Mines-Ponts 2012. Option Informatique.

Corrigé pour serveur UPS par JL. Lamard (jean-louis.lamard@prepas.org)

Exercice sur les langages rationnels.

1) L_1 est décrit par l'expression rationnelle $\Sigma^* f \Sigma^*$ □

2) L_2 est décrit par l'expression rationnelle $\Sigma^* f \Sigma^* f \Sigma^*$ □

3) Soient L_p et L_s les langages ayant le mot f respectivement en préfixe et suffixe.
Ils sont rationnels car décrits respectivement par les expressions rationnelles $f\Sigma^*$ et $\Sigma^* f$.
Donc $L_p \cap L_s$ est rationnel en tant qu'intersection de deux langages rationnels.
De même que le langage L réduit au mot f et $L \cup L_2$ en tant qu'union de deux langages rationnels.
Donc $L_{ps} = (L_p \cap L_s) \setminus (L + L_2)$ également en tant que complémentaire d'un langage rationnel dans un autre.
L_{ps} est le langage des mots formés par exactement deux occurences imbriquées du mot f.

Il en résulte que $L_3 = \Sigma^* L_{ps} \Sigma^*$ est bien rationnel. □

4) $L_4 = L_1 \setminus (L_2 + L_3)$ donc est rationnel. □

Problème d'algorithmique et programmation.

Généralités.

5) Les arêtes $\{0_A, 1_B\}$, $\{1_A, 3_B\}$ et $\{2_A, 2_B\}$ forment un couplage de cardinal 3. □

6) Supposons qu'il existe un couplage de cardinal 4. Alors tous les sommets de A sont couplés. Or 1_A et 3_A n'admettent qu'un seul voisin qui est commun : 3_B. Donc ce couplage contient 2 arêtes incidentes en 3_B. Contradiction. □

7)

```
let verifie G C = let n = vect_length C and ok = ref true and i = ref 0 in
  while (!i <= (n-1)) && (!ok) do
    if C.(!i) = -1 then incr i
    else if not G.(!i).(C.(!i)) then ok := false
    else let k = ref 0 in
      while (!k < !i) && (C.(!k) <> C.(!i)) do incr k done;
      ok := (!k = !i);
      incr i
  done;
  !ok
;;
verifie : bool vect vect -> int vect -> bool = <fun>
```

La complexité minimale est à temps constant atteinte lorsque $C.(0) = j \neq -1$ et que $G.(0).(j) = false$.
La complexité maximale est quadratique et atteinte lorsque C décrit un couplage de cardinal n. En effet la boucle *while* sur i se poursuit jusqu'à $i = n - 1$ et le tour d'indice i comporte i comparaisons. □

8)

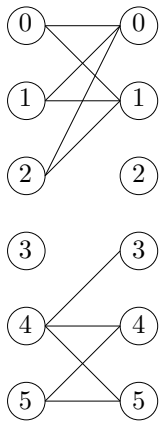
```
let cardinal C = let n = vect_length C and s = ref 0 in
  for i = 0 to (n-1) do
    if (C.(i) <> -1) then incr s
  done;
  !s
;;
cardinal : int vect -> int = <fun>
```

La complexité est évidemment linéaire. □

Un algorithme pour déterminer un couplage maximal.

9) Les arêtes de somme minimum 4 sont $(1,3)$ et $(3,3)$. On choisit par exemple $(1,3)$.
Il reste alors le graphe $\Big((0_A, 2_A, 3_A), (0_B, 1_B, 2_B)\Big)$ avec les arêtes (0,0), ((0,1), (0,2), (2,0), (2,1) et (2,2).
Toutes les arêtes ont pour somme 5 et on choisit par exemple d'éliminer $(0,0)$.
Reste le graphe $\Big((2_A, 3_A), (1_B, 2_B)\Big)$ avec les arêtes (2,1) et (2,2) toutes deux de somme 3.
On élimine par exemple (2,1).
Reste alors le graphe $(3_A, 2_B)$ sans arête et l'algorithme est terminé.
On obtient ainsi le couplage $\Big((1,3), (0,0), (2,1)\Big)$ et on constate que ce couplage est bien maximal. □

10)



L'unique arête de somme minimum est l'arête (3,2) de somme 4.
En la supprimant on obtient le graphe G_2 ci-contre.

