ÉPREUVE MUTUALISÉE AVEC E3A-POLYTECH
ÉPREUVE SPÉCIFIQUE - FILIÈRE MPI

INFORMATIQUE

Durée : 4 heures

N.B. : le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction. Si un candidat est amené à repérer ce qui peut lui sembler être une erreur d'énoncé, il le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.

RAPPEL DES CONSIGNES

— Utiliser uniquement un stylo noir ou bleu foncé non effaçable pour la rédaction de votre composition ; d'autres couleurs, excepté le vert, bleu clair ou turquoise, peuvent être utilisées, mais exclusivement pour les schémas et la mise en évidence des résultats.
— Ne pas utiliser de correcteur.
— Ecrire le mot FIN à la fin de votre composition.

Les calculatrices sont interdites.

Le sujet est composé de trois parties, toutes indépendantes.

Partie I - Grammaire non contextuelle

Soit la grammaire non contextuelle $G = (\Sigma, V, P, S)$, l'ensemble des règles de production P étant défini par :

$$S \to SaS \mid A$$
$$A \to AbA \mid B$$
$$B \to BcB \mid \varepsilon$$

où Σ (respectivement V) est l'ensemble de symboles terminaux (respectivement non terminaux), $S \in V$ est le symbole initial de G et ε dénote le mot vide.
Soit u le mot abc.

Q1. Donner une dérivation à gauche de u. Que peut-on en déduire sur le mot u ?

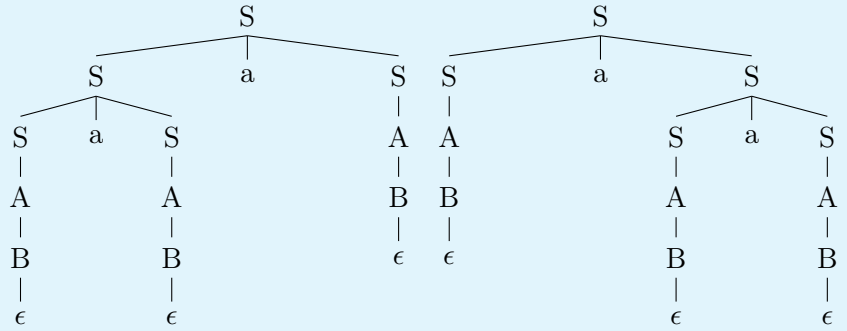
correction

$S \Rightarrow SaS \Rightarrow AaS \Rightarrow BaS \Rightarrow aS \Rightarrow aA \Rightarrow aAbA \Rightarrow aBbA \Rightarrow abA \Rightarrow abB \Rightarrow abBcB \Rightarrow abcB \Rightarrow abc$.
On en déduit que u est dans le langage engendré par la grammaire.

Q2. Donner deux arbres de dérivation pour le mot aa. En déduire que G est ambigüe.

correction

On donne les deux arbres symétriques :



Il y a deux arbres distincts pour engendrer le même mot, donc la grammaire est ambigüe.

Une grammaire est dite récursive gauche directe si elle possède une règle de la forme $A \to A\alpha$, où α est une suite de symboles terminaux et non terminaux. A est dite variable récursive gauche. Une telle grammaire pose divers problèmes en analyse syntaxique descendante, on cherche alors à éliminer cette récursivité. Ceci est toujours possible pour les langages réguliers.

Q3. Identifier dans G les variables récursives gauches.

correction

S, A et B sont récursives gauches.

Pour éliminer la récursivité gauche, on utilise l'algorithme suivant :
Soit $A \to A\alpha \mid \beta$ une règle, où α est une suite de symboles terminaux et non terminaux et β est une suite de symboles terminaux et non terminaux ne commençant pas par A.
On remplace cette règle par :
(i). une règle commençant par A : $A \to \beta A'$,
(ii). une règle pour une nouvelle variable A' : $A' \to \alpha A' \mid \varepsilon$.

