- HAI503I: Algorithmique 4 -

Chap. 5 – Algorithmes d'approximation

L3 informatique
Université de Montpellier

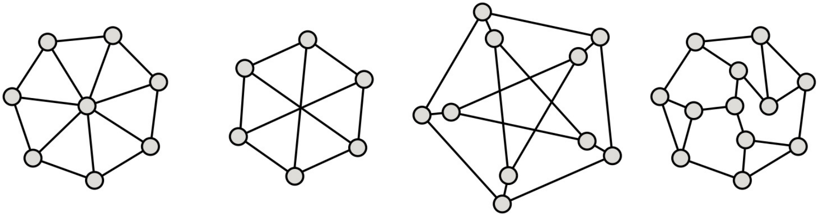
Définition du problème

COUVERTURE-PAR-SOMMETS

Entrée : Un graphe $G = (V, E)$

Sortie : Une **couverture par sommets de** G, c-à-d un sous-ensemble $C \subset V$ de sommets, qui **couvre** toutes les arêtes : pour tout $\{u, v\} \in E$, $u \in C$ ou $v \in C$

Objectif : Trouver C le plus petit possible



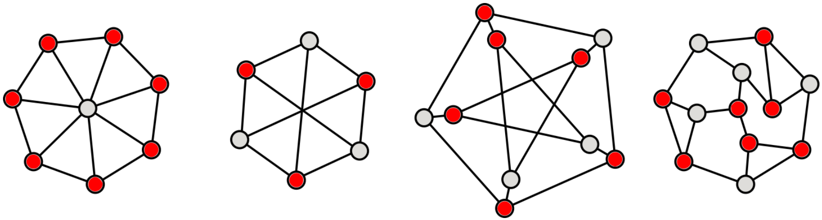
Définition du problème

Couverture-Par-Sommets

Entrée : Un graphe $G = (V, E)$

Sortie : Une **couverture par sommets de** G, c-à-d un sous-ensemble $C \subset V$ de sommets, qui **couvre** toutes les arêtes : pour tout $\{u, v\} \in E$, $u \in C$ 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

- 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

Réduisons nos ambitions...

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

CouvApprox(G) :

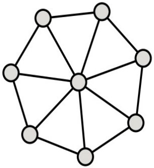
1. $C \leftarrow \emptyset$
2. Tant que G est non vide :
3. Choisir une arête $\{u, v\}$ dans G
4. Ajouter u **et** v dans C
5. Supprimer u et v (et les arêtes incidentes) de G
6. Renvoyer C

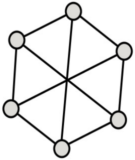
Résultat de l'algorithme CouvApprox

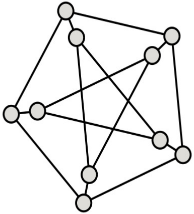
L'algorithme retourne une couverture du graphe G donné en paramètre et a une complexité en $O(n^2)$ (où n est le nombre de sommets de G).

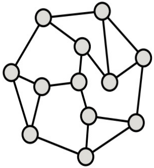
Exemples

- Déroulement de CouvApprox sur certains graphes suivants :



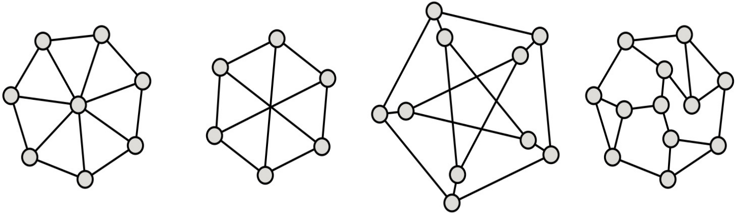






Exemples

- Déroulement de CouvApprox sur certains graphes suivants :

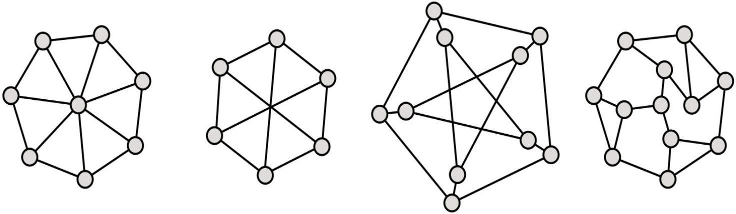


- Taille d'une couverture par sommets minimum :