Ce graphe est la réunion "disjointe" de deux graphes birépartis (non équilibrés) $G_3 = \Big((0_A, 1_A, 2_A), (0_B, 1_B)\Big)$ et $G_4 = \Big((4_A, 5_A), (3_B, 4_B, 5_B)\Big)$.
Il en découle que tout couplage maximal de G_2 est la réunion d'un couplage maximal de G_3 et d'un couplage maximal de G_4 lesquels sont évidemment de cardinal au plus 2. Or G_3 et G_4 admettent chacun un couplage maximal de cardinal 2 : $C_3 = \Big((0,0),(1,1)\Big)$ et $C_4 = \Big((4,4),(5,5)\Big)$.
Il en découle que *algo_approche* va renvoyer un couplage maximal de cardinal 5. □

Or G_1 admet le couplage $\Big((0,0),(1,1),(2,2),(3,3),(4,4),(5,5)\Big)$ de cardinal 6 clairement maximal.

On savait déjà avec l'exemple du graphe G_0 et du couplage C_0 qu'un couplage maximal n'est pas forcément de cardinal maximum parmi l'ensemble de tous les couplages.

Cet exemple montre que *algo_approche* ne fournit pas non plus forcément un couplage de cardinal maximum parmi l'ensemble des couplages maximals.

11) On parcourt la matrice décrivant le graphe (coût quadratique).

À chaque arête rencontrée on calcule sa somme qui est la somme de tous les "true" de sa ligne et de sa colonne (coût linéaire). Reste à retenir l'arête minimum comme indiqué (on initialise la variable min à $2n+1$ valeur forcément supérieure à la somme de toute arête).

Le coût est donc cubique. □

```
let arete_min G a = let n = vect_length G and ok = ref false in
  let min = ref (2*n+1) in
    for i=0 to n-1 do
      for j=0 to n-1 do
        if G.(i).(j) then let s = ref 0 in
          for k=0 to n-1 do s := !s + (if G.(i).(k) then 1 else 0) done;
          for k=0 to n-1 do s := !s + (if G.(k).(j) then 1 else 0) done;
          if (!s < ! min) then
            begin min := !s; a.(0) <- i; a.(1) <- j ; ok := true end
      done
    done;
    !ok
;;
arete_min : bool vect vect -> int vect -> bool = <fun>
```

12) Il suffit de mettre partout des $false$ dans la ligne et la colonne de l'élément correspondant à l'arête à supprimer. Le coût est évidemment linéaire. □

```
let supprimer G a = let n = vect_length G in
  for i=0 to n-1 do G.(i).(a.(1)) <- false done;
  for j=0 to n-1 do G.(a.(0)).(j) <- false done
;;
supprimer : bool vect vect -> int vect -> unit = <fun>
```

13)

```
let dupliquer G  = let n = vect_length G in
  let A = make_vect n (make_vect n true) in
    for i=0 to n-1 do A.(i) <- copy_vect G.(i) done;
    A
;;
duppliquer : bool vect vect -> bool vect vect = <fun>
```

```
let algo_approche G = let n = vect_length G in
  let A = dupliquer G and a = [|-1; -1|] and C = make_vect n (-1) in
    while arete_min A a do
      supprimer A a;
      C.(a.(0)) <- a.(1)
    done;
  C
;;
algo_approche : bool vect vect -> int vect = <fun>
```

Recherche exhaustive d'un couplage de cardinal maximum.

14) On commence par écrire une fonction `une_areteA` de sorte que `une_areteA G xA` où `xA` est un sommet de `A` renvoie -1 si `xA` n'admet aucun voisin et sinon le numéro du premier voisin de `xA`.

```
let une_areteA G xA =
  let n = vect_length G and encore = ref true and xB = ref 0 in
    while ((!encore) && (!xB < n)) do
      if (G.(xA).(!xB) = false) then incr xB
      else encore := false
    done;
    if (!xB = n) then -1 else !xB
;;
une_areteA : bool vect vect -> int -> int = <fun>
```

L'écriture de la fonction demandée est alors immédiate.

```
let une_arete G a =
  let n = vect_length G and encore = ref true and xA = ref 0 in
    while ((!encore) && (!xA < n)) do
      let xB = une_areteA G !xA in
        if (xB = -1) then incr xA
        else begin a.(0) <- !xA; a.(1) <- xB; encore := false end
    done;
    (!xA < n)
;;
une_arete : bool vect vect -> int vect -> bool = <fun>
```