Q4. Construire la grammaire non récursive gauche directe G' équivalente à G.

correction

On obtient en appliquant la construction

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

Q5. Prouver que le langage reconnu par G' (ou G) est $\{a, b, c\}^*$.

correction

NB : on doit parler de langage engendré plutôt que reconnu.
On raisonne sur G.
— On commence par remarquer trivialement que tout mot dans le langage engendré par G est dans $\{a, b, c\}^*$ – a, b et c étant les seuls symboles terminaux possibles.

— Réciproquement :
  — tout mot u constitué uniquement de c est engendré par G en partant du symbole B : si l'on note $|u| = n$, on effectue n dérivations à partir de B avec la règle $B \to BcB$ pour obtenir $BcB \dots cB$ avec n symboles c, puis $n+1$ dérivations remplaçant B par ϵ.
  — tout mot u constitué uniquement de b et c est engendré par G en partant de A : en effet, un tel mot s'écrit $u_0 b u_1 \cdots b u_{n-1}$, où les u_i sont des mots (éventuellement vides) ne contenant que des c (et ou n peut être égal à 0 s'il n'y a pas de B).
  On applique n fois la règle $A \to AbA$ à partir de A pour obtenir $AbAbA \cdots bA$. Alors, on dérive les A en B, et d'après le point précédent, on peut dériver les B en le mot u_i voulu.
  — enfin, tout mot de $\{a, b, c\}^*$ est engendré par G à partir de S : on applique la même construction que précédemment, mais en «découpant» cette fois-ci le mot au niveau des C, puis en appliquant le point précédent.

Partie II - Problème de bin-packing

Cette partie comporte des questions nécessitant un code OCaml.
Soient $n, k \in \mathbb{N}^2$. On appelle rangement de n objets dans k boîtes une fonction $\mathcal{R} : [\![1, n]\!] \to [\![1, k]\!]$. L'ensemble $B_j = \mathcal{R}^{-1}(j) = \{i \in [\![1, n]\!], \mathcal{R}(i) = j\}$ est le contenu de la boîte $j \in [\![1, k]\!]$.
Étant données des tailles $a = (a_1 \cdots a_n) \in (\mathbb{Q} \cap]0, 1])^n$, a_i étant la taille de l'objet i, on dit qu'un rangement $\mathcal{R}$ de n objets dans k boîtes est valide pour les tailles a si pour tout $j \in [\![1, k\|$ on a $\sum_{i \in B_j} a_i \leqslant 1$.
On considère alors le problème de décision BIN-PACKING qui, étant donnés $n, k \in \mathbb{N}^2$ et $a \in (\mathbb{Q} \cap]0, 1])^n$, décide s'il existe un rangement $\mathcal{R}$ de n objets dans k boîtes valide pour les tailles a.
On peut décrire une solution de ce problème par un couple $(k, \mathcal{R})$.

Q6. Montrer que BIN-PACKING est dans NP.

correction

On se donne comme certificat un rangement.
On vérifie en temps polynomial qu'il est valide (en calculant simplement la somme des poids des objets rangés dans chaque boîte).
On en déduit que le problème est dans NP.

Soit le problème PARTITION suivant :
Pour n entiers $c_1 \cdots c_n$, existe-t-il un sous-ensemble S de $[\![1, n]\!]$, tel que $\sum_{i \in S} c_i = \sum_{i \notin S} c_i$?
On admet que le problème PARTITION est NP-complet.

Q7. Montrer par réduction depuis PARTITION que le problème BIN-PACKING est NP-complet.

correction

On se donne une instance $c_1, \cdots c_n$ ne partition – que l'on suppose sans perte de généralité ne pas contenir d'objet de poids nul.
On note S la somme des poids, et l'on construit (en temps linéaire donc polynomial) l'instance suivante de BIN-PACKING :
$\forall i \in [\![1, n]\!], a_i = 2\frac{c_i}{S}$, $k = 2$.
C'est une instance positive de BIN-PACKING
si et seulement si il est possible de séparer les objets en deux parties de poids total $\leqslant 1$
si et seulement si il est possible de séparer les objets en deux parties de poids total 1 (car la somme des poids des objets est 2 par construction)
si et seulement si il est possible de séparer les c_i en deux parties de somme S
si et seulement si les (c_i) forment une instance positive de PARTITION.

