

Chapitre 3

Algorithmes gloutons

HLIN401 : Algorithmique et complexité

L2 Informatique
Université de Montpellier
2020 – 2021

Les méthodes gloutonnes

- ▶ En un mot, **faire à chaque étape le choix possible qui semble le meilleur**. Parfois, ça marche et on obtient des bons algos...
- ▶ On voit dans ce cours des algorithmes gloutons simples et dont on peut prouver l'optimalité
D'autres sont plus *célèbres* mais plus sophistiqués (par ex. l'algorithme de compression de Huffman) et seront étudiés dans des cours plus spécialisés

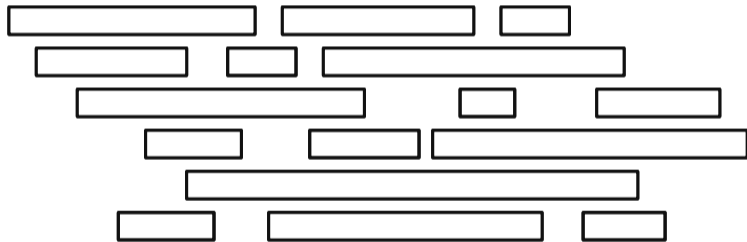
1. Exemple 1 : choix de cours
2. Qu'est-ce qu'un algorithme glouton ?
3. Exemple 2 : le sac-à-dos (fractionnaire)
4. Exemple spécial : approximation pour SETCOVER dans le plan
5. Dernier exemple : arbre couvrant de poids minimal

1. Exemple 1 : choix de cours
2. Qu'est-ce qu'un algorithme glouton ?
3. Exemple 2 : le sac-à-dos (fractionnaire)
4. Exemple spécial : approximation pour SETCOVER dans le plan
5. Dernier exemple : arbre couvrant de poids minimal

Définition du problème

Entrée un ensemble \mathcal{C} de cours $C_i = (d_i, f_i)$ [début, fin], $i = 0, \dots, n - 1$

Sortie un ensemble ordonné maximal de cours $(C_{i_1}, \dots, C_{i_k})$ tels que pour tout $j < k$,
 $f_{i_j} \leq d_{i_{j+1}} \rightsquigarrow$ cours **compatibles**

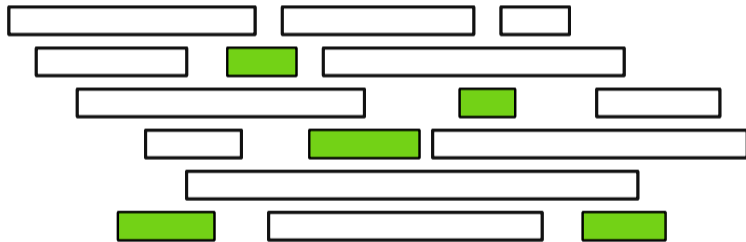


modifié d'après *Algorithms* de J. Erickson

Définition du problème

Entrée un ensemble \mathcal{C} de cours $C_i = (d_i, f_i)$ [début, fin], $i = 0, \dots, n - 1$

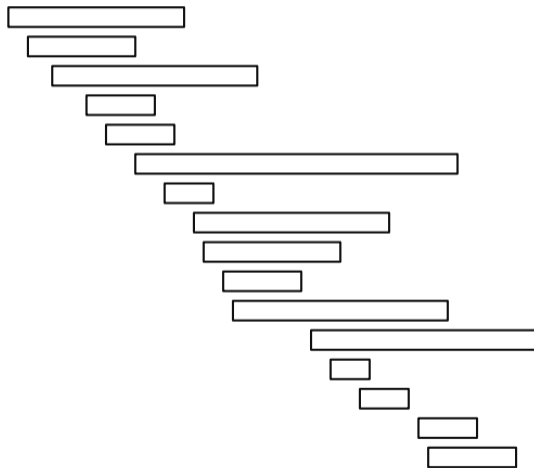
Sortie un ensemble ordonné maximal de cours $(C_{i_1}, \dots, C_{i_k})$ tels que pour tout $j < k$,
 $f_{i_j} \leq d_{i_{j+1}} \rightsquigarrow$ cours **compatibles**



modifié d'après *Algorithms* de J. Erickson

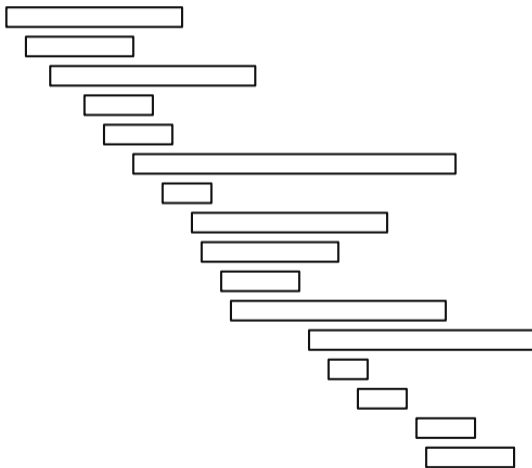
Idée gloutonne 1

- ▶ Tri des cours par **dates de début croissantes**



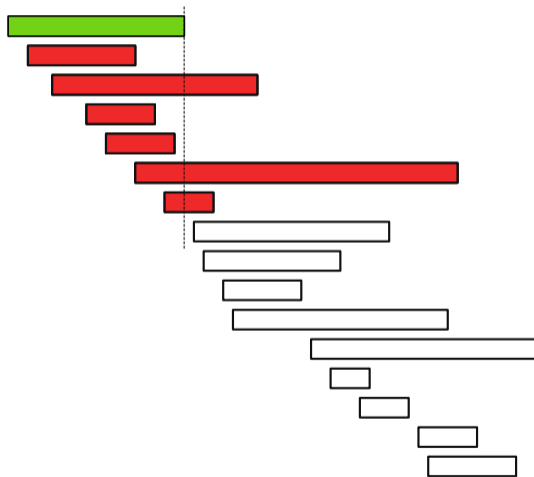
Idée gloutonne 1

- ▶ Tri des cours par **dates de début croissantes**
- ▶ Choix *glouton* : sélectionner le cours qui débute **le plus tôt**



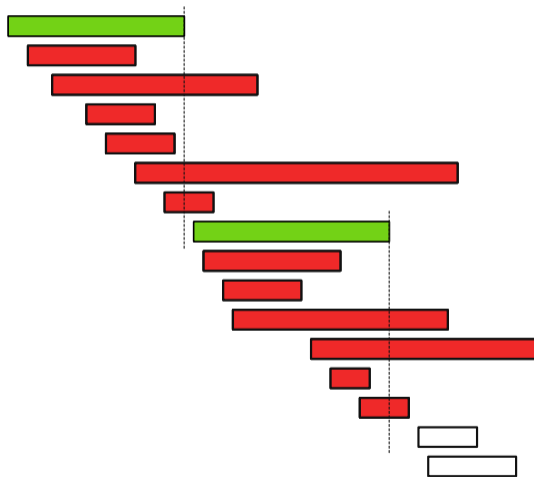
Idée gloutonne 1

- ▶ Tri des cours par **dates de début croissantes**
- ▶ Choix *glouton* : sélectionner le cours qui débute **le plus tôt**



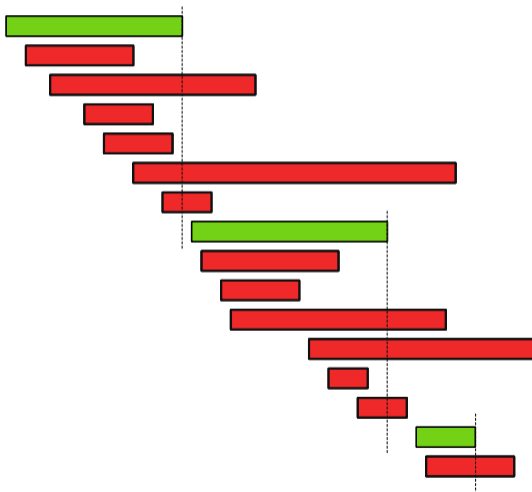
Idée gloutonne 1

- ▶ Tri des cours par **dates de début croissantes**
- ▶ Choix *glouton* : sélectionner le cours qui débute **le plus tôt**



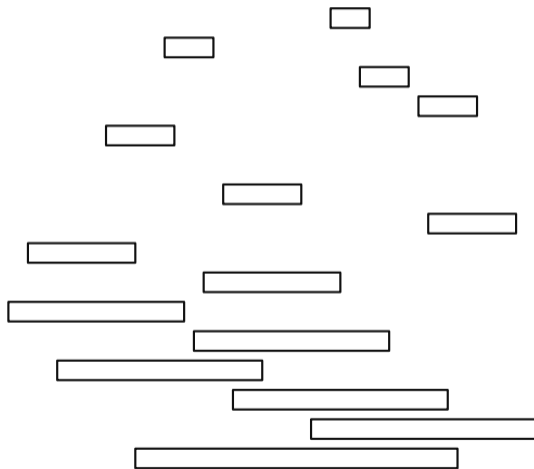
Idée gloutonne 1

- ▶ Tri des cours par **dates de début croissantes**
- ▶ Choix *glouton* : sélectionner le cours qui débute **le plus tôt**



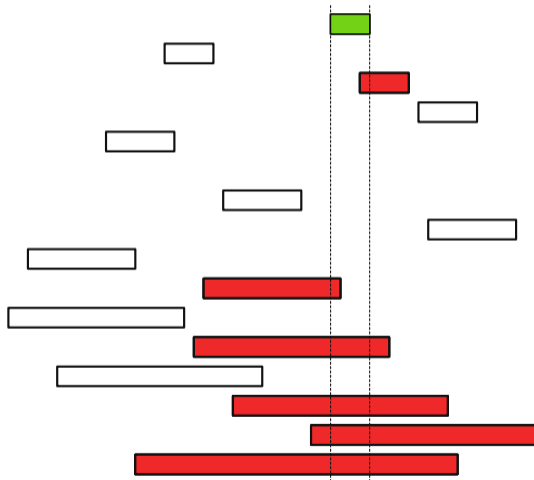
Idée gloutonne 2

- ▶ Tri des cours par **durées croissantes**
- ▶ Choix *glouton* : sélectionner le cours **le plus court**



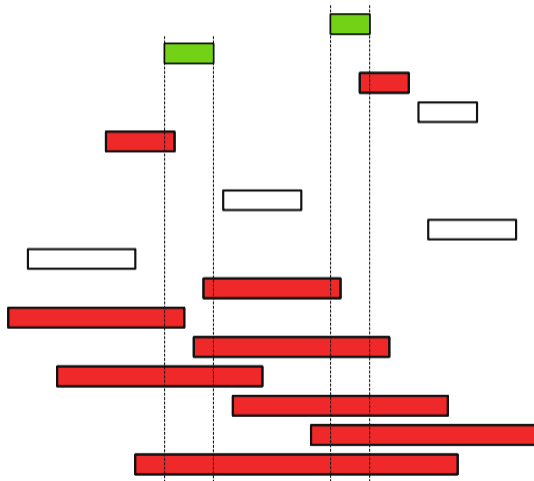
Idée gloutonne 2

- ▶ Tri des cours par **durées croissantes**
- ▶ Choix *glouton* : sélectionner le cours **le plus court**



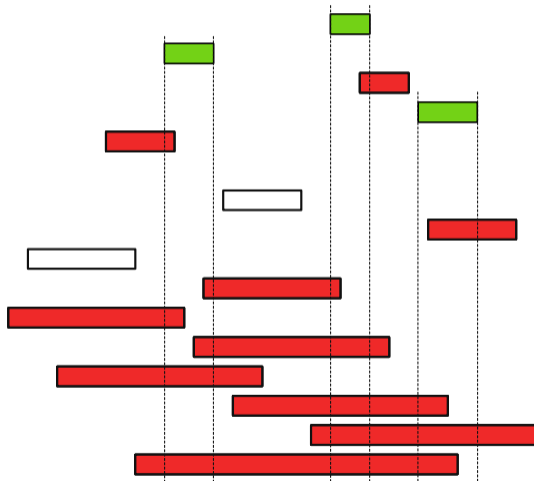
Idée gloutonne 2

- ▶ Tri des cours par **durées croissantes**
- ▶ Choix *glouton* : sélectionner le cours **le plus court**



