

# Corrigé de l'épreuve d'informatique du concours CCINP2025, filière MPI

Florian Bourse (Lycée Colbert - Tourcoing)

May 8, 2025

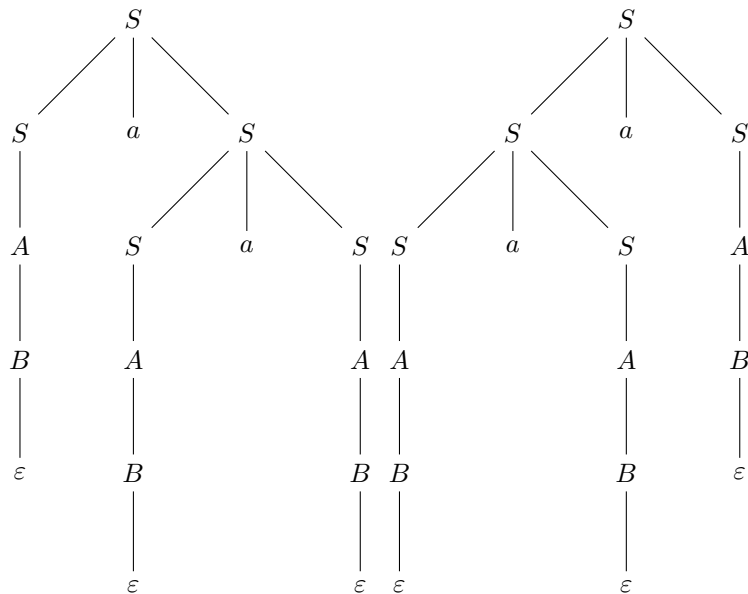
## I - Grammaire non contextuelle

1.

$$\begin{aligned} S &\Rightarrow SaS \Rightarrow AaS \Rightarrow BaS \Rightarrow aS \Rightarrow aA \Rightarrow aAbA \Rightarrow aBbA \\ &\Rightarrow abA \Rightarrow abB \Rightarrow abBcB \Rightarrow abcB \Rightarrow abc = u. \end{aligned}$$

$u$  admet une dérivation gauche donc il appartient au langage engendré par la grammaire  $G$ .

2. Voici deux arbres de dérivation pour  $aa$ . Par définition,  $G$  est ambiguë car un mot possède deux arbres de dérivations (ou de manière équivalente, deux dérivations gauches) différents.



3.  $S$ ,  $A$ , et  $B$ , sont récursives gauches.

4.  $G'$ , obtenue à partir de  $G$  en appliquant l'algorithme donné, est définie par les règles de production suivantes :

$$\begin{aligned} S &\rightarrow AS' \\ S' &\rightarrow aSS' \mid \varepsilon \\ A &\rightarrow BA' \\ A' &\rightarrow bAA' \mid \varepsilon \\ B &\rightarrow B' \\ B' &\rightarrow cBB' \mid \varepsilon \end{aligned}$$

5.  $\supseteq$  : aucun symbol non terminal autre que  $a$ ,  $b$  ou  $c$  n'apparaît dans les règles de production de  $G'$ , donc le langage engendré par  $G'$  est inclus dans  $\{a, b, c\}^*$ .

$\subseteq$  : Montrons par récurrence la propriété  $P(n)$  définie par :

$$\forall u, |u| \leq n,$$

$$\text{Si } u \in \{c\}^*, \text{ alors } B \Rightarrow^* u,$$

$$\text{Si } u \in \{b, c\}^*, \text{ alors } A \Rightarrow^* u,$$

$$\text{Si } u \in \{a, b, c\}^*, \text{ alors } S \Rightarrow^* u.$$

$$P(0) : B \Rightarrow B' \Rightarrow \varepsilon,$$

$$A \Rightarrow BA' \Rightarrow^* A' \Rightarrow \varepsilon,$$

$$S \Rightarrow AS' \Rightarrow^* S' \Rightarrow \varepsilon.$$

$P(n)$  implique  $P(n+1)$  : Soit  $u$  de taille  $n+1$ .

