TD 2 (avec corrigé)

Essais successifs

Exercice 1. Les dominos

On considère des dominos formés de deux syllabes du français (par exemple : ME/TA, NE/RO, SE/RI. Pour assembler deux dominos, il faut que la seconde syllabe du premier et la première syllabe du second forment un mot de la langue française (on suppose disponible un dictionnaire permettant de savoir si un tel assemblage est ou non légal). Ainsi, NE/RO peut être suivi de SE/RI puisque ROSE est un mot français, de même que SE/RI peut être suivi de ME/TA. L'objectif est de déterminer toutes les suites complètes, i.e., utilisant n dominos donnés, commençant par un domino fixé.

Réponse. Eléments de conception selon le canevas donné en cours.

Initialisations du programme principal : X[1] ← domino initial de type doublet s1, s2 ; appel de la procédure avec i = 2

variables de travail : tableau de booléens utilisé ; vecteur X

vecteur : de longueur fixe (n); S_i : tous les dominos non encore utilisés; x_i : un domino

satisfaisant(x_i) : x_i non utilisée dans la "partie actuelle" et mot formé avec la X[i-1].s2 et x_i.s1

enregistrer : X[i] ← x_i ; domino x_i utilisé

soltrouvé : i = n

écriresolution : imprimer le tableau X

encorepossible : i < n

défaire : domino x_i non utilisé

L'algorithme est le suivant :

procédure dominos (ent i) ;
var ent x_i ;
début
  pour x_i **de** 1 **à** n **faire**
    si validarc(X[i – 1], x_i) **alors**
      si $x_i \notin$ pris **alors**
        X[i] ← x_i ;
        pris ← pris ∪ {x_i} ;

```
            si i = n alors écriresolution sinon dominos (i+1) fsi ;
            pris ← pris – {x_i}
        fsi
    fsi
  fait
fin
```

Appel : pris ← {1} ;
X[1] ← 1 ;
n ← ;
dominos(2) ;

Remarque : On cherche tous les chemins hamiltoniens issus de x_1.

Exercice 2. Plus court déplacement d'un cavalier de la case (x_d, y_d) à la case (x_a, y_a), ces deux cases étant supposées distinctes.

On s'intéresse aux parcours par un cavalier partant de la case (x_d, y_d) et devant se rendre à la case d'arrivée (x_a, y_a) et ce en un nombre minimal de déplacements

1. Donner les bornes inférieure et supérieure du nombre de trajets possibles.

Réponse.

On ne sait pas si ce problème admet une solution, mais dès qu'on en trouve une, il y en a une infinité car on peut "ajouter" autant de boucles que l'on veut à un parcours donné. Cette remarque préliminaire montre qu'on doit absolument modifier l'énoncé de façon à éliminer les parcours avec boucles (car sinon on a potentiellement une infinité de solutions et jamais un algorithme ne pourra les construire).

2. Faire l'analyse du problème posé en spécifiant les paramètres de la solution générique

Réponse.

Initialisations dans le programme principal : X[1] ← (x_d, y_d) ; nbdep-opt ← $n^2 + 1$ (ainsi, si cette valeur reste celle-là – test à effectuer en retour de l'appel de la procédure avant d'imprimer le vecteur Y – , cela signifie qu'on n'a pas trouvé de solution); appel de la procédure avec i = 2

variables de travail : tableau de booléens visité ; vecteur X (solution courante) et Y (meilleure solution optimale jusqu'ici) ;

Le vecteur représente un trajet (éventuellement partiel), c'est-à-dire les cases par lesquelles le cavalier est passé jusqu'ici; sa dimension maximale est donc n^2.