5 3 6 7

Exemples

- Déroulement de CouvApprox sur certains graphes suivants :



- Taille d'une couverture par sommets minimum :

5 3 6 7

Théorème, garantie de l'algorithme d'approximation

Soit OPT la taille d'une couverture de taille minimale de G, et C l'ensemble retourné par l'appel CouvApprox(G). Alors

$$|C| \leq 2.\mathrm{OPT}$$

Définition du problème

Somme Partielle

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 $\Sigma S \leq t$ et est maximale
- OPT : valeur de la solution maximale (la meilleure possible)

Exemple

Entrée : $E = \{12, 4, 17, 9, 6\}$ et $t = 28$

Définition du problème

SOMME PARTIELLE

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 $\Sigma S \leq t$ et est maximale
- OPT : valeur de la solution maximale (la meilleure possible)

Exemple

Entrée : $E = \{12, 4, 17, 9, 6\}$ et $t = 28$
Solution : OPT$= 27$ pour $S = \{17, 6, 4\}$

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

- fait partie des problèmes NP-complets *HAI602I*
- Meilleur algorithme connu en $O(2^{\frac{n}{2}}) = O(1,414^n)$ *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 $E_{[i]} \leq t$:
5. Ajouter $E_{[i]}$ à S
6. $t \leftarrow t - E_{[i]}$
7. Renvoyer S

Un algorithme d'approximation

$\textsc{SommePartApprox}(E, t)$:

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

Analyse de $\textsc{SommePartApprox}$

En temps $O(n \log n)$, l'appel $\textsc{SommePartApprox}(E, t)$ retourne une solution S vérifiant

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

Où $\textsc{opt}$ est la valeur d'une solution optimale au problème $\textsc{Somme Partielle}$ pour (E, t)

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** : Somme Partielle, Sac-A-Dos, Choix Cours...
- **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,
    c'est-à-dire, telle que : $\forall s' \in S, c(s') \leq c(s)$
  - **Minimisation** : trouver $s \in S$ telle que $c(s)$ soit minimale,
    c'est-à-dire, telle que : $\forall s' \in S, c(s') \geq c(s)$
- **Valeur optimale** : on note OPT la valeur de la solution optimale
  - maximisation : $\text{OPT} = \max_{s \in S} c(s)$
  - minimisation : $\text{OPT} = \min_{s \in S} c(s)$

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

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*

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

Définition : algorithme d'approximation

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

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

Le réel α est appelé **facteur d'approximation** de l'algorithme.

Algorithmes d'approximation

Algorithmes de compromis

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

Définition : algorithme d'approximation

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

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

Le réel α est appelé **facteur d'approximation** de l'algorithme.

Exemples

- Couverture-Par-Sommets : $c(s) = \#$ sommets renvoyés
  CouvApprox est une 2-approximation
- Somme Partielle : $c(s) =$ la somme retournée
  SommePartApprox est une $\frac{1}{2}$-approximation

Comment concevoir des algorithmes d'approximation ?

Une technique fructueuse : algorithmes glouton

- Approche gloutonne souvent rapide $\rightsquigarrow$ efficacité
- Pas toujours la meilleure solution $\rightsquigarrow$ non exact
- Solution souvent *pas trop mauvaise* $\rightsquigarrow$ 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 CouvApprox...)

Objectif du cours

- Concevoir et analyser des algorithmes d'approximation *simples*.
- Construire des algorithmes d'approximation est un domaine de recherche riche, et certains sont très techniques...

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 $\textsc{opt} \leq c_2$ — $\textsc{opt} \geq c_2$

$\rightsquigarrow$ On peut prendre $\alpha = c_1/c_2$ — $\alpha = c_1/c_2$

Exemple sur CouvApprox

Problème de minimisation. On avait : $c(G) \leq |C|$ $(= c_1)$ et $\textsc{opt} \geq |C|/2$ $(= c_2)$ donc $\alpha = c_1/c_2 = |C|/(|C|/2) = 2$ convient.

