ALGORITHMIQUE, chap. 2

Principaux paradigmes

Diviser pour régner

Un algorithme est dit récursif lorsqu'il est défini à partir de lui même.

Exemple :

POWER (x, n)

Si $n = 0$ **alors** renvoyer 1

sinon renvoyer $x \times$ POWER$(x, n-1)$

cet algorithme calcule de manière récursive la $n - ieme$ puissance de x.

Principes de la récursivité :

- Existence de valeurs de base.
- Méthode pour passer d’un cas compliqué à un cas plus simple.
- On passe des valeurs compliqués à celles de bases de proche en proche.

Avantages :

- Algorithmes plus concis
- Complexité plus faible

Difficultés :

- Conception
- Risque d’erreur augmenté

Analyse des algorithmes récursifs

Le paradigme “diviser pour régner” donne lieu à trois étapes :

- Diviser le problème en sous-problèmes plus simples
- Régner résoudre les sous-problèmes (récursivement ou non)
- Combiner les résultats obtenus en une solution global

Complexité

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{sinon,} \end{cases}$$

Théorème fondamentale

Ce théorème permet de résoudre la plupart des récurrences (mais pas toutes, voir le cas de f superpolynomialement plus petite que le $\log$).

Soient $a \geq 1$ et $b > 1$ deux constantes, soit $f(n)$ une fonction tels que la fonction $T(n)$ soit définie par :

$$T(n) = aT(n/b) + f(n).$$

Alors l'ordre de grandeur de $T(n)$ vaut :

- Si $f(n) = O(n^{(\log_b a)-\epsilon})$ avec $\epsilon > 0$ une constante
  alors $T(n) = \Theta(n^{\log_b a})$
- Si $f(n) = \Theta(n^{\log_b a})$,
  alors $T(n) = \Theta(n^{\log_b a} \log n)$
- Si $f(n) = \Omega(n^{(\log_b a)+\epsilon})$ avec $\epsilon > 0$ une constante
  et si $af(n/b) \leq cf(n)$ avec $c < 1$ une constante et n grand
  alors $T(n) = \Theta(f(n))$

Programmation dynamique

La programmation dynamique, comme la méthode "diviser pour régner" , résout les problèmes en combinant les solutions de sous-problèmes.

Mais La programmation dynamique s’applique quand les sous-problèmes sont dépendants les uns des autres.

Pourquoi ? Car alors un algorithme récursif classique va résoudre plusieurs fois les mêmes sous-problèmes.

En pratique :

Un algorithme de programmation dynamique :

- résout chaque sous-sous-problème une unique fois
- mémorise sa solution dans un tableau
- ne recalcul pas à que le sous-sous-problème est rencontré.

La programmation dynamique est en général appliquée aux problèmes d'optimisation : ces problèmes peuvent admettre plusieurs solutions, parmi lesquelles on veut choisir une solution optimale (maximale ou minimale pour une certaine fonction de coût).

Le développement d'un algorithme de programmation dynamique peut être planifié en quatre étapes :

1. Caractériser la structure d'une solution optimale.

2. Définir récursivement la valeur d'une solution optimale.

3. Calculer la valeur d'une solution optimale depuis les cas de base.

4. Construire une solution optimale pour les informations calculées.

La multiplication de matrices

On suppose que l'on a une suite de n matrices, A_1, ..., A_n, et que l'on souhaite calculer le produit :

$$A_1 A_2 ... A_n.$$

On peut évaluer cette expression après avoir complètement parenthésé cette expression afin de lever toute ambiguïté sur l'ordre des multiplications de matrices.

La multiplication de matrices étant associative, le résultat de la multiplication est indépendant du parenthésage.

Il y a ainsi cinq manières différentes de calculer le produit de quatre matrices :

$$\begin{aligned} A_1A_2A_3A_4 &= (A_1(A_2(A_3A_4))) \\ &= (A_1((A_2A_3)A_4)) \\ &= ((A_1A_2)(A_3A_4)) \\ &= ((A_1(A_2A_3))A_4) \\ &= (((A_1A_2)A_3)A_4) \end{aligned}$$

Le parenthésage du produit peut avoir un impact crucial sur le coût de l'évaluation du produit.

Le produit d'une matrice A de taille $p \times q$ par une matrice B de taille $q \times r$ produit une matrice C de taille $p \times r$ en pqr multiplications scalaires.

Exemple

Trois matrices A_1, A_2 et A_3 de dimensions 10×100, 100×5 et 5×50.

- Parenthésage $((A_1 A_2) A_3$

  Combien de multiplications scalaires ?

- Parenthésage $(A_1 (A_2 A_3))$

  Combien de multiplications scalaires ?

Moralité ?

Problématique de la multiplication d'une suite de matrices

Input

A_1, ..., A_n, n matrices avec

pour $i = 1,2,...,n$ la matrice A_i est de dimensions $p_{i-1} \times p_i$

Output

Un parenthésage du produit $A_1 A_2 ... A_n$ de façon à minimiser le nombre de multiplications scalaires.

L’approche bovine...

Le passage en revue de tous les parenthésages possibles ne donnera pas un algorithme efficace...

...est inefficace...

Soit $P(n)$ le nombre de parenthésages possibles d'une séquence de n matrices. On peut couper une séquence de n matrices entre la $k$ieme et la $(k+1)$ieme, pour k prenant n'importe quelle valeur dans l'intervalle $[1,n-1]$.

On parenthèse alorsles deux sous-séquences résultantes indépendamment.

...c'est clair

D'où la récurrence :

$$P(n) = \begin{cases} 1 & \text{si } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n \geq 2. \end{cases}$$

On peut montrer que

$$P(n) = \frac{1}{n} C_{2n-2}^{n-1} = \Omega\left(\frac{4^n}{n^{3/2}}\right).$$

Le nombre de solutions est donc au moins exponentiel en n...

Heureusement la prog. Dyn. est là !

La première étape du paradigme de la programmation dynamique consiste à caractériser la structure d'une solution optimale.

Nous notons $A_{i..j}$ la matrice résultant de l'évaluation du produit $A_i A_{i+1} \ldots A_{j-1} A_j$. Un parenthésage optimal de $A_1 A_2 \ldots A_n$ sépare le produit entre A_k et A_{k+1} pour une certaine valeur k.

Dans notre solution optimale on commence donc par calculer les matrices $A_{1..k}$ et $A_{k+1..n}$ puis on les multiplie pour obtenir la matrice $A_{1..n}$ recherchée.

Le coût du calcul est donc la somme des coûts des calculs des matrices $A_{1..k}$ et $A_{k+1..n}$ et de leur produit.

Par conséquent le parenthésage de la sous-suite $A_1 \ldots A_k$ (et celui de la sous-suite $A_{k+1}..A_n$) doit être optimal : sinon, on le remplace par un parenthésage plus économique, et on obtient un parenthésage global plus efficace que... le parenthésage optimal !

Par conséquent, une solution optimale à une instance du problème de multiplication d’une suite de matrices utilise uniquement des solutions optimales aux instances des sous-problèmes.

La sous-structure optimale à l’intérieur d’une solution optimale est l’une des garanties qu’il existe une solution “programmation dynamique”.

Rappel

La deuxième étape du paradigme de la programmation dynamique consiste à définir récursivement la valeur d’une solution optimale en fonction de solutions optimales aux sous-problèmes.

• on prend comme sous-problèmes les problèmes consistant à déterminer le coût minimum d'un parenthésage de $A_i A_{i+1} \ldots A_j$.

• on note $m[i,j]$ le nombre minimum de multiplications scalaires pour $A_i A_{i+1} \ldots A_j = A_{i..j}$.

- Pour tout i, $m[i,i] = 0$ car $A_{i..i} = A_i$.

