UFR IEEA

Examen AAC 1ère session
Documents autorisés

8 janvier 2009
M1 Informatique

Remarque: Le correcteur se réserve le droit d'enlever des points si le nombre de réponses incongrues lui paraît déraisonnable.

Exercice 1 : Quelques questions de compréhension du cours

Q 1. La professeur Terrorica a demandé à ses étudiants d'étudier la complexité de six problèmes de décision P_1, P_2, P_3, P_4, P_5, P_6. Chacun énonce ce qu'il a réussi à prouver:

Etudiant 1: Le problème P_1 est NP.
Etudiant 2: Le problème P_2 se réduit polynômialement dans le problème P_1.
Etudiant 3: Le problème P_3 se réduit polynômialement dans le problème P_2.
Etudiant 4: Le problème P_4 se réduit polynômialement dans le problème P_2.
Etudiant 5: Le problème P_5 se réduit polynômialement dans le problème P_4.
Etudiant 6: Le problème P_3 est NP-dur.
Etudiant 7: Le problème P_4 est P.

Bien sûr leurs preuves ne sont pas à mettre en doute. Parmi P_1, P_2, P_3, P_4, P_5, quels sont les problèmes dont on peut affirmer qu'ils sont NP? $NP-$durs? P? Justifier brièvement.

$P_3 \leq_P P_2 \leq_P P_1$, $P_5 \leq_P P_4 \leq_P P_2 \leq_P P_1$

Donc on peut affirmer qu'ils sont tous NP, que P_5 et P_4 sont P et que $P_3, P_2 et P_1$ sont $NP-$durs.

Q 2. Soit un problème d'optimisation où il s'agit de minimiser une fonction objectif, et deux heuristiques $H1$ et $H2$ (qu'on suppose correctes, c.à.d. donnant toutes les deux une solution non nécessairement optimale mais correcte!). Le ratio de garantie de $H1$ est 3, celui de $H2$ est 6.

Soit $fopt(I)$ le coût minimal pour l'instance I, $fh1(I)$ le coût de la solution donnée par $H1$ pour I et $fh2(I)$ celui donné par $H2$; on a donc les contraintes:

$fopt(I) \leq fh1(I) \leq 3 * fopt(I)$

$fopt(I) \leq fh2(I) \leq 6 * fopt(I)$

Q 2.1. Peut-on avoir, pour une certaine donnée, $H1$ qui donne une solution de coût 7 alors qu'$H2$ donne une solution de coût 2? Justifier.

Non, car pour toute instance, le coût de la solution proposée par $H1$ est au plus 3 fois celui de l'optimal donc au plus trois fois celui de la solution proposée par $H2$: $fh1(I) \leq 3 * fopt(I) \leq 3 * fh2(I)$

Q 2.2. Peut-on avoir, pour une certaine donnée, $H1$ qui donne une solution de coût 2 alors qu'$H2$ donne une solution de coût 7? Justifier.

Oui, par exemple si $H1$ donne une solution optimale, i.e. $fopt(I) = fh1(I) = 2$. En fait les contraintes nous donnent $7/6 \leq fopt(I) \leq 2$.

Q 2.3. Peut-on avoir, pour une certaine donnée, $H1$ qui donne une solution de coût 7 alors qu'$H2$ donne une solution de coût 3? Justifier.

Oui, par exemple si $H2$ donne une solution optimale i.e. $fopt(I) = fh2(I) = 3$. En fait les contraintes nous donnent $7/3 \leq fopt(I) \leq 3$.

Q 2.4. Peut-on avoir, pour une certaine donnée, $H2$ qui donne une solution de coût 7 alors qu'$H1$ donne une solution de coût 3? Justifier.

Oui, par exemple si $H1$ donne une solution optimale, i.e. $fopt(I) = fh1(I) = 3$. En fait les contraintes nous donnent $7/6 \leq fopt(I) \leq 3$.

Q 3. Pour chaque affirmation suivante, dire si elle est vraie ou fausse. Justifiez briévement. L'alphabet est $\{a, b\}$.

1. L'ensemble des mots ayant un nombre pair de b est récursif.

   Oui, il est même reconnaissable.

2. L'ensemble des mots ayant autant de a que de b est récursif.

   Oui, il est même algébrique.