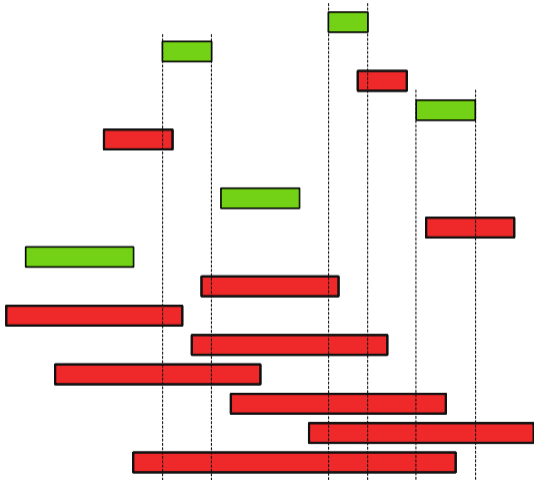
Idée gloutonne 2

- ▶ Tri des cours par **durées croissantes**
- ▶ Choix *glouton* : sélectionner le cours **le plus court**



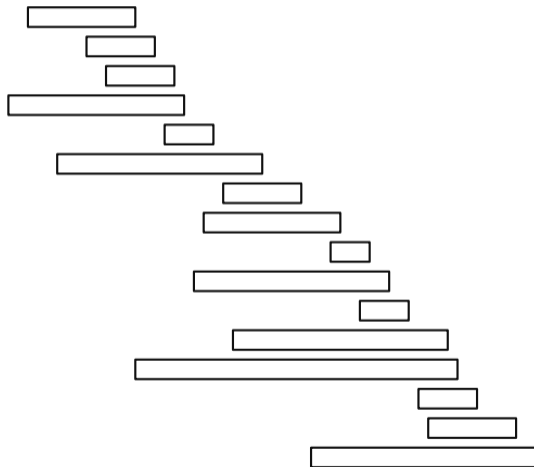
Idée gloutonne 2

- ▶ Tri des cours par **durées croissantes**
- ▶ Choix *glouton* : sélectionner le cours **le plus court**



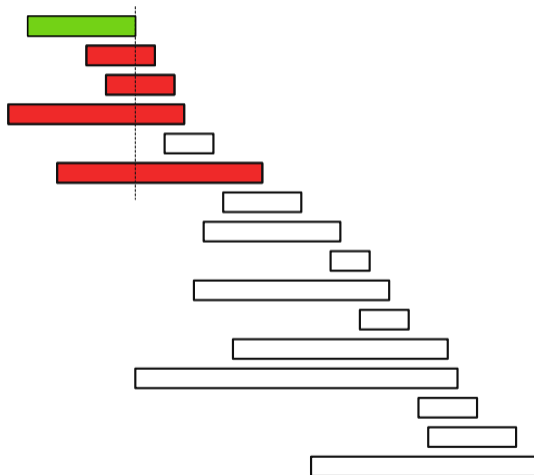
Idée gloutonne 3

- ▶ Tri des cours par **dates de fin croissantes**
- ▶ Choix *glouton* : sélectionner le cours qui finit **le plus tôt**



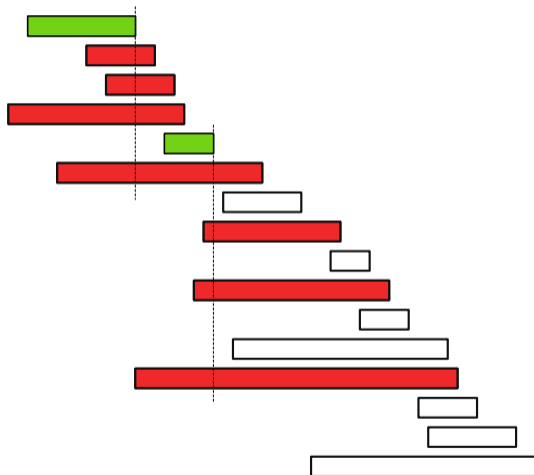
Idée gloutonne 3

- ▶ Tri des cours par **dates de fin croissantes**
- ▶ Choix *glouton* : sélectionner le cours qui finit **le plus tôt**



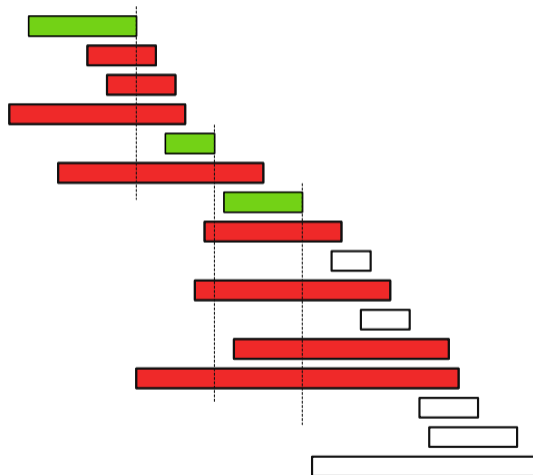
Idée gloutonne 3

- ▶ Tri des cours par **dates de fin croissantes**
- ▶ Choix *glouton* : sélectionner le cours qui finit **le plus tôt**



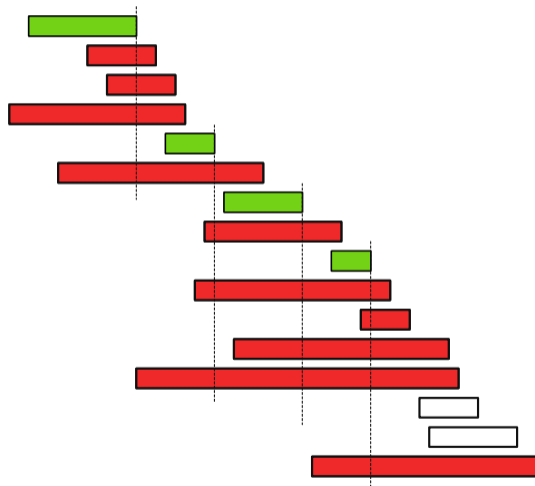
Idée gloutonne 3

- ▶ Tri des cours par **dates de fin croissantes**
- ▶ Choix *glouton* : sélectionner le cours qui finit **le plus tôt**



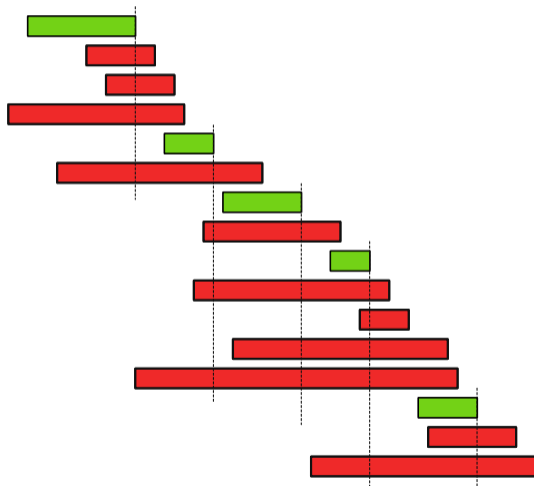
Idée gloutonne 3

- ▶ Tri des cours par **dates de fin croissantes**
- ▶ Choix *glouton* : sélectionner le cours qui finit **le plus tôt**



Idée gloutonne 3

- ▶ Tri des cours par **dates de fin croissantes**
- ▶ Choix *glouton* : sélectionner le cours qui finit **le plus tôt**



Que fait-on ?

- ▶ Idée 1 donne une solution de taille 3 : on l'oublie !

Que fait-on ?

- ▶ Idée 1 donne une solution de taille 3 : on l'oublie !
- ▶ Idées 2 et 3 donnent une solution de taille 5
 - ▶ Les deux sont bonnes ?

Que fait-on ?

- ▶ Idée 1 donne une solution de taille 3 : on l'oublie !
- ▶ Idées 2 et 3 donnent une solution de taille 5
 - ▶ Les deux sont bonnes ?
- ▶ On pourrait avoir d'autres idées...
 - ▶ cours qui crée le moins d'incompatibilités
 - ▶ cours qui commence le plus tard
 - ▶ ...

Que fait-on ?

- ▶ Idée 1 donne une solution de taille 3 : on l'oublie !
- ▶ Idées 2 et 3 donnent une solution de taille 5
 - ▶ Les deux sont bonnes ?
- ▶ On pourrait avoir d'autres idées...
 - ▶ cours qui crée le moins d'incompatibilités
 - ▶ cours qui commence le plus tard
 - ▶ ...

- ▶ Lesquelles fonctionnent ?
 - ▶ Contre-exemples quand ça ne marche pas
 - ▶ Besoin de **preuves** quand ça marche !

Que fait-on ?

- ▶ Idée 1 donne une solution de taille 3 : on l'oublie !
- ▶ Idées 2 et 3 donnent une solution de taille 5
 - ▶ Les deux sont bonnes ?
- ▶ On pourrait avoir d'autres idées...
 - ▶ cours qui crée le moins d'incompatibilités
 - ▶ cours qui commence le plus tard
 - ▶ ...

- ▶ Lesquelles fonctionnent ?
 - ▶ Contre-exemples quand ça ne marche pas
 - ▶ Besoin de **preuves** quand ça marche !
- ▶ Aujourd'hui : choix de l'idée 3 (« finit le plus tôt »)
- ▶ TD : autres choix

Algorithme glouton

Algorithme : CHOIXCOURSGLOUTON(\mathcal{C})

Trier \mathcal{C} en fonction des fins

$I \leftarrow \{\mathcal{C}_{[0]}\}$

// Indice des cours choisis

$f \leftarrow \text{FIN}(\mathcal{C}_{[0]})$

// Fin du dernier cours choisi

pour $i = 1$ à $n - 1$:

si DÉBUT($\mathcal{C}_{[i]}$) $\geq f$:

$I \leftarrow I \cup \{\mathcal{C}_{[i]}\}$

$f \leftarrow \text{FIN}(\mathcal{C}_{[i]})$

renvoyer I

Algorithme glouton

Algorithme : CHOIXCOURSGLOUTON(\mathcal{C})

Trier \mathcal{C} en fonction des fins

$I \leftarrow \{\mathcal{C}_{[0]}\}$

// Indice des cours choisis

$f \leftarrow \text{FIN}(\mathcal{C}_{[0]})$

// Fin du dernier cours choisi

pour $i = 1$ à $n - 1$:

si $\text{DÉBUT}(\mathcal{C}_{[i]}) \geq f$:

$I \leftarrow I \cup \{\mathcal{C}_{[i]}\}$

$f \leftarrow \text{FIN}(\mathcal{C}_{[i]})$

renvoyer I

Question

Quelle est la complexité de CHOIXCOURSGLOUTON ?

Algorithme glouton

Algorithme : CHOIXCOURSGLOUTON(\mathcal{C})

Trier \mathcal{C} en fonction des fins

$I \leftarrow \{\mathcal{C}_{[0]}\}$

// Indice des cours choisis

$f \leftarrow \text{FIN}(\mathcal{C}_{[0]})$

// Fin du dernier cours choisi

pour $i = 1$ à $n - 1$:

si $\text{DÉBUT}(\mathcal{C}_{[i]}) \geq f$:

$I \leftarrow I \cup \{\mathcal{C}_{[i]}\}$

$f \leftarrow \text{FIN}(\mathcal{C}_{[i]})$

renvoyer I

Question

Quelle est la complexité de CHOIXCOURSGLOUTON ?