On a donc bien la réduction polynomiale voulue, donc BIN-PACKING est NP-difficile, donc NP-complet d'après la question précédente.

On s'intéresse par la suite au problème MIN-BIN-PACKING qui demande de déterminer le plus petit k pour lequel il existe un rangement $\mathcal{R}$ de n objets de taille a dans k boîtes. On introduit pour résoudre ce problème une heuristique dite Premier Casier Décroissant (algorithme 2), se basant sur l'heuristique Premier Casier décrite dans l'algorithme 1. Dans cet algorithme, le rangement $\mathcal{R}$ est modélisé par une liste de boîtes $\mathcal{L}$.

```
Algorithme 1 - Algorithme Premier Casier
Entrées : n ∈ ℕ, (a_1 ··· a_n) ∈ (ℚ∩]0,1])^n
Sorties : Une solution (k, ℛ)
début
    ℒ : liste de boîtes ouvertes
    ℒ = ∅
    pour i = 1 à n faire
        Rechercher la première boîte de ℒ dans laquelle l'objet i peut être placé
        si une telle boîte existe alors
            l'objet i est placé à l'intérieur
        sinon
            une nouvelle boîte est ajoutée à ℒ et l'objet i y est placé
    k := longueur (ℒ)
    Déduire ℛ de ℒ.
```

```
Algorithme 2 - Algorithme Premier Casier Décroissant
Entrées : n ∈ ℕ, (a_1 ··· a_n) ∈ (ℚ∩]0,1])^n
Sorties : Une solution (k, ℛ)
début
    σ = permutation telle que a_σ(1) ⩾ a_σ(2) ··· ⩾ a_σ(n)
    k, ℛ_1 := Premier Casier (n, a_σ(1), ··· , a_σ(n)) .
    pour i = 1 à n faire
        ℛ(i) := ℛ_1(σ(i))
```

Nous allons montrer que l'algorithme Premier Casier Décroissant est une $\frac{3}{2}$-approximation pour le problème MIN-BIN-PACKING.
On note dans la suite k le nombre de boîtes retournées par l'algorithme 2 , k^* le nombre de boîtes optimal (défini comme le nombre minimal de boîtes permettant de répondre au problème MIN-BINPACKING) et $k_1 = \lceil 2k/3 \rceil$, où $\lceil . \rceil$ désigne la partie entière supérieure. B_{k_1} fait référence à la boîte déduite du rangement renvoyé par l'algorithme 2.

Q8. Montrer que si B_{k_1} contient un objet i de taille $a_i > \frac{1}{2}$, alors $k^* \geqslant k_1 \geqslant \frac{2}{3}k$.

correction

Par définition de k_1, on sait déjà que $k_1 = \lceil 2k/3 \rceil \geqslant \frac{2}{3}k$.
Supposons que B_{k_1} contienne un objet i de taille $a_i > \frac{1}{2}$.
Cela signifie que tous les objets précédent i sont de poids $\geqslant a_i > \frac{1}{2}$.
Par ailleurs, $k_1 \leqslant i$ (car une récurrence immédiate montre qu'un objet j est nécessairement mis dans une boîte d'indice j ou moins).
Il y a i objets de poids $> 1/2$ (en comptant l'objet i), donc au moins k_1 objets de poids $> 1/2$, qui

devront tous être dans des boîtes distinctes dans toute solution – y compris l'optimale – qui utilise donc au moins k_1 boîtes.
Donc $k^* \geqslant k_1$.

Q9.

(a) Montrer que sinon, les boîtes $B_{k_1} \cdots B_k$ contiennent au moins $2\,(k - k_1) + 1$ objets, aucun d'eux ne pouvant rentrer dans les boîtes $B_1 \cdots B_{k_1 - 1}$.

correction

On suppose donc que B_{k_1} ne contient que des objets de taille $\leqslant 1/2$.

— Par définition de l'algorithme 1, aucun objet que l'on met dans une boîte ne pouvait rentrer dans les boîtes précédente au moment où on l'y a mis - donc a fortiori à la fin de l'exécution.