S_i représente les voisins "accessibles" depuis la case atteinte précédemment X[i-1], c'est-à-dire ceux des huit "voisins" se trouvant dans l'échiquier ; x_i est une case de type (abs, ord)

satisfaisant : la case sélectionnée est non visitée et elle se trouve dans l'échiquier

enregistrer : X[i] ← x_i; mettre la case x_i à "visitée" ; nbdepcour ← +1

soltrouvé : x_i = (x_a, y_a)

optimal : nbdep-cour < nbdep-opt

conserver-solution : Y ← X ; nbdepopt ← nbdepcour

opt-encore-possible : nbdep-cour + nbdep-futur-min < nbdepopt ;

co nbdep-futur-min est calculé en tenant compte du fait qu'on se déplace au plus vite de 2 lignes ou colonnes par déplacement ; autrement dit, on a :

nbdep-futur-min = max(sup(abs(x_i.abs – x_a.abs)/2), sup(abs(x_i.ord – x_a.ord)/2)) **fco**

défaire : mettre la case x_i à "non visitée" ; nbdepcour ← -1

3. Donner le code d'un algorithme associé à cette analyse

Réponse.

```
type case c'est struct ent abs, ord fstruct;
[1 : n * n] case X, Y

proc cavalier c'est fixe (ent i)
case xi; ent j, k;
début pour k de 1 à 8 faire
        co 8 = nombre de voisins fco;
        ci ← (X[i – 1].abs + A[k], X[i – 1].ord + B[k]);
        si xi.abs ∈ [1, n] et xi.ord ∈ [1, n] et non visité[xi.abs, xi.ord] alors
                visité[xi.abs, ci.ord] ← vrai;
                nbdep-cour ← nbdepcour +1 ;
                X[i] ← xi;
                si ci = (xa, ya) et nbdepcour < nbdepopt
                alors Y ← X ; nbdepopt ← nbdepcour
                sinon si nbdepcour + estim-futur(xi) < nbdepopt alors cavalier(i+1)
                        fsi;
                fsi ;
                visité[ci.abs, ci.ord] ← faux ; nbdepcour ← nbdepcour – 1 ;
        fsi
    fait
fin

appel : A ← (1, 1, 2, 2, -1, -1, -2, -2);
    B ← (2, -2, 1, -1, 2, -2, 1, -1);
    visité ← faux;
    lire(n, xd, yd, xa, ya); X[1].abs ← xd; X[1].ord ← yd;
     visité[X[1].abs, X[1].ord] ← faux; nbdepcour ← n*n – 1 ;
    cavalier(2);
    si nbdepopt < n^2 + 1 alors écrire(Y, nbdepopt) sinon écrire("pas de solution") fsi ;
```

Exercice 3. Séparation d'un tableau *T* d'entiers en deux tableaux de même taille dont la différence de somme est minimale.

On considère un tableau trié $T[1 : n]$ de n (n pair > 1) nombres entiers positifs distincts.

On désire partitionner *T* en deux sous-tableaux de taille $n/2$, appelés T_1 et T_2 tels que :

$S_1 = \Sigma_{i=1..n/2}\ T_1[i]$, $S_2 = \Sigma_{i=1..n/2}\ T_2[i]$, $S_1 \leq S_2$, $(S_2 - S_1)$ minimal

1. Montrer que le problème posé est équivalent à :

$S_1 = \Sigma_{i=1..n/2}\ T_1[i]$, $2*S_1 \leq S = \Sigma_{i=1..n}\ T[i]$, $(S - 2*S_1)$ minimal

Réponse.

L'équivalence est évidente car :

$S_1 \leq S_2 \Leftrightarrow 2 * S_1 \leq S_1 + S_2 \Leftrightarrow 2 * S_1 \leq S$

$(S_2 - S_1)$ minimal $\Leftrightarrow (S_2 + S_1 - (S_1 + S_1))$ minimal
$\Leftrightarrow (S - 2 * S_1)$ minimal.

2. Proposer une solution au problème posé par une procédure à essais successifs n'utilisant aucune heuristique et où le tableau T_1 est représenté par un tableau de 0/1 ($T_1[k] = 1$ signifie que $T[k]$ est utilisé dans la solution). On supposera disponible dans *S* la somme des éléments du tableau *T*.

