

# Algorithmique et Analyse d'Algorithmes

L3 Info

Cours 10 : Algorithmes gloutons

Coloration de graphe

Benjamin Wack



2021 - 2022

## La dernière fois

- ▶ Arbre partiellement ordonné
- ▶ Structure de tas
- ▶ Application à la FAP
- ▶ Diviser pour Régner

## Aujourd'hui

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

# Plan

## Problèmes d'optimisation

- Algorithmes gloutons

- Exemple : 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  $v_1, v_2, \dots$  (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**.

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  $Sol := \emptyset$  // ensemble vide ou suite vide ou ...

**while** *Sol est incomplète* et *Utilisables n'est pas vide*

**Sélectionner**  $x$  dans *Utilisables* // selon critère glouton

**if**  $x$  *compatible avec Sol* // dans certains problèmes

        // c'est toujours le cas

**Ajouter**  $x$  à *Sol*

**if** *condition* // true ou  $x$  incompatible ou ...

        Retirer  $x$  de *Utilisables*

Renvoyer *Sol*

# 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 dont  $Sol$  est un préfixe.*

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  $Sol \subseteq S$  une solution optimale et que le choix glouton ajoute à  $Sol$  un élément  $x \notin S$ .*

*Alors il existe un élément  $y \in S \setminus Sol$  tel que  $S \cup \{x\} \setminus \{y\}$  soit une autre solution au moins aussi bonne que  $S$ .*

# 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)

Trier les demandes par **date de fin** croissante.

$Sol := \emptyset$

$dispo := 0$

// Première date disponible

for  $i := 1$  to  $n$

    if  $d_i \geq dispo$

        // demande acceptable

        Insérer  $i$  dans  $Sol$

$dispo := f_i$

# Preuve de l'algorithme glouton

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

Supposons qu'au début de la  $i$ ème itération *Sol* est un préfixe d'une solution optimale  $S$ .

- ▶ Si l'itération n'ajoute pas de demande à *Sol* alors elle reste un préfixe de la même  $S$ .
- ▶ Sinon, elle ajoute la demande  $i$  à *Sol*.

## Preuve de l'algorithme glouton (2)

On distingue alors trois possibilités sur la *première demande* de  $S \setminus Sol$  :

- ▶ c'est une demande  $k < i$  : impossible car dans ce cas, à l'itération  $k$ , on a refusé la demande  $k$  qui était incompatible avec la dernière demande de  $Sol$ .
- ▶ c'est la demande  $k = i$  : alors  $Sol \cup \{i\}$  est encore un préfixe de  $S$ .
- ▶ c'est une demande  $k > i$  : on montre qu'on peut la remplacer par  $i$ .

### Propriété d'échange

Si la première demande de  $S \setminus Sol$  est une demande  $k > i$ , alors  $T = (S \setminus \{k\}) \cup \{i\}$  est également une solution optimale :

- ▶  $f_k \geq f_i$  donc la demande  $i$  est compatible avec toutes les autres demandes de  $S$  (qui ont des dates postérieures à  $f_k$ ).
- ▶  $T$  est de même taille que  $S$ .

# Un autre algorithme glouton déjà connu

L'algorithme de Huffman est **glouton** :

Choix glouton : **tout nœud construit est définitif**

- ▶ Le choix du prochain nœud repose sur le maintien des 2 nœuds de poids minimal.  
(ne nécessite pas de tous les reparcourir si la file à priorité est implantée dans un tas)

Optimalité

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

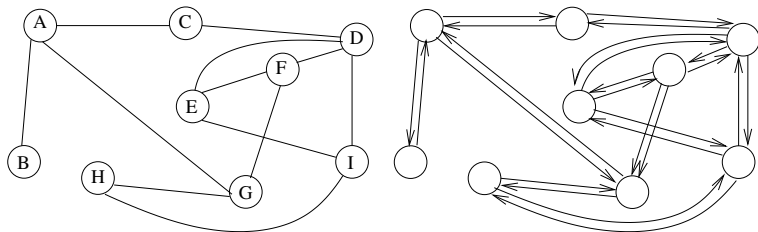
Donc l'algorithme de Huffman produit un code optimal.

# 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  $X$ ).

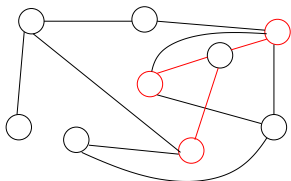


Sur l'exemple :  $R = \{(B, A), (A, C), (C, D), (D, E), (D, F), (D, I), \dots\}$

# 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 **poïds** (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	:	$\text{void} \rightarrow \text{Graphe}$
AjouterSommet	:	$\text{Etiquette} \times \text{Graphe} \rightarrow \text{Graphe}$
AjouterArete	:	$\text{Sommet} \times \text{Sommet} \times \text{Graphe} \rightarrow \text{Graphe}$
ensSommets	:	$\text{Graphe} \rightarrow \text{ensemble}(\text{Sommet})$
existeArete	:	$\text{Sommet} \times \text{Sommet} \times \text{Graphe} \rightarrow \text{bool}$
<b>OU</b> ensVoisins	:	$\text{Sommet} \times \text{Graphe} \rightarrow \text{ensemble}(\text{Sommet})$

+ 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

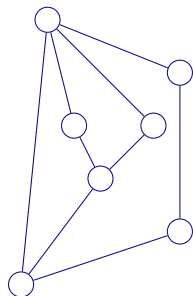
Les structures linéaires possèdent un ordre de parcours naturel.