— Notons i l'indice du premier objet mis dans B_{k_1}. On a donc $a_i \leqslant 1/2$, et pour tous les objets suivants, $a_j \leqslant 1/2$.
Comme tous les objets mis dans $B_{k_1} \cdots B_k$ le sont après l'objet i, il sont tous de poids $\leqslant 1/2$.
Toutes les boîtes à partir de k_1 contiennent au moins 2 objets à la fin de l'exécution – sauf éventuellement la dernière qui en contient au moins 1 – car sinon l'une des boîtes n'aurait pas du être ouverte.
Ces boîtes contiennent donc au moins $2(k - k_1) + 1$ objets comme demandé.

(b) Justifier (dans le cas de la question précédente) que $\sum_{i=1}^{n} a_i > \min\left(k_1 - 1, 2\,(k - k_1) + 1\right)$. En admettant que pour $k \in \mathbb{N}$, $\frac{2}{3}k + \frac{2}{3} \geqslant \lceil 2k/3 \rceil$, en déduire que $k^* \geqslant \left\lceil \frac{2}{3}k \right\rceil \geqslant \frac{2}{3}k$.

correction

Notons α le poids minimal des (au moins) $2(k - k_1) + 1$ objets des boîtes B_{k_1} à B_k. Le poids total de ces objets est donc supérieur ou égal à $\alpha(2(k - k_1) + 1)$.

Comme ces objets ne rentrent pas dans les boîtes de 1 à $k_1 - 1$, ces boîtes sont remplies à strictement plus de $1 - \alpha$. Le poids total des objets qu'elles contiennent est donc strictement supérieur à $(1 - \alpha)(k_1 - 1)$.

On en déduit que $\sum_{i=1}^{n} a_i > \alpha(2(k-k_1)+1) + (1-\alpha)(k_1 - 1)$, qui est un barycentre de $2(k-k_1)+1$ et $k_1 - 1$.

Donc $\sum_{i=1}^{n} a_i > \min\left(k_1 - 1, 2\,(k - k_1) + 1\right)$.

Or $k_1 - 1 \leqslant 2\,(k - k_1) + 1$ ssi $3k_1 \leqslant 2 + 2k$ ssi $\frac{2}{3} + \frac{2}{3}k \geqslant k_1$ – ce qui est vrai d'après l'énoncé, donc le minimum est $k_1 - 1$.

On en déduit que $\sum_{i=1}^{n} a_i > k_1 - 1$.

Finalement, $k^* \geqslant \sum_{i=1}^{n} a_i > k_1 - 1$, d'où $k^* > k_1 - 1$, et comme les deux membres sont entiers, $k^* \geqslant k_1 = \left\lceil \frac{2}{3}k \right\rceil \geqslant \frac{2}{3}k$.

Q10. Que peut-on en déduire pour l'**algorithme 2** ?

correction

(Question absente de l'énoncé original)
L'algorithme 2 est donc une 2/3-approximation pour le problème BIN-PACKING.

On définit des types record pour représenter les objets et les boîtes en OCaml :

```
type objet = {
    id : int ; (*identifiant d'objet *)
    taille : float ; (*taille de l'objet *)
}
type boite = {
    charge : float ; (*somme des tailles des objets de la boîte*)
    objets : objet list ; (*liste courante des tailles des objets dans la ↩
       boîte *)
}
```

Q11. Écrire une constante `boite_vide : boite` représentant une boîte vide.

correction

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

Q12. Écrire une fonction de signature
`ajoute_boite : objet -> boite -> boite` qui renvoie la boîte obtenue en ajoutant un objet à une boîte donnée.

correction

```
let ajoute_boite obj boite = match
    obj, boite with
    {id=id ; taille=t}, {charge=c ; objets=lobj} -> {charge=c+.t ; ↩
       objets=obj ::lobj}
```

Q13. Écrire une fonction récursive de signature
`trouve_boite : boite list -> objet -> int` telle que `trouve_boite bl o` retourne l'indice (à partir de 0) de la première boîte dans `bl` pouvant contenir `o` ou la taille de la liste `bl` si aucune boîte ne peut contenir `o`.