\rightsquigarrow coût du tri : $O(n \log n)$

Correction de l'algorithme

Théorème

L'algorithme CHOIXCOURSGLOUTON est optimal : il renvoie un ensemble maximal de cours compatibles, c'est-à-dire qu'il n'existe pas d'ensemble strictement plus grand de cours compatibles.

Preuve

Correction de l'algorithme

Théorème

L'algorithme CHOIXCOURSGLOUTON est optimal : il renvoie un ensemble maximal de cours compatibles, c'est-à-dire qu'il n'existe pas d'ensemble strictement plus grand de cours compatibles.

Preuve

- ▶ Notons $\mathcal{C} = (C_0, \dots, C_{n-1})$ les cours triés par dates de fin croissantes.

Correction de l'algorithme

Théorème

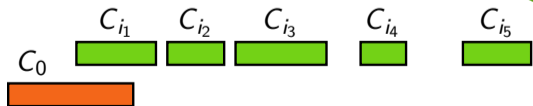
L'algorithme CHOIXCOURSGLOUTON est optimal : il renvoie un ensemble maximal de cours compatibles, c'est-à-dire qu'il n'existe pas d'ensemble strictement plus grand de cours compatibles.

Preuve

- ▶ Notons $\mathcal{C} = (C_0, \dots, C_{n-1})$ les cours triés par dates de fin croissantes.
- ▶ **Petit Lemme** : il existe une solution optimale contenant C_0

En effet, soit $\mathcal{B} = (C_{i_1}, C_{i_2}, \dots, C_{i_k})$ une solution optimale.

- ▶ Si $C_{i_1} = C_0$, on est content...
- ▶ Sinon, par définition de C_0 , on a $f_0 \leq f_{i_1}$ et $(\mathcal{B} \setminus C_{i_1}) \cup C_0$ est aussi une solution optimale, contenant C_0 cette fois.



Correction de l'algorithme

Théorème

L'algorithme CHOIXCOURSGLOUTON est optimal : il renvoie un ensemble maximal de cours compatibles, c'est-à-dire qu'il n'existe pas d'ensemble strictement plus grand de cours compatibles.

Preuve

- ▶ Notons $\mathcal{C} = (C_0, \dots, C_{n-1})$ les cours triés par dates de fin croissantes.
- ▶ **Petit Lemme** : il existe une solution optimale contenant C_0

En effet, soit $\mathcal{B} = (C_{i_1}, C_{i_2}, \dots, C_{i_k})$ une solution optimale.

- ▶ Si $C_{i_1} = C_0$, on est content...
- ▶ Sinon, par définition de C_0 , on a $f_0 \leq f_{i_1}$ et $(\mathcal{B} \setminus C_{i_1}) \cup C_0$ est aussi une solution optimale, contenant C_0 cette fois.

↪ **CHOIXCOURSGLOUTON fait le premier bon choix !**

Correction de l'algorithme

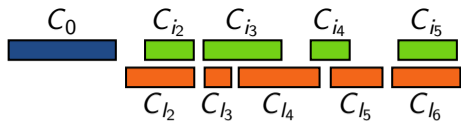
Théorème

L'algorithme CHOIXCOURSGLOUTON est optimal : il renvoie un ensemble maximal de cours compatibles, c'est-à-dire qu'il n'existe pas d'ensemble strictement plus grand de cours compatibles.

Preuve

- ▶ Notons maintenant $\mathcal{C}_0 = \{C_{j_1}, C_{j_2}, \dots, C_{j_{n_1}}\}$ les cours compatibles avec C_0 , triés par dates de fin croissantes (càd les cours dont la date de début est $\geq f_0$).

Correction de l'algorithme



Théorème

L'algorithme CHOIXCOURSGLOUTON est optimal : il renvoie un ensemble maximal de cours compatibles, c'est-à-dire qu'il n'existe pas d'ensemble strictement plus grand de cours compatibles.

Preuve

- ▶ Notons maintenant $\mathcal{C}_0 = \{C_{j_1}, C_{j_2}, \dots, C_{j_{n_1}}\}$ les cours compatibles avec C_0 , triés par dates de fin croissantes (càd les cours dont la date de début est $\geq f_0$).
- ▶ **Moyen Lemme** : il existe une solution optimale formée de C_0 et d'une solution optimale du problème sur l'entrée \mathcal{C}_0

En effet, soit $\mathcal{B} = (C_0, C_{i_2}, \dots, C_{i_k})$ une solution optimale contenant C_0 (ça existe par le petit lemme).

- ▶ Si $(C_{i_2}, \dots, C_{i_k})$ n'est pas une solution optimale pour \mathcal{C}_0 , alors il existerait une meilleure solution $(C_{l_2}, \dots, C_{l_{k+1}})$ sur \mathcal{C}_0 .
- ▶ Mais $(C_{l_2}, \dots, C_{l_{k+1}})$ sont tous compatibles avec C_0 , et $(C_0, C_{l_2}, \dots, C_{l_{k+1}})$ serait meilleure que \mathcal{B} : impossible !

Correction de l'algorithme

Théorème

L'algorithme CHOIXCOURSGLOUTON est optimal : il renvoie un ensemble maximal de cours compatibles, c'est-à-dire qu'il n'existe pas d'ensemble strictement plus grand de cours compatibles.

Preuve

- ▶ Notons maintenant $\mathcal{C}_0 = \{C_{j_1}, C_{j_2}, \dots, C_{j_{n_1}}\}$ les cours compatibles avec C_0 , triés par dates de fin croissantes (càd les cours dont la date de début est $\geq f_0$).
- ▶ **Moyen Lemme** : il existe une solution optimale formée de C_0 et d'une solution optimale du problème sur l'entrée \mathcal{C}_0
- ▶ Du coup, par le Petit Lemme, comme il existe une solution optimale de \mathcal{C}_0 commençant par C_{j_1} , il existe une solution optimale du problème de départ commençant par (C_0, C_{j_1}) .

↪ **CHOIXCOURSGLOUTON fait aussi un second bon choix !**

Correction de l'algorithme

Théorème

L'algorithme CHOIXCOURSGLOUTON est optimal : il renvoie un ensemble maximal de cours compatibles, c'est-à-dire qu'il n'existe pas d'ensemble strictement plus grand de cours compatibles.

Preuve

- ▶ On peut mettre en place la récurrence :

$\mathcal{P}_s = \ll$ Soit $(C_{p_1}, C_{p_2}, \dots, C_{p_s})$ les s premiers choix de cours de CHOIXCOURSGLOUTON et \mathcal{C}_s les cours compatibles avec tous ces cours là. Alors il existe une solution au problème initial formée de $(C_{p_1}, C_{p_2}, \dots, C_{p_s})$ et d'une solution optimale sur \mathcal{C}_s . \gg

Correction de l'algorithme

Théorème

L'algorithme CHOIXCOURSGLOUTON est optimal : il renvoie un ensemble maximal de cours compatibles, c'est-à-dire qu'il n'existe pas d'ensemble strictement plus grand de cours compatibles.

Preuve

- ▶ On peut mettre en place la récurrence :

$\mathcal{P}_s = \ll$ Soit $(C_{p_1}, C_{p_2}, \dots, C_{p_s})$ les s premiers choix de cours de CHOIXCOURSGLOUTON et \mathcal{C}_s les cours compatibles avec tous ces cours là. Alors il existe une solution au problème initial formée de $(C_{p_1}, C_{p_2}, \dots, C_{p_s})$ et d'une solution optimale sur \mathcal{C}_s . \gg

- ▶ \mathcal{P}_1 vraie par le Moyen lemme.

Correction de l'algorithme

Théorème

L'algorithme CHOIXCOURSGLOUTON est optimal : il renvoie un ensemble maximal de cours compatibles, c'est-à-dire qu'il n'existe pas d'ensemble strictement plus grand de cours compatibles.

Preuve

- ▶ On peut mettre en place la récurrence :

$\mathcal{P}_s = \ll$ Soit $(C_{p_1}, C_{p_2}, \dots, C_{p_s})$ les s premiers choix de cours de CHOIXCOURSGLOUTON et \mathcal{C}_s les cours compatibles avec tous ces cours là. Alors il existe une solution au problème initial formée de $(C_{p_1}, C_{p_2}, \dots, C_{p_s})$ et d'une solution optimale sur \mathcal{C}_s . \gg

- ▶ Si \mathcal{P}_s vraie, alors par le Moyen Lemme, il existe une solution opt. sur \mathcal{C}_s formée du premier cours $C_{p_{s+1}}$ de \mathcal{C}_s et d'une solution opt. sur les cours de \mathcal{C}_s compatibles avec $C_{p_{s+1}}$. Et \mathcal{P}_{s+1} est vraie !

Correction de l'algorithme

Théorème

L'algorithme CHOIXCOURSGLOUTON est optimal : il renvoie un ensemble maximal de cours compatibles, c'est-à-dire qu'il n'existe pas d'ensemble strictement plus grand de cours compatibles.

Preuve

- ▶ On peut mettre en place la récurrence :

$\mathcal{P}_s = \ll$ Soit $(C_{p_1}, C_{p_2}, \dots, C_{p_s})$ les s premiers choix de cours de CHOIXCOURSGLOUTON et \mathcal{C}_s les cours compatibles avec tous ces cours là. Alors il existe une solution au problème initial formée de $(C_{p_1}, C_{p_2}, \dots, C_{p_s})$ et d'une solution optimale sur \mathcal{C}_s . \gg

\rightsquigarrow **CHOIXCOURSGLOUTON renvoie une solution optimale au problème !** ■

1. Exemple 1 : choix de cours
2. Qu'est-ce qu'un algorithme glouton ?
3. Exemple 2 : le sac-à-dos (fractionnaire)
4. Exemple spécial : approximation pour SETCOVER dans le plan
5. Dernier exemple : arbre couvrant de poids minimal

Idée générale

Un **algorithme glouton** fait à chaque étape un choix **localement optimal** dans le but d'obtenir à la fin un **optimum global**.

Idée générale

Un **algorithme glouton** fait à chaque étape un choix **localement optimal** dans le but d'obtenir à la fin un **optimum global**.

Exemple du choix de cours

- ▶ Optimum local : cours qui minimise les incompatibilités
- ▶ Optimum global : maximum de cours compatibles

Idée générale

Un **algorithme glouton** fait à chaque étape un choix **localement optimal** dans le but d'obtenir à la fin un **optimum global**.

Exemple du choix de cours

- ▶ Optimum local : cours qui minimise les incompatibilités
- ▶ Optimum global : maximum de cours compatibles

Remarques

- ▶ Construction pas-à-pas d'une solution
 - ▶ Algorithmes simples à concevoir... mais pas toujours parfaits!
- ↪ Résolution exacte, approximation, heuristique

Concevoir des algorithmes gloutons

1. Décider d'un **choix glouton**

- ▶ Ajout d'un nouvel élément à la solution en construction
- ▶ Recommencer sur le sous-problème restant

Concevoir des algorithmes gloutons

1. Décider d'un **choix glouton**
 - ▶ Ajout d'un nouvel élément à la solution en construction
 - ▶ Recommencer sur le sous-problème restant
2. Chercher un cas où **ça ne marche pas**
 - ▶ Si on en trouve, retourner en 1.
 - ▶ Sinon, continuer en 3.

Concevoir des algorithmes gloutons

Tri par durées croissantes
 $\{(11, 14), (7, 12), (13, 19)\}$



1. Décider d'un **choix glouton**

- ▶ Ajout d'un nouvel élément à la solution en construction
- ▶ Recommencer sur le sous-problème restant

2. Chercher un cas où **ça ne marche pas**

- ▶ Si on en trouve, retourner en 1.
- ▶ Sinon, continuer en 3.

Concevoir des algorithmes gloutons

Tri par durées croissantes
 $\{(11, 14), (7, 12), (13, 19)\}$



1. Décider d'un **choix glouton**

- ▶ Ajout d'un nouvel élément à la solution en construction
- ▶ Recommencer sur le sous-problème restant

2. Chercher un cas où **ça ne marche pas**

- ▶ Si on en trouve, retourner en 1.
- ▶ Sinon, continuer en 3.