3. Si L_1 et L_2 ne sont pas récursifs, $L_1 \cup L_2$ ne l'est pas non plus.

   Non, on peut prendre pour L_1 un langage non récursif -il en existe- et pour $L2$ son complémentaire -qui n'est donc pas récursif, puisque les récursifs sont clos par complémentaire et que le complémentaire du complémentaire d'un langage est ce langage -: l'union est $\{a, b\}^*$ et est un langage récursif.

4. Si $L_1 \cup L_2$ n'est pas récursif, L_1 ne l'est pas non plus.

   Non, on peut choisir $L_1 = \emptyset$ -qui est récursif- , L_2 un langage non récursif, et donc $L_1 \cup L_2 = L_2$ n'est pas récursif.

5. Si $L_1 \cap L_2$ n'est pas récursif, L_1 ne l'est pas non plus.

   Non, on peut choisir $L_1 = \{a, b\}^*$ -qui est récursif- , L_2 un langage non récursif, et donc $L_1 \cap L_2 = L_2$ n'est pas récursif.

6. Si $L_1 \cup L_2$ n'est pas récursif, L_1 ou L_2 ne l'est pas non plus.

   Oui, car si les deux sont récursifs, l'union l'est aussi.

   Remarque: l'affirmation est vraie Ssi sa contraposée est vraie; or sa contraposée est:

   Si L_1 et L_2 sont récursifs, $L_1 \cup L_2$ l'est.

7. Si $L_1 \cap L_2$ n'est pas récursif, L_1 ou L_2 ne l'est pas non plus.

   Oui, car si les deux sont récursifs, l'intersection l'est aussi.

   Remarque: l'affirmation est vraie Ssi sa contraposée est vraie; or sa contraposée est:

   Si L_1 et L_2 sont récursifs, $L_1 \cap L_2$ l'est.

Rappel: un langage est récursif si il existe un algorithme pour décider l'appartenance à ce langage.

Exercice 2 : Maximum

Soit un tableau T de n entiers; on suppose qu'il existe un indice p, $0 \leq p < n$ tel que la suite $T[0], T[1], ..., T[p]$ soit strictement croissante et la suite $T[p], T[p+1], ..., T[n-1]$ soit strictement décroissante. Par exemple, si $n = 6$ et $T[0] = 1, T[1] = 6, T[2] = 12, T[3] = 16, T[4] = 9, T[5] = 4$, alors $p = 3$.

Proposer un algorithme en $O(\log n)$ pour déterminer p.

```
// il existe p tel que T[0]< T[1]<..<T[p] > T[p+1]>.....T[n-1];
a=0;
b=n-1;
while (a<b){ // invariant : a <=p <=b
m=a+b/2; // a<=m<b car a<b
```

```
if (T[m] <T[m+1]) //ok car m+1<=b
   a=m+1;
else b=m;
}
//a=p=b
return a;
```

Remarque on a bien $(b-a)$ qui décroît strictement à chaque itération, d'où l'arrêt. La correction est prouvée en utilisant les assertions qui décorent l'algorithme;

DEUX erreurs faites souvent dans les copies (et peu pénalisées):

. algo qui boucle comme par exemple si on remplace `a=m+1` ci-dessus par `a=m`;

. accès à $T[i]$ avec i hors des bornes du tableau par exemple ici avec un test $T[m-1] < T[m]$

Exercice 3 : Au suivant

On veut pouvoir générer tous les mots sur $\{a, b\}$ de longueur inférieure ou égale à une valeur donnée dans l'ordre alphabétique. Par exemple l'énumération pour 3 doit produire:

```
a
aa
aaa
aab
ab
aba
abb
b
ba
baa
bab
bb
bba
bbb
```

Q 1. Pour cela, complétez le code des méthodes *Suivant*() et *EstDernier*() de la classe suivante:

```
class Mot {
//le mot
private char[] cont;

//la longueur du mot courant
private int longueur;

//la longueur maxi
private int longueurmaxi;

//constructeur initialise le mot au mot vide, la longueur au maxi
Mot(int longueurmaxi){
```

```
this.cont=new char[longueurmaxi];
this.longueur=0;
this.longueurmaxi=longueurmaxi;
}

//tranforme le mot en son suivant pour l'ordre alphabétique
//on peut supposer que this.EstDernier() est faux;
void Suivant(){ A faire}

//retourne vrai Ssi le mot est le dernier dans l'ordre alphabétique: bbbb...bbbbb
boolean EstDernier(){A Faire}
......
}
```

```
//tranforme le mot en son suivant pour l'ordre alphabétique
void Suivant(){
if (longueur <longueurmaxi) {longueur++;cont[longueur-1]='a';}
else {
while (cont[longueur-1]=='b') longueur- -; // evt cont[longueur]=' ';
//remarque déclenche une exception si Estdernier() est vrai
cont[longueur-1]='b';
}}

//retourne vrai Ssi le mot est le dernier dans l'ordre alphabétique bbbb...bbbbb
boolean EstDernier(){
if (longueur<longueurmaxi) return false;
for (int i=0;i <longueur; i++) {
if (this.cont[i]=='a') return false;
}
return true;
}
```