15) Lorsque l'état courant du graphe contient au moins une arête a, on réalise deux copies du graphe courant.
Dans l'une, notée $A1$, on supprime l'arête a et toutes ses arêtes incidentes et dans l'autre, notée $A2$, on supprime uniquement l'arête a.
On appelle récursivement la fonction sur ces deux graphes. Comme ils possèdent chacun au moins une arête en moins cela assure la terminaison de la récursion sur le cas de base où il n'y a pas d'arête.
En ajoutant l'arete a au couplage renvoyé par l'appel de la fonction sur $A1$ on obtient un couplage maximum parmi ceux qui contiennent l'arête a.
L'appel de la fonction sur $A2$ renvoie un couplage maximum parmi ceux qui ne contiennent pas l'arête a.
Il reste à choisir celui de ces deux couplage de cardinal maximum.

```
let rec meilleur_couplage G =
  let n = vect_length G and a = [|-1; -1|] in let C = make_vect n (-1) in
    match une_arete G a with
      | false -> C
      | _ -> let A1 = dupliquer G and A2 = dupliquer G in
               supprimer A1 a;
               A2.(a.(0)).(a.(1)) <- false;
               let C1 = meilleur_couplage A1
               and C2 = meilleur_couplage A2 in
                 if (cardinal C2 > (cardinal C1) + 1) then C2
                 else begin C1.(a.(0)) <- a.(1); C1 end
;;
meilleur_couplage : bool vect vect -> int vect = <fun>
```

L'algorithme hongrois.

16) Le seul sommet de A non couplé est 2_A et le seul de B non couplé est 4_B.
Donc s'il existe une chaîne alternée augmentante son origine est $x_0 = 2_A$ et son extrémité 4_B.
On vérifie alors que $(2_A, 2_B, 3_A, 3_B, 4_A, 4_B)$ est une chaîne alternée augmentante de $C1$. □

17) Supposons que $(x_0, x_1, \ldots, x_{2p+1})$ avec $p \geqslant 0$ soit une chaîne alternée augmentante relativement à un couplage C.
Soit le couplage C' obtenu à partir de C en supprimant les p arêtes (x_{2k+2}, x_{2k+1}) pour $0 \leqslant k < p$ si $p \geqslant 1$
Les sommets intervenant dans C' ne comprennent évidemment pas les sommets des arêtes retirées c'est à dire x_1, x_2, …, x_{2p}.
Ils ne comprennent pas non plus x_0 ni x_{2p+1} car x_0 et x_{2p+1} ne sont pas couplés dans C par définition d'une chaîne alternée augmentante relativement à C.
Ainsi en ajoutant les $p+1$ arêtes (x_0, x_1), (x_2, x_3), …, (x_{2p}, x_{2p+1}) à C', on obtient bien un nouveau couplage C^+.
Or $\text{card}(C^+) = \text{card}(C) + 1$ □

Dans l'exemple de la question précédente cela fournit le couplage
$C_1^+ = \Big((0_A, 0_B), (1_A, 1_B), (2_A, 2_B), (3_A, 3_B), (4_A, 4_B), (5_A, 5_B)\Big)$ de cardinal 6.

18) En partant de l'extrémité 3_B, on remonte la chaîne grâce aux marques.
On obtient ainsi la chaîne alternée $(1_A, 0_B, 0_A, 1_B, 2_A, 2_B, 3_A, 3_B)$.
Comme 3_B n'est pas couplé dans C_1', il s'agit bien d'une chaîne alternée augmentante de C_1'. □

La question précédente fournit alors le couplage $C_1'^+ = \Big((0_A, 1_B), (1_A, 0_B), (2_A, 2_B), (3_A, 3_B), (4_A, 4_B), (5_A, 5_B)\Big)$

19) On remonte la chaîne augmentante grâce aux marques en partant de l'extrémité et, suivant la méthode expliquée dans la question 17, chaque fois qu'on obtient un couple (x_A, x_B) de cette chaîne, on modifie $C(x_A)$ et $R(x_B)$ de manière à inclure l'arête (x_A, x_B) dans le couplage.
On poursuit tant que la marque de x_A est différente de -1.

```
let actualiser C R mA mB numero =
  let xB = ref numero and xA = ref (mB.(numero)) in let marqueA = ref mA.(!xA) in
    C.(!xA) <- !xB;
    R.(!xB) <- !xA;
    while (!marqueA <> -1) do
      xB := !marqueA;
      xA := mB.(!xB);
      C.(!xA) <- !xB;
      R.(!xB) <- !xA;
      marqueA := mA.(!xA)
    done
;;
actualiser : int vect -> int vect -> int vect -> int vect -> int -> unit = <fun>
```

