Algorithmique II
Backtracking – Programmation Dynamique

(Correction)

Exercice 1 : Encore de l'argent.

Nous disposons d'un stock illimité de pièces de monnaie de m valeurs différentes $p_1, p_2, \cdots, p_m$. On veut représenter un montant M avec ces pièces. Par exemple, $M = 12$ pourra être fourni comme $2+3+7$ mais aussi $5+1+6$. Le nombre de représentations de M ne doit donc pas tenir compte de l'ordre des pièces.

1. Soit $P(i, j)$ le nombre de représentations du montant j avec les i premières valeurs du tableau p. Définir une récurrence sur $P(i, j)$.

▷
- *Si* $j = 0$ *alors* $P(i, j) = 1$ *(*$M = 0.p_1 + \cdots + 0.p_m$ *uniquement).*
- *Si* $i = 0$ *alors pas de pièces pas de représentations* $P(i, j) = 0$.
- *Si* $j < p_i$ *alors on ne va pas pouvoir utiliser* p_{i+1} *et* $P(i, j) = P(i-1, j)$.
- *Sinon* $P(i, j) = P(i, j - p_i) + P(i-1, j)$.

2. Ecrire alors un algorithme pour calculer $P(u, v)$

▷ *L'idée est de mémoiser les résultats intermédiaires. On commence par créer un tableau* $T[0 \cdots m-1, 0 \cdots k]$ *qu'on initialise partout à* -1.

```
int P(u,v) {
 int res;
 if (T[u,v]>-1) return T(u,v)
 (* on l'a déjà calculé *)
 if (v==0) res = 1
 else
     if (u==0) res = 0
     else if (u==1) && (v % i == 0)
          res = 1
          else if (u==1) && (v % i !=0)
               res = 0
               else if (v < p[u])
                     res = T[u-1][j]
                    else
                        res = T[u][v-p[i]] + T[u-1,v]
 T[u][v] = res; (* on mémoize *)
 return res; (* et on renvoie *)
}
```

3. Analysez votre algorithme.

▷ *L'initialisation coûte* $O(m \times k)$. *A chaque fois on teste* $v == 0$, *puis* $u == 0$ *puis* $(u == 1)\&\&(v\%i == 0)$, $(u == 1)\&\&(v\%i! = 0)$ *et enfin* $v < p[u]$ *: donc* $O(1)$ *tests par case du tableau : au total donc* $O(m \times k)$ *étapes globales.*

Exercice 2 : Dominos horizontaux.

Dans cette partie, on considère des dominaux horizontaux avec des lettres donc de la forme $\boxed{\text{a} \mid \text{b}}$. On définit alors une *chaîne* comme une suite dominos telles que les lettres voisines coïncident comme par exemple $\boxed{\text{a} \mid \text{c}}\boxed{\text{c} \mid \text{g}}\boxed{\text{g} \mid \text{g}}\boxed{\text{g} \mid \text{t}}$.

Dans cet exercice, on se donne en **entrée** n dominos (qu'on ne peut pas retourner) $D_1 = \boxed{x_1 \mid y_1}$, $\cdots$, $D_n = \boxed{x_n \mid y_n}$.

1. Ecrire un algorithme du type *backtracking* qui à partir des D_i trouve une chaîne si une telle chaîne existe et renvoie une erreur sinon.

▷

– *Comme les dominos ne peuvent pas tourner, on peut les représenter par deux tableaux* $x[1], \cdots, x[n]$ *et* $y[1], \cdots, y[n]$.

– *Soit alors chaîne*$[1 \cdots k]$ *la solution partielle qui contiendra les pièces de dominos formant une chaîne : la fonction chercher(k) cherche en profondeur une solution commençant par chaîne*$[1 \cdots k]$ *:*

```
int membre(int *liste, int i, int len) {
/* retourne vrai si i est dans la liste */
int c=0
for (;c<len;c++)
   if (liste[c]==i) return 1;
return 0;
}

int *pas_membre(int *liste, int a, int b, int len) {
/* renvoie les entiers entre a et b qui ne sont
   pas dans liste */
int *res=NULL;
int cur=0;
for (int i=a; i<=b;i++) {
   if (membre(liste,i,len)) {
      res=realloc(res,sizeof(int));
      res[cur++]=i;
   }
}
LEN=cur;
return res;
}

int compatible(int i, int j) {
/* renvoie 1 si le domino i peut être placé à gauche du domino j */
  return y[i] == x[j];
}

void chercher(int k) {
/* retourne une liste d'entiers formant la chaîne */
int *notmember;
if (k==n) /* chaine[1]...chaine[k] est déjà une chaîne */
   Afficher(chaine);
notmember = pas_membre(chaine,1,n,k);
for (int l=0;l<LEN;l++)
    if (k==0) || compatible(chaine[k],notmember[l]) {
        chaine[k+1] = notmember[l];
        chercher(k+1);
    }
}

/* main */
initialisez;
chercher(0);
```

