# Chap. 5 – Algorithmes d'approximation

## HAI503I – Algorithmique 4

Bruno Grenet

Université de Montpellier – Faculté des Sciences

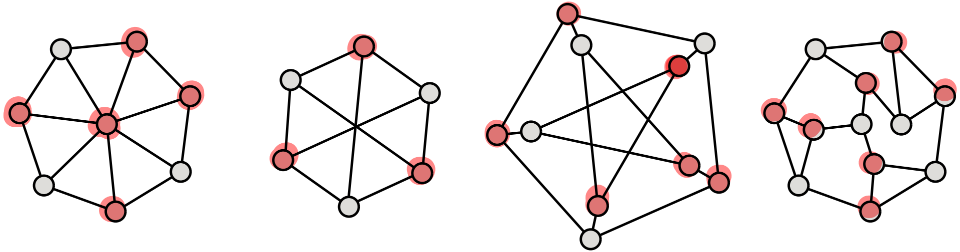
# Définition du problème

**Couverture** **Vertex Cover**

*Entrée :* Un graphe $G = (S, A)$

*Sortie :* Un sous-ensemble $C \subset S$ de sommets, qui *couvre* toutes les arêtes : pour tout $\{u, v\} \in A$, $u \in C$ ou $v \in C$

*Objectif :* Trouver $C$ le plus petit possible



# Solution exacte

## Algorithme par recherche exhaustive

- Tester tous les sous-ensembles possibles, par taille croissante
- Complexité : $O(2^n n^2)$ où $n$ est le nombre de sommets
  - $O(2^k n^2)$ si la couverture minimale est de taille $k$

## *A priori* pas d'algorithme polynomial

- COUVERTURE fait partie des problèmes NP-complets *HAI602I*
- Meilleurs algorithmes connus en $O(2^k n)$, voire $O(1,2738^k + kn)$ *difficile !*

Que peut-on faire en *temps polynomial* ?

# Un algorithme d'approximation

On ne cherche plus la couverture la plus petite possible mais *une couverture assez petite*

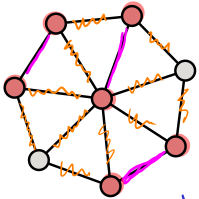
COUVAPPROX($G$) :

1. $C \leftarrow \emptyset$
2. Tant que $G$ est non vide :
3. Choisir une arête $\{u, v\}$ dans $G$
4. Ajouter $u$ **et** $v$ dans $C$
5. Supprimer $u$ et $v$ (et les arêtes incidentes) de $G$
6. Renvoyer $C$

## Complexité

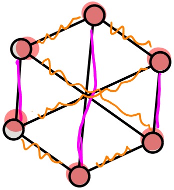
L'algorithme COUVAPPROX a une complexité $O(n^2)$

# Exemples



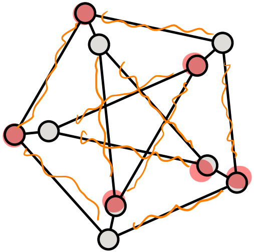
Solution de taille 6

optimal : 5



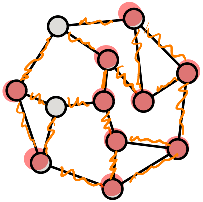
6

3



6

6



10

7

# Garantie de l'algorithme d'approximation

## Théorème

Soit OPT la taille d'une couverture de taille minimale de $G$, et $C \leftarrow \textsc{CouvApprox}(G)$. Alors

$$|C| \leq 2\,\text{OPT}$$

B : ensemble des arêtes choisies par l'algo pour créer C

Dans une solution optimale, toutes les arêtes de B doivent être couvertes.

Les arêtes de B n'ont pas de sommet en commun,

donc $\text{OPT} \geq |B|$, $|C| = 2|B| \Rightarrow |C| = 2|B| \leq 2\,\text{OPT}$ □

# Définition du problème

**Somme Partielle** — *Subset Sum*

*Entrée :* Un ensemble $E$ d'entiers strictement positifs, un entier cible $T$

*Sortie :* Un sous-ensemble $S \subset E$ dont la somme est $\leq T$

*Objectif :* Trouver $S$ de somme la plus grande possible (la plus proche possible de $T$)

## Notations

- Pour $S \subset E$, $\Sigma S = \sum_{x \in S} x$
- Objectif : trouver $S$ tel que $\Sigma S \leq T$ est maximale
- OPT : valeur de la solution maximale (la meilleure possible)