- Considérons un couple (i,j) avec $i < j$. Supposons qu'un parenthésage optimal sépare le produit $A_i A_{i+1} ... A_j$ entre A_k et A_{k+1}.

Alors le coût du calcul de $A_{i..j}$ est égal au coût du calcul de $A_{i..k}$, plus celui de $A_{k+1..j}$, plus celui du produit de ces deux matrices.

Nous avons donc :

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j.$$

Cette équation nécessite la connaissance de la valeur de k, connaissance que nous n'avons pas. Il nous faut donc passer en revue tous les cas possibles et il y en a $j - i$:

$$m[i,j] = \begin{cases} 0 & \text{si } i = j, \\ \text{Min}_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{si } i < j. \end{cases}$$

$m[i,j]$ est le coût d'une solution optimale. Pour pouvoir construire une telle solution on note $s[i,j]$ une valeur k telle que $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$.

En suivant la récursion, on pourra faire la “chose” suivante :

LA CHOSE(p, i, j)

si $i = j$ **alors retourner** 0

$m[i,j] \leftarrow +\infty$

pour $k \leftarrow 1$ **à** $j - 1$ **faire**

$q \leftarrow$ LA CHOSE(p, i, k)

$+$ LA CHOSE(p, $k + 1$, j)

$+ p_{i-1} p_k p_j$

si $q < m[i,j]$ **alors** $m[i,j] \leftarrow q$

renvoyer $m[i,j]$

La complexité de cet algorithme est donné par la récurrence :

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ 1 + \sum_{k=1}^{n-1} \left(T(k) + T(n-k) + 1\right) & \text{pour } n > 1. \end{cases}$$

Dans le cas général, cette complexité peut se récrire :

$$T(n) = 2\sum_{i=1}^{n-1} T(i) + n.$$

On a donc $T(n) = \Omega(2^n)$.

La quantité totale de travail effectué par l'appel LA CHOSE(P, 1, n) est donc au moins exponentiel ! aie aie aie ...

Faisons mieux...

Comment mieux faire :

Le nombre de sous-problèmes est assez réduit : pour chaque choix de i et de j tels que $1 \leq i \leq j \leq n$, il y a $C_n^2 + n = \Theta(n^2)$ choix.

L'algorithme récursif rencontre chaque sous-problème un grand nombre de fois (ici, un nombre exponentiel de fois) d'où sa complexité.

Cette propriété, dite des sous-problèmes superposés, est celle qui permet de faire de la programmation dynamique.

Plutôt que d'implémenter de manière récursive l'équation on aborde la troisième étape du paradigme de la prog. dyn. : on calcule le coût optimal en utilisant une approche ascendante.

L'algo. suivant prend en entrée : $p_0,p_1,\ldots,p_n$ (les dimensions des matrices).

Cet algorithme calcul le coût optimal $m[i,j]$ et enregistre un indice $s[i,j]$ permettant de l'obtenir.

ORDONNER-CHAÎNEDEMATRICES(p)

$n \leftarrow longueur(p) - 1$
pour $i \leftarrow 1$ **à** n **faire** $m[i,i] \leftarrow 0$
pour $l \leftarrow 2$ **à** n **faire**
  pour $i \leftarrow 1$ **à** $n - l + 1$ **faire**
    $j \leftarrow i + l - 1$
    $m[i,j] \leftarrow +\infty$
    pour $k \leftarrow 1$ **à** $j - 1$ **faire**
      $q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$
      si $q < m[i,j]$ **alors** $m[i,j] \leftarrow q$
                $s[i,j] \leftarrow k$
renvoyer m **et** s

Petites remarques :

• L'algorithme remplit le tableau m en considérant des suites de matrices de longueur croissante.

• En effet que le calcul du coût d'un produit de m matrices ne dépend que des coûts de calcul de suites de matrices de longueur strictement inférieure.

• La boucle sur l est une boucle sur la longueur des suites considérées.

Un simple coup d'oeil à l'algorithme montre que sa complexité est en $O(n^3)$. Plus précisément :

$$\begin{aligned} T(n) &= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} \sum_{k=i}^{i+l-2} 1 \\ &= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} (l-1) \\ &= \sum_{l=2}^{n} (n-l+1)(l-1) \\ &= \sum_{l=2}^{n} (-l^2 + l(n+2) - (n+1)) \\ &= \frac{n^3+5n+12}{6} \end{aligned}$$

sachant que $\sum_{i=1}^{n} i^3 = \frac{n(n+1)(2n+1)}{6}$.

La complexité de l'algorithme ORDONNER-CHAÎNEDEMATRICES est donc en $\Theta(n^3)$ ce qui est infiniment meilleur que la solution naïve énumérant tous les parenthésages ou que la solution récursive, toutes deux de complexité exponentielle.

L'algorithme ORDONNER-CHAÎNEDEMATRICES calcule le coût d'un parenthésage optimal, mais n'effectue pas la multiplication de la suite de matrices.

Par contre, l'information nécessaire à la réalisation d'un calcul suivant un parenthésage optimal est stockée au fur et à mesure dans le tableau s.

On examine ici les deux caractéristiques principales que doit posséder un problème d'optimisation pour que la programmation dynamique soit applicable : une sous-structure optimale et des sous-problèmes superposés. On examinera aussi une variante de ce paradigme : le recensement.

Un problème fait apparaître une **sous-structure optimale** si une solution optimale au problème fait apparaître des solutions optimales aux sous-problèmes. La présence d'une sous-structure optimale est un bon indice de l'utilité de la programmation dynamique (mais cela peut aussi signifier qu'une stratégie gloutonne est applicable). La sous-structure optimale d'un problème suggère souvent une classe de sous-problèmes pertinents auxquels on peut appliquer la programmation dynamique.

La seconde caractéristique que doit posséder un problème d’optimisation pour que la programmation dynamique soit applicable est "l’étroitesse" de l’espace des sous-problèmes, au sens où un algorithme récursif doit résoudre constamment les mêmes sous-problèmes, plutôt que d’en engendrer toujours de nouveaux.

En général, le nombre de sous-problèmes distincts est polynomial par rapport à la taille de l’entrée. Quand un algorithme récursif repasse sur le même problème constamment, on dit que le problème d’optimisation contient des **sous-problèmes superposés**.

A contrario, un problème pour lequel l’approche "diviser pour régner" est plus adaptée génère le plus souvent des problèmes nouveaux à chaque étape de la récursivité.

Les algorithmes de programmation dynamique tirent parti de la superposition des sous-problèmes en résolvant chaque sous-problème une unique fois, puis en conservant la solution dans un tableau où on pourra la retrouver au besoin avec un temps de recherche constant.

Il existe une variante de la programmation dynamique qui offre souvent la même efficacité que l'approche usuelle, tout en conservant une stratégie descendante. Son principe est de **recenser** les actions naturelles, mais inefficaces, de l'algorithme récursif. Comme pour la programmation dynamique ordinaire, on conserve dans un tableau les solutions aux sous-problèmes, mais la structure de remplissage du tableau est plus proche de l'algorithme récursif.

En pratique, si tous les sous-problèmes doivent être résolus au moins une fois, un algorithme ascendant de programmation dynamique bat en général un algorithme descendant avec recensement d’un facteur constant car il élimine le temps pris par les appels récursifs et prend moins de temps pour gérer le tableau.

En revanche, si certains sous-problèmes de l’espace des sous-problèmes n’ont pas besoin d’être résolus du tout, la solution du recensement présente l’avantage de ne résoudre que ceux qui sont vraiment nécessaires.

Programmation Dynamique, partie 2

Rappel Partie 1

Nous avons vu comment, au travers de l’exemple du problème de la multiplication de matrices, désigner un algorithme de programmation dynamique.

Aujourd’hui

L’exemple du problème de la plus longue sous séquence commune (PLSSC ou PLSC).