$$\text{Si } u = c^{n+1}, B \Rightarrow B' \Rightarrow cBB' \Rightarrow cB \Rightarrow^* cc^n = c^{n+1} \text{ (HR)}$$

$$\text{De plus, } A \Rightarrow BA' \Rightarrow B \Rightarrow^* u, \text{ et } S \Rightarrow AS' \Rightarrow A \Rightarrow^* u$$

$$\text{Si } u \in \{b, c\}^* \setminus \{c\}^*, \exists (u_1, u_2) \in \{c\}^* \times \{b, c\}^*, u = u_1bu_2.$$

$$\text{On a } A \Rightarrow BA' \Rightarrow BbAA' \Rightarrow BbA \Rightarrow^* u_1bA \Rightarrow^* u_1bu_2 \text{ (HR).}$$

$$\text{De plus, } S \Rightarrow AS' \Rightarrow A \Rightarrow^* u.$$

$$\text{Si } u \in \{a, b, c\}^* \setminus \{b, c\}^*, \exists (u_1, u_2) \in \{b, c\}^* \times \{a, b, c\}^*, u = u_1au_2.$$

$$\text{On a } S \Rightarrow AS' \Rightarrow AaSS' \Rightarrow AaS \Rightarrow^* u_1aS \Rightarrow^* u_1au_2. \text{ (HR).}$$

## II - Problème de bin\_packing

6. Si  $(n, k, a)$  est une solution valide, alors il existe un certificat  $\mathcal{R}$  de taille polynomiale ( $|\mathcal{R}| = |a|$ ) que l'on peut vérifier en temps polynomial :

```

verifie(n,k,a,R):
  Pour j = 1 .. n :
    Si R(j) > n ou R(j) < 1 :
      Renvoyer F
  Pour i = 1 .. k :
    acc = 0
    Pour j = 1 .. n :
      Si R(j) = i :
        acc = acc + a[j]
    Si acc > 1 :
      Renvoyer F
  Renvoyer V

```

Si le certificat  $\mathcal{R}$  existe, alors  $(n, k, a)$  admet une solution valide.

Donc BIN-PACKING  $\in$  NP.

