

# Conception d'algorithmes et applications (LI325)

## Cours 7 et 8: Algorithmes Gloutons

Ana BUŠIĆ

`ana.busic@inria.fr`

# Principe général

- ▶ Technique pour résoudre des problèmes d'optimisation
- ▶ Solution optimale obtenue en effectuant une suite de choix :  
à chaque étape on fait le meilleur choix local
- ▶ Pas de retour en arrière : à chaque étape de décision dans l'algorithme, le choix qui semble le meilleur à ce moment est effectué.
- ▶ Progression descendante : choix puis résolution d'un problème plus petit

# Stratégie gloutonne

- ▶ **Propriété du choix glouton** : Il existe toujours une solution optimale commençant par un choix glouton.
- ▶ **Propriété de sous-structure optimale** : trouver une solution optimale contenant le premier choix glouton se réduit à trouver une solution optimale pour un sous-problème de même nature.

# Plan

## Introduction

Principe général

Exemple : Location d'un camion

## Arbre Couvrant Minimum

## Codage de Huffman

## Matroïdes

Définitions et propriétés

Algorithme GROUTON

Exemple : Ordonnancements

## Exemple : Location d'un camion

- ▶ Un unique véhicule à louer et maximiser le nombre de clients satisfaits (autres fonctions possibles à optimiser)
- ▶  $E$  ensemble de demandes ; chaque demande  $e_i$  est caractérisée par une date de début  $d_i$  et une date de fin  $f_i$ .
- ▶ Les demandes  $e_i$  et  $e_j$  sont compatibles ssi

$$]d_i, f_i[ \cap ]d_j, f_j[ = \emptyset.$$

- ▶ On cherche un sous-ensemble maximal de  $E$  de demandes 2 à 2 compatibles.

# Algorithme glouton

Exemple

$i$	1	2	3	4	5	6	7	8	9
$d_i$	1	2	4	1	5	8	9	13	11
$f_i$	8	5	7	3	9	11	10	16	14

Algorithme glouton : trier selon critère et satisfaire les demandes par ordre croissant.

# Algorithme glouton

## Exemple

$i$	1	2	3	4	5	6	7	8	9
$d_i$	1	2	4	1	5	8	9	13	11
$f_i$	8	5	7	3	9	11	10	16	14

Algorithme glouton : trier selon critère et satisfaire les demandes par ordre croissant.

Quel critère ?

- ▶ durée ?
- ▶ date de début ?
- ▶ date de fin ?

# Algorithme glouton

## Exemple

$i$	1	2	3	4	5	6	7	8	9
$d_i$	1	2	4	1	5	8	9	13	11
$f_i$	8	5	7	3	9	11	10	16	14

Algorithme glouton : trier selon critère et satisfaire les demandes par ordre croissant.

Quel critère ?

- ▶ durée ? **NON**
- ▶ date de début ? **NON**
- ▶ date de fin ? **OUI**



# Solution optimale

Demandes :

$i$	1	2	3	4	5	6	7	8	9
$d_i$	1	2	4	1	5	8	9	13	11
$f_i$	8	5	7	3	9	11	10	16	14

# Solution optimale

Demandes :

$i$	1	2	3	4	5	6	7	8	9
$d_i$	1	2	4	1	5	8	9	13	11
$f_i$	8	5	7	3	9	11	10	16	14

Etape 1 : tri selon la date de fin

$i$	4	2	3	1	5	7	6	9	8
$d_i$	1	2	4	1	5	9	8	11	13
$f_i$	3	5	7	8	9	10	11	14	16

# Solution optimale

Demandes :

$i$	1	2	3	4	5	6	7	8	9
$d_i$	1	2	4	1	5	8	9	13	11
$f_i$	8	5	7	3	9	11	10	16	14

Etape 1 : tri selon la date de fin

$i$	4	2	3	1	5	7	6	9	8
$d_i$	1	2	4	1	5	9	8	11	13
$f_i$	3	5	7	8	9	10	11	14	16

Etape 2 : satisfaire les demandes par ordre croissant

$i$	4	2	3	1	5	7	6	9	8
$d_i$	1	2	4	1	5	9	8	11	13
$f_i$	3	5	7	8	9	10	11	14	16

Solution : demandes 4, 3, 7, 9.

# Variantes

$E$  trié selon dates de fin :  $f_1 \leq f_2 \leq \dots \leq f_n$

$E, d, f$  variables globales

(Convention :  $f_0 := 0$ .)

## Réursive

---

Fonction  $R(i)$

---

- 1  $m = i + 1$ ;
  - 2 **tant que**  $m \leq n$  et  $d_m < f_i$  **faire**
  - 3     $m = m + 1$ ;
  - 4 **si**  $m \leq n$  **alors**
  - 5    **return**  $R(m)$
  - 6 **sinon**
  - 7    **return**  $\emptyset$
- 

Appel  $R(0)$

# Variantes

$E$  trié selon dates de fin :  $f_1 \leq f_2 \leq \dots \leq f_n$

$E, d, f$  variables globales

(Convention :  $f_0 := 0$ .)

## Réursive

---

Fonction  $R(i)$

---

```
1  $m = i + 1$ ;  
2 tant que  $m \leq n$  et  $d_m < f_i$  faire  
3   |  $m = m + 1$ ;  
4 si  $m \leq n$  alors  
5   | return  $R(m)$   
6 sinon  
7   | return  $\emptyset$ 
```

---

Appel  $R(0)$

## Itérative

---

Fonction  $P$

---

```
1  $s_1 = e_1$ ;  
2  $k = 1$ ;  
3 pour  $i$  allant de 2 à  $n$  faire  
4   | si  $d_i \geq f_k$  alors  
5     |  $s_{k+1} = e_i$ ;  
6     |  $k = k + 1$ ;  
7 return  $(s_1, \dots, s_k)$ 
```

---

# Variantes

$E$  trié selon dates de fin :  $f_1 \leq f_2 \leq \dots \leq f_n$

$E, d, f$  variables globales

(Convention :  $f_0 := 0$ .)

## Réursive

---

Fonction  $R(i)$

---

```
1  $m = i + 1$ ;  
2 tant que  $m \leq n$  et  $d_m < f_i$  faire  
3    $m = m + 1$ ;  
4 si  $m \leq n$  alors  
5   return  $R(m)$   
6 sinon  
7   return  $\emptyset$ 
```

---

Appel  $R(0)$

Complexité  $O(n \log n)$  (tri des  $n$  dates de fin)

## Itérative

---

Fonction  $P$

---

```
1  $s_1 = e_1$ ;  
2  $k = 1$ ;  
3 pour  $i$  allant de 2 à  $n$  faire  
4   si  $d_i \geq f_k$  alors  
5      $s_{k+1} = e_i$ ;  
6      $k = k + 1$ ;  
7 return  $(s_1, \dots, s_k)$ 
```

---

# Preuve de l'algorithme

Théorème : Le résultat de l'algorithme est optimal.

1. Il existe une solution optimale qui commence par  $e_1$  : soit  $A$  une sol. opt., et soit  $e_{a_1}$  le premier client ; si  $e_{a_1} \neq e_1$ , alors  $A - e_{a_1} + e_1$  est aussi une sol. opt.

# Preuve de l'algorithme

Théorème : Le résultat de l'algorithme est optimal.

1. Il existe une solution optimale qui commence par  $e_1$  : soit  $A$  une sol. opt., et soit  $e_{a_1}$  le premier client ; si  $e_{a_1} \neq e_1$ , alors  $A - e_{a_1} + e_1$  est aussi une sol. opt.
2. Le problème se ramène à trouver une solution optimale d'éléments de  $E$  compatibles avec  $e_1$ . Donc si  $A$  est une solution optimale pour  $E$ , alors  $A' = A - e_1$  est une solution optimale pour  $E' = \{e_i; d_i \geq f_1\}$ . En effet si l'on pouvait trouver  $B'$  une solution optimale pour  $E'$  contenant plus de clients que  $A'$ , alors ajouter  $e_1$  à  $B'$  offrirait une solution optimale pour  $E$  contenant plus de clients que  $A$ , ce qui contredirait l'hypothèse que  $A$  est optimale.