$E = \{1, 7, 28, 3, 9, 41, 11, 8\}$

$T = 16 \rightsquigarrow \{7, 9\}$

$T = 30 \rightsquigarrow \{7, 3, 9, 11\}$

$T = 33 \rightsquigarrow \{28, 3, 1\} \rightsquigarrow 32$

# Solution exacte

## Recherche exhaustive et *backtrack* — TD2 Ex. 1

- Parcours de tous les sous-ensembles $S \subset E$
  - Complexité $O(n2^n)$ où $n = |E|$
- *Backtrack* si entiers tous positifs
  - Complexité $O(2^n)$

## *A priori* pas d'algorithme polynomial

- Somme Partielle fait partie des problèmes NP-complets — *HAI602I*
- Meilleur algorithme connu en $O(2^{n/2}) = O(1,414^n)$ — *difficile !*

Que peut-on faire en *temps polynomial* ?

# Un algorithme d'approximation

SommePartApprox($E$, $T$) :

1. Trier $E$ par ordre décroissant
2. $S \leftarrow \emptyset$
3. Pour $i = 0$ à $|E| - 1$ :
4. Si $T \geq E_{[i]}$ :
5. Ajouter $E_{[i]}$ à $S$
6. $T \leftarrow T - E_{[i]}$
7. Renvoyer $S$

$E = \{1, 7, 28, 3, 9, 41, 11, 8\}$

$\hookrightarrow \underline{E} = \{41, 28, 11, 9, 8, 7, 3, 1\}$

$T = 30$ : $\{28, 1\}$ 29 au lieu de 30

$T = 16$ : $\{11, 3, 1\}$ 15 — 16

$T = 33$ : $\{28, 3, 1\}$ 32 — 32

## Complexité

L'algorithme SommePartApprox a une complexité $O(n \log n)$

# Garantie de l'algorithme d'approximation

## Théorème

Soit $S \leftarrow \textsc{SommePartApprox}(E, T)$ et $\textsc{opt}$ la valeur de la solution optimale. Alors

$$\Sigma S \geq \frac{1}{2}\,\textsc{opt}$$

- On élimine de $E$ tous les éléments $> T$.
- Si $S = E$, la solution est optimale.
- Sinon : soit $E[i]$ le 1er élément non choisi par l'algo
  + $E[0] \geq E[1] \geq \dots \geq E[i] \rightsquigarrow E[i] \leq \sum_{j=0}^{i-1} E[j]$
  + $E[i] + \sum_{j=0}^{i-1} E[j] > T \rightsquigarrow$ sinon l'algo sélectionne $E[i]$

$\Rightarrow \textsc{opt} \leq T < E[i] + \sum_{j=0}^{i-1} E[j] \leq 2 \sum_{j=0}^{i-1} E[j] \leq 2\,\Sigma S \Rightarrow \Sigma S > \frac{1}{2}\,\textsc{opt}$ ∎

# Problèmes d'optimisation

## Cadre général

- *Problème de maximisation* : sur une entrée, trouver une solution qui *maximise* une certaine fonction
- *Problème de minimisation* : sur une entrée, trouver une solution qui *minimise* une certaine fonction

## Exemples

Max : SommePartielle, Sac-à-dos, ChoixCours

Min : Couverture, Voyageur de commerce

# Formalisation

## Définition

- Ingrédients :
  - Ensemble $I$ des instances (entrées)
  - Pour chaque $x \in I$, l'ensemble $S$ des solutions *acceptables* (sorties possibles)
  - Une fonction de coût $c : S \to \mathbb{R}$ (valeur d'une solution)
- Objectifs :
  - maximisation : trouver $s \in S$ telle que $c(s)$ soit maximale $\quad \forall s' \in S, c(s') \leq c(s)$
  - minimisation : trouver $s \in S$ telle que $c(s)$ soit minimale $\quad \forall s' \in S, c(s') \geq c(s)$
- Valeur optimale : on note OPT la valeur de la solution optimale
  - maximisation : OPT $= \max_{s \in S} c(s)$
  - minimisation : OPT $= \min_{s \in S} c(s)$

COUV : $c(s)$ = #sommets
S.P. : $c(s)$ = somme des élts de s

# Résolution exacte

Comment résoudre un problème d'optimisation de manière exacte ?