3. **Démontrer** que l'algorithme est correct

- ▶ Il existe une solution optimale contenant le choix local
- ▶ Choix local + glouton pour le reste \rightsquigarrow solution optimale

Concevoir des algorithmes gloutons

Tri par durées croissantes
 $\{(11, 14), (7, 12), (13, 19)\}$



1. Décider d'un **choix glouton**
 - ▶ Ajout d'un nouvel élément à la solution en construction
 - ▶ Recommencer sur le sous-problème restant
2. Chercher un cas où **ça ne marche pas**
 - ▶ Si on en trouve, retourner en 1.
 - ▶ Sinon, continuer en 3.
3. **Démontrer** que l'algorithme est correct
 - ▶ Il existe une solution optimale contenant le choix local
 - ▶ Choix local + glouton pour le reste \rightsquigarrow solution optimale
4. Étudier la **complexité** de l'algorithme

Algorithme glouton générique

Problème générique

Entrée Un ensemble fini X , avec une **valeur** v_x pour tout $x \in X$

Paramètre Une propriété \mathcal{P} définissant les sous-ensembles $A \subset X$ **acceptables**, telle que si A vérifie \mathcal{P} , tout sous-ensemble $B \subset A$ vérifie \mathcal{P} (propriété *monotone vers le bas*)

Sortie Un sous-ensemble $A \subset X$, acceptable, et qui maximise/minimise $v_A = \sum_{x \in A} v_x$ (parmi les sous-ensembles acceptables)

Algorithme glouton générique

Problème générique

Entrée Un ensemble fini X , avec une **valeur** v_x pour tout $x \in X$

Paramètre Une propriété \mathcal{P} définissant les sous-ensembles $A \subset X$ **acceptables**, telle que si A vérifie \mathcal{P} , tout sous-ensemble $B \subset A$ vérifie \mathcal{P} (propriété *monotone vers le bas*)

Sortie Un sous-ensemble $A \subset X$, acceptable, et qui maximise/minimise $v_A = \sum_{x \in A} v_x$ (parmi les sous-ensembles acceptables)

Choix de cours

- ▶ $X = \{(d_i, f_i) : 0 \leq i < n\}$; $v_x = 1$ pour tout $x \in X$ ($X = \mathcal{C}$)
- ▶ $\mathcal{P} : A \subset X$ vérifie \mathcal{P} si les cours de A sont compatibles
- ▶ On cherche l'ensemble de cours compatibles *le plus grand*

Algorithme glouton générique

Problème générique

Entrée Un ensemble fini X , avec une **valeur** v_x pour tout $x \in X$

Paramètre Une propriété \mathcal{P} définissant les sous-ensembles $A \subset X$ **acceptables**, telle que si A vérifie \mathcal{P} , tout sous-ensemble $B \subset A$ vérifie \mathcal{P} (propriété *monotone vers le bas*)

Sortie Un sous-ensemble $A \subset X$, acceptable, et qui maximise/minimise $v_A = \sum_{x \in A} v_x$ (parmi les sous-ensembles acceptables)

Algorithme : GLOUTONGÉNÉRIQUE(X, \mathcal{P})

Trier X par *valeurs* croissantes/décroissantes + critère glouton

$S \leftarrow \emptyset$

pour $x \in X$ (*dans l'ordre du tri*) :

si $S \cup \{x\}$ vérifie \mathcal{P} : ajouter x à S

 // $S \leftarrow S \cup \{x\}$

renvoyer S

Théorème des algorithmes gloutons

Théorème

On considère le problème générique avec la propriété \mathcal{P} . Si pour toute entrée X , il existe une solution optimale S telle que

- ▶ *le premier élément x_0 de X , **dans l'ordre du tri**, appartienne à S*
- ▶ *$S \setminus x_0$ soit une solution optimale sur l'entrée $X \setminus x_0$ du problème de paramètre \mathcal{P}' où A vérifie \mathcal{P}' si $A \cup \{x_0\}$ vérifie \mathcal{P}*

Alors GLOUTONGÉNÉRIQUE est optimal.

Théorème des algorithmes gloutons

Théorème

On considère le problème générique avec la propriété \mathcal{P} . Si pour toute entrée X , il existe une solution optimale S telle que

- ▶ *le premier élément x_0 de X , **dans l'ordre du tri**, appartienne à S*
- ▶ *$S \setminus x_0$ soit une solution optimale sur l'entrée $X \setminus x_0$ du problème de paramètre \mathcal{P}' où A vérifie \mathcal{P}' si $A \cup \{x_0\}$ vérifie \mathcal{P}*

Alors GLOUTONGÉNÉRIQUE est optimal.

Exemple du choix de cours

- ▶ X : ensemble des cours ($v_x = 1$ pour tout x); \mathcal{P} : cours compatibles entre eux
- ▶ Tri : dates de fin croissantes
- ▶ Preuve :
 - ▶ Il existe un ensemble de cours optimal contenant le 1^{er} cours
 - ▶ En enlevant le 1^{er} cours, il reste un ensemble optimal pour les cours commençant après la fin du 1^{er} cours

Théorème des algorithmes gloutons

Théorème

On considère le problème générique avec la propriété \mathcal{P} . Si pour toute entrée X , il existe une solution optimale S telle que

- ▶ *le premier élément x_0 de X , **dans l'ordre du tri**, appartienne à S*
- ▶ *$S \setminus x_0$ soit une solution optimale sur l'entrée $X \setminus x_0$ du problème de paramètre \mathcal{P}' où A vérifie \mathcal{P}' si $A \cup \{x_0\}$ vérifie \mathcal{P}*

Alors GLOUTONGÉNÉRIQUE est optimal.

Preuve par récurrence sur $|X|$

- ▶ Si $|X| = 0$, la solution optimale est \emptyset
- ▶ Soit X une entrée avec $|X| > 0$. Par hyp. de récurrence, GLOUTONGÉNÉRIQUE trouve une solution optimale S' pour $X \setminus x_0$ avec la propriété \mathcal{P}' . Donc $S' \cup \{x_0\}$ est optimale pour X avec la propriété \mathcal{P} , sinon on obtient une contradiction... ■

En pratique

- ▶ Il existe une théorie générale des algorithmes gloutons
 - ▶ basée sur la notion de **matroïde**
 - ▶ mais certains algorithmes « type glouton » ne rentrent pas exactement dans le moule

En pratique

- ▶ Il existe une théorie générale des algorithmes gloutons
 - ▶ basée sur la notion de **matroïde**
 - ▶ mais certains algorithmes « type glouton » ne rentrent pas exactement dans le moule
- ▶ Dans ce cours : étude de plusieurs exemples
 - ▶ utilisation du théorème pour faciliter les preuves

En pratique

- ▶ Il existe une théorie générale des algorithmes gloutons
 - ▶ basée sur la notion de **matroïde**
 - ▶ mais certains algorithmes « type glouton » ne rentrent pas exactement dans le moule
- ▶ Dans ce cours : étude de plusieurs exemples
 - ▶ utilisation du théorème pour faciliter les preuves

Objectifs :

- ▶ Savoir tenter une stratégie gloutonne
- ▶ Savoir détecter si elle marche ou non
- ▶ Savoir l'analyser (correction et complexité)

1. Exemple 1 : choix de cours
2. Qu'est-ce qu'un algorithme glouton ?
3. Exemple 2 : le sac-à-dos (fractionnaire)
4. Exemple spécial : approximation pour SETCOVER dans le plan
5. Dernier exemple : arbre couvrant de poids minimal

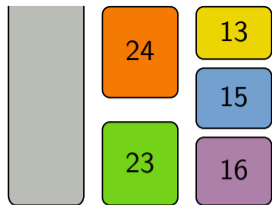
Problème du sac-à-dos

Définition du problème

Entrée un ensemble d'objets, ayant une taille t_i et une valeur v_i
une taille T de sac-à-dos

Sortie un sous-ensemble des objets qui

- ▶ rentre dans le sac : $\sum_i t_i \leq T$
- ▶ maximise la valeur totale $V = \sum_i v_i$



Problème du sac-à-dos

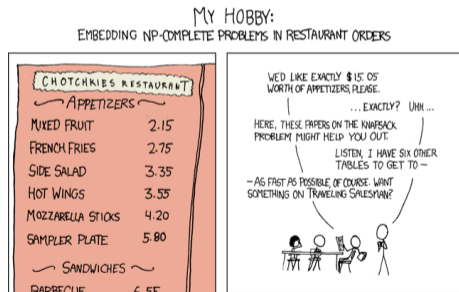
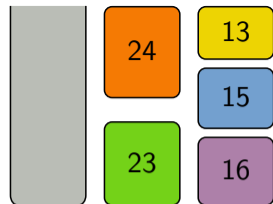
Définition du problème

Entrée un ensemble d'objets, ayant une taille t_i et une valeur v_i
une taille T de sac-à-dos

Sortie un sous-ensemble des objets qui

- ▶ rentre dans le sac : $\sum_i t_i \leq T$
- ▶ maximise la valeur totale $V = \sum_i v_i$

- ▶ Problème célèbre car utile
 - ▶ en théorie
 - ▶ en pratique
 - ▶ en cryptographie
- ▶ Difficile (NP-complet \rightsquigarrow HLIN612)



<https://xkcd.com/287/>

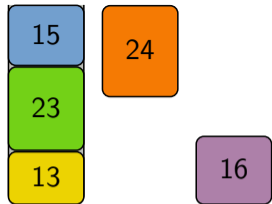
Problème du sac-à-dos

Définition du problème

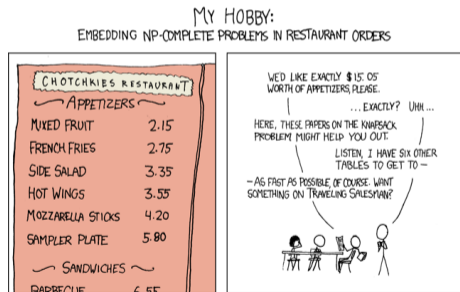
Entrée un ensemble d'objets, ayant une taille t_i et une valeur v_i
une taille T de sac-à-dos

Sortie un sous-ensemble des objets qui

- ▶ rentre dans le sac : $\sum_i t_i \leq T$
- ▶ maximise la valeur totale $V = \sum_i v_i$



- ▶ Problème célèbre car utile
 - ▶ en théorie
 - ▶ en pratique
 - ▶ en cryptographie
- ▶ Difficile (NP-complet \rightsquigarrow HLIN612)



<https://xkcd.com/287/>

Problème du sac-à-dos fractionnaire

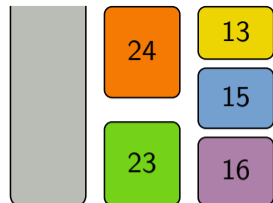
Objets *fractionnables* : on peut n'en prendre qu'une partie

Définition

Entrée un ensemble d'objets, ayant une taille t_i et une valeur v_i
une taille T de sac-à-dos

Sortie une fraction $x_i \in [0, 1]$ pour chaque objet, telle que

- ▶ le total ne dépasse pas la taille du sac : $\sum_i x_i t_i \leq T$
- ▶ la valeur totale $V = \sum_i x_i v_i$ est maximale



Problème du sac-à-dos fractionnaire

Objets *fractionnables* : on peut n'en prendre qu'une partie

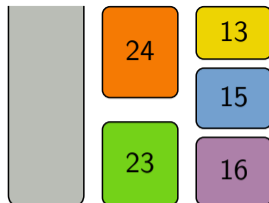
Définition

Entrée un ensemble d'objets, ayant une taille t_i et une valeur v_i
une taille T de sac-à-dos

Sortie une fraction $x_i \in [0, 1]$ pour chaque objet, telle que

- ▶ le total ne dépasse pas la taille du sac : $\sum_i x_i t_i \leq T$
- ▶ la valeur totale $V = \sum_i x_i v_i$ est maximale