# Arbre Couvrant Minimum

$G = (S, A, w)$ , graphe connexe non orienté  
( $n$  sommets et  $m$  arêtes)

$w : A \rightarrow R$  (valuation sur les arêtes)

**Def.** Un arbre couvrant minimum (ACM) est un sous-graphe connexe sans cycle  $T = (S, B)$  ( $B \subset A$ ) qui minimise

$$w(T) = \sum_{(u,v) \in B} w(u, v).$$

# Algorithme ACM

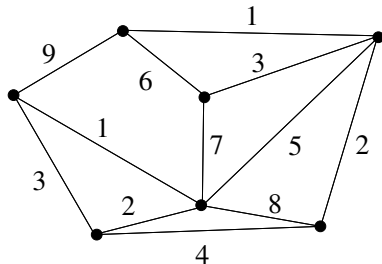
Calcul d'un ACM par stratégie gloutonne : la suite des minimums locaux (ajout d'une arête minimale à chaque étape) produit un minimum global

---

Proc ACM(G)

---

- 1  $B = \emptyset$ ;
  - 2 **tant que**  $B$  n'est pas un arbre couvrant ( $n-1$  étapes) **faire**
  - 3     ajouter une arête minimale  
   qui ne compromet pas  $B$
- 



# Algorithme ACM

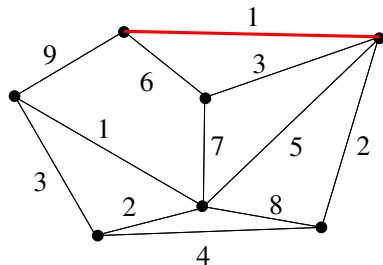
Calcul d'un ACM par stratégie gloutonne : la suite des minimums locaux (ajout d'une arête minimale à chaque étape) produit un minimum global

---

Proc ACM(G)

---

- 1  $B = \emptyset$ ;
  - 2 **tant que**  $B$  n'est pas un arbre couvrant ( $n-1$  étapes) **faire**
  - 3     ajouter une arête minimale  
   qui ne compromet pas  $B$
- 



# Algorithme ACM

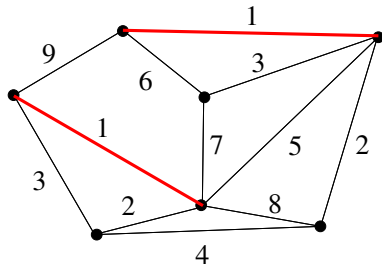
Calcul d'un ACM par stratégie gloutonne : la suite des minimums locaux (ajout d'une arête minimale à chaque étape) produit un minimum global

---

Proc ACM(G)

---

- 1  $B = \emptyset$ ;
  - 2 **tant que**  $B$  n'est pas un arbre couvrant ( $n-1$  étapes) **faire**
  - 3     ajouter une arête minimale  
   qui ne compromet pas  $B$
- 



# Algorithme ACM

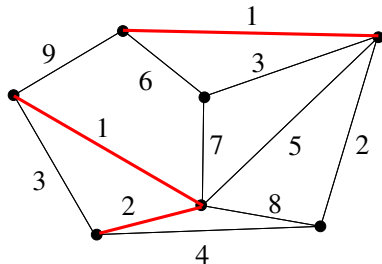
Calcul d'un ACM par stratégie gloutonne : la suite des minimums locaux (ajout d'une arête minimale à chaque étape) produit un minimum global

---

Proc ACM(G)

---

- 1  $B = \emptyset$ ;
  - 2 **tant que**  $B$  n'est pas un arbre couvrant ( $n-1$  étapes) **faire**
  - 3     ajouter une arête minimale  
   qui ne compromet pas  $B$
- 



# Algorithme ACM

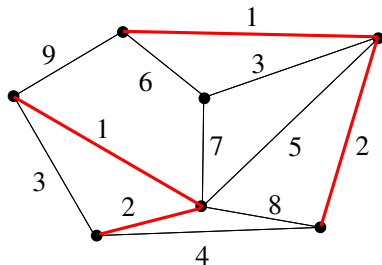
Calcul d'un ACM par stratégie gloutonne : la suite des minimums locaux (ajout d'une arête minimale à chaque étape) produit un minimum global

---

Proc ACM(G)

---

- 1  $B = \emptyset$ ;
  - 2 **tant que**  $B$  n'est pas un arbre couvrant ( $n-1$  étapes) **faire**
  - 3     ajouter une arête minimale  
   qui ne compromet pas  $B$
- 



# Algorithme ACM

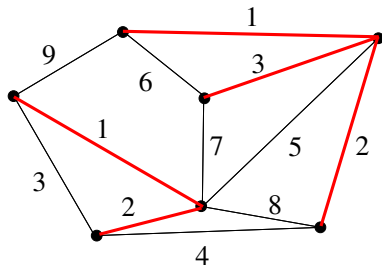
Calcul d'un ACM par stratégie gloutonne : la suite des minimums locaux (ajout d'une arête minimale à chaque étape) produit un minimum global

---

Proc ACM(G)

---

- 1  $B = \emptyset$ ;
  - 2 **tant que**  $B$  n'est pas un arbre couvrant ( $n-1$  étapes) **faire**
  - 3     ajouter une arête minimale  
   qui ne compromet pas  $B$
- 



# Algorithme ACM

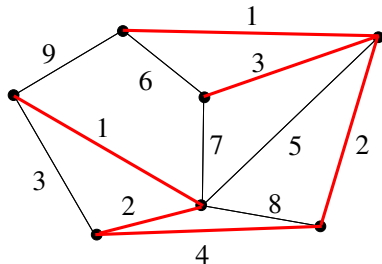
Calcul d'un ACM par stratégie gloutonne : la suite des minimums locaux (ajout d'une arête minimale à chaque étape) produit un minimum global

---

Proc ACM(G)

---

- 1  $B = \emptyset$ ;
  - 2 **tant que**  $B$  n'est pas un arbre couvrant ( $n-1$  étapes) **faire**
  - 3     ajouter une arête minimale  
   qui ne compromet pas  $B$
- 





# Principe

**Déf.** une coupe de  $G = (S, A)$  est une partition  $(R, S - R)$  de  $S$ . Une arête  $(u, v)$  traverse la coupe si l'une de ses extrémités est dans  $R$  et l'autre dans  $S - R$ . Une coupe respecte un ensemble d'arêtes  $B$  si aucune arête de  $B$  ne traverse la coupe.

**Proposition.** Soit  $G = (S, A, w)$ , et soit  $B$  un sous-ensemble de  $A$  inclus dans un  $ACM(G)$ ; soit  $(R, S - R)$  une coupe de  $G$  qui respecte  $B$  et soit  $(u, v)$  une arête minimale traversant  $(R, S - R)$ . Alors  $B \cup (u, v)$  est inclus dans un  $ACM(G)$ .

# Principe

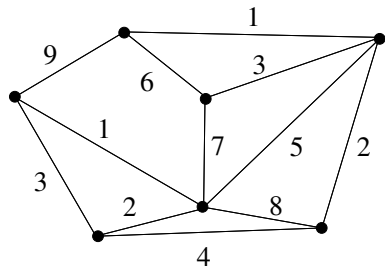
**Déf.** une coupe de  $G = (S, A)$  est une partition  $(R, S - R)$  de  $S$ . Une arête  $(u, v)$  traverse la coupe si l'une de ses extrémités est dans  $R$  et l'autre dans  $S - R$ . Une coupe respecte un ensemble d'arêtes  $B$  si aucune arête de  $B$  ne traverse la coupe.