## Recherche exhaustive et *backtrack* — Chap. 2

- Parcours (intelligent) de toutes les solutions, en gardant la meilleure
- Fonctionne toujours ; complexité (en général) exponentielle

## Algorithmes gloutons — Cours L2

- Construction d'une solution en optimisant localement à chaque étape
- Fonctionne parfois… ; complexité *souvent assez bonne*

## Programmation dynamique — Cours L2

- Décomposition du problème en sous-problèmes, et résolution par tailles croissantes
- Fonctionne souvent ; complexité (en général) exponentielle mais meilleure qu'en recherche exhaustive

# Algorithmes d'approximation

## Algorithmes de compromis

- Algorithmes efficaces $\rightarrow$ complexité polynomiale, voire linéaire
- Algorithmes non exacts $\rightarrow$ solution de valeur proche de l'optimal

## Définition

Un algorithme d'$\alpha$-approximation est un algorithme qui *pour tout entrée x* renvoie une solution $s \in S$ telle que

- maximisation : $\alpha \cdot \textsc{opt} \leq c(s) \leq \textsc{opt}$ $\qquad 0 < \alpha < 1$
- minimisation : $\textsc{opt} \leq c(s) \leq \alpha \cdot \textsc{opt}$ $\qquad \alpha > 1$

Le réel $\alpha$ est appelé facteur d'approximation de l'algorithme.

## Exemples

CouvertureApprox : 2-approximation

S.P. Approx : 1/2-approximation.

# Comment concevoir des algorithmes d'approximation ?

Très vaste sujet, dépasse (très) largement le cadre de ce cours !

## Une technique fructueuse : algorithmes glouton

- Approche gloutonne souvent rapide → efficacité
- Pas toujours la meilleure solution → non exact
- Solution souvent *pas trop mauvaise* → compromis

## Remarque

- On ne cherche pas une solution *optimale*, mais *pas trop mauvaise*
- Parfois intéressant de faire des choix *un peu bêtes* mais pas loin de l'optimal
  - Exemple de COUVERTURE : ajouter les 2 extrémités de l'arête choisie

## Objectif du cours

- Concevoir et analyser des algorithmes d'approximation *simples*

# Comment analyser un algorithme d'approximation ?

## Objectif

Montrer que pour tout entrée, l'algorithme renvoie une solution $s$ vérifiant

- $c(s) \geq \alpha \cdot \textsc{opt}$ (si maximisation)
- $c(s) \leq \alpha \cdot \textsc{opt}$ (si minimisation)

## Deux bornes à trouver (cas max.) (cas min.)

- Trouver une borne $c_1$ telle que $c(s) \geq c_1$ — $c(s) \leq c_1$
- Trouver une borne $c_2$ telle que $\textsc{opt} \leq c_2$ — $\textsc{opt} \geq c_2$

$\rightarrow$ On en déduit que $\alpha \geq c_1/c_2$ — $\alpha \leq c_1/c_2$

> Pour trouver le facteur d'approximation, il faut aussi une borne sur la valeur optimale !

# Définition du problème

## Informellement

- Ensemble de *n tâches* à exécuter, chacune ayant une *durée*
- À disposition : *m processeurs*
- Objectif : répartir les tâches sur les processeurs, pour *minimiser* le temps total de calcul

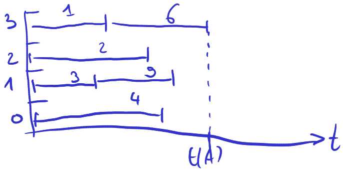
### Équilibrage — *Load Balancing*

*Entrée :* Tableau $D$ d'entiers positifs (*durées*) et entier $m$

*Sortie :* Tableau $A$ : affectation de chaque tâche à un processeur

- tâche $i$ affectée au processeur $j$ : $A_{[i]} = j$

*Objectif :* Minimiser le temps total, calculé comme : $t(A) = \max_{0 \le j \le m-1} \left( \sum_{i:A_{[i]}=j} D_{[i]} \right)$




NP-difficile

# Algorithme glouton *à la volée*

Scénario : les tâches arrivent les unes après les autres, on doit les traiter dans l'ordre

- Traduction : on ne peut pas trier le tableau $D$
- Idée de l'algo. : on affecte la prochaine tâche au processeur le moins occupé

ÉQUILIBRAGEGLOUTON($D$, $m$) :