Plus Longue Sous-séquence Commune : PLSC

Définition

Soit $X = x_1x_2x_3x_4...x_n$ et $Z = z_1z_2z_3z_4...z_k$ deux séquences de caractères.

Z est une sous-séquence de X si il existe une suite croissante $1 \leq i_1 \leq i_2 \leq ... \leq i_k \leq m$ d'indices de X telle que :

$$x_{i_j} = z_j \text{ pour } j = 1,2,3,...,k.$$

Exemple

$Z = bcdb$

$X = abcbdab$

définition Z est une sous-séquence commune de deux séquences X et Y si Z est une sous-séquence de X et de Y.

Exemple

$X = abcbdab$

$Y = bdcaba$

bca sous séquence commune.

définition

Z est une plus longue sous-séquence commune de deux séquences X et Y si Z est une sous-séquence de X et de Y et si c'est la plus longue parmi celles-ci.

Problème : PLSC

Input

Deux séquences de caractères $X = x_1x_2...x_m$ et $Y = y_1y_2...y_n$

Output

a. Une PLSC de X et Y

b. La longueur d'une PLSC de X et Y

Définition

le i-ème préfixe de $X = x_1x_2x_3...$ est $X_i = x_1x_2...x_i$.

Problème PLSC

- trouver la longueur d'une PLSC par programmation dynamique (ici le cout est donné par la longueur)
- bonus : donner la PLSC

Théorème : caractérisation d'une PLSC

Soit $X = x_1x_2...x_m$ et $Y = y_1y_2...y_n$ deux séquences, et soit $Z = z_1z_2...z_k$ une PLSC de X et Y, alors :

- $x_m = y_n \quad \Longrightarrow \quad x_k = x_m = y_n$ et Z_{k-1} est une PLSC de X_{m-1} et Y_{n-1}.
- $x_m \neq y_n$ et $z_k \neq x_m \quad \Longrightarrow \quad Z$ est une PLSC de X_{m-1} et Y.
- $x_m \neq y_n$ et $z_k \neq y_n \quad \Longrightarrow \quad Z$ est une PLSC de Y_{n-1} et X.

Solution par récurrence

Soit $C[i,j] =$ la longueur d'une PLSC de X_i et Y_j. On a alors :

- $C[i,j] = 0$ si $i = 0$ ou $j = 0$
- $C[i,j] = C[i-1,j-1] + 1$ si $i,j > 0$ et $x_i = y_j$
- $C[i,j] = \text{Max}(C[i-1,j],C[i,j-1])$ si $i,j > 0$ et $x_i \neq y_j$

Complexité de la procédure associée

$\mathcal{O}(m.n)$

La procédure

```
for i=1 to m do C[i,0]:=0
for j=1 to n do C[0,j]:=0
for i=1 to m
   for j=1 to n
      if x_i = y_j then C[i,j]:=C[i-1,j-1]+1
      else if C[i-1,j] >= C[i,j-1]
         then C[i,j]:=C[i-1,j]
      else C[i,j]:=C[i,j-1]
```

Algorithmes Gloutons (Greedy)

La gloutonnerie ? pour quoi faire ?

Les algorithmes gloutons ont pour vocation de résoudre des problèmes d'optimisation (comme les algorithmes de programmation dynamique).

Mais leur méthode est complétement différente.

Qu'est ce qu'un glouton ?

C’est un mammifère de l’espèce des *Golu Golu*...

Cet animal trapu et robuste, apparenté à la famille des belettes, se trouve dans presque toutes les régions du Canada, bien qu’il ait disparu des régions les plus méridionales.

Il est courageux, hardi et curieux comme toutes les belettes et autres loutres. Il se nourrit de toute une variété de racines et de baies, de menu gibier et de poissons. On l’a vu attaquer et tuer des animaux aussi gros que le caribou et la chèvre de montagne.

Mais pour nous ce sera...

Un algorithme glouton c'est :

- Un ensemble de candidats, que l'on regarde un par un.
- Une fonction qui vérifie si un ensemble de candidats donnent une solution au problème.
- Une fonction qui vérifie si un ensemble de candidats est faisable.
- Une fonction de selection qui indique quel est le meilleur prochain candidat.
- Une fonction qui donne une valeur à une solution.

De manière plus intuitive, un algorithme glouton est un algorithme qui prend les éléments à analyser un par un et qui ne fait que des choix locaux.

De manière évidente, un tel algorithme est toujours incrémentale et à toujours la même structure.

La structure :

- Au début, l'ensemble des candidats choisis est vide.
- A chaque étape, on ajoute le meilleur prochain candidat.
- Si l'ensemble n'est plus faisable, alors on enlève le dernier candidat ajouté.
- A chaque étape, on regarde si on a une solution au problème.

Et au final

Un algorithme glouton correct renvoie comme première solution la solution optimale.

La structure (suite):

C est l'ensemble des candidats S est l'ensemble solution

```
While  not(solution(S)) and C
       Soit x l'élément de C qui
                  maximise select(x)
       C <- C - x
       if  feasible(S + {x})
              then S <- S + {x}
       return  S
```

Remarques :

- La fonction de selection n'est pas forcément identique à celle qui donne la valeur de la solution.
- A chaque étape l'algorithme prend le candidat le meilleur sans faire de prévisions sur les possibles futurs.
- Une fois vu, un candidat n'est plus jamais considéré.
- Les algorithmes gloutons ne fournissent pas une solution optimale pour tous les problèmes.

Exemples de problèmes “gloutons” :

- Sélection d’activité
- Ordonnancement de deadlines
- Minimum spanning tree (arbre de recouvrement minimal)
- Coloriage de graphes
- Voyageur de commerce
- Plus court chemin (Dijkstra)

Ce qui fait l’approche gloutonne...

La propriété du choix Glouton

Une solution globalement optimale peut être trouvée en faisant des choix localement optimaux (choix gloutons).

Par rapport à la Prog. Dyn.

Pour examiner les solution, la prog. dyn. préfère une approcge bottom up, alors que les algorithmes gloutons fonctionnent en top down.

La propriété de la sous-structure optimale

Une solution optimale pour le problème contient également la solution optimale de n'importe quel sous-problème.

Ce qui signifie :

Si S est une solution optimale qui contient un candidat s_1, alors l'ensemble $S \setminus \{s_1\}$ est solution optimale du sous-problème qui correspond au problème d'origine sans s_1.

Exemple : sac à dos binaire

Une loutre est au bord d'une rivère en train de pêcher des poissons pour sa petite famille, les poissons ne pèsent pas tous le même poids et ont des valeurs caloriques différentes. La loutre veut maximiser le nombre de calories qu'elle va ramener sachant qu'elle ne peut porter qu'au plus 10 kilos de poisson.

Peut-on appliquer l'approche gloutonne ? (réponse : non)

Exercice :

Ecrire un algorithme Glouton pour le problème de la conversion de monnaie :

Je veux donner un ensemble minimal (au sens du nombre) de pièces choisies parmi des pièces de 1, 2 et 5 euros qui corresponde à une somme entière x.

Par exemple, si $x = 8$, alors l'ensemble $\{1,2,5\}$ est minimal et l'ensemble $\{2,2,2,2\}$ ne l'est pas.

Codage de Huffman

Principe :

Le codage de Huffman est une méthode de compression de texte.

Chaque caractère est représenté de manière optimale par une chaîne binaire, appelé un code.

Pour économiser de l'espace, on utilise des codes de longueur variables.

Ce code est basé sur une propriété statistique : plus un caractère est fréquent, plus son code est de longueur petite.

Codes préfixes

Nous ne considérerons que des codes préfixes, c'est à dire des codes où tout mot de code est préfixe du code encodé.

Ainsi, aucun code ne sera préfixe d'un autre code.