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 :
- D, tableau d'entiers positifs de taille n correspondant aux *durées* des tâches à répartir
- m entier correspondant aux nombres de processeurs

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 : $t(A) = \max_{1 \leq j \leq m} \left(\sum_{i:A[i]=j} D[i] \right)$

Algorithme glouton *à la volée*

Scénario/modèle 1

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 d'un algorithme glouton** : on affecte la prochaine tâche au processeur le moins occupé

Algorithme glouton *à la volée*

Scénario/modèle 1

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 d'un algorithme glouton** : 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 $i = 1$ à n :
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

Algorithme glouton *à la volée*

ÉquilibrageGlouton(D, m) :

1. $T \leftarrow$ tableau de taille m, intialisé à 0 (*temps total par processeur*)
2. Pour $i = 1$ à n :
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

Théorème

L'algorithme ÉquilibrageGlouton est une 2-approximation pour le problème Équilibrage et a une complexité $O(nm)$ (ou $O(n \log m)$ avec un tas)

Algorithme glouton avec tri

Scénario/modèle 2

On connaît toutes les tâches à l'avance $\rightsquigarrow$ fait-on mieux ?

- **Idée** : On peut trier les tâches par durée décroissante et les affecter comme précédemment selon cet ordre.

Algorithme et complexité

- Même algorithme 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$\rightsquigarrow$ $O(n \log n)$ car $n \geq m$

Théorème

Si D est trié par ordre décroissant, le facteur d'approximation α de ÉquilibrageGlouton vérifie $\alpha \leq \frac{3}{2}$

Bilan sur l'équilibrage de charge

Cas non trié

- L'algorithme glouton est une 2-approximation
- Analyse plus fine de l'algo glouton : $(2 - \frac{1}{m})$-approximation
- Facteur d'approximation atteint

Cas trié

- L'algorithme glouton fournit une $\frac{3}{2}$-approximation
- Analyse plus fine de l'algo glouton : $(\frac{4}{3} - \frac{1}{m})$-approximation
  meilleure borne sur OPT

Encore mieux ?

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

Rappel du Chapitre 2 : le problème SAT

Le problème SAT

Définition

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

Sortie : une affectation des variables qui satisfasse φ ; « 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 φ
- $(x_1, x_2, x_3) = (\text{VRAI}, \text{FAUX}, \text{VRAI})$ ne satisfait pas φ

Le problème MaxSat

Définition : MaxSat

Entrée : un ensemble $C_1, \ldots C_m$ de m clauses disjonctives sur n variables booléennes

Sortie : une affectation des variables

Objectif : maximiser le nombre de clauses satisfaites

Exemple

$$(x_1 \vee \neg x_2 \vee x_3),\ (\neg x_1 \vee x_3),\ (\neg x_3),\ (x_2 \vee x_3)$$

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

On voit facilement qu'on ne peut pas satisfaire plus de 3 *clauses sur les* 4

Algorithmes exacts

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

L'algorithme MaxSatRand

MaxSatRand($C_1, \ldots, C_m$) :

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

Analyse de MaxSatRand

L'algorithme MaxSatRand a une complexité de $O(n)$ et l'espérance du nombre de clauses qu'il satisfait est $\geq \frac{1}{2}$ OPT, où OPT est le OPT est le nombre maximum de clauses qui peuvent être satisfaites par une affectation.

Bilan sur MaxSat

Un algorithme d'approximation

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

Mieux ?

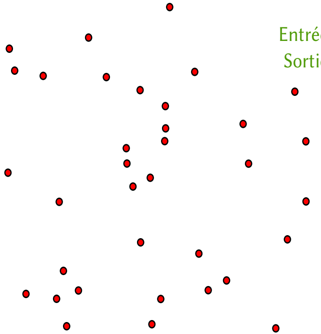
- Si la plus petite clause est de taille $k \rightsquigarrow (1 - 1/2^k)$-approximation
- Il existe un algorithme de $\frac{3}{4}$-approximation, quelque soit k
- On peut *dérandomiser* MaxSatRand

Remarque :exemple de la méthode probabiliste en combinatoire

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 !

Rappel du chapitre 2 : le voyageur de commerce

Le voyageur de commerce



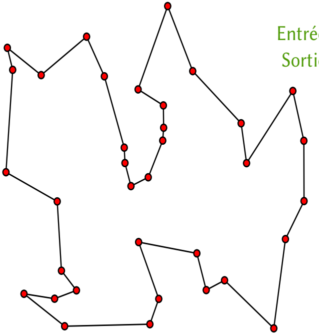
Entrée : Un ensemble de points du plan

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 du chapitre 2 : 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

Problème plus général

Définition : VdC graphique

Entrée : un graphe $G = (V, E)$ avec une longueur $\ell(u, v)$ pour chaque arête vérifiant **l'inégalité triangulaire** : $\ell(u, w) \leq \ell(u, v) + \ell(v, w)$ si les arêtes uv, vw et uw correspondantes existent.