1. $T \leftarrow$ tableau de taille $m$, intialisé à 0 (*temps total par processeur*)
2. Pour $i = 0$ à $n-1$ :
3. $\quad j \leftarrow$ indice du minimum de $T$
4. $\quad A_{[i]} \leftarrow j$
5. $\quad T_{[j]} \leftarrow T_{[j]} + D_{[i]}$
6. Renvoyer $A$

## Complexité

L'algorithme ÉQUILIBRAGEGLOUTON a une complexité $O(nm)$ (ou $O(n \log m)$ avec un tas)

# Garantie de l'algorithme glouton

## Théorème

L'algorithme ÉquilibrageGlouton est un algorithme de 2-approximation pour le problème Équilibrage

- $OPT \geq \max_i D[i]$ car il faut affecter le max à un proc.
- $OPT \geq \frac{1}{m} \sum_{i=0}^{n-1} D[i]$ car $\sum_{i=0}^{n-1} D[i] = \sum_{j=0}^{m-1} T^{OPT}[j] \leq m \cdot OPT$

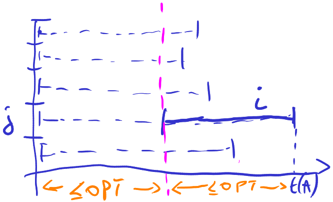
Soit A l'affectation calculée par l'algo et j tq $T[j] = \max_k T[k]$

Soit i la dernière tâche affectée à j.

$\Rightarrow T[j] - D[i] \leq T[k]$ pour tout k

$$T[j] - D[i] \leq \frac{1}{m} \sum_{k=0}^{m-1} T[k] = \frac{1}{m} \sum_{i=0}^{n-1} D[i] \leq OPT$$

$$\Rightarrow T[j] \leq OPT + D[i] \leq 2\,OPT$$



# Algorithme glouton avec tri

Nouveau scénario : on connaît toutes les tâches à l'avance $\rightarrow$ fait-on mieux ?

- On peut trier les tâches par durée décroissante et affecter les tâches les plus longues en premier

## Algorithme et complexité

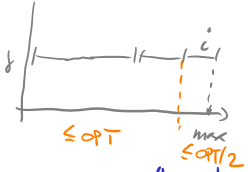
- Même algorithme ÉquilibrageGlouton, avec tri de $D$ initialement
- Complexité : $O(n \log n)$ pour le tri, puis pareil
  - $O(n(m + \log n))$ avec recherche *naïve* de minimum
  - $O(n(\log n + \log m))$ avec un tas $\rightarrow O(n \log n)$ car $n \geq m$

# Garanties de l'algorithme glouton *avec tri*

## Théorème

Si $D$ est trié par ordre décroissant, le facteur d'approximation $\alpha$ de ÉQUILIBRAGEGLOUTON est $\leq 3/2$

- $j$ le numéro du processeur le plus chargé ⇝ on veut borner $T_{[j]}$
  - × Si $j$ n'a qu'une tâche, alors $T_{[j]} = OPT$
  - × Sinon : soit $i$ la dernière tâche affectée au proc. $j$




(1) $T_{[j]} - D_{[i]} \leq OPT$ (cf preuve précédente)

(2) $i \geq m+1$ car les $m$ premières tâches vont à des processeurs différents.

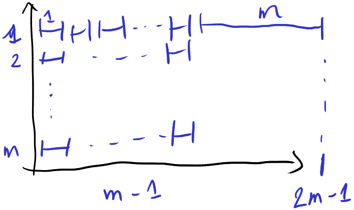
(3) ⇒ dans une solution optimale, il existe forcément deux tâches $k, \ell$ affectées à un même processeur ⇒ $D[k] + D[\ell] \leq OPT$

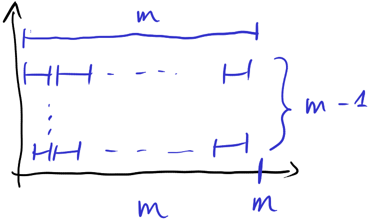
$$\Rightarrow \quad D_{[i]} \leq D_{[m+1]} \leq \min\left(D_{[k]}, D_{[\ell]}\right) \leq \frac{OPT}{2} \quad \Rightarrow \quad T_{[j]} \leq D_{[i]} + OPT \leq \frac{3}{2} OPT.$$