Cela permet de décompresser facilement : il suffit de lire les bits du texte encodé jusqu'a ce qu'on reconnaisse un mot de code, puis à recommencer...

Exemple :

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

Soit F un fichier composé de 100000 caractères selon les frequences du tableau, calculer la taille du fichier encodé selon que l'on utilise un code de longueur variable ou de longueur fixe.

Représentation

Un arbre binaire peut être utilisé pour représenter n'importe quel codage binaire.

Les arêtes de l'arbre sont étiquetés par des 0 (gauche) et 1 (droite).
Le code de chaque lettre est donc un chemin dans l'arbre.

chaque feuille correspond à une lettre donnée.

Décoder un caractère se fait donc en suivant un chemin de la racine à une feuille, le codage est l'opération inverse.

Représentation

Pour pouvoir travailler sur ces arbres, on va rajouter une information sur les noeuds :

- Chaque feuille va contenir la fréquence de la lettre qu'elle représente.
- Chaque noeud interne va contenir la somme des fréquences des feuilles de ses sous-arbres.

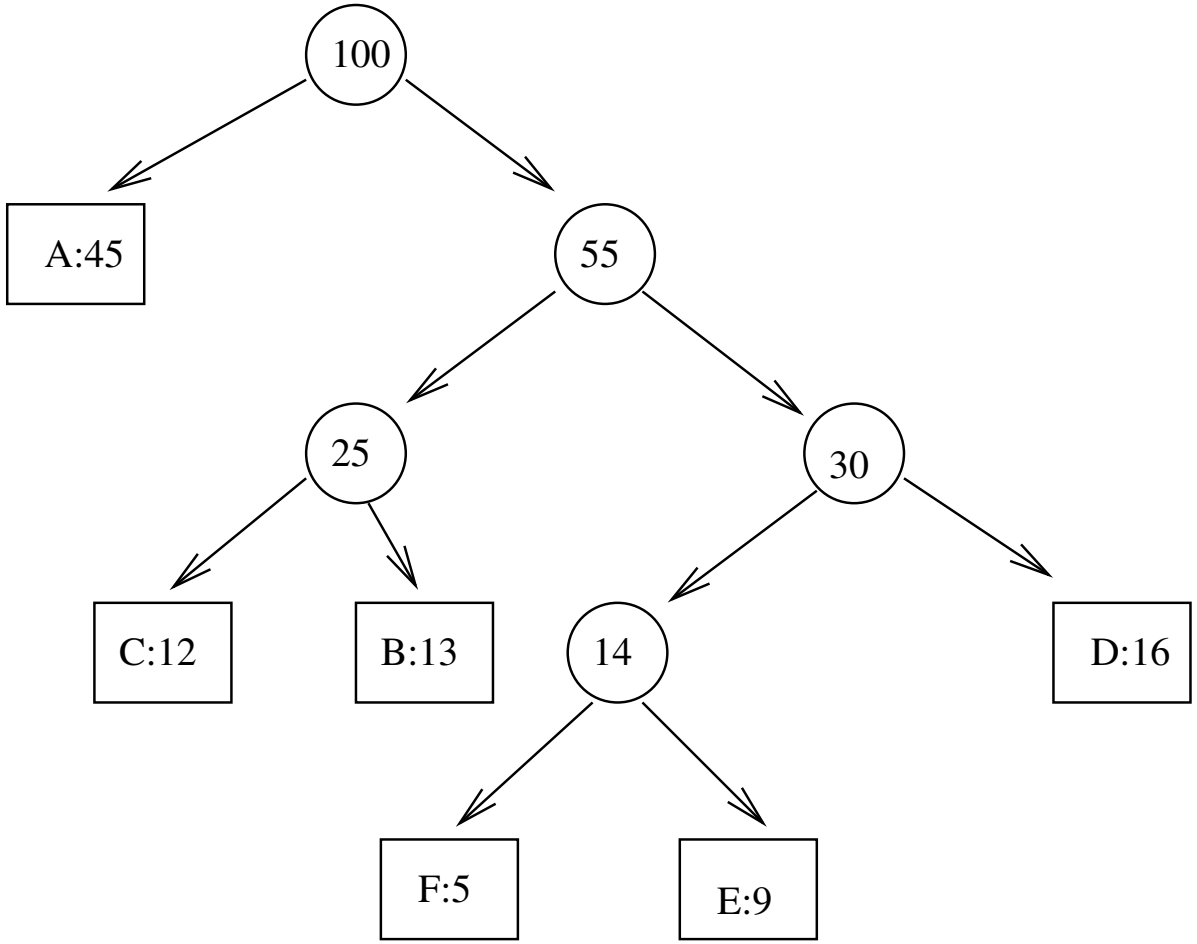
Remarque

Le codage d'un alphabet de taille n est optimal si l'arbre associé au codage a n feuilles et $n-1$ noeuds internes.

conséquence

Le codage à longueur variable de l'exemple est optimal car l'arbre associé est un arbre binaire complet (chaque noeud interne a deux fils).

Exemple (suite) :



Soit un arbre T pour un codage préfixe, le nombre de bits pour coder une chaîne de caractère pris dans un alphabet C est :

$$B(T) = \Sigma_{c \in C} f(c) \cdot d_T(c)$$

avec :

- $f(c)$ est la fréquence de c dans la chaîne.
- $d_T(c)$ est la profondeur de la feuille pour c (et donc la longueur du code de c).

L’algorithme :

```
HUFFMAN(C)
    n <- |C|
    Q <-  C
    pour i allant de 1 à n-1
        faire allouer un nouveau noeud a
        gauche[a] <- x <- EXTRAIRE_MIN(Q)
        droite[a] <- y <- EXTRAIRE_MIN(Q)
        f[a] <- f[x] + f[y]
        INSERER(Q, a)
    retourner EXTRAIRE_MIN(Q)
```

Exercice

Construire l’arbre de Huffman pour le premier exemple.

Cet algo est optimal (preuve omise)

Théorème :

L’algorithme glouton de Huffman produit des codes préfixes optimaux.

Attention !

Cela ne signifie pas qu’il n’existe pas de meilleure méthode de compression !

Arbre de recouvrement minimal

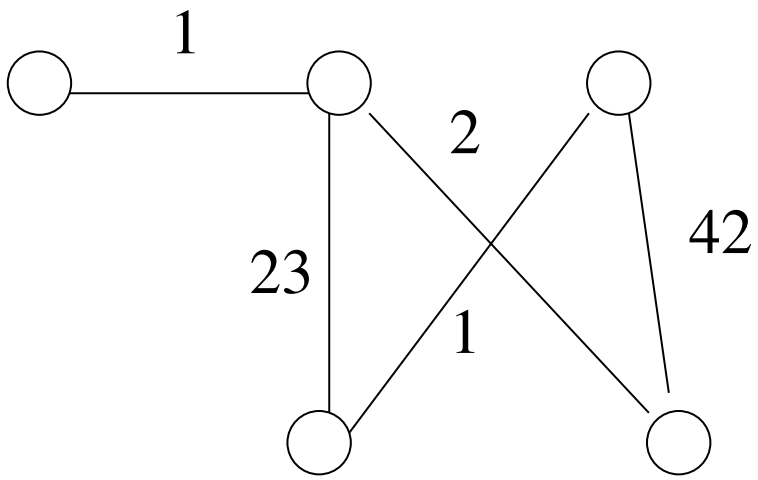
Le problème :

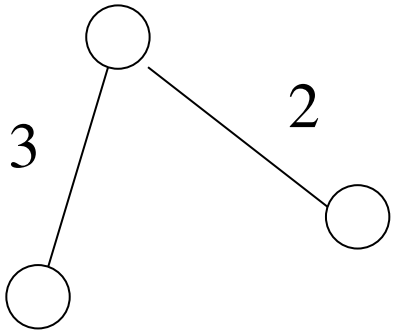
Soit $G = (V,E)$ un graphe non orienté où à chaque arête $e \in E$ est associée un coût (poids) $c(e)$ tel que $c : E \longrightarrow \mathbb{R}$.