**Proposition.** Soit  $G = (S, A, w)$ , et soit  $B$  un sous-ensemble de  $A$  inclus dans un  $ACM(G)$ ; soit  $(R, S - R)$  une coupe de  $G$  qui respecte  $B$  et soit  $(u, v)$  une arête minimale traversant  $(R, S - R)$ . Alors  $B \cup (u, v)$  est inclus dans un  $ACM(G)$ .

*Preuve.* Si  $B \cup (u, v)$  n'est pas inclus dans un  $ACM(G)$ , alors il existe un autre  $(x, y) \in ACM(G)$  tel que  $x \in R, y \in S - R$ . Alors  $Z = ACM(G) \cup (u, v) - (x, y)$  est un autre arbre couvrant et  $w(Z) \leq w(ACM(G))$ .

# Variantes

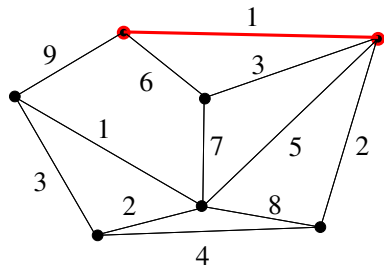
Algorithme de Kruskal :  $B$  est un ensemble de sous-arbre de  $ACM(G)$  ; la coupe s'appuie sur les sommets de l'un de ces sous-arbres.



Algorithme de Prim :  $B$  est sous-arbre racine de  $ACM(G)$  ; la coupe s'appuie sur les sommets de  $B$ .

# Variantes

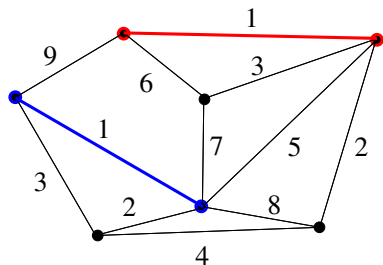
Algorithme de Kruskal :  $B$  est un ensemble de sous-arbre de  $ACM(G)$  ; la coupe s'appuie sur les sommets de l'un de ces sous-arbres.



Algorithme de Prim :  $B$  est sous-arbre racine de  $ACM(G)$  ; la coupe s'appuie sur les sommets de  $B$ .

# Variantes

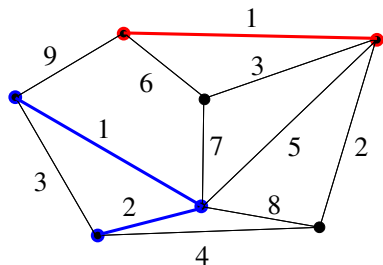
Algorithme de Kruskal :  $B$  est un ensemble de sous-arbre de  $ACM(G)$  ; la coupe s'appuie sur les sommets de l'un de ces sous-arbres.



Algorithme de Prim :  $B$  est sous-arbre racine de  $ACM(G)$  ; la coupe s'appuie sur les sommets de  $B$ .

# Variantes

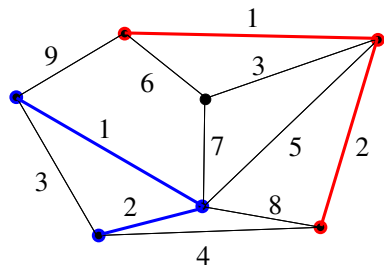
Algorithme de Kruskal :  $B$  est un ensemble de sous-arbre de  $ACM(G)$  ; la coupe s'appuie sur les sommets de l'un de ces sous-arbres.



Algorithme de Prim :  $B$  est sous-arbre racine de  $ACM(G)$  ; la coupe s'appuie sur les sommets de  $B$ .

# Variantes

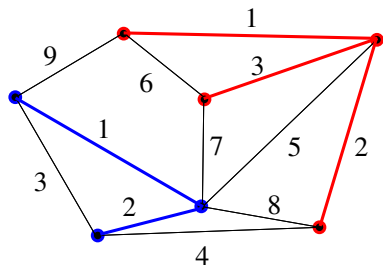
Algorithme de Kruskal :  $B$  est un ensemble de sous-arbre de  $ACM(G)$  ; la coupe s'appuie sur les sommets de l'un de ces sous-arbres.



Algorithme de Prim :  $B$  est sous-arbre racine de  $ACM(G)$  ; la coupe s'appuie sur les sommets de  $B$ .

# Variantes

Algorithme de Kruskal :  $B$  est un ensemble de sous-arbre de  $ACM(G)$  ; la coupe s'appuie sur les sommets de l'un de ces sous-arbres.

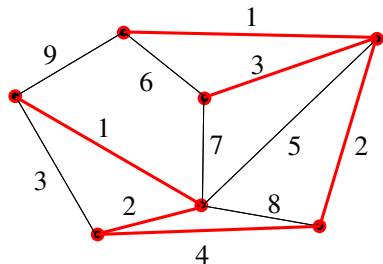


Algorithme de Prim :  $B$  est sous-arbre racine de  $ACM(G)$  ; la coupe s'appuie sur les sommets de  $B$ .



# Variantes

Algorithme de Kruskal :  $B$  est un ensemble de sous-arbre de  $ACM(G)$  ; la coupe s'appuie sur les sommets de l'un de ces sous-arbres.

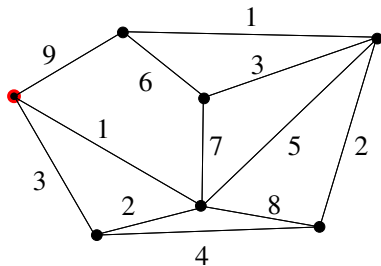


Algorithme de Prim :  $B$  est sous-arbre racine de  $ACM(G)$  ; la coupe s'appuie sur les sommets de  $B$ .

# Variantes

Algorithme de Kruskal :  $B$  est un ensemble de sous-arbre de  $ACM(G)$  ; la coupe s'appuie sur les sommets de l'un de ces sous-arbres.

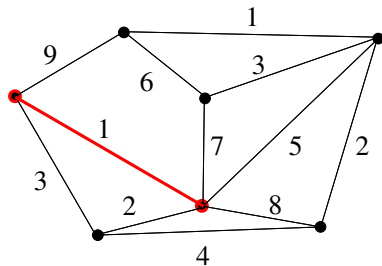
Algorithme de Prim :  $B$  est sous-arbre racine de  $ACM(G)$  ; la coupe s'appuie sur les sommets de  $B$ .



# Variantes

Algorithme de Kruskal :  $B$  est un ensemble de sous-arbre de  $ACM(G)$  ; la coupe s'appuie sur les sommets de l'un de ces sous-arbres.

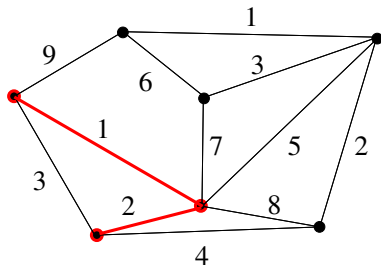
Algorithme de Prim :  $B$  est sous-arbre racine de  $ACM(G)$  ; la coupe s'appuie sur les sommets de  $B$ .



# Variantes

Algorithme de Kruskal :  $B$  est un ensemble de sous-arbre de  $ACM(G)$  ; la coupe s'appuie sur les sommets de l'un de ces sous-arbres.

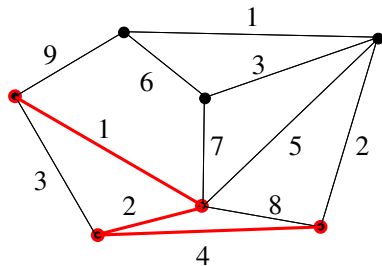
Algorithme de Prim :  $B$  est sous-arbre racine de  $ACM(G)$  ; la coupe s'appuie sur les sommets de  $B$ .



