** : une donnée sur laquelle l'algorithme échoue.

Notre adversaire

Soit le graphe suivant :



Les sommets A, B, C doivent clairement prendre 3 couleurs distinctes.

Supposons que l'algorithme glouton colore D,E,F dans cet ordre :

- s'il colore D et/ou E en bleu, alors on ajoute 2 arêtes A-F et C-F
- s'il colore D en jaune et E en rouge alors on ajoute l'arête B-F

Dans les deux cas l'algo glouton aura besoin d'une 4è couleur pour F, alors qu'une 3-coloration suffisait.

En résumé

Aujourd'hui

- Les graphes constituent un bon support à beaucoup d'algorithmes.
- Plusieurs parcours d'un graphe sont possibles, avec des précautions supplémentaires à cause des éventuels **cycles**.
- La complexité d'un algorithme de graphes s'exprime en fonction du nombre de **sommets et** du nombre d'**arêtes**.
- Les graphes se prêtent à de nombreux problèmes d'optimisation.
- Un algorithme glouton permet de résoudre efficacement un problème d'optimisation, mais **il n'en existe pas toujours**.

La prochaine fois

- Arbre couvrant
- Algorithmes de Prim et Kruskal
- Un problème de graphe orienté : le tri topologique