Un tel graphe est dit valué.

Exemple :

G





Définition

Le coût d'un sous graphe G' de G est la somme des coûts des arêtes de G'.

Définition Un arbre de recouvrement de G est un sous-graphe de G qui contient tous les sommets de G et qui est un arbre.

Un ARM est un arbre de recouvrement qui est minimal respectivement à la fonction de coût sur les arêtes.

Application

Réseau de comunication

Existence

Si et seulement si le graphe est connexe.

Sinon on parle de forêt de recouvrement minimal.

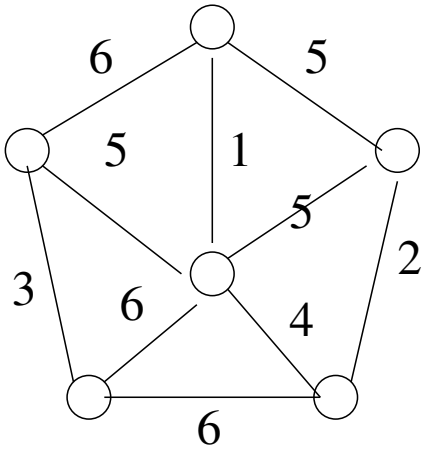
Unicité

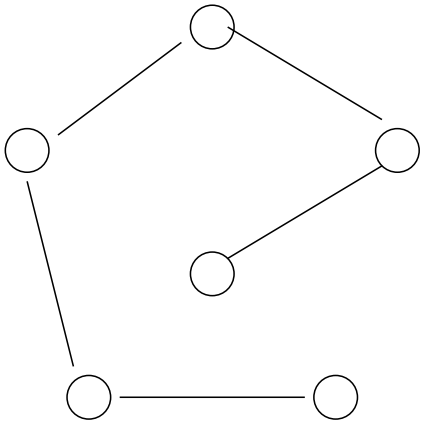
Non

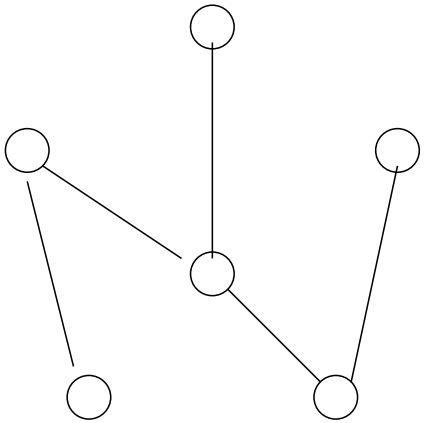
Théorème

Si les coûts des arêtes sont tous différents, alors l’ARM est unique.

Exemple







ou est l'ARM ?

Exercices

1. Donnez un graphe valué qui admet plusieurs ARM.

2. Soit un graphe valué tel que les coûts des différentes arêtes soient tous distincts. Montrer qu'il y a unicité de l'ARM.

3. Soit e une arête de poids minimal dans un graphe. Tout ARM pour le graphe contient l'arête e, VRAI ou FAUX ? et pourquoi ?

Dans la suite

On va donner deux algorithmes pour résoudre le problème de l'arbre de recouvrement minimal.

Mais à propos :

Exercice

Soit A un algorithme qui fournit un ARM dans le cas d'un graphe où les coûts sont positifs. Montrer qu'une modification simple de l'algorithme A fournit aussi un arbre de recouvrement minimal pour les graphes dont les coûts peuvent être quelconques.

Proposition

Soit $G = (V,E)$ un graphe connexe valué.

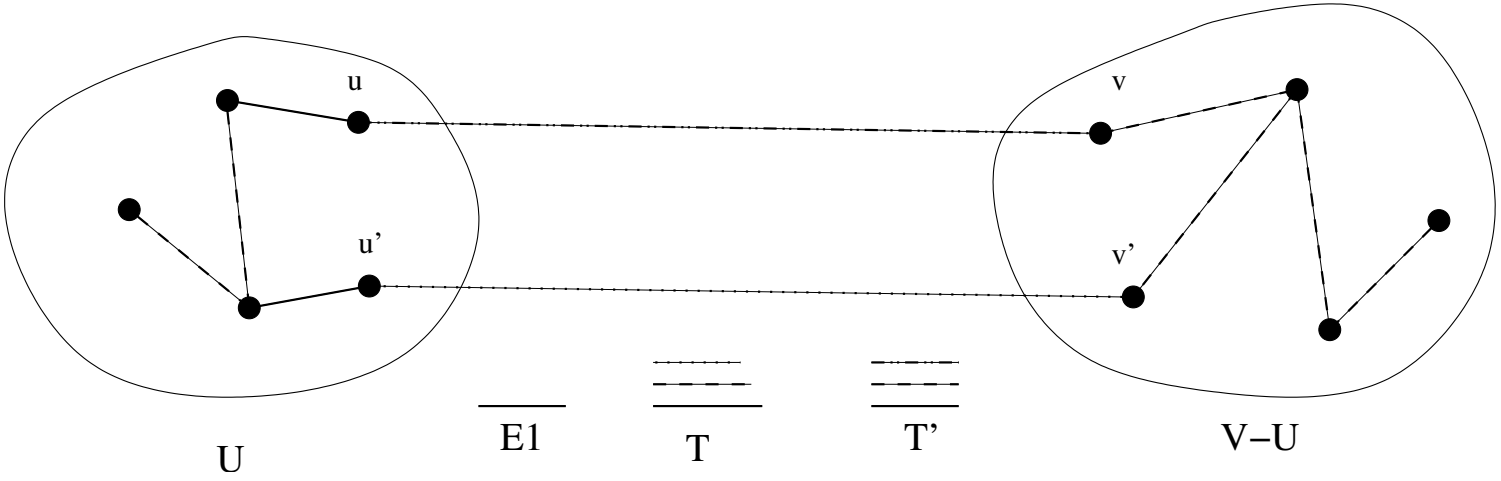
Soit $E_1 \subseteq E$ un sous-ensemble des arêtes qui fait partie d'un ARM de G.

Soit $U \subseteq V$ un sous-ensemble des sommets qui contient tous les sommets adjacents à E_1.

Soit $\{u,v\}$ une arête de coût minimal telle que $u \in U$ et $v \in V - U$.

Alors il existe un ARM qui contient $E_1 \cup \{\{u,v\}\}$.

Preuve



Soit T un ARM qui contient E, mais pas u,v. L'arête u,v forme un cycle avec la chaîne de T qui va de u à v.
Sur cette chaîne, il existe au moins une arête u',v' telle que $u' \in U$ et $v' \in V - U$.