# Variantes

Algorithme de Kruskal :  $B$  est un ensemble de sous-arbre de  $ACM(G)$  ; la coupe s'appuie sur les sommets de l'un de ces sous-arbres.

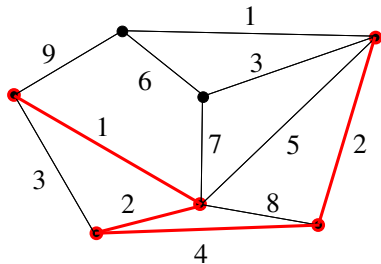
Algorithme de Prim :  $B$  est sous-arbre racine de  $ACM(G)$  ; la coupe s'appuie sur les sommets de  $B$ .



# Variantes

Algorithme de Kruskal :  $B$  est un ensemble de sous-arbre de  $ACM(G)$  ; la coupe s'appuie sur les sommets de l'un de ces sous-arbres.

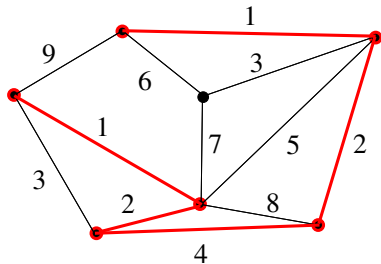
Algorithme de Prim :  $B$  est sous-arbre racine de  $ACM(G)$  ; la coupe s'appuie sur les sommets de  $B$ .



# Variantes

Algorithme de Kruskal :  $B$  est un ensemble de sous-arbre de  $ACM(G)$  ; la coupe s'appuie sur les sommets de l'un de ces sous-arbres.

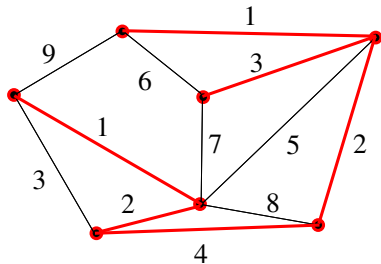
Algorithme de Prim :  $B$  est sous-arbre racine de  $ACM(G)$  ; la coupe s'appuie sur les sommets de  $B$ .



# Variantes

Algorithme de Kruskal :  $B$  est un ensemble de sous-arbre de  $ACM(G)$  ; la coupe s'appuie sur les sommets de l'un de ces sous-arbres.

Algorithme de Prim :  $B$  est sous-arbre racine de  $ACM(G)$  ; la coupe s'appuie sur les sommets de  $B$ .





# Codage de Huffman

**Problème** : Compresser un fichier texte T (archivage, transfert).

Idée : 2 lectures du texte T

1. 1ère passe : calculer la fréquence de chaque caractère dans T
2. 2ème passe : encoder chaque caractère en binaire ; les caractères les plus fréquents ont les codes les plus courts.

# Exemple

Fichier de 100000 caractères (6 caractères distincts)

Caractère	a	b	c	d	e	f
Fréquence	45%	13%	12%	16%	9%	5%
Code fixe	000	001	010	011	100	101
Code variable	0	101	100	111	1101	1100

Taille du fichier compressé :

1. Code fixe (sur 3 bits) :  $3 * 100000 = 300000$  bits
2. Code variable :  $1 * 45000 + 3 * 13000 + 3 * 12000 + 3 * 16000 + 4 * 9000 + 4 * 5000 = 224000$  bits

# Exemple

Fichier de 100000 caractères (6 caractères distincts)

Caractère	a	b	c	d	e	f
Fréquence	45%	13%	12%	16%	9%	5%
Code fixe	000	001	010	011	100	101
Code variable	0	101	100	111	1101	1100

Taille du fichier compressé :

1. Code fixe (sur 3 bits) :  $3 * 100000 = 300000$  bits
2. Code variable :  $1 * 45000 + 3 * 13000 + 3 * 12000 + 3 * 16000 + 4 * 9000 + 4 * 5000 = 224000$  bits

**Code préfixe** : l'encodage d'un caractère n'est le préfixe d'aucun autre  
→ simple concaténation à l'encodage et pas d'ambiguïté au décodage.

# Exemple

Fichier de 100000 caractères (6 caractères distincts)

Caractère	a	b	c	d	e	f
Fréquence	45%	13%	12%	16%	9%	5%
Code fixe	000	001	010	011	100	101
Code variable	0	101	100	111	1101	1100

Taille du fichier compressé :

1. Code fixe (sur 3 bits) :  $3 * 100000 = 300000$  bits
2. Code variable :  $1 * 45000 + 3 * 13000 + 3 * 12000 + 3 * 16000 + 4 * 9000 + 4 * 5000 = 224000$  bits

**Code préfixe** : l'encodage d'un caractère n'est le préfixe d'aucun autre  
→ simple concaténation à l'encodage et pas d'ambiguïté au décodage.

Ex. 001011101.

Code fixe : 001 011 101 → *bdf*.

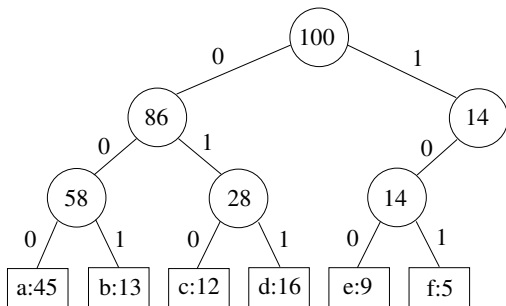
Code variable : 0 0 101 1101 → *aabe*.

# Code préfixe et Arbre digital

Un code préfixe peut toujours être représenté à l'aide d'un arbre binaire (arbre digital).

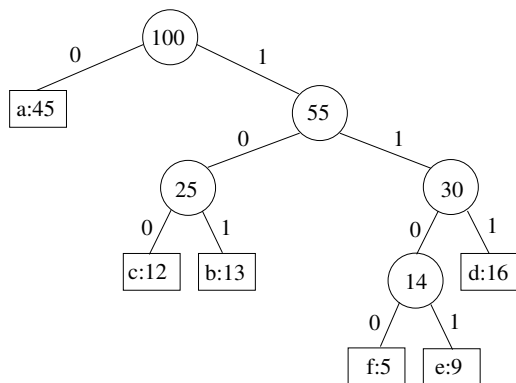
Si l'arbre n'est pas complet alors le code n'est pas optimal.

Caractère	a	b	c	d	e	f
Fréquence	45%	13%	12%	16%	9%	5%
Code fixe	000	001	010	011	100	101



# Code préfixe et Arbre digital

Caractère	a	b	c	d	e	f
Fréquence	45%	13%	12%	16%	9%	5%
Code variable	0	101	100	111	1101	1100



# Taille du fichier compressé

Texte  $T$  sur l'alphabet  $A$

- ▶  $f(c)$  dénote la fréquence du caractère  $c \in A$  dans  $T$
- ▶  $prof(c)$  dénote la profondeur du caractère  $c$  dans l'arbre digital (= taille du codage de  $c$ )