– *Dans cette solution, une solution partielle est composée de* k *dominos parmi les* n *possibles. Une telle solution peut-être étendue à droite en rajoutant une pièce compatible non utilisée. On fait donc une recherche en profondeur dans l'arbre des solutions partielles en commençant par la racine (*0 *solution = vide). On stoppe quand* $k = n$ *car on a trouver une solution complète.*

Par exemple pour l'entrée
$\boxed{a \mid h}$ $\boxed{a \mid z}$ $\boxed{k \mid a}$ $\boxed{h \mid k}$ $\boxed{z \mid v}$
on aura comme suite d'exécutions

- $\emptyset$
- $\boxed{a \,/\, h}$
- $\boxed{a \,/\, h}\ \boxed{h \,/\, k}$
- $\boxed{a \,/\, h}\ \boxed{h \,/\, k}\ \boxed{k \,/\, a}$
- $\boxed{a \,/\, h}\ \boxed{h \,/\, k}\ \boxed{k \,/\, a}\ \boxed{a \,/\, z}$
- $\boxed{a \,/\, h}\ \boxed{h \,/\, k}\ \boxed{k \,/\, a}\ \boxed{a \,/\, z}\ \boxed{z \,/\, v}$

2. Ecrire un algorithme du type *programmation dynamique* qui trouve la plus longue sous-séquence de dominos (sans interchanger) qui forme une chaîne.

▷

- *Qui dit prog. dyn. dit : récurrence puis mémoïzation.*
- *Pour la récurrence, on définit $c(i)$ comme étant la longueur de la chaîne la plus longue pour l'entrée $D_1 D_2 \cdots D_i$. Nous avons*
  1. $c(1) = 1$
  2. *Pour $i > 1$, pour $j < i$ D_j est compatible avec D_i (on peut placer D_i à gauche de D_j) donc on peut prendre la chaîne la plus longue jusqu'à D_j ce qui donne pour $i > 1$ la récurrence*

$$c(i) = \max(1, c(j) + 1) \text{ où le max porte sur tous les } j < i \text{ tel que } j \text{ est compatible avec } i\,.$$

  3. *D'où un algo avec mémoïzation :*

```
c(1) = 1 ; sous-chaine[1]=0
m = 1
pour i de 2 à n faire
  pour j de 1 à i-1 faire
       si compatible(j,i) et c(j)+1 > m alors
          m=c(j)+1; sous-chaine[i] = j;
       finsi
  finpour
c(i) = m /* on mémoïze! */
finpour
```

Exercice 3 : MAX-3-COL.

Un graphe est dit 3-colorable si tous les sommets de ce graphe peuvent être colorés en rouge, vert, bleu sans que deux sommets adjacents ne soient voisins.
Dans cette partie, on considère le problème d'optimisation consistant à trouver dans un graphe le plus grand (grand en nombre de sommets et nombre d'arêtes) sous-graphe 3-colorable.

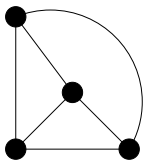
1. En vous inspirant du cours, définir une fonction Score portant sur les sommets et les arêtes d'un graphe pouvant résoudre ce problème d'optimisation.

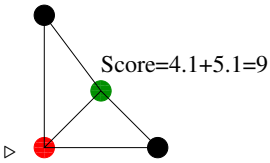
▷

- *Quand un graphe n'est pas 3-colorable, on peut toujours éliminer des arêtes pour qu'un sous-graphe 3-colorable émerge. On peut donc définir par exemple*

$$s(\emptyset) = 0 \text{ (Pour l'init.)}$$
$$s(\textit{rouge}) = s(\textit{vert}) = s(\textit{bleu}) = +1 \text{ (Pour les sommets.)}$$
$$s((u,v)) = -12 \text{ si } \textit{couleur}(u) = \textit{couleur}(v) + 1 \text{ sinon. (Pour les arêtes.)}$$

2. A partir du graphe g donné ci-après dessiner le plus grand sous-graphe sg 3-colorable "extrait" de g. Montrer que votre fonction Score le calcule effectivement.





▷

3. Dans le reste de l'exercice, on ne considère que les graphes de degrés 1, 2 et 3 (chaque sommet a 1 ou 2 ou 3 voisins). Décrire un algorithme qui calcule le score obtenu à partir du plus grand sous-graphe 3-colorable d'un graphe donné en entrée. Montrer sur un petit exemple comment votre algorithme calcule ce score.

▷

- *On mémoïze à partir des feuilles les gains si cette feuille a une couleur rouge/bleue/verte.*
- *On tombe sur un graphe réduit avec que des sommets de degré* ≥ 2 *: on mémoïze à nouveau en remplaçant chacune des séquences de sommets de degré* 2 *par une arête avec le score de cette arête.*
- *On branche avec* 3 *couleurs pour chaque sommet de degré* 3*.*

4. Quelle est la complexité de votre algorithme ?

▷ *On ne "branche" sur les sommets de degré* 3 *:*

$$O(3^{nbr\ sommets\ de\ degré\ 3} + Polynome(nbr\ total\ sommets))\,.$$