Par définition, on a : $c(u,v) \leq c(u',v')$
On définit alors $T' = T \cup u,v - u',v'$
T est un AR et on a $c(T') \leq c(T)$

L'algorithme de PRIM

- Glouton
- n-1 itérations

Description d'une itération :

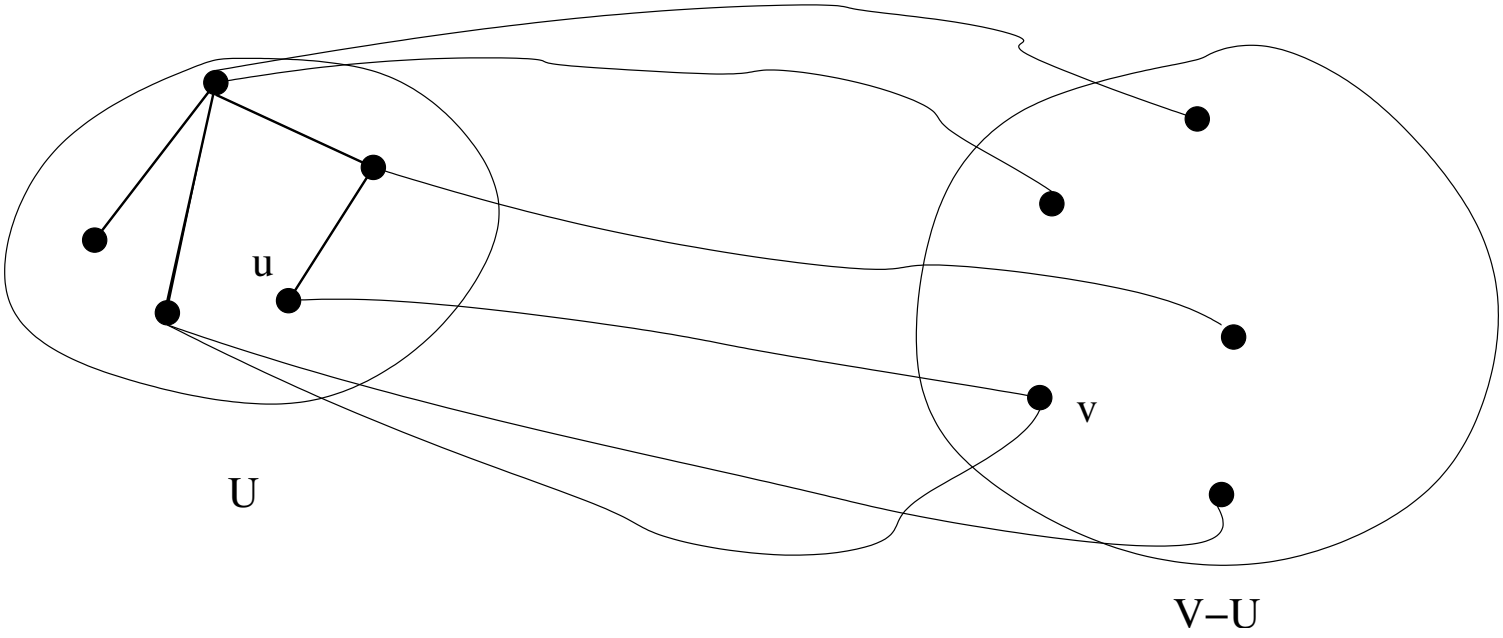
T : ensemble des arêtes déja choisies = un arbre

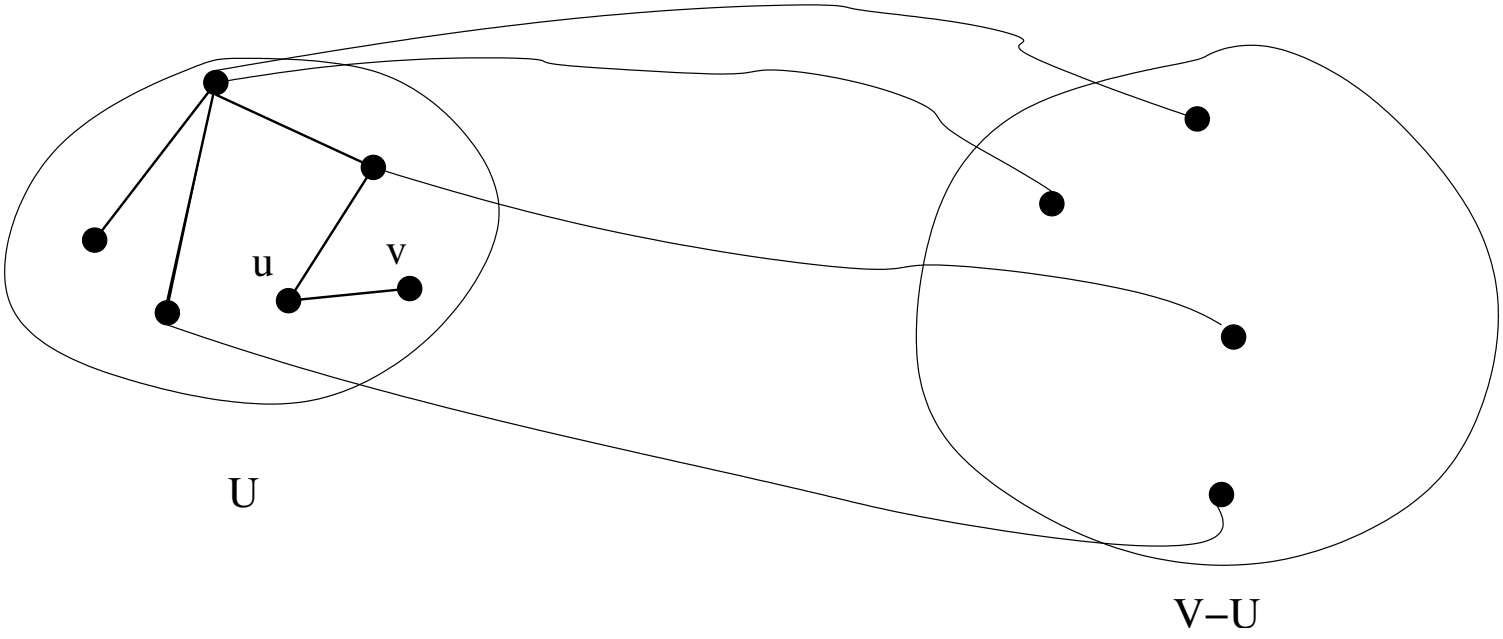
E1 : T

U : ensemble des sommets adjacents à T

u,v : arête de coût minimal tel que $u \in U$ et $v \in V - U$

$T := T \cup u,v$


u
v
U
V−U


u
v
U
V−U

```
procedure PRIM (G graphe valué, var T ens d'arêtes)

var U : ens de sommets
    u,v : sommets

begin
    T=null
    U={1}
    While U<>V do begin
          Soit u,v l'arête de coût minimal
          telle que u in U et v in V-U
          T := T union u,v
          U := U union v
    end
end
```