Le nombre de bits requis pour encoder le texte  $T$  (i.e. la taille du fichier compressé) est

$$B(T) = \sum_{c \in A} f(c)prof(c).$$

# Construction de l'arbre de Huffman

**Entrées :** Fréquences  $(f_1, \dots, f_n)$  des lettres  $(a_i)$  d'un texte  $T$ .

**Sortie :** Arbre digital donnant un code préfixe minimal pour  $T$ .

1. Créer, pour chaque lettre  $a_i$ , un arbre (réduit à une feuille) qui porte comme poids la fréquence  $f(i)$ .
2. Itérer le processus suivant :
  - ▶ choisir 2 arbres  $G$  et  $D$  de poids minimum
  - ▶ créer un nouvel arbre  $R$ , ayant pour sous-arbre gauche (resp. droit)  $G$  (resp.  $D$ ) et lui affecter comme poids la somme des poids de  $G$  et  $D$ .
3. Arrêter lorsqu'il ne reste plus qu'un seul arbre :  
c'est l'arbre de Huffman



f:5

e:9

c:12

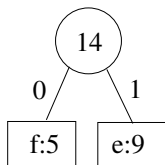
b:13

d:16

a:45

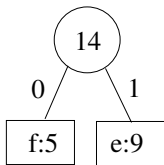
c:12

b:13

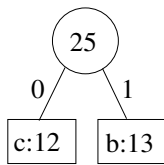


d:16

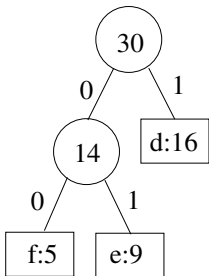
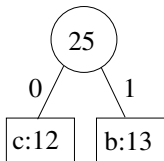
a:45



d:16

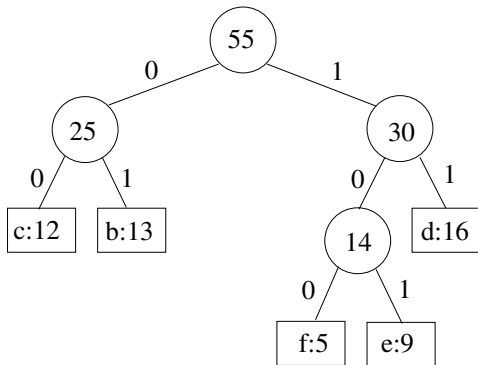


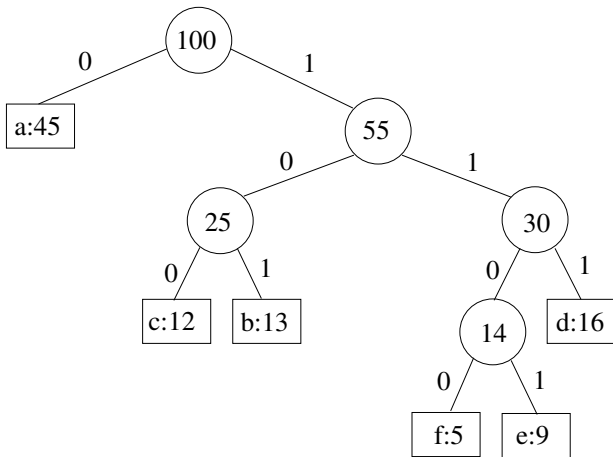
a:45



a:45

a:45





# Complexité de la construction

1. La construction initiale et le tri :  $O(n \log n)$
2. A chaque itération :
  - ▶ extraire 2 fois l'arbre de poids min et fabriquer un nouvel arbre à partir des 2 précédents :  $O(1)$
  - ▶ rajouter ce nouvel arbre à la liste triée :  $O(\log n)$
3. Il y a  $n - 1$  itérations.

La construction de l'arbre de Huffman est donc en  $O(n \log n)$  comparaisons.

# Optimalité

**Optimalité** : Le code préfixe complet de l'arbre de Huffman est minimal  
i.e. aucun arbre digital ne compresse mieux le texte  $T$  que  
 $Huffman(A, (f_i))$ .



# Optimalité

**Optimalité** : Le code préfixe complet de l'arbre de Huffman est minimal i.e. aucun arbre digital ne compresse mieux le texte  $T$  que  $Huffman(A, (f_i))$ .

- ▶ **Propriété de choix glouton**

Soient  $x$  le caractère ayant la moins grande fréquence, et  $y$  le caractère ayant la seconde moins grande fréquence. Il existe un codage préfixe optimal tq.  $x$  et  $y$  ont le même père.

# Optimalité

**Optimalité** : Le code préfixe complet de l'arbre de Huffman est minimal i.e. aucun arbre digital ne compresse mieux le texte  $T$  que  $Huffman(A, (f_i))$ .

- ▶ **Propriété de choix glouton**

Soient  $x$  le caractère ayant la moins grande fréquence, et  $y$  le caractère ayant la seconde moins grande fréquence. Il existe un codage préfixe optimal tq.  $x$  et  $y$  ont le même père.

- ▶ **Propriété de sous-structure optimale**

On considère l'alphabet  $A' = A - \{x, y\} + z$ , où  $z$  est une nouvelle lettre ayant une fréquence  $f(z) = f(x) + f(y)$ .

Soit  $T'$  l'arbre d'un codage optimal pour  $A'$ , alors l'arbre  $T$  obtenu à partir de  $T'$  en remplaçant la feuille associée à  $z$  par un noeud interne ayant  $x$  et  $y$  comme feuilles représente un codage optimal pour  $A$ .

# Matroïdes

**Définition.**  $(E, \mathcal{I})$  est un matroïde si  $E$  est un ensemble de  $n$  éléments,  $\mathcal{I}$  est une famille de parties de  $E$ ,  $\mathcal{I} \subset P(E)$  vérifiant :

- ▶ l'hérédité :  $X \in \mathcal{I} \implies \forall Y \subset X, Y \in \mathcal{I}$ .
- ▶ l'échange :  $(A, B \in \mathcal{I}, |A| < |B|) \implies \exists x \in B - A$  tel que  $A \cup \{x\} \in \mathcal{I}$ .

Si  $X \in \mathcal{I}$ , on dit que  $X$  est un indépendant.

# Matroïdes

**Définition.**  $(E, \mathcal{I})$  est un matroïde si  $E$  est un ensemble de  $n$  éléments,  $\mathcal{I}$  est une famille de parties de  $E$ ,  $\mathcal{I} \subset P(E)$  vérifiant :

- ▶ l'hérédité :  $X \in \mathcal{I} \implies \forall Y \subset X, Y \in \mathcal{I}$ .
- ▶ l'échange :  $(A, B \in \mathcal{I}, |A| < |B|) \implies \exists x \in B - A$  tel que  $A \cup \{x\} \in \mathcal{I}$ .

Si  $X \in \mathcal{I}$ , on dit que  $X$  est un indépendant.

Exemples :

- ▶ **Les matroïdes vectoriels.**  $E = \{v_1, \dots, v_n\}$  un ensemble de vecteurs d'un espace vectoriel et  $\mathcal{I}$  la famille de tous les sous-ensembles de vecteurs de  $E$  linéairement indépendants. Alors  $M = (E, \mathcal{I})$  est un matroïde.

# Matroïdes

**Définition.**  $(E, \mathcal{I})$  est un matroïde si  $E$  est un ensemble de  $n$  éléments,  $\mathcal{I}$  est une famille de parties de  $E$ ,  $\mathcal{I} \subset P(E)$  vérifiant :

- ▶ l'hérédité :  $X \in \mathcal{I} \implies \forall Y \subset X, Y \in \mathcal{I}$ .
- ▶ l'échange :  $(A, B \in \mathcal{I}, |A| < |B|) \implies \exists x \in B - A$  tel que  $A \cup \{x\} \in \mathcal{I}$ .

Si  $X \in \mathcal{I}$ , on dit que  $X$  est un indépendant.

Exemples :

- ▶ **Les matroïdes vectoriels.**  $E = \{v_1, \dots, v_n\}$  un ensemble de vecteurs d'un espace vectoriel et  $\mathcal{I}$  la famille de tous les sous-ensembles de vecteurs de  $E$  linéairement indépendants. Alors  $M = (E, \mathcal{I})$  est un matroïde.
- ▶ **Les matroïdes graphiques** (les forêts d'un graphe). Soit  $G = (S, E)$  un graphe non orienté et  $\mathcal{I}$  la famille des forêts de  $G$  :  $F \in \mathcal{I}$  si et seulement si  $F$  est acyclique. Alors  $M = (E, \mathcal{I})$  est un matroïde.