- ▶ Problème simplifié!
- ▶ Approche pour résoudre le sac-à-dos



Problème du sac-à-dos fractionnaire

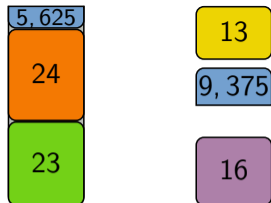
Objets *fractionnables* : on peut n'en prendre qu'une partie

Définition

Entrée un ensemble d'objets, ayant une taille t_i et une valeur v_i
une taille T de sac-à-dos

Sortie une fraction $x_i \in [0, 1]$ pour chaque objet, telle que

- ▶ le total ne dépasse pas la taille du sac : $\sum_i x_i t_i \leq T$
- ▶ la valeur totale $V = \sum_i x_i v_i$ est maximale



- ▶ Problème simplifié!
- ▶ Approche pour résoudre le sac-à-dos

Algorithme glouton

Choix glouton : choisir l'objet de meilleur *rapport qualité - prix*

Algorithme glouton

Choix glouton : choisir l'objet de meilleur *rapport qualité - prix*

Algorithme : SÀDFRACGLOUTON(O, T)

Trier les objets $O_i = (t_i, v_i)$ par v_i/t_i **décroissant**

$R \leftarrow T$ // Espace encore libre dans le sac-à-dos

pour $i = 0$ à $n - 1$ (dans l'ordre du tri) :

si $t_i \leq R$:

$x_i \leftarrow 1$

$R \leftarrow R - t_i$

sinon :

$x_i \leftarrow R/t_i$

$R \leftarrow 0$

renvoyer (x_0, \dots, x_{n-1})

Algorithme glouton

Choix glouton : choisir l'objet de meilleur *rapport qualité - prix*

Algorithme : SÀDFRACGLOUTON(O, T)

Trier les objets $O_i = (t_i, v_i)$ par v_i/t_i **décroissant**

$R \leftarrow T$ // Espace encore libre dans le sac-à-dos

pour $i = 0$ à $n - 1$ (dans l'ordre du tri) :

si $t_i \leq R$:

$x_i \leftarrow 1$

$R \leftarrow R - t_i$

sinon :

$x_i \leftarrow R/t_i$

$R \leftarrow 0$

renvoyer (x_0, \dots, x_{n-1})

Lemme

La complexité de SÀDFRACGLOUTON est $O(n \log n)$.

Correction de l'algorithme

Lemme

Soit $O = \{(t_0, v_0), \dots, (t_{n-1}, v_{n-1})\}$ un ensemble d'objets et T une taille de sac-à-dos, où $v_0/t_0 \geq v_1/t_1 \geq \dots \geq v_{n-1}/t_{n-1}$. Alors il existe une solution optimale (x_0, \dots, x_{n-1}) sur l'entrée (O, T) telle que

- ▶ $x_0 = \begin{cases} 1 & \text{si } t_0 \leq T \\ T/t_0 & \text{sinon} \end{cases}$
- ▶ (x_1, \dots, x_{n-1}) est solution optimale sur l'entrée $\{(t_1, v_1), \dots, (t_{n-1}, v_{n-1})\}$ et $T - t_0$

Preuve : en TD!

↪ Optimalité de SÀDFRACGLOUTON d'après le théorème des algorithmes gloutons!

1. Exemple 1 : choix de cours
2. Qu'est-ce qu'un algorithme glouton ?
3. Exemple 2 : le sac-à-dos (fractionnaire)
4. Exemple spécial : approximation pour SETCOVER dans le plan
5. Dernier exemple : arbre couvrant de poids minimal

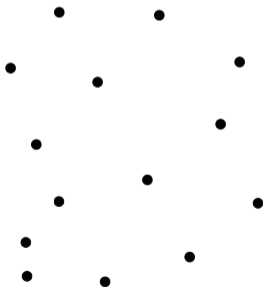
Le problème

SETCOVER

Entrée n maisons placées dans le plan

Sortie un ensemble minimal de maisons où placer une antenne Wifi :

- ▶ chaque antenne a une portée de 100 m
- ▶ toutes les maisons doivent être couvertes



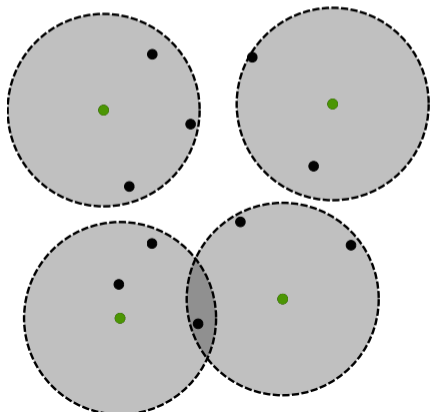
Le problème

SETCOVER

Entrée n maisons placées dans le plan

Sortie un ensemble minimal de maisons où placer une antenne Wifi :

- ▶ chaque antenne a une portée de 100 m
- ▶ toutes les maisons doivent être couvertes



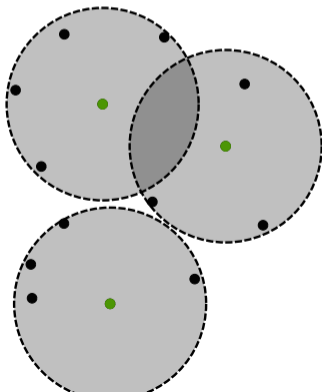
Le problème

SETCOVER

Entrée n maisons placées dans le plan

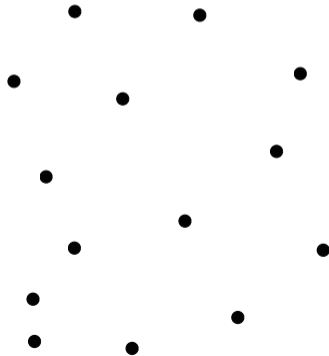
Sortie un ensemble minimal de maisons où placer une antenne Wifi :

- ▶ chaque antenne a une portée de 100 m
- ▶ toutes les maisons doivent être couvertes



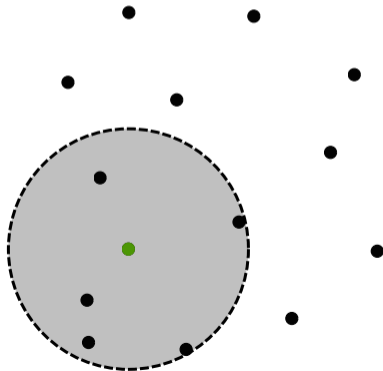
Choix glouton

À chaque étape, on choisit la maison qui permet de couvrir le plus de maisons non encore couvertes



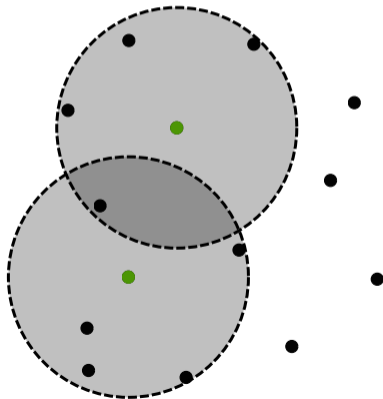
Choix glouton

À chaque étape, on choisit la maison qui permet de couvrir le plus de maisons non encore couvertes



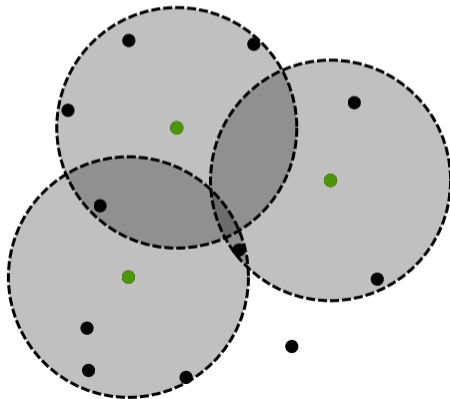
Choix glouton

À chaque étape, on choisit la maison qui permet de couvrir le plus de maisons non encore couvertes



Choix glouton

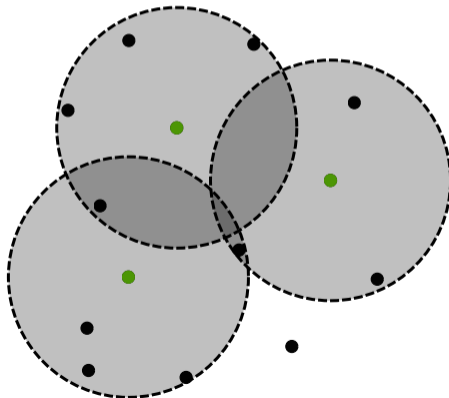
À chaque étape, on choisit la maison qui permet de couvrir le plus de maisons non encore couvertes



Choix glouton

À chaque étape, on choisit la maison qui permet de couvrir le plus de maisons non encore couvertes

Choix non optimal mais...



Propriétés du choix glouton

Lemme

L'algorithme présenté peut être implanté en temps $O(n^3)$

Preuve : en TD !

Propriétés du choix glouton

Lemme

L'algorithme présenté peut être implanté en temps $O(n^3)$

Preuve : en TD !

Lemme

Si k est le nombre minimal d'antennes nécessaires, l'algorithme trouve toujours une solution avec $\leq k \ln n$ antennes

Preuve Notons n_t le nb de maisons *non couvertes* après l'étape t .

Propriétés du choix glouton

Lemme

L'algorithme présenté peut être implanté en temps $O(n^3)$

Preuve : en TD !

Lemme

Si k est le nombre minimal d'antennes nécessaires, l'algorithme trouve toujours une solution avec $\leq k \ln n$ antennes

Preuve Notons n_t le nb de maisons *non couvertes* après l'étape t .

► $n_0 = n$

Propriétés du choix glouton

Lemme

L'algorithme présenté peut être implanté en temps $O(n^3)$

Preuve : en TD !

Lemme

Si k est le nombre minimal d'antennes nécessaires, l'algorithme trouve toujours une solution avec $\leq k \ln n$ antennes

Preuve Notons n_t le nb de maisons *non couvertes* après l'étape t .

- ▶ $n_0 = n$
- ▶ Puisque k antennes suffisent pour couvrir *toutes* les maisons, k antennes suffisent pour les n_t maisons non encore couvertes
- ▶ Donc l'emplacement qui couvre le plus de maisons non encore couvertes en couvre $\geq n_t/k$ (preuve : par contradiction)

Propriétés du choix glouton

Lemme

L'algorithme présenté peut être implanté en temps $O(n^3)$

Preuve : en TD !

Lemme

Si k est le nombre minimal d'antennes nécessaires, l'algorithme trouve toujours une solution avec $\leq k \ln n$ antennes

Preuve Notons n_t le nb de maisons *non couvertes* après l'étape t .

- ▶ $n_0 = n$
- ▶ Puisque k antennes suffisent pour couvrir *toutes* les maisons, k antennes suffisent pour les n_t maisons non encore couvertes
- ▶ Donc l'emplacement qui couvre le plus de maisons non encore couvertes en couvre $\geq n_t/k$ (preuve : par contradiction)
- ▶ Alors
$$n_{t+1} \leq n_t - n_t/k = \left(1 - \frac{1}{k}\right)n_t$$

Propriétés du choix glouton

Lemme

L'algorithme présenté peut être implanté en temps $O(n^3)$

Preuve : en TD !

Lemme

Si k est le nombre minimal d'antennes nécessaires, l'algorithme trouve toujours une solution avec $\leq k \ln n$ antennes

Preuve Notons n_t le nb de maisons *non couvertes* après l'étape t .