Correction

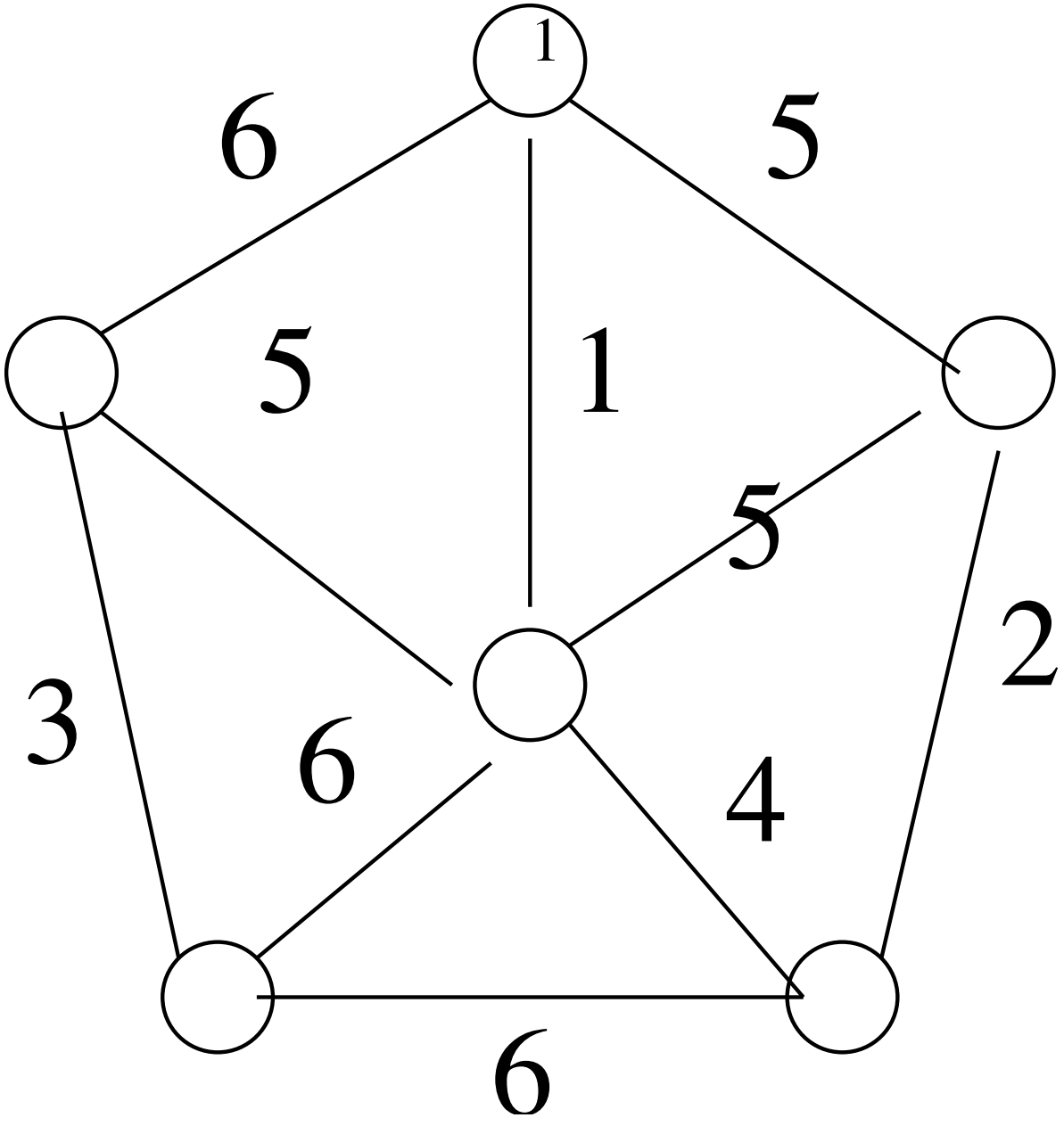
Par la proposition

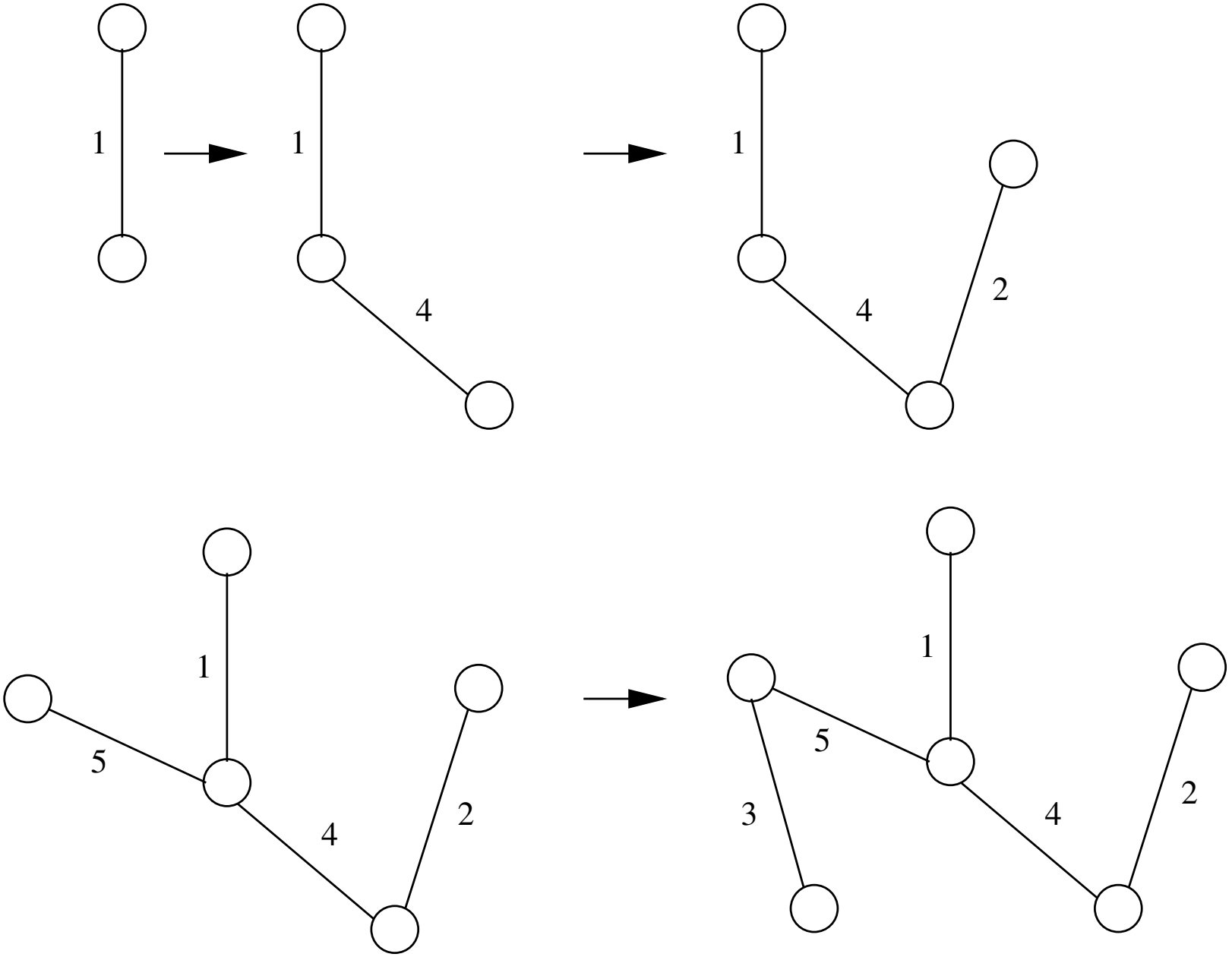
Complexité

- $n-1$ itération
- dans chaque itération, calcul d'un minimum parmi les poids des arêtes restantes

et donc : ???