<u>Exercice 4</u> : Plaques

Soit n plaques rectangulaires de dimension (x_i, y_i) avec $x_i \geq y_i$, $0 \leq i \leq n-1$; ces plaques ont toute la même épaisseur;

Q 1. On cherche à empiler le maximum de ces plaques en respectant la contrainte suivante: la plaque j ne peut être empilée au-dessus de la plaque i que si $x_j \leq x_j$ et $y_j \leq y_i$. Par exemple, si on a 6 plaques de dimensions respectives $(12, 7)$, $(11, 8)$, $(7, 7)$, $(9, 6)$, $(5, 5)$, $(9, 4)$ on ne pourra pas empiler toutes les plaques, mais on pourra empiler les plaques $0, 2, 4$ dans cet ordre (il y a d'autres solutions optimales comme $1, 2, 4$). Le problème est donc:

Donnée: un entier n, le nombre de plaques
un tableau x de taille n contenant les longueurs des plaques
un tableau y de taille n contenant les largeurs des plaques

Sortie: $J \subset [0, n-1]$ de cardinal maximal tel que pour tout i de J et tout j de J, soit $x[i] \leq x[j]$ et $y[i] \leq y[j]$, soit $x[i] \geq x[j]$ et $y[i] \geq y[j]$

Q 1.1. A quelle condition peut-on empiler toutes les plaques en respectant la contrainte?

Il faut juste que pour tout i et tout j, si $x[i] \leq x[j]$, alors $y[i] \leq y[j]$

Q 1.2. *Glouton?...* Pensez-vous que l'algorithme de type glouton suivant donne toujours la solution optimale? Justifier.

```
Trier les plaques par  x[i] décroissant.
Initialiser J à l'ensemble vide;
Ajouter 0 à J;
largeur=y[0];
pour i de 1 à n-1
  si y[i]< =largeur {
      ajouter i à J;
     largeur=y[i];}
```

Non, contre-exemple: $(10, 1), (8, 3), (7, 2)$

Q 1.3. *ou dynamique?* En utilisant la programmation dynamique, proposer un algorithme en $O(n^2)$ pour résoudre le problème. Vous pourrez supposer les plaques triées par $x[i]$ décroissant.

Remarque: vous pouvez répondre partiellement en calculant uniquement le nombre maximal de plaques qu'on peut empiler (i.e. *card* J).

```
// les plaques sont triées par  $x[i]$ décroissant
int[] nb=new int [n];
//nb[i] sera le nombre  de plaques maximum qu'on peut empiler au-dessus de la plaque i
int[] suiv=new int[n];
//suiv[i] sera le no de la première plaque qu'on empile au-dessus de i dans une solution optimale.
int max=0; la hauteur maxi d'une pile
int indmax;
for (int i=n-1;i>=0;i- -)
  nb[i]=0;
  suiv[i]=i;
  for (int j=i+1; j<n;j++} {
      if (y[i]>=y[j]) && (nb[j]>=nb[i]) {
           nb[i]=1+nb[j];suiv[i]=j}  }
  if (nb[i]>=max) max=1+nb[i]; indmax=i;}
}
ensemble J;
ind=indmax;
J.ajouterr(ind); // première plaque est plaque ind
while nb[ind]>0 {ind=suiv[ind];J.ajouter(ind);} //empiler plaque ind
return(J);
}
```

Remarque: cela correspond exactement à l'algorithme de recherche d'une plus longue sous-suite croissante, qui était l'objet d'un exercice d'une feuille de TD!

Q 2. On cherche maintenant à choisir un nombre minimum de plaques telle que pour toute plaque, il existe une plaque parmi les plaques choisies qui puisse la recouvrir. Dans l'exemple ci-dessus, on choisirait donc les plaques $\{0, 1\}$.