- ▶ En résumé : $n_0 = n$ et pour tout $t \geq 0$ on a $n_{t+1} \leq (1 - \frac{1}{k})n_t$

Propriétés du choix glouton

Lemme

L'algorithme présenté peut être implanté en temps $O(k \ln n)$

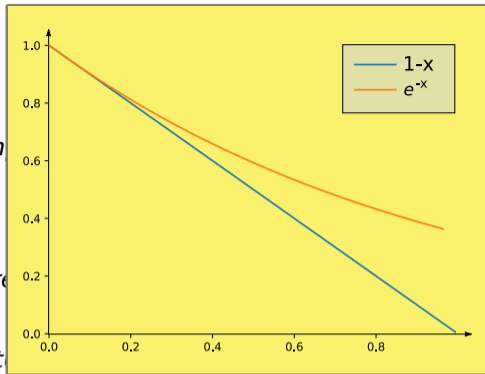
Preuve : en TD!

Lemme

Si k est le nombre minimal d'antennes nécessaire pour couvrir n maisons, alors il existe une solution avec $\leq k \ln n$ antennes

Preuve Notons n_t le nb de maisons non couvertes à l'étape t

- ▶ En résumé : $n_0 = n$ et pour tout $t \geq 0$ on a $n_{t+1} \leq (1 - \frac{1}{k})n_t$
- ▶ Or $1 - x \leq e^{-x}$ pour tout x , donc $n_{t+1} \leq e^{-\frac{1}{k}} n_t$



Propriétés du choix glouton

Lemme

L'algorithme présenté peut être implanté en temps $O(n^3)$

Preuve : en TD !

Lemme

Si k est le nombre minimal d'antennes nécessaires, l'algorithme trouve toujours une solution avec $\leq k \ln n$ antennes

Preuve Notons n_t le nb de maisons *non couvertes* après l'étape t .

- ▶ En résumé : $n_0 = n$ et pour tout $t \geq 0$ on a $n_{t+1} \leq (1 - \frac{1}{k})n_t$
- ▶ Or $1 - x \leq e^{-x}$ pour tout x , donc $n_{t+1} \leq e^{-\frac{1}{k}} n_t$
- ▶ D'où $n_t \leq e^{-\frac{1}{k}} n_{t-1} \leq e^{-\frac{2}{k}} n_{t-2} \leq \dots \leq e^{-\frac{t}{k}} n_0 = e^{-\frac{t}{k}} n$

Propriétés du choix glouton

Lemme

L'algorithme présenté peut être implanté en temps $O(n^3)$

Preuve : en TD !

Lemme

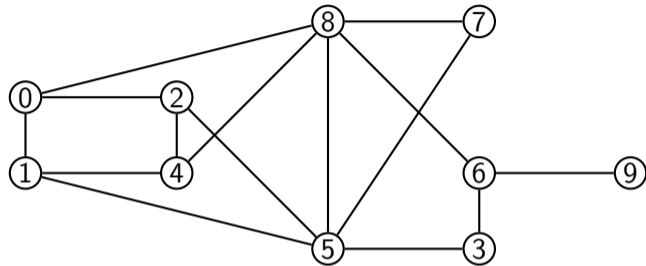
Si k est le nombre minimal d'antennes nécessaires, l'algorithme trouve toujours une solution avec $\leq k \ln n$ antennes

Preuve Notons n_t le nb de maisons *non couvertes* après l'étape t .

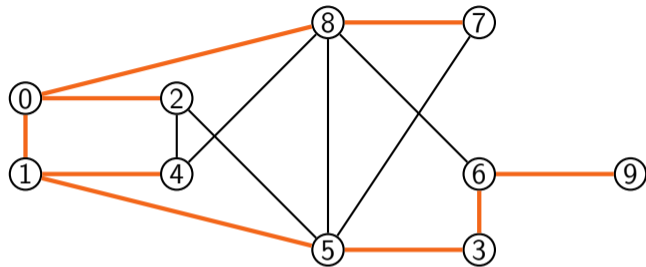
- ▶ En résumé : $n_0 = n$ et pour tout $t \geq 0$ on a $n_{t+1} \leq (1 - \frac{1}{k})n_t$
- ▶ Or $1 - x \leq e^{-x}$ pour tout x , donc $n_{t+1} \leq e^{-\frac{1}{k}} n_t$
- ▶ D'où $n_t \leq e^{-\frac{1}{k}} n_{t-1} \leq e^{-\frac{2}{k}} n_{t-2} \leq \dots \leq e^{-\frac{t}{k}} n_0 = e^{-\frac{t}{k}} n$
- ▶ Toutes les maisons sont couvertes dès que $e^{-\frac{t}{k}} n < 1$, c'ad $t > k \ln n$ ■

1. Exemple 1 : choix de cours
2. Qu'est-ce qu'un algorithme glouton ?
3. Exemple 2 : le sac-à-dos (fractionnaire)
4. Exemple spécial : approximation pour SETCOVER dans le plan
5. Dernier exemple : arbre couvrant de poids minimal

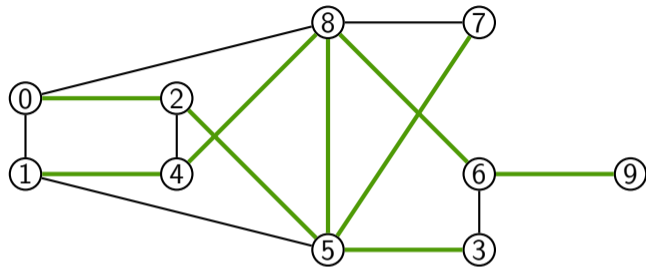
Arbre couvrant



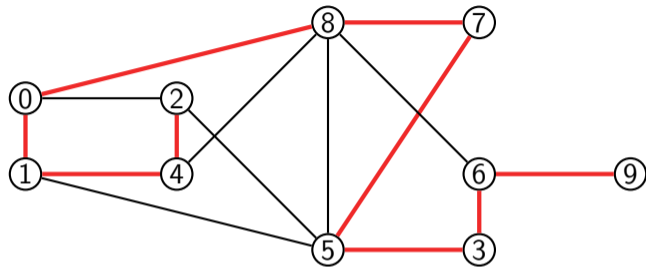
Arbre couvrant



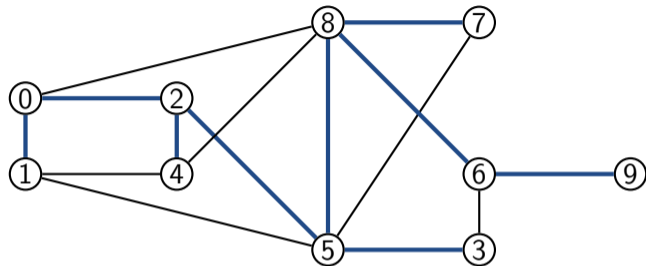
Arbre couvrant



Arbre couvrant



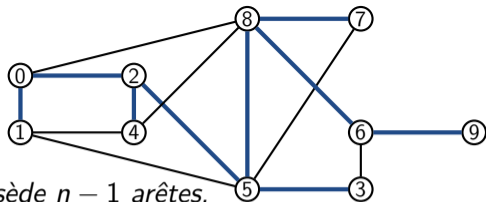
Arbre couvrant



Définition

Un **arbre couvrant** d'un graphe $G = (S, A)$ est un sous-ensemble $B \subset A$ des arêtes tel que $T = (S, B)$ est un arbre (graphe connexe et sans cycle).

Propriétés



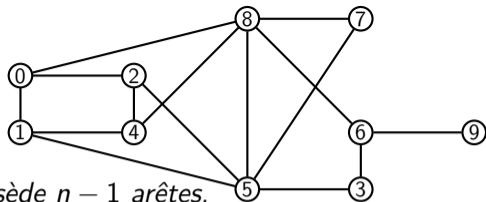
Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

Preuve par récurrence sur n (cas $n = 1$ trivial) :

- ▶ L'arbre possède au moins un sommet de degré 1 (*feuille*), sinon il a un cycle
- ▶ Si on supprime ce sommet et son arête, on a un arbre couvrant de $n - 1$ sommets, donc $n - 2$ arêtes par hypothèse de récurrence. ■

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

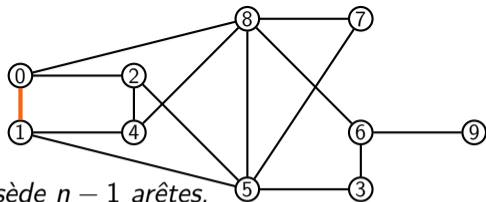
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

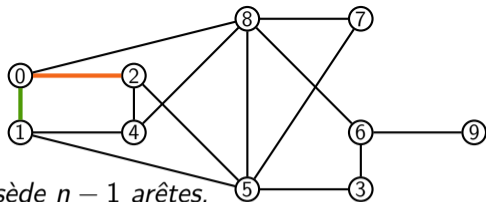
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

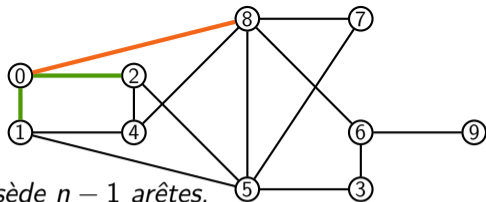
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

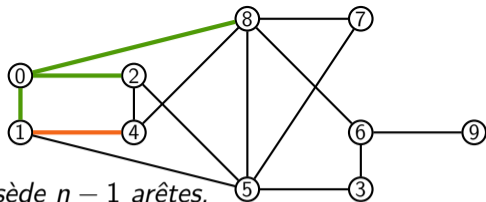
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

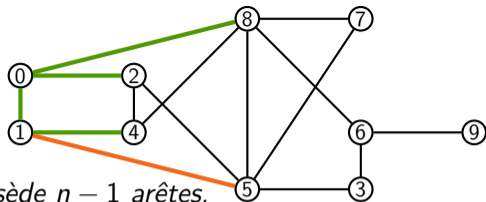
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

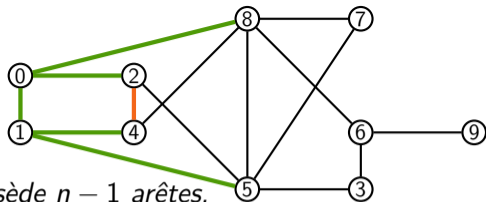
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

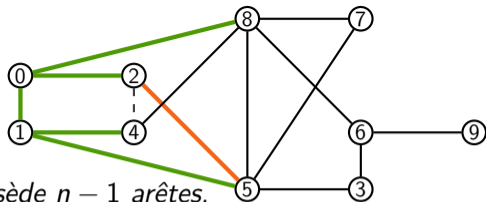
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

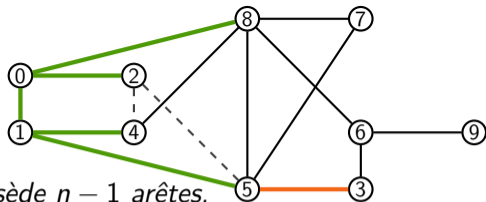
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

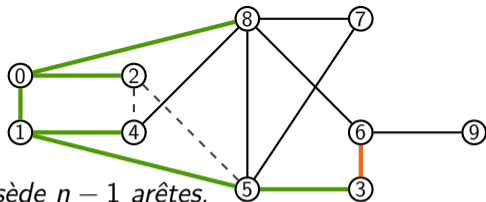
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

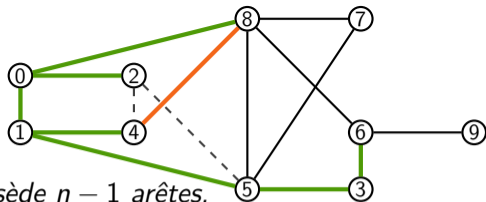
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

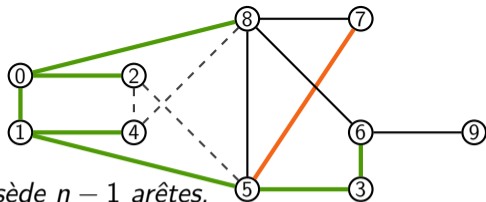
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