exemple




1
1
4
1
4
2
1
5
4
2
1
5
3
4
2

Exercice

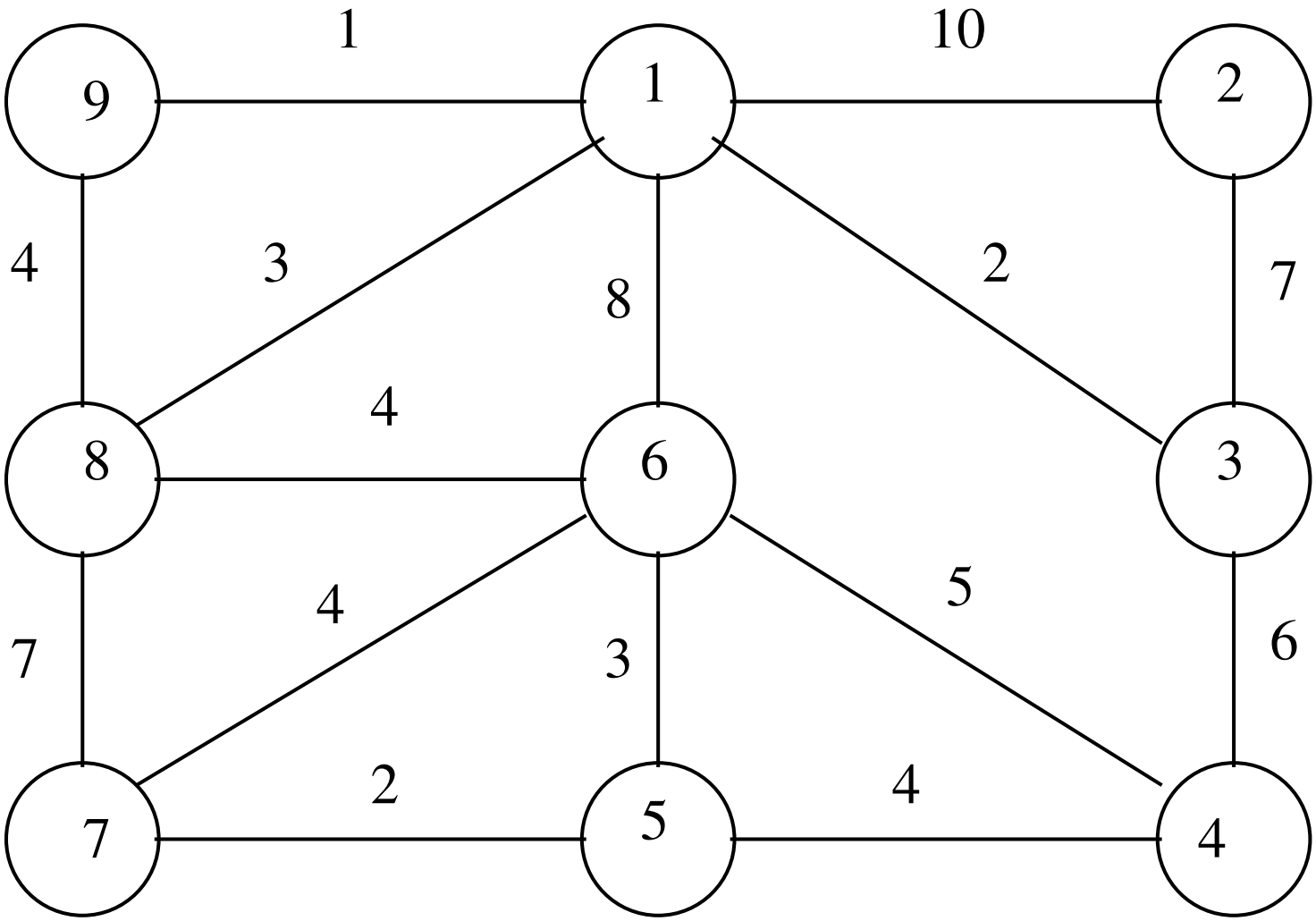
Comment améliorer PRIM (de n^3 vers n^2)?

Pensez à utiliser des tableaux auxiliaires pour diminuer le nombre d'arêtes à considérer à chaque itération.

Donner une version de l'algorithme amélioré.

Exercice

Appliquer PRIM sur le graphe valué suivant :



Algorithme de Kruskal

- Glouton
- au plus n itérations (nombre de sommets, pas d'arêtes)

```
procedure Kruskal

T:=null
L:= liste triée des arêtes selon leurs poids
While |T|<n-1 do begin
      e:=l’arête de coût minimal dans L
      L:=L-e
      If T union e ne contient pas de cycle
      alors T:=T union e
end
```

Correction

Par la proposition

Complexité

- Trier les arêtes : $m \log m$
- Pour le reste, on a besoin de deux opérations :
- UNION : union de deux composantes de T
- FIND : trouve la composante d'un sommet

Exercice

Appliquer Kruskal sur l’exemple précédent.

Une structure de donée pour UNION/FIND

Le but : maintenir une collection $S = \{S_1, S_2, \ldots, S_k\}$ d'ensembles dynamiques sur lequel on veut deux opérations :

UNION(A,B) : remplace A et B par leur union.

FIND(x) : renvoie l'ensemble qui contient x.

On veut d'autre part analyser la complexité d'une structure pour ce problème en fonction de m opérations FIND et $m - 1$ opérations UNION.

- Chaque ensemble est représenté par un arbre
- le nom d'un ensemble est celui de sa racine
- chaque elt dans un ensemble pointe vers son père dans l'arbre représentatif (sauf la racine)
- la racine pointe vers un entier qui est la taille de l'ensemble

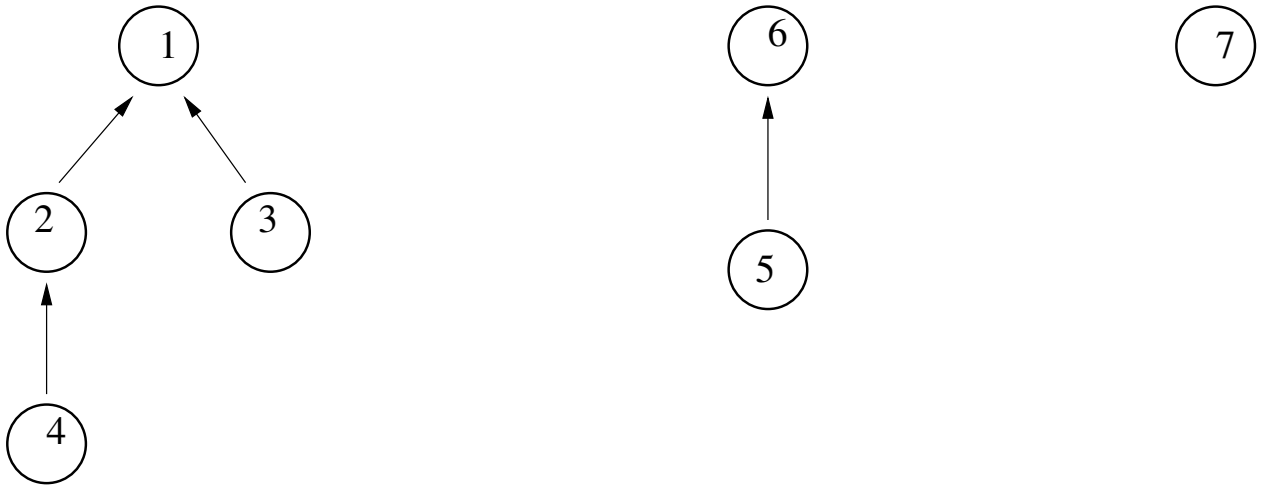
FIND(x) : retourne la racine de l'arbre de x

UNION(A,B) : la racine de A ou B devient la racine de l'union

Exemple

$X = \{1,2,3,4,5,6,7\}$

1→R4 ; 2→1 ; 3→1 ....



1 6 7

Donner le résultat de UNION(1,6) et FIND(4).

Comment choisir la racine de UNION(A,B) ?

La racine du plus petit des deux arbres devient fils de la racine du plus grand. Pou décider, on utilise les pointeurs des racines.

Lemme

Avec ce choix, la profondeur de chaque arbre est au plus $\log m$.

Preuve

Chaque fois qu'un élément change de racine (d'ensemble), il se passe deux choses :

- sa distance à la racine augmente de 1.
- son nouvel ensemble contient au moins deux fois le nombre d'éléments de son ancien ensemble.

Comme le nombre d'léments est m, chaque élément change de racine au plus $\log m$ fois.

Théorème

Si l'on implémente UNION/FIND par des arbres, $m-1$ opérations UNION et m opérations FIND s'éxécutent en temps $\mathcal{O}(m \log m)$.

Preuve

• 1 opération UNION prend un temps $\mathcal{O}(1)$.

• 1 opération FIND prend un temps proportionnel à la profondeur de l'arbre où se trouve l'élément donc $\mathcal{O}(\log m)$ (lemme précédent).

Au total, cela nous fait
$(m-1).\mathcal{O}(1) + m.\mathcal{O}(\log m) = \mathcal{O}(m \log m)$

Remarque

Si dans le cas d'une UNION, on choisit la nouvelle racine sans se préoccuper des tailles des arbres, le temps d'éxecution devriendrait quadratique.

Corollaire

La complexité de Kruskal avec UNION/FIND par arbres est $\mathcal{O}(m \log m)$.