7. Soit  $c_1, \dots, c_n$ ,  $n$  entiers. Pour  $i \in \{1, \dots, n\}$ , on pose  $a_i = \frac{2c_i}{\sum_{j=1..n} c_j}$ .  
 $c \mapsto (n, 2, a)$  est une réduction de PARTITION à BIN-PACKING :  
 Si  $(c_i)_{i=1..n}$  admet une solution  $S$ , alors  $\mathcal{R} : i \mapsto 1$  si  $i \in S$ , 2 sinon.  
 On a  $\sum_{i \in S} c_i = \sum_{i \notin S} c_i = \frac{1}{2} \sum_{i=1..n} c_i$ , donc  $\sum_{i \in S} a_i = \sum_{i \notin S} a_i = 1$ .  
 Donc  $(n, 2, a)$  admet une solution  $\mathcal{R}$ .  
 Réciproquement, si  $\sum_{i \in B_1} a_i \leq 1$  alors  $\sum_{i \in S} c_i \leq \frac{1}{2} \sum_{i=1..n} c_i$ .  
 De même,  $\sum_{i \notin S} c_i \leq \frac{1}{2} \sum_{i=1..n} c_i$ .  
 Donc  $\sum_{i \in S} c_i = \sum_{i=1..n} c_i - \sum_{i \notin S} c_i \geq \frac{1}{2} \sum_{i=1..n} c_i$ .  
 On donc  $\sum_{i \in S} c_i = \frac{1}{2} \sum_{i=1..n} c_i$ , et de même pour  $\sum_{i \notin S} c_i$ .  
 Donc  $(c_i)_{i=1..n}$  admet une solution.  
 De plus,  $c \mapsto (n, 2, a)$  se calcule en temps polynomial  
 donc PARTITION  $\leq_P$  BIN-PACKING.  
 De plus, PARTITION est NP-complet, donc NP-dur,  
 donc BIN-PACKING est NP-dur.  
 Or, on a montré qu'il était dans NP, donc il est NP-complet.
8. Si  $B_{k_1}$  contient un objet de taille  $a_i > \frac{1}{2}$ , alors chaque boîte  $B_j$ ,  $j < k_1$  contiennent un objet de taille supérieure à  $a_i > \frac{1}{2}$ .  
 On ne peut pas placer deux tels objets dans une même boîte, donc  $k^* \geq k_1$ .  
 $k_1 \geq \frac{2}{3}k$  est donné par la définition de  $k_1 = \lceil \frac{2}{3}k \rceil$ .
9. Sinon, chacune des boîtes  $B_{k_1}, \dots, B_{k-1}$  contient au moins 2 objets car toute combinaison de 2 objets peut tenir dans une boîte si le poids de chacun est inférieur à  $\frac{1}{2}$ .  
 De plus, la boîte  $k$  contient au moins 1 objet.  
 Les boîtes  $B_{k_1}, \dots, B_k$  contiennent donc au moins  $2(k-1-k_1+1)+1 = 2(k-k_1)+1$  objets.  
 Aucun d'eux ne peut entrer dans les boîtes  $B_1, \dots, B_{k_1-1}$  sinon l'**algorithme 2** les y aurait placé.
10. Soit  $k_2 = \min(2(k-k_1)+1, k_1-1)$ .  
 Si on prend  $k_2$  des  $2(k-k_1)+1$  objets de la question précédente et qu'on les ajoute à  $k_2$  des  $k_1-1$  boîtes  $B_0, \dots, B_{k_1-1}$ , on a  $k_2$  boîtes dont la taille est supérieure strictement à 1.  
 Donc  $\sum_{i=1..n} a_i > k_2$ .  
 On aurait donc besoin de au moins  $k_2+1$  boîtes pour répondre au problème :  
 $k^* \geq k_2 + 1$ .  
 Si  $k_2 = k_1 - 1$  alors  $k^* \geq k_1 = \lceil \frac{2}{3}k \rceil \geq \frac{2}{3}k$ .

Si  $k_2 = 2(k - k_1) + 1$  alors

$$\begin{aligned}
 k^* &\geq 2(k - k_1 + 1) \\
 &\geq 2(k - \left\lceil \frac{2}{3}k \right\rceil + 1) \\
 &\geq 2(k - \frac{2}{3}k - \frac{2}{3} + 1) \\
 &\geq \frac{2}{3}k + \frac{2}{3} \\
 &\geq \left\lceil \frac{2}{3}k \right\rceil \geq \frac{2}{3}k
 \end{aligned}$$

11. Il n'y a qu'à initialiser un élément d'un type enregistrement :

```
let boite_vide = {charge = 0.; objets = []}
```

12. Idem, en accédant aux champs d'un élément :

```
let ajoute_boite o b =
  {charge = b.charge +. o.taille;
   objets = o :: b.objets}
```

13. On parcourt la liste jusqu'au premier élément qui vérifie une condition :

```
let rec trouve_boite bl o = match bl with
| [] -> 0
| t :: q -> if t.charge +. o.taille < 1.
              then 0
              else 1 + trouve_boite q o
```

14. Encore un parcours de liste :

```
let rec transforme d f i l = match l with
| [] -> [f d]
| t :: q -> if i = 0
              then f t :: q
              else t :: transforme d f (i-1) q
```

15. Remarque : contrairement à la description de l'**algorithme 1**, l'implémentation donnée ci-dessous commence par traiter les éléments en fin de liste et termine par la tête. Ce qui nous arrange car nous appellerons à la place l'**algorithme 1** sur une liste triée dans l'ordre croissant plutôt que décroissant dans la fonction suivante.

```
let rec premier_casier ol = match ol with
| [] -> []
| t :: q ->
  let bl = premier_casier q in
  transforme boite_vide (ajoute_boite t) (trouve_boite bl t) bl
```