# Bilan sur l'équilibrage de charge

## Cas non trié

- L'algorithme glouton est une 2-approximation
- Un peu mieux : $(2 - 1/m)$-approximation
- Facteur d'approximation atteint




## Cas trié

- L'algorithme glouton fournit une $3/2$-approximation
- On peut dire mieux : $(4/3 - 1/m)$-approximation

*meilleure borne sur* OPT

## Encore mieux ?

- Pour tout $\varepsilon > 0$, il existe un algorithme qui est une $(1 + \varepsilon)$-approximation

# Rappel : le problème SAT

## Le problème SAT

### Définition

Entrée : une formule logique $\varphi$ à $n$ variables booléennes, sous *forme normale conjonctive* (CNF)

Sortie : une affectation des variables qui satisfasse $\varphi$ ; « insatisfiable » sinon

### Formule logique CNF : *conjonction de disjonction de littéraux*

- Littéraux : $x_1, \neg x_1, \ldots, x_n, \neg x_n$
- Disjonction : $C = x_1 \vee \neg x_3 \vee \neg x_4$ (clause)
- Conjonction : $C_1 \wedge C_2 \wedge \cdots \wedge C_k$

$$\varphi(x_1, x_2, x_3) = (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge \neg x_2$$

### Affectation satisfaisante ou non

- $(x_1, x_2, x_3) = (\text{FAUX}, \text{FAUX}, \text{FAUX})$ satisfait $\varphi$
- $(x_1, x_2, x_3) = (\text{VRAI}, \text{FAUX}, \text{VRAI})$ ne satisfait pas $\varphi$

# Le problème MaxSat

## Définition

**Entrée :** une formule logique $\varphi$ à $n$ variables booléennes, sous *forme normale conjonctive* (CNF)

**Sortie :** une affectation des variables

**Objectif :** maximiser le nombre de clauses satisfaites

## Exemple

$$\varphi(x_1, x_2, x_3) = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_3) \wedge (x_2 \vee x_3)$$

- (VRAI, VRAI, VRAI) $\rightarrow$ clause n° 3 non satisfaite
- (VRAI, VRAI, FAUX) $\rightarrow$ clause n° 2 non satisfaite
- (VRAI, FAUX, FAUX) $\rightarrow$ clauses n$^{\text{os}}$ 2 et 4 non satisfaites
- …

$\rightarrow$ Au max, 3 clauses sur 4 sont satisfiables

## Algorithmes exacts

Modification des algorithmes exhaustifs et *backtrack* pour trouver la *meilleure* solution

# L'algorithme RandSat

**RandSat($\varphi$) :**

*Affectation aléatoire des variables :*

1. Pour $i = 0$ à $n - 1$ :
2. $\quad b \leftarrow$ bit aléatoire
3. $\quad$ Si $b = 1$ : $A_{[i]} \leftarrow$ VRAI
4. $\quad$ Sinon : $A_{[i]} \leftarrow$ FAUX
5. Renvoyer $A$

## Complexité

L'algorithme RandSat a une complexité $O(n)$.

# Garantie de RandSat

## Lemme

Soit $\varphi$ une formule, et OPT le nombre maximal de clauses simultanément satisfiable. Alors l'espérance du nombre de clauses satisfaites par RandSat($\varphi$) est $\geq \frac{1}{2}$ OPT.

- OPT $\leq m$ où $m$ est le nb de clauses.
- Soit $C$ une clause de taille $k$ ($k$ littéraux)

$$\Pr[C \text{ satisfaite}] = 1 - 1/2^k \geqslant 1/2$$

- $\mathbb{E}[\#\text{clauses satisfaite}] = \sum_C \Pr[C \text{ satisfaite}] \geqslant m/2$.

$$\Rightarrow \mathbb{E}[\#\text{clauses satisfaites}] \geqslant \frac{1}{2} \text{ OPT}$$

# Garantie de RandSat

## Lemme

Soit $\varphi$ une formule, et OPT le nombre maximal de clauses simultanément satisfiable. Alors l'espérance du nombre de clauses satisfaites par RandSat($\varphi$) est $\geq \frac{1}{2}$ OPT.

## Remarque

Si l'espérance du nombre de clauses satisfaites est $\geq \frac{1}{2}$ OPT, alors il existe *au moins une affectation* satisfaisant $\geq \frac{1}{2}$ OPT clauses.