# Propriétés des matroïdes

Étant donné un matroïde  $M = (E, \mathcal{I})$ , on dit qu'un élément  $x \notin F$  est une **extension** de  $F \in \mathcal{I}$  si  $F \cup \{x\} \in \mathcal{I}$ .

Si  $F$  est un sous-ensemble indépendant d'un matroïde  $M$ , on dit que  $F$  est **maximal** s'il ne possède aucune extension (i.e. il est maximal au sens de l'inclusion).

# Propriétés des matroïdes

Étant donné un matroïde  $M = (E, \mathcal{I})$ , on dit qu'un élément  $x \notin F$  est une **extension** de  $F \in \mathcal{I}$  si  $F \cup \{x\} \in \mathcal{I}$ .

Si  $F$  est un sous-ensemble indépendant d'un matroïde  $M$ , on dit que  $F$  est **maximal** s'il ne possède aucune extension (i.e. il est maximal au sens de l'inclusion).

**Thm.** Tous les sous-ensembles indépendants maximaux d'un matroïde ont le même cardinal.

# Propriétés des matroïdes

Étant donné un matroïde  $M = (E, \mathcal{I})$ , on dit qu'un élément  $x \notin F$  est une **extension** de  $F \in \mathcal{I}$  si  $F \cup \{x\} \in \mathcal{I}$ .

Si  $F$  est un sous-ensemble indépendant d'un matroïde  $M$ , on dit que  $F$  est **maximal** s'il ne possède aucune extension (i.e. il est maximal au sens de l'inclusion).

**Thm.** Tous les sous-ensembles indépendants maximaux d'un matroïde ont le même cardinal.

*Preuve.* Supposons au contraire que  $F$  soit un sous-ensemble indépendant maximal de  $M$  et qu'il en existe un autre  $H$ , plus grand. Alors, la propriété d'échange implique que  $F$  peut être étendu à un ensemble indépendant  $F \cup \{x\}$  pour un certain  $x \in H \setminus F$ , ce qui contredit l'hypothèse que  $F$  est maximal.



# Matroïdes pondérés

On dit qu'un matroïde  $M = (E, \mathcal{I})$  est **pondéré** si l'on dispose d'une fonction de pondération  $w$  de  $E$  dans  $\mathbb{R}^+$  qui affecte un poids strictement positif  $w(x)$  à chaque élément  $x \in E$ .

Pour  $F \subset E$  :  $w(F) = \sum_{x \in F} w(x)$ .

**Question** : Trouver un indépendant de poids maximal (optimal).

**Algorithme glouton**

---

**GLOUTON**

---

- 1 Trier les éléments de  $E$  par poids décroissant :  
 $w(e_1) \geq w(e_2) \geq \dots \geq w(e_n)$
  - 2  $A = \emptyset$ ;
  - 3 **pour**  $i$  de 1 à  $|E|$  **faire**
  - 4     **si**  $A \cup e_i \in \mathcal{I}$  **alors**
  - 5          $A = A \cup \{e_i\}$ ;
  - 6 **return**  $A$ ;
-

# Validité de l'algorithme

Les matroïdes vérifient la propriété du choix glouton

**Lemme** Supposons  $M = (E, \mathcal{I}, w)$  (avec  $\mathcal{I} \neq \{\emptyset\}$ ) un matroïde pondéré; et  $E$  trié par ordre de poids décroissant. Soit  $x$  le premier élément de  $E$  tel que  $\{x\}$  soit indépendant, alors il existe un sous-ensemble optimal  $F$  de  $E$  qui contient  $\{x\}$ .

# Validité de l'algorithme

Les matroïdes vérifient la propriété du choix glouton

**Lemme** Supposons  $M = (E, \mathcal{I}, w)$  (avec  $\mathcal{I} \neq \{\emptyset\}$ ) un matroïde pondéré; et  $E$  trié par ordre de poids décroissant. Soit  $x$  le premier élément de  $E$  tel que  $\{x\}$  soit indépendant, alors il existe un sous-ensemble optimal  $F$  de  $E$  qui contient  $\{x\}$ .

*Preuve.*

- ▶ Soit  $H$  un indépendant de poids optimal. Supposons que  $x \notin H$ . Aucun élément de  $H$  n'a un poids supérieur à  $w(x)$ .
  
- ▶ Construire à partir de  $H$  un ensemble  $F$  indépendant de poids optimal et qui contient  $x$ .

# Validité de l'algorithme

Les matroïdes vérifient la propriété du choix glouton

**Lemme** Supposons  $M = (E, \mathcal{I}, w)$  (avec  $\mathcal{I} \neq \{\emptyset\}$ ) un matroïde pondéré ; et  $E$  trié par ordre de poids décroissant. Soit  $x$  le premier élément de  $E$  tel que  $\{x\}$  soit indépendant, alors il existe un sous-ensemble optimal  $F$  de  $E$  qui contient  $\{x\}$ .

*Preuve.*

- ▶ Soit  $H$  un indépendant de poids optimal. Supposons que  $x \notin H$ .  
Aucun élément de  $H$  n'a un poids supérieur à  $w(x)$ .  
Preuve :  $y \in H$  implique que  $\{y\}$  est indépendant, puisque  $H \in \mathcal{I}$  et  $\mathcal{I}$  est héréditaire. Donc  $w(x) \geq w(y)$  pour tout  $y \in H$ .
- ▶ Construire à partir de  $H$  un ensemble  $F$  indépendant de poids optimal et qui contient  $x$ .

# Validité de l'algorithme

Les matroïdes vérifient la propriété du choix glouton

**Lemme** Supposons  $M = (E, \mathcal{I}, w)$  (avec  $\mathcal{I} \neq \{\emptyset\}$ ) un matroïde pondéré ; et  $E$  trié par ordre de poids décroissant. Soit  $x$  le premier élément de  $E$  tel que  $\{x\}$  soit indépendant, alors il existe un sous-ensemble optimal  $F$  de  $E$  qui contient  $\{x\}$ .

*Preuve.*

- ▶ Soit  $H$  un indépendant de poids optimal. Supposons que  $x \notin H$ .  
Aucun élément de  $H$  n'a un poids supérieur à  $w(x)$ .  
Preuve :  $y \in H$  implique que  $\{y\}$  est indépendant, puisque  $H \in \mathcal{I}$  et  $\mathcal{I}$  est héréditaire. Donc  $w(x) \geq w(y)$  pour tout  $y \in H$ .
- ▶ Construire à partir de  $H$  un ensemble  $F$  indépendant de poids optimal et qui contient  $x$ .  
Commencer avec  $F = \{x\}$ . En se servant de la propriété d'échange, on trouve itérativement un nouvel élément de  $H$  pouvant être ajouté à  $F$  jusqu'à ce que  $|F| = |H|$ , en préservant l'indépendance de  $F$ .

# Validité de l'algorithme

Les matroïdes vérifient la propriété du choix glouton

**Lemme** Supposons  $M = (E, \mathcal{I}, w)$  (avec  $\mathcal{I} \neq \{\emptyset\}$ ) un matroïde pondéré; et  $E$  trié par ordre de poids décroissant. Soit  $x$  le premier élément de  $E$  tel que  $\{x\}$  soit indépendant, alors il existe un sous-ensemble optimal  $F$  de  $E$  qui contient  $\{x\}$ .