Sortie : Une numérotation u_0, ..., u_{n-1} des sommets de G (c-à-d de V) telle que $u_i u_{i+1}$ existe pour $i = 0, \ldots, n-2$ ainsi que $u_{n-1} u_0$ et qui minimise la longueur totale $\sum_{i=0}^{n-1} \ell(u_i, u_{i+1}) + \ell(u_{n-1}, u_0)$

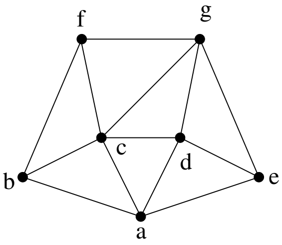
Remarques :

- Le VdC graphique est plus général que le VdC euclidien...
- À venir : 2-approximation pour le VdC graphique basé sur le calcul d'un arbre couvrant de poids minimum (ici le poids sera la longeur ℓ).

Quelques notions de graphes (rappels L2...)

Pour un graphe $G = (V, E)$,

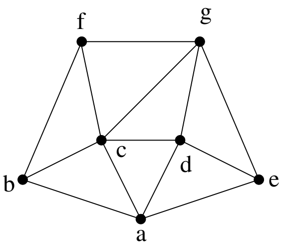
- un **chemin** de G est une suite $v_1, v_2, \dots, v_k$ de sommets de G telle que $v_i v_{i+1}$ est une arête de G pour tout $i = 1, \dots, k-1$.
- un **cycle** de G est une suite $v_1, v_2, \dots, v_k$ de sommets de G telle que $v_i v_{i+1}$ est une arête de G pour tout $i = 1, \dots, k-1$ **et** que $v_k v_1$ soit aussi une arête de G.
- G est **connexe** si pour tous sommets u et v de G, il existe un chemin de u à v.

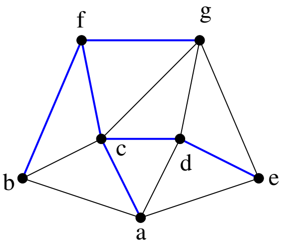


Quelques notions de graphes (rappels L2...)

Pour un graphe $G = (V, E)$,

- un **chemin** de G est une suite $v_1, v_2, \ldots, v_k$ de sommets de G telle que $v_i v_{i+1}$ est une arête de G pour tout $i = 1, \ldots, k-1$.
- un **cycle** de G est une suite $v_1, v_2, \ldots, v_k$ de sommets de G telle que $v_i v_{i+1}$ est une arête de G pour tout $i = 1, \ldots, k-1$ **et** que $v_k v_1$ soit aussi une arête de G.
- G est **connexe** si pour tous sommets u et v de G, il existe un chemin de u à v.
- un **arbre** est un graphe connexe sans cycle.
- un **arbre couvrant** de G est un sous-graphe de G, contenant tous ses sommets et qui est un arbre.





Quelques notions de graphes (rappels L2...)

Théorèmes : connexité et arbre couvrant

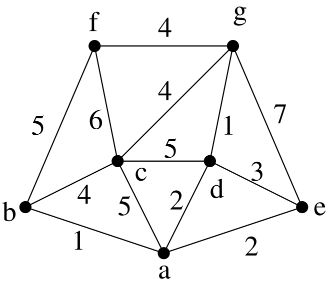
- Un graphe G est connexe si, et seulement si, il admet un arbre couvrant.
- Un arbre sur n sommets contient $n-1$ arêtes.

Quelques notions de graphes (rappels L2...)

Théorèmes : connexité et arbre couvrant

- Un graphe G est connexe si, et seulement si, il admet un arbre couvrant.
- Un arbre sur n sommets contient $n-1$ arêtes.

- On ajoute maintenant une **fonction de poids** (ou **de distance**) $\ell : E \to \mathbb{R}$ sur les arêtes de G.



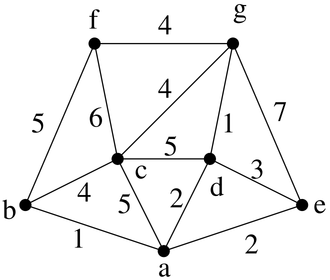
Quelques notions de graphes (rappels L2...)