Donnée: un entier n, le nombre de plaques

x, un tableau de taille n contenant les longueurs des plaques

y , un tableau de taille n contenant les largeurs des plaques

Sortie: $J \subset [0, n-1]$ de cardinal minimal tel que pour tout i de $[0, n-1]$, il existe j dans J tel que $x[i] \leq x[j]$ et $y[i] \leq y[j]$.

Remarque: On suppose que deux plaques différentes ont au moins une dimension différente, c.à.d. si $i \neq j$, $x[i] \neq x[j]$ ou $y[i] \neq y[j]$.

Q 2.1. Quelle est la solution optimale pour $(17, 5), (12, 7), (9, 5), (9, 7), (8, 6)$?

$(17, 5), (12, 7)$

Q 2.2. Peut-il y avoir deux solutions optimales différentes pour la même donnée? Justifier.

Non! Supposons qu'on ait deux solutions différentes S_1, S_2. Donc, il existe une plaque i qui est dans l'une et qui n'est pas dans l'autre: on peut supposer sans perte de généralité qu'elle est dans S_1 et pas dans S_2. Cette plaque i est couverte par une plaque j de S_2 puisque S_2 est solution: $x[i] \leq x[j], y[i] \leq y[j]$ avec $x[i] < x[j]$ ou $y[i] < y[j]$ d'après l'hypothèse: deux plaques différentes ont au moins une dimension différente. Mais comme S_1 est solution la plaque j est couverte par une plaque k de S_1: $x[j] \leq x[k], y[j] \leq y[k]$. Donc $x[i] \leq x[k], y[i] \leq y[k]$ avec $x[i] < x[k]$ ou $y[i] < y[k]$: la plaque i est couverte par la plaque k de S_1: mais alors S_1 privée de la plaque i est toujours solution -puisque toute plaque recouverte par i l'est par k- et S_1 ne serait pas optimale.

Q 2.3. Proposez un algorithme de type glouton pour le problème. Justifiez sa correction et analysez sa complexité.

```
Trier les plaques par  x[i] décroissant, PUIS par y[i] décroissant en cas d'égalité des x
Initialiser J à l'ensemble vide;
Ajouter 0 à J;
largeurmaxi=y[0];
pour i de 1 à n-1
  si y[i] >largeurmaxi {
      ajouter i à J;
     largeurmaxi=y[i];}
```

Preuve: l'ensemble J ainsi construit est bien une couverture puisque si une plaque n'est pas ajoutée, c'est qu'elle est couverte par une plaque examinée auparavant. C'est bien une couverture optimale: quand on ajoute i à J, on est sûr qu'elle ne peut être couverte par aucune autre plaque: une plaque qui la couvrirait aurait un `x >x[i]` et un `y >= y[i]` ou un `x >x[i]` et un `y >=y[i]` (car elles sont toutes différentes): mais alors on aurait déjà examinée cette plaque, elle serait donc couverte et on aurait `largeurmaxi>=y[i]`.

Exercice 5 : Processus

Soit un système temps réel avec n processus asynchrones et m ressources. Quand un processus est actif, il bloque un certain nombre de ressources et une ressource ne peut être utilisée que par un seul processus. On cherche à activer simultanément k processus.

Le problème de décision DecProc est donc:

Donnée:

n, le nombre de processus

m le nombre de ressources

pour chaque processus i, la liste P_i des ressources qu'il bloque.

k le nombre de processus que l'on souhaite activer

Sortie:

Oui, si on peut activer k processus simultanément, non sinon.

Par exemple si $n = 4$, $m = 5$, et $P_1 = \{1, 2\}, P_2 = \{1, 3\}, P_3 = \{2, 4, 5\}, P_4 = \{1, 2, 4\}$ on peut activer simultanément les processus 2 et 3 et donc la réponse est Oui pour $k = 2$ mais la réponse est Non pour $k = 3$.

Q 1. Montrer que le problème DecProc est NP.

Exemple de solution:

Remarque préliminaire: la taille du problème est au moins $n + m$.

Un certificat est simplement un ensemble de processus, donc un sous-ensemble de $\{1, ..., n\}$. Un certificat peut donc être codé par un vecteur de n booléens et la taille d'un certificat est donc n, donc inférieure à la taille du problème.