Réponse.

Les éléments d'analyse sont les suivants :

satisfaisant : 2 * (somcour + xi * T[i]) ≤ S

enregistrer : somcour ← somcour + xi * T[i]
nombredenombres ← nombredenombres + xi
T1[i] ← xi

soltrouvé : nombredenombre = n div 2 et somcour > somopt

optencorepossible : i < n et nombredenombres < n div 2

défaire : somcour ← somcour – xi * T[i]
nombredenombres nombredenombres - xi

La trame d'une procédure est la suivante :

```
proc part-ecart-mini c'est fixe (ent i)
var ent j, k;
début pour j de 0 à 1 faire
        k ← T[i] * j;
        si 2* (somcour + k) ≤ S
```

```
        alors somcour ← somcour + k;
                nbnombre ← nbnombre + j;
                T1[i] ← j;
                si nbnombre = n div 2 et i = n
                alors si somcour > somopt
                        alors gardersolution
                        fsi
                sinon si i < n et nbnombre < n div 2
                         alors part-ecart-mini(i + 1)
                         fsi
                fsi;
                somcour ← somcour – k;
                nbnombre ← nbnombre – j
        fsi
        fait
fin
```

Remarque 1 : on pourrait modifier soltrouvé de sorte que dès qu'on a n **div** 2 nombres on regarde l'optimalité de la solution; dans ce cas gardersolution doit compléter T1 avec des 0 de i+1 à n.

Remarque 2 : on pourrait séparer le cas 0 du cas 1.

3. Proposer le principe de conditions d'élagage améliorant la procédure précédente.

Réponse.

On va gérer une variable k_1 contenant le nombre de nombres restant à prendre (au départ (n **div** 2)). Il est nécessaire que :

a) il reste encore au moins k_1 nombres possibles $(n – i + 1) \geq k_1$

b) somcour + somme des k_1 plus petits nombres encore possibles (T[i] à T[i + k_1 – 1]) reste inférieure à S **div** 2;

c) somcour + somme des k_1 nombres les plus grands encore possibles (T[n – k_1 + 1] à T[n]) permette de dépasser s **div** 2.

Exercice 4.

On cherche à résoudre des problèmes qui se présentent sous la forme d'une opération arithmétique sur des lettres. Le but est d'affecter un chiffre (0 à 9) à une lettre, sous la condition que le même chiffre ne soit pas affecté à deux lettres, et qu'une lettre vaille toujours le même chiffre. Le résultat de cette affectation doit fournir une opération arithmétiquement correcte en base 10.
Par exemple le problème de crypto-arithmétique :

NEUF + UN + UN = ONZE

que l'on peut aussi écrire :

```
 N E U F
       +
   U N
       +
   U N
-----------
 O N Z E
```

a une solution que l'on peut représenter par :

(N = 1 , E = 9 , U = 8 , O = 2 , Z = 4 , F = 7)

parce que :

1987 + 81 + 81 = 2149

Les trois exemples suivants montrent des affectations incorrectes :

– La proposition 1988 + 81 + 81 = 2150 suppose : U = F = 8
– La proposition 1986 + 82 + 82 = 2150 suppose que 1 et 2 sont affectés à N et 0 et 9 à E.
– Quant à l'affectation (N = 2 , E = 9 , U = 8 , O = 3 , Z = 4 , F = 7), qui correspond à

2987 + 82 + 82 = 3242,

elle est arithmétiquement fausse.

On supposera qu'il existe une procédure *CalculExact* qui, appelée pour une affectation des lettres, rend *VRAI* si l'opération est arithmétiquement correcte et *FAUX* sinon.

1. Donner le principe d'un algorithme permettant de trouver toutes les solutions à tout problème de crypto-arithmétique. On emploiera la méthodologie « essais successifs », sans chercher d'élagages.

Principe : pour chaque lettre différente du problème, on essaie de lui affecter un chiffre, en prenant les chiffres les uns après les autres par essais successifs.