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.



# Notion de marquage

Les structures linéaires possèdent un ordre de parcours naturel.

Dans les arbres on va assez naturellement :

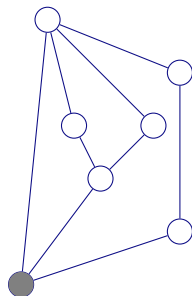
- ▶ de la racine vers les feuilles
- ▶ de la gauche vers la droite

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 :

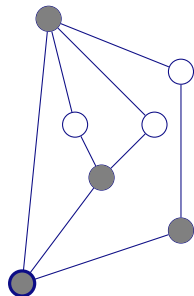
- ▶ de la racine vers les feuilles
- ▶ de la gauche vers la droite

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 :

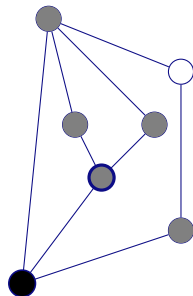
- ▶ de la racine vers les feuilles
- ▶ de la gauche vers la droite

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 :

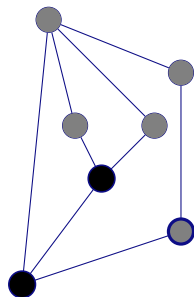
- ▶ de la racine vers les feuilles
- ▶ de la gauche vers la droite

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 :

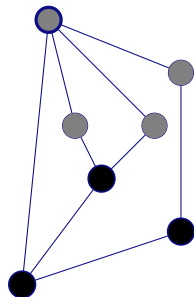
- ▶ de la racine vers les feuilles
- ▶ de la gauche vers la droite

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 :

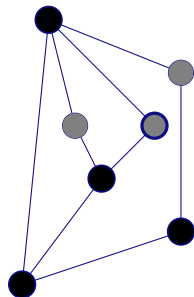
- ▶ de la racine vers les feuilles
- ▶ de la gauche vers la droite

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 :

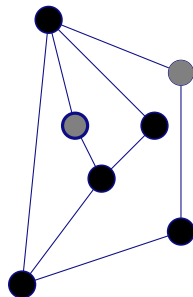
- ▶ de la racine vers les feuilles
- ▶ de la gauche vers la droite

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 :

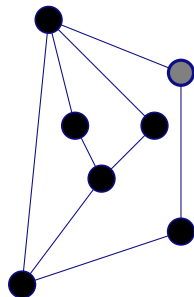
- ▶ de la racine vers les feuilles
- ▶ de la gauche vers la droite

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 :

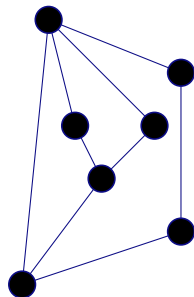
- ▶ de la racine vers les feuilles
- ▶ de la gauche vers la droite

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  $origine$  )

$L = \text{ListeVide}()$

Marquer(  $origine$  )

Insérer(  $origine, L$  )

**while**  $\neg \text{EstVide}(L)$

$x = \text{Premier}(L)$

$L = \text{ExtrairePremier}(L)$

    Traiter(  $x$  )

**foreach**  $y \in \text{ensVoisins}(x, g)$  **do**

**if**  $y$  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

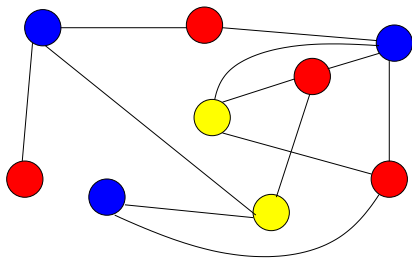
## Énoncé du problème

Soit un graphe  $G = (X, R)$ .

Une coloration de  $G$  est une fonction  $c : X \rightarrow \mathbb{N}$

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

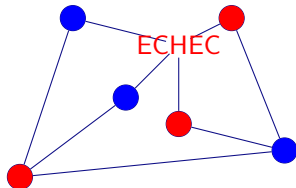
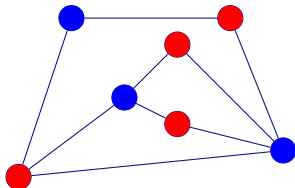
On cherche de plus à utiliser le moins possible de couleurs.



## 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 colorié comme  $x$ , **échec**
  - ▶ si  $y$  est déjà colorié différemment de  $x$ , rien à faire
  - ▶ sinon ( $y$  n'est pas encore colorié) : colorer  $y$  différemment de  $x$  (et le marquer pour le visiter plus tard)



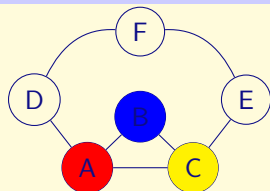
## Un graphe $G$ est-il 3-colorable ?

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 4<sup>e</sup> 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