*Preuve.*

- ▶ Soit  $H$  un indépendant de poids optimal. Supposons que  $x \notin H$ .  
Aucun élément de  $H$  n'a un poids supérieur à  $w(x)$ .  
Preuve :  $y \in H$  implique que  $\{y\}$  est indépendant, puisque  $H \in \mathcal{I}$  et  $\mathcal{I}$  est héréditaire. Donc  $w(x) \geq w(y)$  pour tout  $y \in H$ .
- ▶ Construire à partir de  $H$  un ensemble  $F$  indépendant de poids optimal et qui contient  $x$ .  
Commencer avec  $F = \{x\}$ . En se servant de la propriété d'échange, on trouve itérativement un nouvel élément de  $H$  pouvant être ajouté à  $F$  jusqu'à ce que  $|F| = |H|$ , en préservant l'indépendance de  $F$ .  
Alors,  $F = H \setminus \{y\} \cup x$  pour un certain  $y \in H$ , et donc  $w(F) = w(H) - w(y) + w(x) \geq w(H)$ .  $F$  est optimal et  $x \in F$ .

# Les matroïdes satisfont la propriété de sous-structure optimale

**Lemme.** Soit  $x$  le premier élément de  $E$  choisi par GLOUTON (c'est-à-dire le singleton indépendant de poids maximal) pour le matroïde pondéré  $M = (E, \mathcal{I}, w)$ . Trouver un sous-ensemble indépendant de poids maximal contenant  $x$ , revient à trouver un sous-ensemble indépendant de poids maximal du matroïde pondéré  $M' = (E', \mathcal{I}', w')$ , où  $E' = \{y \in E \setminus \{x\} : \{x, y\} \in \mathcal{I}\}$ ,  $\mathcal{I}' = \{H \subset E \setminus \{x\} : H \cup \{x\} \in \mathcal{I}\}$ , et la fonction de pondération de  $M'$  est celle de  $M$ , restreinte à  $E'$ . (On appelle  $M'$  la *contraction* de  $M$  par l'élément  $x$ ).

# Les matroïdes satisfont la propriété de sous-structure optimale

**Lemme.** Soit  $x$  le premier élément de  $E$  choisi par GROUTON (c'est-à-dire le singleton indépendant de poids maximal) pour le matroïde pondéré  $M = (E, \mathcal{I}, w)$ . Trouver un sous-ensemble indépendant de poids maximal contenant  $x$ , revient à trouver un sous-ensemble indépendant de poids maximal du matroïde pondéré  $M' = (E', \mathcal{I}', w')$ , où  $E' = \{y \in E \setminus \{x\} : \{x, y\} \in \mathcal{I}\}$ ,  $\mathcal{I}' = \{H \subset E \setminus \{x\} : H \cup \{x\} \in \mathcal{I}\}$ , et la fonction de pondération de  $M'$  est celle de  $M$ , restreinte à  $E'$ . (On appelle  $M'$  la *contraction* de  $M$  par l'élément  $x$ ).

*Preuve.* Si  $F$  est un sous-ensemble indépendant de poids maximal de  $M$  contenant  $x$ , alors  $F' = F \setminus \{x\}$  est un sous-ensemble indépendant de  $M'$ . Inversement, un sous-ensemble indépendant  $F'$  de  $M'$  engendre un sous-ensemble indépendant  $F = F' \cup \{x\}$  de  $M$ . Comme nous avons dans les deux cas  $w(F) = w(F') + w(x)$ , une solution de poids maximal pour  $M$  contenant  $x$  donne une solution de poids maximal sur  $M'$ , et réciproquement.



# Optimalité

**Théorème.** L'algorithme GROUTON donne une solution optimale.

*Preuve.* Soit  $e_k$  le premier élément indépendant de  $E$ , i.e. le premier indice  $i$  de l'algorithme tel que  $e_i \in \mathcal{I}$ .

- ▶ Il existe une solution optimale qui contient  $e_k$  (choix glouton).
- ▶ Puis, par récurrence on obtient que GROUTON donne une solution optimale (sous-structure optimale) : on se restreint à une solution contenant  $e_k$ , et on recommence avec  $E' = E - e_k$  et  $\mathcal{I}' = \{X \in E'; X \cup \{e_k\} \in \mathcal{I}\}$ .

# Optimalité

**Théorème.** L'algorithme GROUTON donne une solution optimale.

*Preuve.* Soit  $e_k$  le premier élément indépendant de  $E$ , i.e. le premier indice  $i$  de l'algorithme tel que  $e_i \in \mathcal{I}$ .

- ▶ Il existe une solution optimale qui contient  $e_k$  (choix glouton).
- ▶ Puis, par récurrence on obtient que GROUTON donne une solution optimale (sous-structure optimale) : on se restreint à une solution contenant  $e_k$ , et on recommence avec  $E' = E - e_k$  et  $\mathcal{I}' = \{X \in E'; X \cup \{e_k\} \in \mathcal{I}\}$ .
- ▶ Il suffit de regarder  $E' = \{e_{k+1}, \dots, e_n\}$  car les éléments  $e_j, j < k$  ne peuvent pas être extension d'un indépendant (sinon, par hérédité, aussi indépendants et choisis au lieu de  $e_k$ ).

# Optimalité

**Théorème.** L'algorithme GROUTON donne une solution optimale.

*Preuve.* Soit  $e_k$  le premier élément indépendant de  $E$ , i.e. le premier indice  $i$  de l'algorithme tel que  $e_i \in \mathcal{I}$ .

- ▶ Il existe une solution optimale qui contient  $e_k$  (choix glouton).
- ▶ Puis, par récurrence on obtient que GROUTON donne une solution optimale (sous-structure optimale) : on se restreint à une solution contenant  $e_k$ , et on recommence avec  $E' = E - e_k$  et  $\mathcal{I}' = \{X \in E'; X \cup \{e_k\} \in \mathcal{I}\}$ .
- ▶ Il suffit de regarder  $E' = \{e_{k+1}, \dots, e_n\}$  car les éléments  $e_j$ ,  $j < k$  ne peuvent pas être extension d'un indépendant (sinon, par hérédité, aussi indépendants et choisis au lieu de  $e_k$ ).

Exemple des forêts d'un graphe : algorithme glouton de Kruskal pour construire un arbre de poids minimal. Trier toutes les arêtes par poids croissant, et sélectionner au fur et à mesure celles qui ne créent pas de cycle si les on a joute à l'ensemble courant.

# Complexité

---

## GLOUTON

---

- 1 Trier les éléments de  $E$  par poids décroissant :  
 $w(e_1) \geq w(e_2) \geq \dots \geq w(e_n)$
  - 2  $A = \emptyset$ ;
  - 3 **pour**  $i$  de 1 à  $|E|$  **faire**
  - 4     **si**  $A \cup e_i \in I$  **alors**
  - 5          $A = A \cup \{e_i\}$ ;
  - 6 **return**  $A$ ;
-

# Complexité

---

## GLOUTON

---

- 1 Trier les éléments de  $E$  par poids décroissant :  
 $w(e_1) \geq w(e_2) \geq \dots \geq w(e_n)$
  - 2  $A = \emptyset$ ;
  - 3 **pour**  $i$  de 1 à  $|E|$  **faire**
  - 4     **si**  $A \cup e_i \in I$  **alors**
  - 5          $A = A \cup \{e_i\}$ ;
  - 6 **return**  $A$ ;
- 

Soit  $n = |E|$ , la phase de tri de GLOUTON prend un temps  $O(n \log n)$ .

Chaque exécution de la ligne 4 impose de vérifier si l'ensemble  $F \cup \{x\}$  est ou non indépendant.

Si cette vérification prend un temps  $f(n)$  et les comparaisons se font en temps constants, l'algorithme GLOUTON prend un temps

$$O(n \log n) + nf(n).$$

# Ordonnement sur une machine

## Problème :

- ▶  $n$  tâches  $T_1 \dots T_n$ , toutes de durée 1, deadlines associées :  $d_1, \dots, d_n$  avec pénalités  $w_1, \dots, w_n$  si deadline dépassé.
- ▶ Un ordonnancement est une permutation des tâches : première tâche commence au temps 0 et termine au temps 1, deuxième tâche commence au temps 1 et termine au temps 2 ...
- ▶ Si une tâche  $T_i$  commence après sa deadline  $d_i$  alors on a la pénalité  $w_i$ .
- ▶ But : trouver un ordonnancement qui minimise la pénalité totale des tâches en retard.