16. Il ne reste qu'à appliquer la fonction précédentes et celles rappelées dans l'énoncé :

```

let rec premier_casier_decroissant ol =
  premier_casier (List.sort compare ol)

```

### III - Algorithmes de couplage

17.  $A = \emptyset$   
 $A = \{(v_1, u_2)\}$   $v_1$  propose à  $u_2$   
 $A = \{(v_1, u_2), (v_2, u_1)\}$   $v_2$  propose à  $u_1$   
 $A = \{(v_1, u_2), (v_3, u_1)\}$   $v_3$  propose à  $u_1$   
 $A = \{(v_2, u_2), (v_3, u_1)\}$   $v_2$  propose à  $u_2$   
 $A = \{(v_2, u_2), (v_1, u_1)\}$   $v_1$  propose à  $u_1$   
 $A = \{(v_2, u_2), (v_1, u_1)\}$   $v_3$  propose à  $u_2$   
 $A = \{(v_2, u_2), (v_1, u_1), (v_3, u_3)\}$   $v_3$  propose à  $u_3$
18. Si on note  $\mathcal{N}(v)$  l'ensemble des éléments de  $U$  auxquels  $v$  n'a pas encore proposé,  
la quantité  $\sum_{v \in V} |\mathcal{N}(v)|$  est un variant :  
c'est un entier positif qui décroît strictement à chaque itération.
19.  $A$  est un couplage : tout  $v \in V$  n'est apparié qu'à au plus un élément de  $u$  et tout  $u \in U$  n'est apparié qu'à au plus un élément de  $v$ .
20. Par l'absurde : Supposons que le couplage parfait renvoyé n'est pas stable.  
 $\exists ((v_1, u_1), (v_2, u_2)) \in A^2$  tels que  $u_2 >^{v_1} u_1$  et  $v_1 >^{u_2} v_2$ .  
 $v_1$  a déjà proposé à  $u_2$  avant de proposer à  $u_1$  car  $u_2 >^{v_1} u_1$ .  
Que  $v_1$  ait proposé avant ou après  $v_2$  à  $u_2$ ,  $v_2$  ne peut pas être apparié à  $u_2$  car  $v_1 >^{u_2} v_2$ .
21. Le variant donné en question 18 commence à  $n^2$  et décroît d'au moins 1 à chaque itération.
22. Si  $v$  préfère  $A$  et  $v$  préfère  $A$ , alors on a :  
 $u >^v m_{A'}(v)$  et  $v >^u m_{A'}(u)$ . Donc  $A'$  n'est pas stable.
23. Par l'absurde : soit  $A$  le couplage donné par l'**algorithme 3**,  
et  $A'$  un couplage parfait stable contenant  $(v, u)$ .  
Notons  $u' = m_A(v)$ , et  $v' = m_{A'}(u')$ .  
 $v$  a été rejeté par  $u$  donc  $u >^v u'$ ,  $v$  préfère  $A'$ .  
Comme  $v$  préfère  $A'$ , son partenaire  $u'$  dans  $A$  préfère  $A'$  :  $v' >^{u'} v$ .  
On a donc  $A$  qui n'est pas stable, ce qui donne une contradiction avec le résultat de la question 21.
24. (a)  $M(v)$  ne peut pas rejeter  $v$  car il est réalisable pour  $v$ .  
Si  $v$  est apparié, il est donc forcément apparié avec un élément  $u$  à qui il a proposé avant  $M(v)$  ou avec  $M(v)$ .  
  
(b) De même, comme  $M(v)$  ne peut pas rejeter  $v$ , si  $v$  n'est pas apparié, c'est qu'il n'a pas encore proposé à  $M(v)$ .

25. À la fin de l'algorithme, le couplage obtenu est stable, donc chaque élément  $v$  est apparié avec un élément  $u$  qui est réalisable pour  $v$ . Le seul qui vérifie la condition montrée à la question précédente est  $M(v)$ .  
Donc l'algorithme produit l'ensemble  $A^*$ .