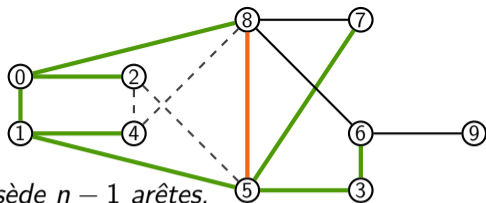
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

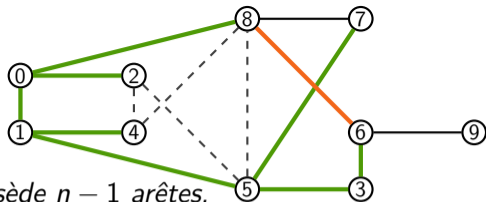
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

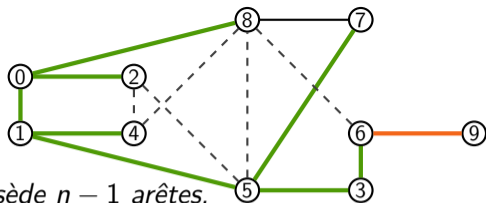
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

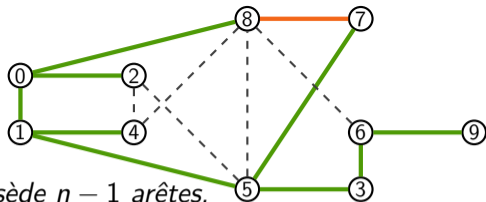
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

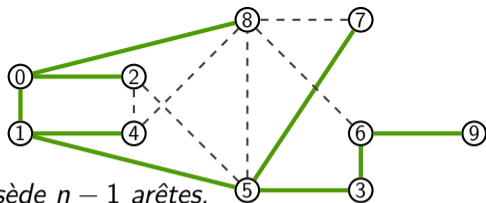
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

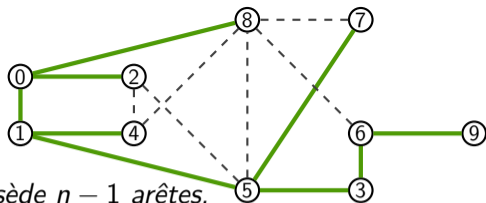
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
- ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle

Propriétés



Lemme

Un arbre couvrant d'un graphe à n sommets possède $n - 1$ arêtes.

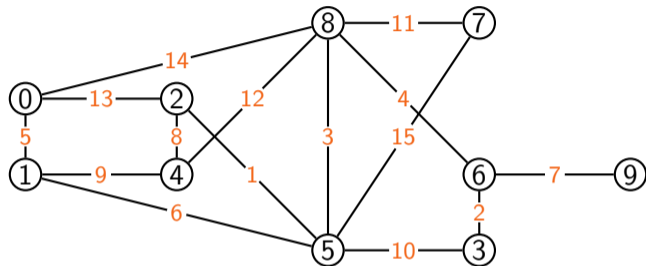
Lemme

Un graphe possède (au moins) un arbre couvrant si et seulement s'il est connexe.

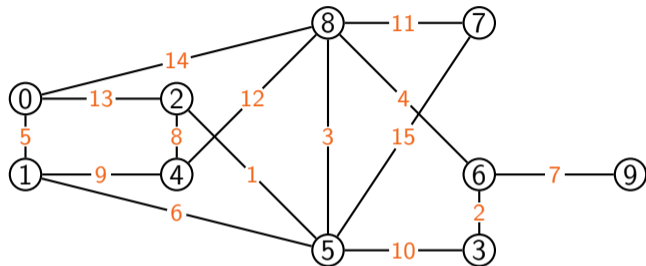
Preuve

- ▶ Si G a un arbre couvrant $T = (S, B)$, il est connexe
 - ▶ Si G est connexe : algorithme *glouton* de construction de B
 - ▶ Pour chaque arête $e \in A$ (ordre qcq), ajouter e à B si elle ne crée pas de cycle
- $\rightsquigarrow (S, B)$ est sans cycle, et connexe (sinon on pourrait ajouter des arêtes à B) ■

Arbre couvrant de poids minimum



Arbre couvrant de poids minimum

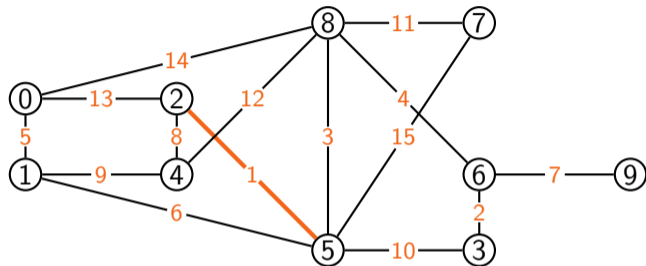


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum

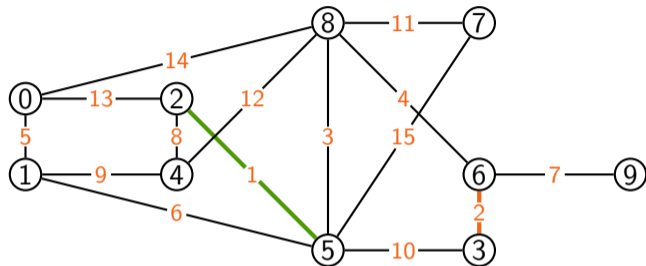


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum

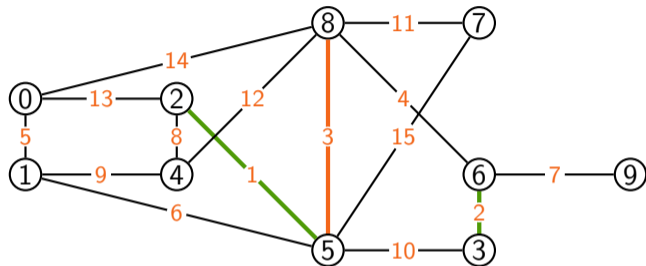


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum

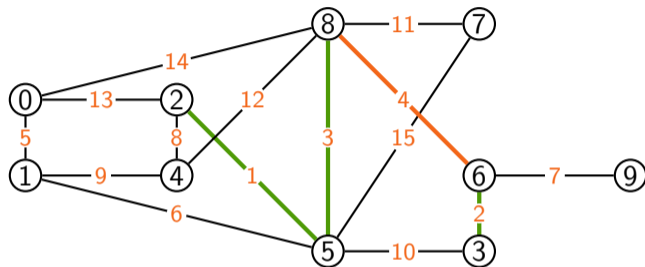


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum

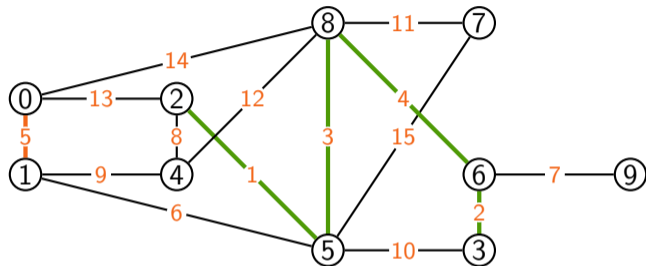


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum

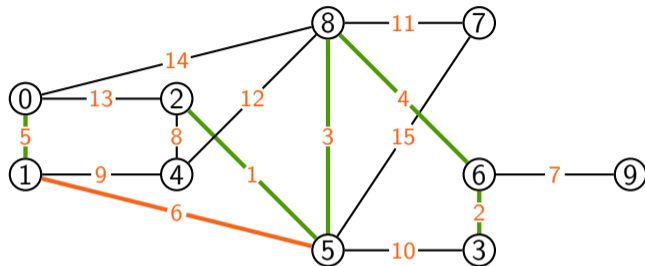


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum

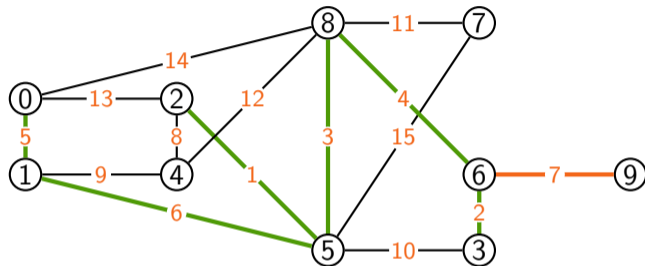


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum

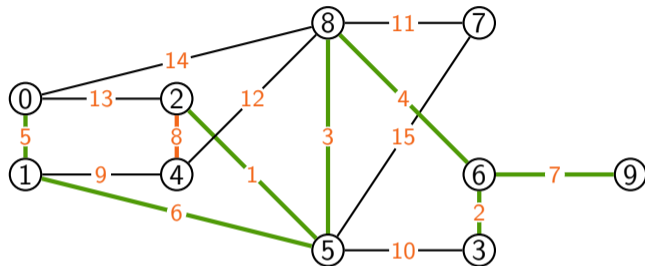


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum

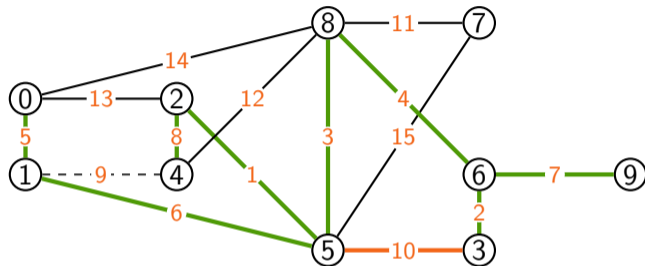


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum

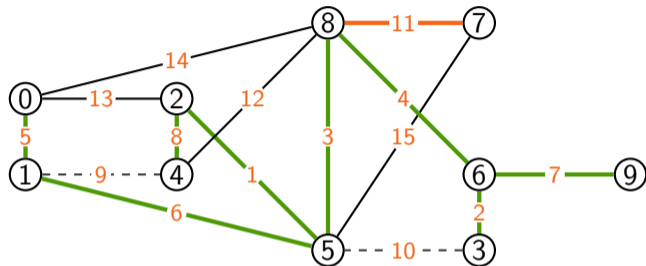


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum

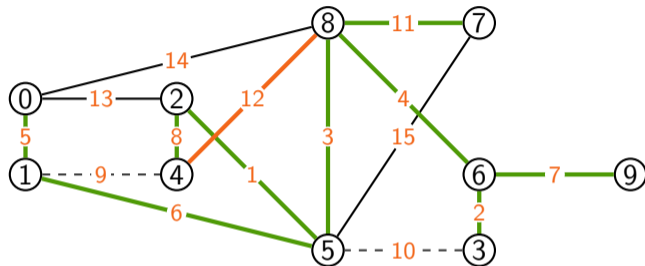


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum

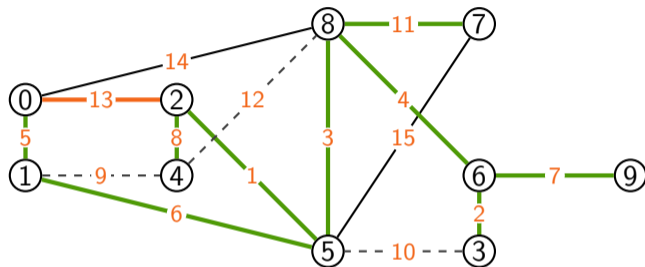


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum

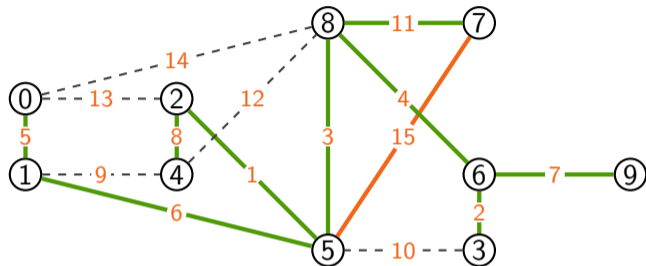


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum

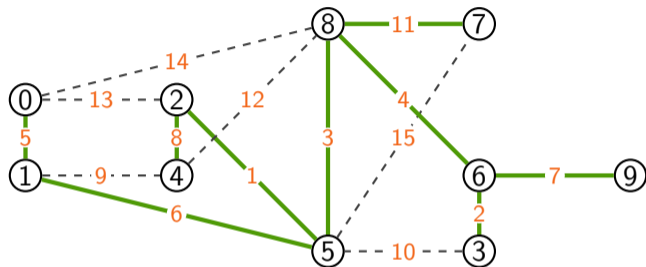


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum

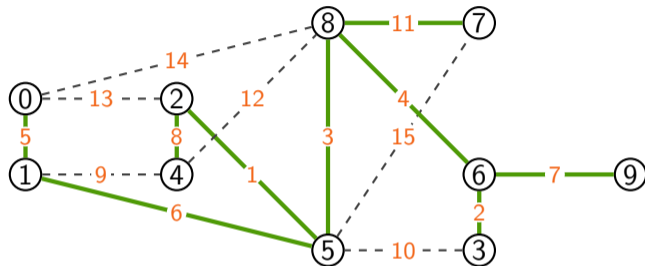


Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Arbre couvrant de poids minimum



Arbre couvrant de poids $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 11 = 47$.