2. Définir les paramètres du programme : le vecteur de la récursion, sa taille, le domaine de chaque composante, les procédures *Satisfaisant*, *SolTrouvée*, *Enregistrer* et *Défaire*.

Vecteur X : sa taille est le nombre de lettres différentes qui apparaissent dans le problème, soit n. Dans l'exemple, n = 6.
Domaine de chaque composante de X : les chiffres de 0 à 9.
Satisfaisant : le chiffre courant n' a pas encore été affecté.
On va gérer un ensemble des chiffres déja affectés : E.
SolTrouvée : i = n et *CalculExact* = VRAI

3. Ecrire le pseudo-code.

```
procedure TouteSol(i)
début
   pour x_i de 0 à 9 faire
      si x_i ∉ E alors
         E = E + {x_i}
         X[i] = x_i
         si i = n et CalculExact= VRAI alors
            EcrireSol
         sinon
            si i < n alors
               TouteSol (i+1)
            fsi
         fsi
         E = E − {x_i}
      fsi
   fait
fin
```

Lancement du programme : E = vide ; *TouteSol*(1)
Ce programme demande (au pire) n^{10} appels récursifs.

4. Pour un problème particulier comme celui donné ci-dessus, comment pourrait-on utiliser la procédure *SolEncorePossible* pour faire des élagages ? Vous paraît-il facile de trouver une technique générale pour l'utiliser ?

On peut rechercher des contraintes propres au problème et construire la procédure *SolEncorePossible* à partir d'elles. Dans notre exemple, les contraintes O = N +1, N = E +2−10 apparaissent assez facilement mais la contrainte U > 6 est indirecte.
Il est difficile d'imaginer un système pour extraire systématiquement ce genre contraintes : elles se présentent rarement comme des équations indépendantes.

Variante
On peut permuter les rôles des lettres et des chiffres. C'est alors un peu différent, car certains chiffres n'ont pas forcément de lettre correspondante. Il faut donc prévoir le cas où aucune lettre n'est affectée à un chiffre.
Il s'agit dans ce cas d'un algorithme par essais successifs où le vecteur peut comporter des « trous ».
On obtient une complexité au pire de 10^{27}.
Cette variante est moins intéressante.

Exercice 5. Le jeu du taquin.

On considère un taquin n × n. On cherche à déterminer une façon optimale (nombre de coups minimal) pour aller de la configuration initiale à la configuration finale.

1. Ecrire un algorithme à essais successifs sans élagage.

Attention aux circuits : il faut mémoriser les différentes configurations de la branche courante.

solution = liste chainée de configurations

S_i = 4 mouvements possibles (L, R, U, D)

satisfaisant(x_i) : (le trou reste à l'intérieur du taquin) et (on ne revient pas à une configuration déjà rencontrée dans la branche courante)

enregistrer(x_i) : on ajoute la nouvelle configuration à la fin de la liste ; nbcoups ← nbcoups + 1

soltrouvée : configuration atteinte = configuration cible

meilleure : nbcoups < nbopt

encorepossible : nbcoups < nbopt +1

défaire : on enlève la dernière configuration de la queue de la liste ; nbcoups ← ncoups – 1

Raffinement : pour S_i, on peut ne considérer que trois mouvements et non quatre en éliminant le symétrique du dernier mouvement effectué (ce qui implique de conserver celui-ci dans une variable).

2. Introduire une (ou des) condition(s) d'élagage.

Soit nbopt l'optimal courant.
Soit nbcoups le nombre courant de déplacements (du trou) effectués.
Premier critère :
si nbcoups + nbre de cases mal placées ≥ nbopt, il n'y a pas d'espoir de faire mieux.
Raffinement :
si (nbcoups + somme des distances de Manhattan associées à chaque case) ≥ nbopt, c'est fichu.

Rappel :

$d_{Manhattan}(A, B) = | X_B – X_A | + | Y_B – Y_A |$.