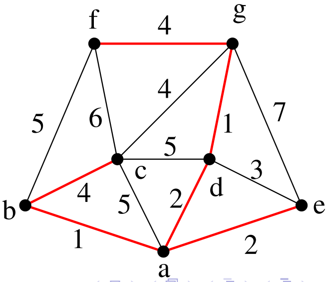
Théorèmes : connexité et arbre couvrant

- Un graphe G est connexe si, et seulement si, il admet un arbre couvrant.
- Un arbre sur n sommets contient $n-1$ arêtes.

- On ajoute maintenant une **fonction de poids** (ou **de distance**) $\ell : E \to \mathbb{R}$ sur les arêtes de G.
- Le **poids d'un arbre couvrant T de G** est la somme des poids des arêtes de T :

$$\ell(T) = \sum_{e \text{ arête de } T} \ell(e)$$





Arbre couvrant de poids minimum (rappels L2...)

Définition : ArbreCouvrantPoidsMin

Entrée : un graphe $G = (V, E)$ connexe avec une fonction de poids ℓ sur ses arêtes.

Sortie : un arbre couvrant de T **de poids minimal** parmi tous les arbres couvrants de G.

Arbre couvrant de poids minimum (rappels L2...)

Définition : ArbreCouvrantPoidsMin

Entrée : un graphe $G = (V, E)$ connexe avec une fonction de poids ℓ sur ses arêtes.

Sortie : un arbre couvrant de T **de poids minimal** parmi tous les arbres couvrants de G.

Kruskal(G, ℓ)

1. Trier les arêtes de G par poids croissant selon ℓ ;
2. $T \leftarrow \emptyset$; /
3. Pour toute arête uv de G selon l'ordre calculé :
4. Si uv ne forme pas un cycle avec les arêtes de T déjà choisies
5. $T \leftarrow T \cup \{uv\}$;
6. Renvoyer T ;

Remarque

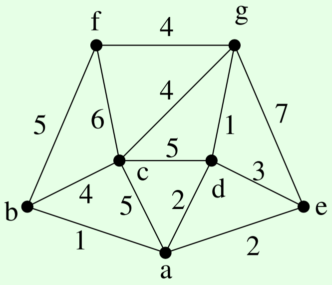
▷ C'est un algo glouton !

Algorithme de Kruskal (rappels L2...)

$\textsc{Kruskal}(G, \ell)$

1. Trier les arêtes de G par poids croissant selon ℓ ;
2. $T \leftarrow \emptyset$;
3. Pour toute arête uv de G selon l'ordre calculé :
4. Si uv ne forme pas un cycle avec les arêtes de T déjà choisies
5. $T \leftarrow T \cup \{uv\}$;
6. Renvoyer T ;

Exemple

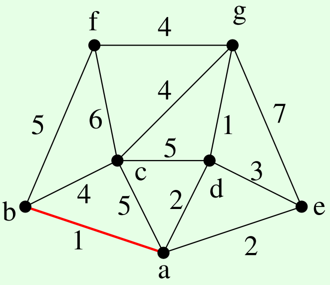


Algorithme de Kruskal (rappels L2...)

KRUSKAL(G, ℓ)

1. Trier les arêtes de G par poids croissant selon ℓ ;
2. $T \leftarrow \emptyset$;
3. Pour toute arête uv de G selon l'ordre calculé :
4. Si uv ne forme pas un cycle avec les arêtes de T déjà choisies
5. $T \leftarrow T \cup \{uv\}$;
6. Renvoyer T ;

Exemple

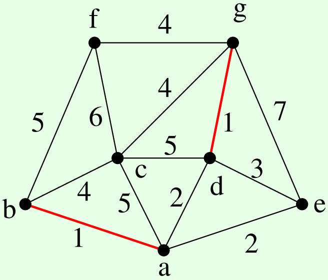


Algorithme de Kruskal (rappels L2...)

KRUSKAL(G, ℓ)

1. Trier les arêtes de G par poids croissant selon ℓ ;
2. $T \leftarrow \emptyset$;
3. Pour toute arête uv de G selon l'ordre calculé :
4. Si uv ne forme pas un cycle avec les arêtes de T déjà choisies
5. $T \leftarrow T \cup \{uv\}$;
6. Renvoyer T ;

Exemple

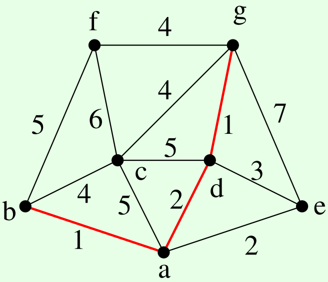


Algorithme de Kruskal (rappels L2...)