# Ordonnements canoniques

## Définitions.

- ▶ tâche à l'heure : tâche finie avant la dead line
- ▶ tâche en retard : tâche finie après la dead line

## Ordre canonique

on peut se restreindre :

- ▶ aux ordonnancements où les tâches à l'heure précèdent les tâches en retard ;
- ▶ aux ordonnancement où les tâches à l'heure sont classées par ordre de deadline croissant.

Remarque : minimiser la pénalité totale des tâches en retard = maximiser la pénalité des tâches à l'heure.

# Indépendants

**Définition.** Un ensemble de tâches est dit indépendant ssi les tâches peuvent être exécutées en étant toutes à l'heure.

**Lemme.**  $A$  ensemble de tâches,  $N_t(A)$  = nombre de tâches de deadline  $\leq t$ .

Les propositions suivantes sont équivalentes :

1.  $A$  est indépendant
2. Pour tout  $t = 1, 2, \dots, n$ ,  $N_t(A) \leq t$ .
3. Ordre canonique  $\rightarrow$  pas de tâches en retard.



# Exemple

Exemple : 7 tâches

tâche	1	2	3	4	5	6	7
deadline	4	2	4	3	1	4	6
pénalité	7	6	5	4	3	2	1

Trier les tâches par  $w_i$  décroissants ; puis appliquer l'algorithme glouton en prenant la tâche si elle tient sans pénalité (réorganiser selon l'ordre canonique à chaque nouvel ajout).

# Exemple

Exemple : 7 tâches

tâche	1	2	3	4	5	6	7
deadline	4	2	4	3	1	4	6
pénalité	7	6	5	4	3	2	1

Trier les tâches par  $w_i$  décroissants ; puis appliquer l'algorithme glouton en prenant la tâche si elle tient sans pénalité (réorganiser selon l'ordre canonique à chaque nouvel ajout).

Sur l'exemple :

$T_1$

# Exemple

Exemple : 7 tâches

tâche	1	2	3	4	5	6	7
deadline	4	2	4	3	1	4	6
pénalité	7	6	5	4	3	2	1

Trier les tâches par  $w_i$  décroissants ; puis appliquer l'algorithme glouton en prenant la tâche si elle tient sans pénalité (réorganiser selon l'ordre canonique à chaque nouvel ajout).

Sur l'exemple :

$T_1$

$T_2 \rightarrow \{T_2, T_1\}$

# Exemple

Exemple : 7 tâches

tâche	1	2	3	4	5	6	7
deadline	4	2	4	3	1	4	6
pénalité	7	6	5	4	3	2	1

Trier les tâches par  $w_i$  décroissants ; puis appliquer l'algorithme glouton en prenant la tâche si elle tient sans pénalité (réorganiser selon l'ordre canonique à chaque nouvel ajout).

Sur l'exemple :

$T_1$

$T_2 \rightarrow \{T_2, T_1\}$

$T_3 \rightarrow \{T_2, T_1, T_3\}$

# Exemple

Exemple : 7 tâches

tâche	1	2	3	4	5	6	7
deadline	4	2	4	3	1	4	6
pénalité	7	6	5	4	3	2	1

Trier les tâches par  $w_i$  décroissants ; puis appliquer l'algorithme glouton en prenant la tâche si elle tient sans pénalité (réorganiser selon l'ordre canonique à chaque nouvel ajout).

Sur l'exemple :

$T_1$

$T_2 \rightarrow \{T_2, T_1\}$

$T_3 \rightarrow \{T_2, T_1, T_3\}$

$T_4 \rightarrow \{T_2, T_4, T_1, T_3\}$

# Exemple

Exemple : 7 tâches

tâche	1	2	3	4	5	6	7
deadline	4	2	4	3	1	4	6
pénalité	7	6	5	4	3	2	1

Trier les tâches par  $w_i$  décroissants ; puis appliquer l'algorithme glouton en prenant la tâche si elle tient sans pénalité (réorganiser selon l'ordre canonique à chaque nouvel ajout).

Sur l'exemple :

$T_1$

$T_2 \rightarrow \{T_2, T_1\}$

$T_3 \rightarrow \{T_2, T_1, T_3\}$

$T_4 \rightarrow \{T_2, T_4, T_1, T_3\}$

$T_5$  et  $T_6$  impossibles

# Exemple

Exemple : 7 tâches

tâche	1	2	3	4	5	6	7
deadline	4	2	4	3	1	4	6
pénalité	7	6	5	4	3	2	1

Trier les tâches par  $w_i$  décroissants ; puis appliquer l'algorithme glouton en prenant la tâche si elle tient sans pénalité (réorganiser selon l'ordre canonique à chaque nouvel ajout).

Sur l'exemple :

$T_1$

$T_2 \rightarrow \{T_2, T_1\}$

$T_3 \rightarrow \{T_2, T_1, T_3\}$

$T_4 \rightarrow \{T_2, T_4, T_1, T_3\}$

$T_5$  et  $T_6$  impossibles

$T_7 \rightarrow \{T_2, T_4, T_1, T_3, T_7\}$

# Exemple

Exemple : 7 tâches

tâche	1	2	3	4	5	6	7
deadline	4	2	4	3	1	4	6
pénalité	7	6	5	4	3	2	1

Trier les tâches par  $w_i$  décroissants ; puis appliquer l'algorithme glouton en prenant la tâche si elle tient sans pénalité (réorganiser selon l'ordre canonique à chaque nouvel ajout).

Sur l'exemple :

$T_1$

$T_2 \rightarrow \{T_2, T_1\}$

$T_3 \rightarrow \{T_2, T_1, T_3\}$

$T_4 \rightarrow \{T_2, T_4, T_1, T_3\}$

$T_5$  et  $T_6$  impossibles

$T_7 \rightarrow \{T_2, T_4, T_1, T_3, T_7\}$

La solution optimale est  $\{T_2, T_4, T_1, T_3, T_7\}$  et la pénalité minimale est  $w_5 + w_6 = 5$ .



# Résolution par matroïde

**Proposition.** Soit  $E$  ensemble de tâches, et  $\mathcal{I}$  famille des sous-ensembles de tâches indépendantes, alors  $(E, \mathcal{I})$  est un matroïde.

- ▶ Hérité : trivial.
- ▶ Echange : Soit  $A, B$  indépendants et  $|A| < |B|$ . Existe-t-il  $x$  appartenant à  $B$  tel que  $A \cup \{x\}$  soit indépendant ?

# Résolution par matroïde

**Proposition.** Soit  $E$  ensemble de tâches, et  $\mathcal{I}$  famille des sous-ensembles de tâches indépendantes, alors  $(E, \mathcal{I})$  est un matroïde.

- ▶ Hérité : trivial.
- ▶ Echange : Soit  $A, B$  indépendants et  $|A| < |B|$ . Existe-t-il  $x$  appartenant à  $B$  tel que  $A \cup \{x\}$  soit indépendant ?

Idée : Comparer  $N_t(A)$  et  $N_t(B)$  pour  $t = 1, \dots, n$ .

On cherche donc le plus grand  $t \leq n$  tel que  $N_t(A) < N_t(B)$ .

On sait que  $0 \leq t < n$ , donc  $N_t(A) \geq N_t(B)$ ,

$N_{t+1}(A) < N_{t+1}(B), \dots, N_n(A) < N_n(B)$ .

Dans  $B$  il y a plus de tâches de deadline  $t + 1$  que dans  $A$ ; on en prend une,  $x$ , qui n'est pas déjà dans  $A$  et  $A \cup \{x\}$  est OK.