# Bilan sur RandSat

## Un algorithme d'approximation

- RandSat est un algorithme de $\frac{1}{2}$-approximation pour MaxSat, de type *Monte Carlo*
- Il existe une version *Las Vegas* $\rightarrow$ tirer des affectations tant qu'elles ne satisfont pas suffisamment de clauses

## Mieux ?

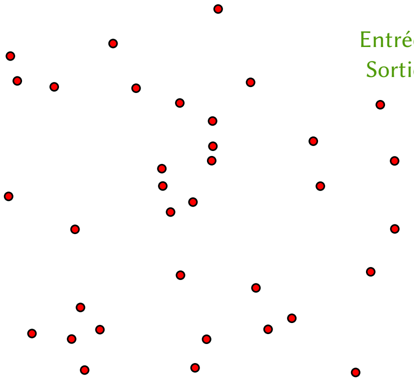
- Si la plus petite clause est de taille $k \rightarrow (1 - 1/2^k)$-approximation
- Il existe un algorithme de $\frac{3}{4}$-approximation, quelque soit $k$
- On peut *dérandomiser* RandSat

## Remarque : méthode probabiliste

Pour toute formule $\varphi$, il existe une affectation qui satisfait au moins la moitié des clauses

# Rappel : le voyageur de commerce

## Le voyageur de commerce

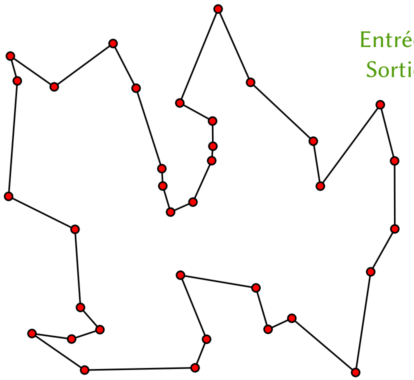


Entrée : Un ensemble de points du plan
Sortie : Un ordre de parcours des points $u_0 \to u_1 \to \cdots \to u_{n-1} \to u_0$ qui minimise la distance totale

Source : Wikipédia

# Rappel : le voyageur de commerce

## Le voyageur de commerce



**Entrée :** Un ensemble de points du plan

**Sortie :** Un ordre de parcours des points $u_0 \to u_1 \to \cdots \to u_{n-1} \to u_0$ qui minimise la distance totale

Source : Wikipédia

# Formalisation du problème

## Définition

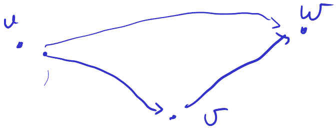
**Entrée :** Graphe $G = (S, A)$ avec une longueur $\ell(u, v)$ pour chaque arête *vérifiant l'inégalité triangulaire :* $\ell(u, w) \leq \ell(u, v) + \ell(v, w)$ *pour tous* $u, v, w$

**Sortie :** Une numérotation $u_0, \ldots, u_{n-1}$ des sommets

**Objectif :** Minimiser la longueur totale $\sum_{k=0}^{n-1} \ell(u_i, u_{i+1}) + \ell(u_{n-1}, u_0)$

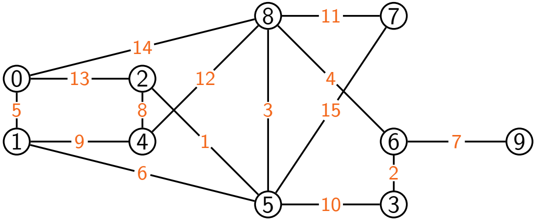
## Algorithmes exacts

- Recherche exhaustive : $O(n \times n!)$ — *Chap. 2*
- Programmation dynamique : $O(n^2 2^n)$ — *HLIN401 – Chap. 5*



# Arbre couvrant de poids minimum (rappel)

## Arbre couvrant de poids minimum



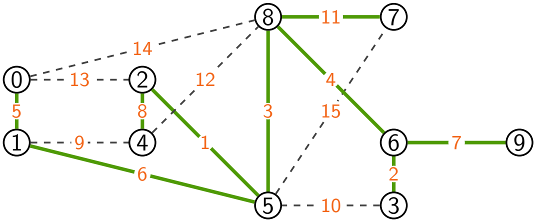

**Problème ACPM**

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

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

# Arbre couvrant de poids minimum (rappel)

## 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 $p : A \to \mathbb{R}_+$

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

# Arbre couvrant de poids minimum (rappel)

## Algorithme de Kruskal

**Algorithme :** $\textsc{Kruskal}(S, A, p)$

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

# Un algorithme de 2-approximation

VoyageurDeCommerce$_2(G)$ :