KRUSKAL(G, ℓ)

1. Trier les arêtes de G par poids croissant selon ℓ ;
2. $T \leftarrow \emptyset$;
3. Pour toute arête uv de G selon l'ordre calculé :
4. Si uv ne forme pas un cycle avec les arêtes de T déjà choisies
5. $T \leftarrow T \cup \{uv\}$;
6. Renvoyer T ;

Exemple

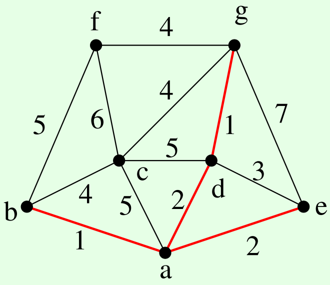


Algorithme de Kruskal (rappels L2...)

KRUSKAL(G, ℓ)

1. Trier les arêtes de G par poids croissant selon ℓ ;
2. $T \leftarrow \emptyset$;
3. Pour toute arête uv de G selon l'ordre calculé :
4. Si uv ne forme pas un cycle avec les arêtes de T déjà choisies
5. $T \leftarrow T \cup \{uv\}$;
6. Renvoyer T ;

Exemple

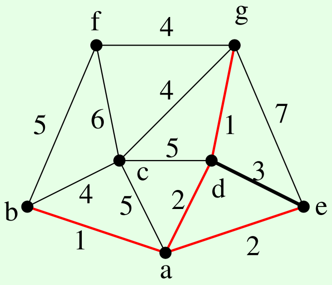


Algorithme de Kruskal (rappels L2...)

$\textsc{Kruskal}(G, \ell)$

1. Trier les arêtes de G par poids croissant selon ℓ ;
2. $T \leftarrow \emptyset$;
3. Pour toute arête uv de G selon l'ordre calculé :
4. Si uv ne forme pas un cycle avec les arêtes de T déjà choisies
5. $T \leftarrow T \cup \{uv\}$;
6. Renvoyer T ;

Exemple

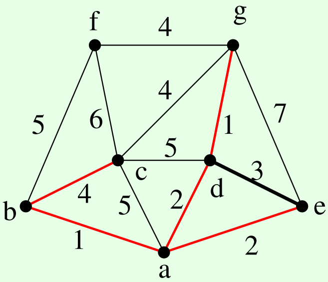


Algorithme de Kruskal (rappels L2...)

KRUSKAL(G, ℓ)

1. Trier les arêtes de G par poids croissant selon ℓ ;
2. $T \leftarrow \emptyset$;
3. Pour toute arête uv de G selon l'ordre calculé :
4. Si uv ne forme pas un cycle avec les arêtes de T déjà choisies
5. $T \leftarrow T \cup \{uv\}$;
6. Renvoyer T ;

Exemple

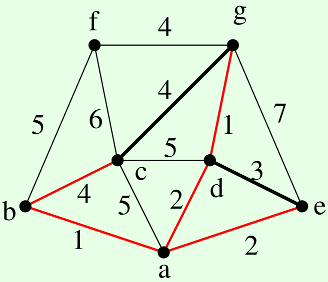


Algorithme de Kruskal (rappels L2...)

KRUSKAL(G, ℓ)

1. Trier les arêtes de G par poids croissant selon ℓ ;
2. $T \leftarrow \emptyset$;
3. Pour toute arête uv de G selon l'ordre calculé :
4. Si uv ne forme pas un cycle avec les arêtes de T déjà choisies
5. $T \leftarrow T \cup \{uv\}$;
6. Renvoyer T ;

Exemple

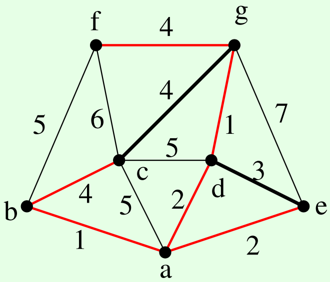


Algorithme de Kruskal (rappels L2...)