26. Ajoutons la condition supplémentaire manquante dans l'énoncé :  
 $V \cup U \neq \emptyset$ .

Il existe alors un sommet  $s_0$ . Supposons que  $\mathcal{G}$  n'admette pas de circuit.  
Alors pour tout  $i \in \mathbb{N}$ , le sommet  $s_{i+1}$  étant le premier choix de  $s_i$  n'appartient pas à  $\{s_0, \dots, s_i\}$  car  $\mathcal{G}$  n'admet pas de circuit.  
On a une contradiction pour  $i = |S| - 1$  car  $s_{i+1} \notin S$ .  
 $\mathcal{G}$  contient donc au moins un circuit.

27. Supposons que  $x \in S$  est dans au moins deux circuits :

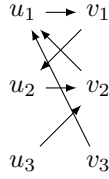
$$x = s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n = x$$

$$x = t_0 \rightarrow t_1 \rightarrow \dots \rightarrow t_m = x$$

Soit  $i$  le plus petit entier tel que  $s_i \neq t_i$ . Il existe car les deux circuits sont différents, et il est supérieur à 0 car  $s_0 = t_0 = x$ .

Alors  $s_{i-1} = t_{i-1}$  a deux voisins, ce qui est absurde par définition du graphe  $\mathcal{G}$ .

28. Voici le graphe  $\mathcal{G}$  :



On a le circuit  $u_1 \rightarrow v_1 \rightarrow u_2 \rightarrow v_2 \rightarrow u_1$ .

On pose  $A = \{(v_1, u_2), (v_2, u_1)\}$ . et il reste le graphe suivant :

$$u_3 \leftrightarrow v_3$$

On a le circuit  $u_3 \rightarrow v_3 \rightarrow u_3$ .

On conclut avec  $A = \{(v_1, u_2), (v_2, u_1), (u_3, v_3)\}$ .

29. Dans l'ensemble  $A$  suivant, on a  $(v_1, u_2)$  et  $(v_2, u_1)$ .

Or  $u_1 >^{v_1} u_2$  et  $v_1 >^{u_1} v_2$ . Donc  $A$  n'est pas stable.

30. Supposons qu'il existe  $A$  produit par l'algorithme 4 qui n'est pas P-stable.

Alors, il existe  $(v, v') \in V^2$  telle que  $v$  préfère  $m_A(v')$  et  $v'$  préfère  $m_A(v)$ .  
Au moment où  $(v, m_A(v))$  a été ajouté à  $A$ ,  $m_A(v)$  était le premier choix de  $v$  parmi les éléments de  $U$  restants. Donc  $(v', m_A(v'))$  avait déjà été ajouté.

De même, au moment où  $(v', m_A(v'))$  a été ajouté,  $m_A(v')$  était le premier choix de  $v'$  et  $m_A(v)$  avait déjà été ajouté.

Chacun de ces couples a été ajouté strictement avant l'autre, ce qui est absurde.

31. Comme montré en question 17, l'algorithme 3 produit l'ensemble  $A = \{(u_1, v_1), (u_2, v_2), (u_3, v_3)\}$  pour le jeu de données  $\mathcal{D}$ .

Or,  $v_1$  préfère  $u_2$  et  $v_2$  préfère  $u_1$ . Donc  $A$  n'est pas P-stable.

32. Il s'agit de parcourir la matrice. Deux boucles imbriquées font l'affaire.

```
void calcul_rang(int Lu[n][n], int Rang[n][n]){
    for (int u = 0; u < n; u = u + 1)
        for (int j = 0; j < n; j = j + 1){
            int v = Lu[u][j];
            Rang[u][v] = j;
        }
}
```