1. $\mathcal{A} \leftarrow$ arbre couvrant de poids minimum de $G$ — Algo. de Kruskal
2. $\mathcal{P} \leftarrow$ parcours en profondeur de $\mathcal{A}$
3. $(u_0, \ldots, u_{n-1}) \leftarrow$ sommets de $G$, dans l'ordre de première apparition dans $\mathcal{P}$
4. Renvoyer $(u_0, \ldots, u_{n-1})$

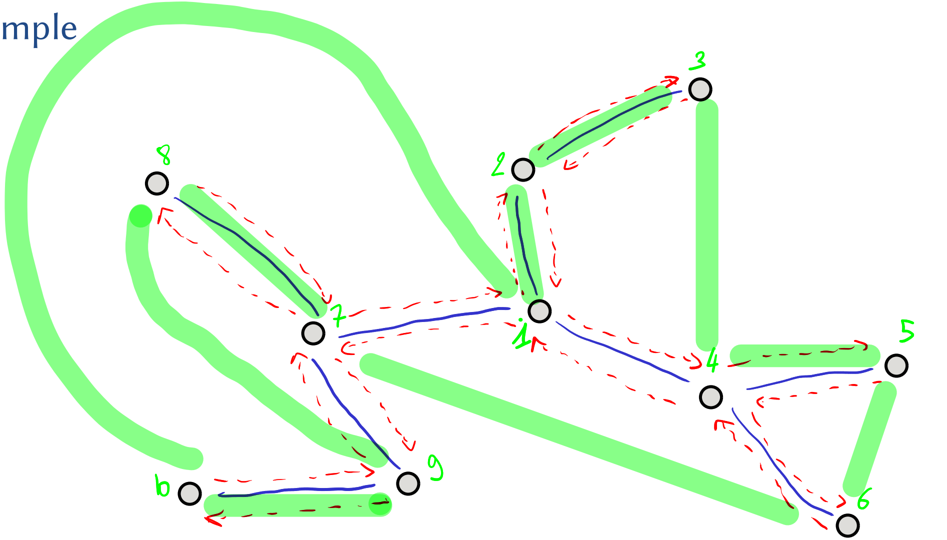
## Complexité

$G$ : graphe à $n$ sommets et $m$ arêtes

- Kruskal : $O(m \log n)$
- Parcours : $O(n)$

> L'algorithme VoyageurDeCommerce$_2$ s'exécute en temps $O(m \log n)$

# Exemple



# Garantie de l'algorithme VoyageurDeCommerce$_2$

**Théorème**

L'algorithme VoyageurDeCommerce$_2$ est un algorithme de 2-approximation

- Poids Arbre $\leq$ OPT :

  Si on prend un cycle qui par tous les sommets et qu'on enlève une arête $\leadsto$ on obtient un arbre couvrant $A'$.

  On a : poids$(A') \leq$ OPT car on a enlevé une arête

  poids$(A') \geq$ poids de l'arbre couvrant de poids min

- Si on parcourt tout l'ACPM (parcours rouge), la lgr totale est $2 \times$ poids(ACPM)

- le parcours obtenu avec les "raccourcis" (vert) est de lgr $\leq 2$ poids(ACPM) (inégalité triangulaire)

# Couplage parfait de poids minimum

## Remarque

Soit $I \subset S$ l'ensemble des sommets dont le degré dans $\mathcal{A}$ est *impair*. Alors $|I|$ est *pair*