KRUSKAL(G, ℓ)

1. Trier les arêtes de G par poids croissant selon ℓ ;
2. $T \leftarrow \emptyset$;
3. Pour toute arête uv de G selon l'ordre calculé :
4. Si uv ne forme pas un cycle avec les arêtes de T déjà choisies
5. $T \leftarrow T \cup \{uv\}$;
6. Renvoyer T ;

Exemple

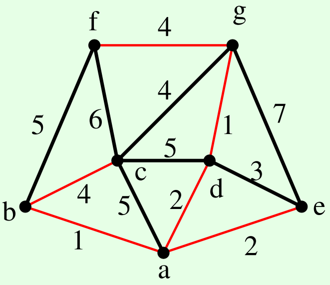


Algorithme de Kruskal (rappels L2...)

KRUSKAL(G, ℓ)

Trier les arêtes de G par poids croissant selon ℓ ;
$T \leftarrow \emptyset$;
Pour toute arête uv de G selon l'ordre calculé :
    Si uv ne forme pas un cycle avec les arêtes de T déjà choisies
        $T \leftarrow T \cup \{uv\}$;
Renvoyer T ;

Exemple



Algorithme de Kruskal (rappels L2...)

$\textsc{Kruskal}(G, \ell)$

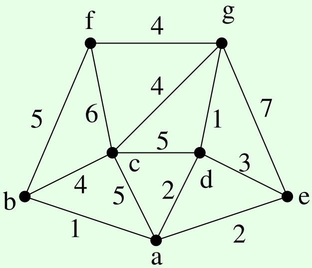
1. Trier les arêtes de G par poids croissant selon ℓ ;
2. $T \leftarrow \emptyset$;
3. Pour toute arête uv de G selon l'ordre calculé :
4. Si uv ne forme pas un cycle avec les arêtes de T déjà choisies
5. $T \leftarrow T \cup \{uv\}$;
6. Renvoyer T ;

Théorème : validité de l'algorithme de Kruskal

L'algorithme de Kruskal retourne un arbre couvrant minimum du graphe connexe G donné en instance. Si on implémente le test de l'existence de cycle par gestion des composantes composantes connexes formées par les arrêtes sélectionnées, alors l'algorithme de Kruskal peut avoir une complexité en $O(m \log n)$.

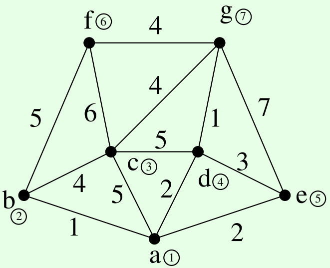
Implémentation de l'algorithme de Kruskal (rappels de L2...)

Exemple



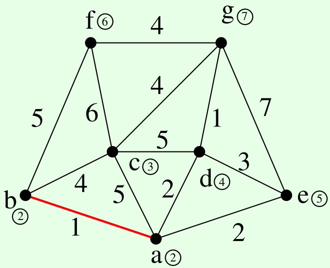
Implémentation de l'algorithme de Kruskal (rappels de L2...)

Exemple



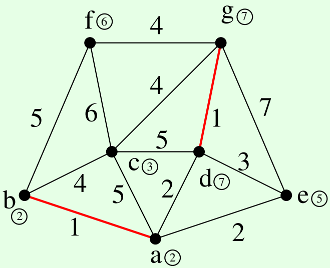
Implémentation de l'algorithme de Kruskal (rappels de L2...)

Exemple



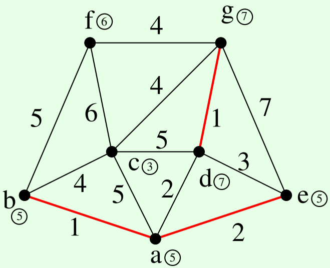
Implémentation de l'algorithme de Kruskal (rappels de L2...)

Exemple



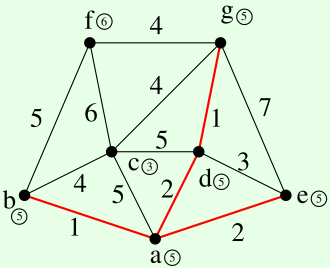
Implémentation de l'algorithme de Kruskal (rappels de L2...)

Exemple



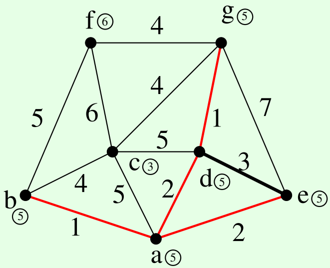
Implémentation de l'algorithme de Kruskal (rappels de L2...)