Problème ACPM

Entrée Graphe **pondéré** $G = (S, A, \mathbf{p})$, avec $p : A \rightarrow \mathbb{R}_+$

Sortie Arbre couvrant $T = (S, B)$ de G de **poids minimum** : $P = \sum_{e \in B} p(e)$

Algorithme de Kruskal

Algorithme : KRUSKAL(S, A, p)

Trier les arêtes par poids croissants

$B \leftarrow \emptyset$ // aucune arête

pour chaque arête $e \in A$ dans l'ordre :

si $(S, B \cup \{e\})$ n'a pas de cycle :

 Ajouter e à B

renvoyer B

Algorithme de Kruskal

Algorithme : KRUSKAL(S, A, p)

Trier les arêtes par poids croissants

$B \leftarrow \emptyset$ // aucune arête

pour chaque arête $e \in A$ dans l'ordre :

si $(S, B \cup \{e\})$ n'a pas de cycle :

 Ajouter e à B

renvoyer B

Théorème

L'algorithme KRUSKAL renvoie un arbre couvrant de poids minimum de $G = (S, A, p)$.

Remarque : on suppose que les poids sont distincts deux-à-deux

Algorithme de Kruskal

Algorithme : KRUSKAL(S, A, p)

Trier les arêtes par poids croissants

$B \leftarrow \emptyset$ // aucune arête

pour chaque arête $e \in A$ dans l'ordre :

si $(S, B \cup \{e\})$ n'a pas de cycle :

 Ajouter e à B

renvoyer B

Théorème

L'algorithme KRUSKAL renvoie un arbre couvrant de poids minimum de $G = (S, A, p)$.

Remarque : on suppose que les poids sont distincts deux-à-deux

Lemme On ajoutant une arête à un arbre couvrant, on crée un unique cycle.

Preuve Supposons qu'en ajoutant une arête uv , deux cycles $u - v - \dots - u$ distincts sont créés. Alors il existe deux chemins distincts (et distincts de l'arête uv) entre u et v , càd un cycle qui passe par u et v . ■



Algorithme de Kruskal

Algorithme : KRUSKAL(S, A, p)

Trier les arêtes par poids croissants

$B \leftarrow \emptyset$ // aucune arête

pour chaque arête $e \in A$ dans l'ordre :

si $(S, B \cup \{e\})$ n'a pas de cycle :

 Ajouter e à B

renvoyer B

Théorème

L'algorithme KRUSKAL renvoie un arbre couvrant de poids minimum de $G = (S, A, p)$.

Remarque : on suppose que les poids sont distincts deux-à-deux

Preuve Soit C un ensemble d'arêtes tq (S, C) soit un arbre couvrant. Soit e_i l'arête la plus légère qui est dans B mais pas dans C .

- ▶ $(S, C \cup \{e_i\})$ contient un cycle $u_1 - u_2 - \dots - u_k - u_1$ (avec $e_i = u_k u_1$)
- ▶ Il existe $e_j = u_t u_{t+1}$ tq $p(e_j) > p(e_i)$: sinon e_i ne serait pas choisie par KRUSKAL
- ▶ Donc $(S, C \cup \{e_i\} \setminus \{e_j\})$ est un arbre couvrant de poids moindre :
 - ▶ Cycle supprimé en supprimant e_j
 - ▶ Connexe car il suffit de remplacer e_j par $u_{t+1} - \dots - u_k - u_1 - \dots - u_t$
 - ▶ Poids plus faible car $p(e_i) < p(e_j)$.

Questions d'implantation et de complexité

Comment tester **efficacement** à chaque étape si ajouter e à B crée un cycle ?

Questions d'implantation et de complexité

Comment tester **efficacement** à chaque étape si ajouter e à B crée un cycle ?

1. Retenir la *composante connexe* de chaque sommet
 - ▶ Numéro d'un des sommets de la composante, unique pour la composante
2. Ajout d'une arête :
 - ▶ Seulement si composantes différentes (sinon création de cycle)
 - ▶ Nécessité de faire l'*union* des deux composantes

Questions d'implantation et de complexité

Comment tester **efficacement** à chaque étape si ajouter e à B crée un cycle ?

1. Retenir la *composante connexe* de chaque sommet
 - ▶ Numéro d'un des sommets de la composante, unique pour la composante
2. Ajout d'une arête :
 - ▶ Seulement si composantes différentes (sinon création de cycle)
 - ▶ Nécessité de faire l'*union* des deux composantes

Structure de données de type **UNION-FIND**

1. Données :
 - ▶ Tableau c des composantes : $c[u] = v$ si la composante de u est v
 - ▶ Pile $P_{c[u]}$ pour chaque composante, contenant tous les sommets de la composante
2. Union de deux composantes $c[u]$ et $c[v]$:
 - ▶ Vider la pile $P_{c[u]}$ dans $P_{c[v]}$, en mettant à jour $c[w]$ pour chaque $w \in P_{c[u]}$
 - ▶ Rôles u et v inversés si $P_{c[u]}$ est plus grande que $P_{c[v]}$

Algorithme de Kruskal (complet)

Algorithme : KRUSKAL(S, A, p)

Trier les arêtes par poids croissants

$B \leftarrow \emptyset$ // aucune arête

pour chaque arête $e = uv \in A$ dans l'ordre :

si $c[u] \neq c[v]$: // e ne crée pas de cycle
 Ajouter e à B

Algorithme de Kruskal (complet)

Algorithme : KRUSKAL(S, A, p)

Trier les arêtes par poids croissants

$B \leftarrow \emptyset$ // aucune arête

pour chaque sommet $v \in S$:

$c_{[v]} \leftarrow v$
 $P_{c_{[v]}} \leftarrow \text{Pile } \{v\}$ // un seul élément
 $t_{c_{[v]}} \leftarrow 1$ // taille de la pile

pour chaque arête $e = uv \in A$ dans l'ordre :

si $c_{[u]} \neq c_{[v]}$: // e ne crée pas de cycle
 Ajouter e à B
 UNION($c_{[u]}$, $c_{[v]}$)

Algorithme de Kruskal (complet)

Algorithme : KRUSKAL(S, A, p)

Trier les arêtes par poids croissants

$B \leftarrow \emptyset$ // aucune arête

pour chaque sommet $v \in S$:

$c[v] \leftarrow v$
 $P_{c[v]} \leftarrow \text{Pile } \{v\}$ // un seul élément
 $t_{c[v]} \leftarrow 1$ // taille de la pile

pour chaque arête $e = uv \in A$ dans l'ordre :

si $c[u] \neq c[v]$: // e ne crée pas de cycle
 Ajouter e à B
 UNION($c[u], c[v]$)

Algorithme : UNION(x, y)

si $t_x > t_y$: UNION(y, x)

sinon :

$t_y \leftarrow t_y + t_x$
 tant que P_x est non vide :
 $w \leftarrow$ dépiler P_x
 $c[w] \leftarrow y$
 Empiler w sur P_y

Complexité de l'algorithme de Kruskal

Théorème

L'algorithme KRUSKAL a une complexité $O(m \log n)$, où n est le nombre de sommets et m le nombre d'arêtes.

Complexité de l'algorithme de Kruskal

Théorème

L'algorithme KRUSKAL a une complexité $O(m \log n)$, où n est le nombre de sommets et m le nombre d'arêtes.

Lemme

Pour tout sommet u , le nombre total de mises à jour de la case $c_{[u]}$ au cours de l'algorithme est $\leq \log n$.

Preuve La mise à jour a lieu dans UNION, avec $t_{c_{[v]}} \geq t_{c_{[u]}}$. Donc après mise à jour, $t_{c_{[u]}}$ a au moins doublé. Comme $t_{c_{[u]}} \leq n$, le nombre de mises à jour est $\leq \log n$. ■

Complexité de l'algorithme de Kruskal

Théorème

L'algorithme KRUSKAL a une complexité $O(m \log n)$, où n est le nombre de sommets et m le nombre d'arêtes.

Lemme

Pour tout sommet u , le nombre total de mises à jour de la case $c_{[u]}$ au cours de l'algorithme est $\leq \log n$.

Preuve du théorème

- ▶ Tri des arêtes : $O(m \log m)$, or $m \leq n^2$ donc $\log m \leq 2 \log n$ $\rightsquigarrow O(m \log n)$
- ▶ Parcours de toutes les arêtes une fois $\rightsquigarrow O(m)$
- ▶ Pour chaque sommet, $\leq \log n$ mises à jour de $c_{[u]}$ $\rightsquigarrow O(n \log n)$

Conclusion

Bilan

Pourquoi des algorithmes gloutons ?

- ▶ Algorithmes souvent simples et rapides...
- ▶ ... parfois optimaux
- ▶ ... parfois avec de bonnes propriétés
- ▶ ... parfois qui marchent en pratique
- ▶ ... parfois parfaitement inutiles !

Bilan

Pourquoi des algorithmes gloutons ?

- ▶ Algorithmes souvent simples et rapides...
- ▶ ... parfois optimaux
- ▶ ... parfois avec de bonnes propriétés
- ▶ ... parfois qui marchent en pratique
- ▶ ... parfois parfaitement inutiles !

Comment les utiliser ?

1. Chercher un choix glouton
2. Démontrer que c'est un bon choix (en **théorie** ou pratique)
3. Étudier la complexité obtenue

Aussi vu dans ce cours

- ▶ Algorithme d'**approximation** (SETCOVER) : ne résout pas le problème exactement, mais fournit une **solution approchée**
- ▶ Complexité **amortie** (UNION) : technique d'analyse de complexité qui remplace le pire cas par la **somme** (ou **moyenne**) des coûts de plusieurs exécutions

Étudiés dans le cours d'algorithmique de L3

Aussi vu dans ce cours

- ▶ Algorithme d'**approximation** (SETCOVER) : ne résout pas le problème exactement, mais fournit une **solution approchée**
- ▶ Complexité **amortie** (UNION) : technique d'analyse de complexité qui remplace le pire cas par la **somme** (ou **moyenne**) des coûts de plusieurs exécutions

Étudiés dans le cours d'algorithmique de L3

- ▶ Structure de donnée **UNION-FIND** pour stocker des **ensembles disjoints** :
 - ▶ Trois opérations : MAKESET, UNION et FIND
 - ▶ Plusieurs implantations possibles :
 - ▶ Tableau + Piles : complexité amortie $O(n \log n)$
 - ▶ Meilleure implantation possible : $O(n\alpha(n))$

inverse de la fonction d'Ackermann : $\alpha(n) \leq 4$ pour $n \leq 2^{2^{2^{65536}}} - 3$

Voir <https://fr.wikipedia.org/wiki/Union-find>