33. Pour les tableaux `coupleV` et `coupleU`, il s'agit de choisir une valeur ne pouvant pas apparaître lorsque les éléments sont appariés, on peut proposer  $-1$  ou  $n$ . Pour `Usuiv`, il faut absolument commencer avec le premier choix de chaque  $v$ , c'est-à-dire le rang 0.

```
void init(int CoupleV[n], int CoupleU[n], int Usuiv[n]){
    for (int i = 0; i < n; i = i + 1){
        CoupleV[i] = -1;
        CoupleU[i] = -1;
        Usuiv[i] = 0;
    }
}
```

34. On retient le nombre de couples formés pour éviter de le recalculer. On pourrait améliorer la complexité en utilisant par exemple une file de priorité pour retenir le prochain élément de  $V$  à considérer.

```

void gale_shapley(int Lv[n][n], int Lu[n][n], int CoupleU[n], int CoupleV[n])
{
    int couples = 0;
    int Usuiv[n];
    int Rang[n][n];
    init(CoupleV, CoupleU, Usuiv);
    calcul_rang(Lu, Rang);
    while (couples < n){
        int v = 0;
        while (CoupleV[v] != -1) // v est apparié
            v = v + 1;
        int u = Usuiv[v];
        Usuiv[v] = u + 1;
        if (CoupleU[u] == -1){ // u n'est pas apparié
            CoupleU[u] = v;
            CoupleV[v] = u;
            couples = couples + 1;
        }
        else{
            int v2 = CoupleU[u];
            if (Rang[u][v] < Rang[u][v2]){ // u préfère v
                CoupleV[v2] = -1;
                CoupleV[v] = u;
                CoupleU[u] = v;
            }
        }
    }
}

```

35. Il s'agit juste d'ajouter un élément à une liste chaînée. On l'ajoute en tête car c'est le plus efficace ( $O(1)$ ).

```

void ajoute_arc(graphe *g, int i, int j){
    liste vi = malloc(sizeof(struct noeud));
    vi->individu = j;
    vi->suivant = g->liste_adjacence[i];
    g->liste_adjacence[i] = vi;
}

```

36. Le comportement attendu si  $(i, j)$  n'est pas dans  $\mathcal{G}$  n'est pas précisé. J'ai choisi de ne rien faire pour simplifier l'implémentation de la fonction suivante `supprime_sommet`.

La gestion de la mémoire n'est pas explicitement demandée dans l'énoncé.

```
void supprime_arc(graphe *g, int i, int j){
    liste curr = g->liste_adjacence[i];
    if (curr == NULL) return; // l'arc (i,j) n'est pas dans g
    while (curr->suivant != NULL && curr->suivant->individu != j)
        curr = curr->suivant;
    if (curr->suivant == NULL) return; // l'arc (i,j) n'est pas dans g
    liste a_supprimer = curr->suivant;
    curr->suivant = a_supprimer->suivant;
    free(a_supprimer);
}
```

37. La consigne semble se contredire. On ne supprime effectivement pas le sommet  $s$ , mais plutôt tous les arcs dont il est origine ou destination. Encore une fois, la gestion de la mémoire n'est pas explicitement demandée dans l'énoncé.

```
void supprime_sommet(graphe *g, int s){
    for (int i = 0; i < g->nb_sommet; i = i + 1)
        supprime_arc(g, i, s); // suppression des arcs entrants
    liste curr = g->liste_adjacence[s];
    while (curr != NULL){
        liste next = curr->suivant;
        free(curr);
        curr = next;
    }
    g->liste_adjacence[s] = NULL; // suppression des arcs sortants
}
```

38. Pas de difficulté particulière. On pense bien à allouer dynamiquement la mémoire.

```
graphe *construit_graphe(int Lv[n][n], int Lu[n][n]){
    graphe *g = malloc(sizeof(graphe));
    g->nb_sommet = 2 * n;
    g->liste_adjacence = malloc(2 * n * sizeof(liste));
    for (int i = 0; i < 2 * n; i = i + 1){
        g->liste_adjacence[i] = NULL;
    }
    for (int i = 0; i < n; i = i + 1){
        ajoute_arc(g, i, n + Lv[i][0]);
        ajoute_arc(g, n + i, Lu[i][0]);
    }
    return g;
}
```