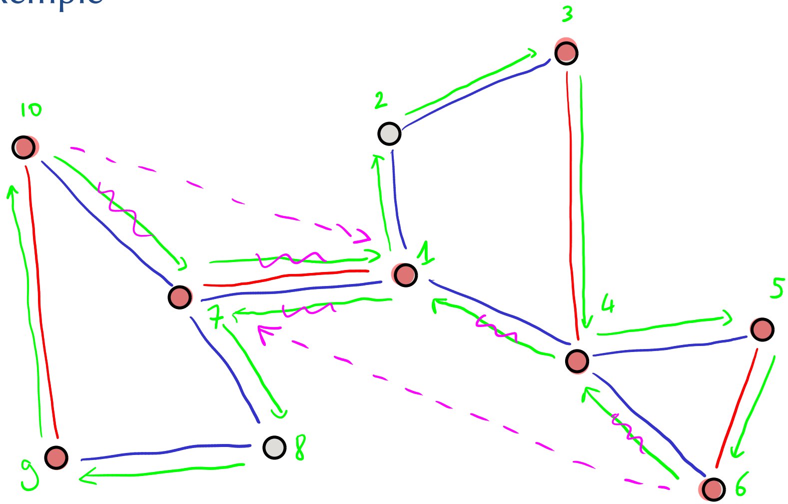
## Définition

- Un *couplage parfait* de $I$ est un ensemble $\mathcal{C}$ d'arêtes telles que tout sommet de $I$ appartient exactement à une arête de $\mathcal{C}$
- Le *poids* de $\mathcal{C}$ est la somme des longueurs des arêtes de $\mathcal{C}$

## Théorème (admis)

On peut calculer en temps polynomial un couplage parfait de poids minimum de $I$

# Retour à l'exemple



# L'algorithme de Christofides

CHRISTOFIDES($G$) :

1. $\mathcal{A} \leftarrow$ arbre couvrant de poids minimum de $G$ — Algo. de KRUSKAL
2. $I \leftarrow$ sommets de degré impair de $\mathcal{A}$
3. $\mathcal{C} \leftarrow$ couplage parfait de poids minimum de $I$
4. $\mathcal{M} \leftarrow$ *multigraphe* $\mathcal{A} \cup \mathcal{C}$
5. $\mathcal{P} \leftarrow$ parcours *eulérien* de $\mathcal{M}$ — une fois et une seule par chaque arête
6. $(u_0, \ldots, u_{n-1}) \leftarrow$ sommets de $G$, dans l'ordre de première apparition dans $\mathcal{P}$
7. Renvoyer $(u_0, \ldots, u_{n-1})$

## Complexité

- L'algorithme de Christofides a une complexité *polynomiale*

## Correction

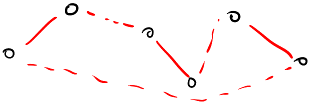
- Dans $\mathcal{M}$, tous les sommets ont degré pair $\rightarrow$ il existe un chemin eulérien

# Garantie de l'algorithme de Christofides

## Théorème

L'algorithme Christofides est une $3/2$-approximation pour le problème du voyageur de commerce

- Parcours eulérien : lgr = poids($A$) + poids($C$)
- poids($A$) $\leq$ OPT (cf preuve précédente)
- Tour optimal passant par tous les sommets de $I$ : lgr $\leq$ OPT
- Mais lgr —— : $\geq 2$ poids($C$)



$\Rightarrow$ poids($C$) $\leq \frac{1}{2}$ OPT

poids(parcours renvoyé) $\leq$ poids(parcours eulérien) $\leq \frac{3}{2}$ OPT

# Bilan sur le voyageur de commerce

## Algorithmes d'approximation

Avec l'inégalité triangulaire :

- algorithme *relativement simple* de 2-approximation
- algorithme plus complexe de $^3/_2$-approximation *Christofides (1976)*
- algorithme très complexe de $(^3/_2 - ^1/_{10^{36}})$-approximation *Karlin, Klein, Gharan (**2021**)*

## Algorithmes exacts

- Programmation dynamique en $O(n^2 2^n) \rightarrow$ nécessite l'inégalité triangulaire
- Algorithme exhaustif en $O(n \times n!) \rightarrow$ marche même sans inégalité triangulaire

## Approximation sans inégalité triangulaire ?

- Pas d'algorithme d'approximation polynomial, sauf si $\text{P} \neq \text{NP}$

# Conclusion générale

Les algorithmes d'approximation sont extrêmement utiles !

## Beaucoup de problèmes sont difficiles

- Théorie de la NP-complétude *HAI602I*
- Deux solutions :
  - algorithme exponentiel $\rightarrow$ petites instances
  - algorithme d'approximation $\rightarrow$ résultat approché

## Conception d'un algorithme d'approximation

- Multitude de techniques $\rightarrow$ toutes celles des algorithmes *standard*, et d'autres !

## Analyse d'un algorithme d'approximation

- Montrer que l'algorithme n'est jamais pire qu'un facteur $\alpha$
- Deux ingrédients :
  - borner la valeur optimale OPT
  - borner la valeur renvoyée par l'algorithme