Exemple



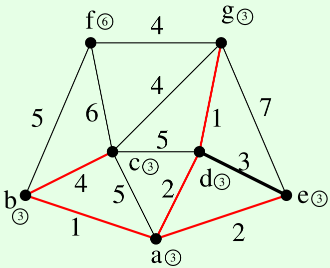
Implémentation de l'algorithme de Kruskal (rappels de L2...)

Exemple



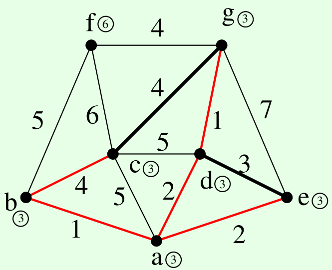
Implémentation de l'algorithme de Kruskal (rappels de L2...)

Exemple



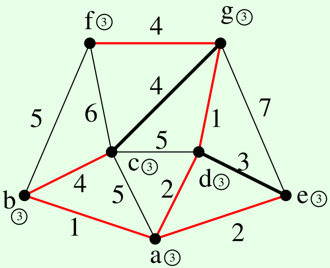
Implémentation de l'algorithme de Kruskal (rappels de L2...)

Exemple



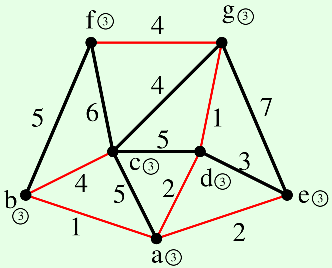
Implémentation de l'algorithme de Kruskal (rappels de L2...)

Exemple



Implémentation de l'algorithme de Kruskal (rappels de L2...)

Exemple



Remarque :

- A la fin de l'algorithme, tous les sommets ont le même numéro de composante...
- Une implémentation possible :

Implémentation de l'algorithme de Kruskal (rappels de L2...)

$\textsc{Kruskal}(G, \ell)$

Trier les arêtes de G par poids croissant selon ℓ ;
$T \leftarrow \emptyset$;
$i \leftarrow 1$;
Pour tout $x \in V$
    $comp(x) \longleftarrow i$; $i \leftarrow i+1$;
Pour toute arête uv de G selon l'ordre calculé :
    Si $comp(u) \neq comp(v)$
        $T \leftarrow T \cup \{uv\}$;
        $aux \leftarrow comp(u)$;
        Pour tout $w \in V$
            Si $comp(w) = aux$ alors $comp(w) \leftarrow comp(v)$;
Retourner T ;

Un algorithme de 2-approximation pour le VdC

VoyageurDeCommerce$_{2APPROX}(G = (V, E), \ell)$:

1. $\mathcal{A} \leftarrow$ arbre couvrant de poids minimum de G ;
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 $(u_0, \ldots, u_{n-1})$;

Un algorithme de 2-approximation pour le VdC

$\textsc{VoyageurDeCommerce}_{2APPROX}(G = (V, E), \ell)$:

1. $\mathcal{A} \leftarrow$ arbre couvrant de poids minimum de G ;
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 $(u_0, \ldots, u_{n-1})$;

Exemple...

Un algorithme de 2-approximation pour le VDC

VOYAGEURDECOMMERCE$_{2APPROX}$$(G = (V, E), \ell)$:

1. $\mathcal{A} \leftarrow$ arbre couvrant de poids minimum de G ;
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 $(u_0, \ldots, u_{n-1})$;

Exemple...

Analyse de VOYAGEURDECOMMERCE$_{2APPROX}$

L'algorithme VOYAGEURDECOMMERCE$_{2APPROX}$ s'exécute en temps $O(m \log n)$ et fournit une 2-approximation au problème du VOYAGEUR DE COMMERCE

Bilan sur le voyageur de commerce

Algorithmes d'approximation

Avec l'inégalité triangulaire :

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

Algorithmes exacts

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

Approximation sans inégalité triangulaire ?

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

Conclusion générale

Beaucoup de problèmes sont difficiles

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

Conception d'un algorithme d'approximation

- Multitude de techniques $\rightsquigarrow$ toutes celles des algorithmes *standard*, notamment les algorithmes gloutons, et d'autres...

Analyse d'un algorithme d'approximation

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