La vérification consiste juste à vérifier que deux processus actifs ne partagent pas une ressource, et que le cardinal est bien k, soit:

```
//probleme donne par n, P1, ...Pi
//certificat : ensemble de processus
boolean correct(certificat certif){
ensemble R=ensemble vide; //les ressources utilisees
int card=0; // pour calculer card (certif), le nb de processus actives
pour i de 1 à n
  si certif.appartient(i) alors{
     card++;
     si intersection( Pi, R) est non vide
     alors  return False; //conflit sur au moins une ressource
     sinon R=union(R, Pi);}
  fsi
fpour
return (card == k);}
```

L'algorithme de vérification est bien polynômial; sa complexité exacte dépend de la complexité de `appartient(), intersection(,) et union(,)` mais sera de toute façon polynômiale, le coût de ces opérations étant polynômial

Remarque 1: si on veut préciser cette complexité -ce qui était non demandé-, soit $P_a(x)$ (resp. $P_{in}(x), P_u(x)$) bornant la complexité de `appartient()` pour un ensemble de cardinal au plus x (resp. le test du vide de l'intersection(resp. le calcul de l'union) de deux ensembles de cardinal au plus x), le coût de l'algo est en $O(n * (P_a(n) + P_{in}(m) + P_u(m))^2)$ soit $O(n * (P_a(t) + P_{in}(t) + P_u(t))^2)$, si t est la taille du problème.

Remarque 2: pour beaucoup, la notion de certificat semble encore confuse; un certificat est juste "un essai de solution", pas forcément une solution correcte;

Q 2. En utilisant le fait que le problème Independent Set étudié en TP est NP-dur, montrer que le problème DecProc est NP-dur.

Remarque: Attention à ne pas se tromper de sens dans la réduction!!!!!

Il fallait réduire Independent Set , connu NP-dur, dans DecProc, et non le contraire.

Il faut donc associer à toute instance I de Independent Set , une instance $red(I)$ de DecProc telle que $red(I)$ soit positive Ssi I l'est.

Une instance de Independent Set est la donnée d'un graphe $(Sommets, Arcs)$ et d'un entier k_0.

On définit $red(I)$ par:

n, le nombre de processus par $n = card(Sommets)$

m le nombre de ressources $p = card(Arcs)$

pour chaque processus i, la liste P_i des ressources qu'il bloque par P_i = l'ensemble des arcs adjacents au sommet i.

k le nombre de processus que l'on souhaite activer par $k = k_o$

La réduction est bien polynomiale (il faut juste calculer $card(Sommets), card(Arcs)$ et pour chaque sommet, les arcs adjacents).

Elle est bien exacte: si I est une instance positive de INDEPENDENT SET, il existe donc un sous-ensemble de cardinal k de sommets indépendants: les k processus associés ne partagent donc aucune ressource et forment donc un sous-ensemble de cardinal k de processus qu'on peut activer simultanément: $red(I)$ est positive.

Réciproquemment, si $red(I)$ est positive, il existe un sous-ensemble de k processus qu'on peut activer simultanément: les k sommets associés ne partagent donc pas d'arcs et sont indépendants: I est bien positive.

Q 3. Que pensez-vous de la complexité du problème DECPROC si chaque processus utilise une seule ressource?

Le problème devient bien sûr polynômial et peut par exemple être résolu par un glouton:

```
//probleme donne par n, P1, ...Pi
//certificat : ensemble de processus
boolean solve(){
ensemble R=ensemble vide; //les ressources utilisees
int card=0; // pour calculer e nb de processus actives
pour i de 1 à n
     si intersection( Pi, R) est   vide
     alors { card++; R=union(R, Pi);}
  fsi
fpour
return (card >= k);}
```

Q 4. Que pensez-vous de la complexité du problème DECPROC si chaque ressource est utilisée par au plus deux processus?

Il est toujours $NP-$dur puisque, dans les instances obtenues par réduction dans la question 2, chaque ressource (un arc) est utilisée par au plus deux processus (ses extrémités).

Q 5. Que pensez-vous de la complexité du problème d'optimisation associé OPTPROC:

Donnée:

n, le nombre de processus

m le nombre de ressources

pour chaque processus i, la liste P_i des ressources qu'il bloque.

Sortie:

k maximum tel qu'on puisse activer k processus simultanément.

On peut dire qu'il est $NP-$ dur: si on avait un algorithme polynômial pour le problème d'optimisation, on en aurait un pour celui de décision! (puisque $(I(n, m, (Pi), k).DecProc() = (I(n, m, (Pi)).OptProc() >= k)$).