20) En partant de 1_A, on peut atteindre 3_B ou 4_B
On obtient les deux seules possibilités suivantes : $(1_A, 3_B, 3_A, 4_B, 4_A)$ et $(1_A, 4_B, 4_A, 3_B, 3_A)$.
On obtient les mêmes sommets dans un ordre différent et aucune des deux chaînes n'est augmentante. □

21) On commence par écrire une fonction `voisinA` de sorte que `voisinA G C R mA mB xA` où `xA` est le numéro d'un sommet de A renvoie :
- le premier numéro d'un voisin de xA non encore atteint et non couplé dans C s'il en existe au moins un;
- sinon le premier numéro d'un voisin de xA non encore atteint et non couplé avec xA s'il en existe au moins un;
- sinon -1.

```
let voisinA G C R mA mB xA =
  let xB = ref (-1) and j = ref 0 and encore = ref true in
  while ((!j < vect_length C) && !encore) do
    if ((G.(xA).(!j)) && (C.(xA) <> !j) && (mB.(!j) = -1)) then
      if (R.(!j) = -1) then begin xB := !j; encore := false end
      else if (!xB = -1) then begin xB := !j ; incr j end
           else incr j
    else incr j
  done;
  !xB
;;
voisinA : bool vect vect -> int vect -> int vect -> 'a -> int vect -> int -> int = <fun>
```

Les deux fonctions `chercheA` et `chercheB` s'écrivent alors facilement en récursivité croisée :

```
let rec chercheB G C R mA mB xB = match R.(xB) with
  | -1 -> xB
  | xA -> mA.(xA) <- xB;
          chercheA G C R mA mB xA
and chercheA G C R mA mB xA = let xB = voisinA G C R mB xA in
  if (xB = -1) then -1
  else begin mB.(xB) <- xA; chercheB G C R mA mB xB end
;;
chercheB : bool vect vect -> int vect -> int vect -> int vect -> int vect -> int -> int = <fun>
chercheA : bool vect vect -> int vect -> int vect -> int vect -> int vect -> int -> int = <fun>
```

22) On parcourt dans l'ordre les sommets de A jusqu'à trouver un éventuel sommet non couplé origine d'une chaîne alternée augmentante.

```
let chaine_alternee G C R mA mB =
  let n = vect_length C and xA = ref 0 and extremite = ref (-1) in
    while (!xA < vect_length C) && (!extremite = -1) do
      if (C.(!xA) = -1) then
        begin
          for i=0 to (n-1) do mA.(i) <- -1 done;
          for i=0 to (n-1) do mB.(i) <- -1 done;
          let num = chercheA G C R mA mB !xA in
              if (num <> -1) then extremite := num else incr xA;
        end
      else incr xA
    done;
    !extremite
;;
chaine_alternee : bool vect vect -> int vect -> int vect -> int vect -> int vect -> int = <fun>
```

23) On commence par initialiser les deux tableaux C et R avec la valeur -1.
Puis tant qu'une éventuelle chaîne augmentante peut être trouvée on initialise de même les deux tableaux de marques mA et mB et on cherche avec la fonction `chaine_alternee` une chaîne augmentante.
Si on en trouve une, on actualise les tableaux C et R grâce à la fonction `actualiser` et on recommence.
Sinon on s'arrête et on renvoie l'état actuel du tableau C.

Naturellement l'algorithme s'arrête bien car à chaque tour de boucle le cardinal du couplage augmente de une unité de sorte que la boucle effectue au plus n tours.

```
let algorithme_hongrois G = let n = vect_length G in
  let C = make_vect n (-1) and R = make_vect n (-1) and encore = ref true in
    while !encore do
      let mA = make_vect n (-1) and mB = make_vect n (-1) in
      let num = chaine_alternee G C R mA mB in
        if (num <> -1) then actualiser C R mA mB num
        else encore := false
    done;
    C
;;
algorithme_hongrois : bool vect vect -> int vect = <fun>
```

FIN