correction

```
let trouve_boite bl {id=id ; taille=t} =
  let rec aux i bl =
    match bl with
      [] -> i (*i sera bien la taille de la liste dans ce cas*)
    | {charge=c ; objets=lobj} ::_ when t <= 1. -. c -> i
    | _ ::qu -> aux (i+1) qu

  in aux 0 bl
```

Q14. Écrire une fonction récursive de signature
`transforme : 'a -> ('a -> 'a)-> int -> 'a list -> 'a list`
telle que `transforme d f i l` retourne la liste `l` où l'élément `x` à l'indice `i` (à partir de 0) a été remplacé par `f`. Si `i` est plus grand que la taille de la liste, `f d` est ajouté à la fin.

correction

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

Q15. Écrire une fonction de signature
`premier_casier : objet list -> boite list`
qui applique l'**algorithme 1**. On retournera directement la liste de boîtes, on ne cherchera pas à calculer la fonction $\mathcal{R}$. On pourra utiliser la fonction transforme précédemment écrite.

correction

```
let rec premier_casier lobjets =
  let rec aux lobjets lboites =
    match lobjets with
      [] -> lboites
    | t ::q -> aux q (transforme boite_vide (ajoute_boite t) (↩
       trouve_boite lboites t) lboites)
  in aux lobjets []
```

Q16. Écrire une fonction de signature
`premier_casier_decroissant : objet list -> boite list`
qui applique l'**algorithme 2**.
On pourra utiliser `List.sort : ('a -> 'a -> int)-> 'a list -> 'a list` qui trie la liste passée en argument dans l'ordre croissant suivant la fonction passée en argument (qui renvoie -1 , 0 ou 1 pour inférieur, égal ou supérieur respectivement). On pourra utiliser la fonction de comparaison générique `compare : 'a -> '↩ a -> int`. Là encore, on retournera directement la liste des boîtes.

correction

```
let premier_casier_decroissant lobjets =
  premier_casier (List.sort (fun {id = _; taille = t} {id = _; taille↩
      = t'} -> if t < t' then 1 else if t > t' then -1 else 0) ↩
    lobjets)
```

Partie III - Algorithmes de couplage

Cette partie comporte des questions nécessitant un code en C.

On s'intéresse ici à un problème d'appariement entre deux groupes d'individus U et V, que l'on suppose de même cardinalité n.
On cherche à apparier les éléments de V aux éléments de U, chaque élément devant être apparié à exactement un élément de l'autre ensemble. Chaque élément de V (respectivement de U) a, de plus, un ordre de préférence total entre tous les éléments de U (respectivement de V).

Définition 1 (Couplage, couplage parfait)
Un ensemble de paires $A \subseteq V \times U$ est un couplage *si tout $x \in V$ et tout $y \in U$ apparaissent dans au plus un élément de A. Le couplage est dit parfait si tout $x \in V$ et tout $y \in U$ apparaissent dans un et un seul élément de A.*

Notations
Dans toute la suite on notera :
— $|E|$ le cardinal de l'ensemble E,
— $>^x$ l'ordre associé à l'élément $x \in V \cup U$: si $u_1 \in U$ préfère strictement $v_1 \in V$ à $v_2 \in V$ alors $v_1 >^{u_1} v_2$. Par extension, on définit de même la notion de préférence $\geqslant^x$.
— $m_A(x)$ le partenaire de $x \in V \cup U$ dans le couplage parfait A.

Définition 2 (Couplage parfait stable)
Un couplage parfait A est dit stable *si, pour tous couples (v_1, u_1) et (v_2, u_2) de A, il est impossible que l'on ait $u_2 >^{v_1} u_1$ et $v_1 >^{u_2} v_2$.*

Définition 3 (Pareto-optimalité)
Soient A et A' deux couplages parfaits stables et $E \subseteq V \cup U$. On dit que A Pareto-domine *A' sur E si et seulement si pour tout $x \in E, m_A(x) \geqslant^x m_{A'}(x)$ et il existe $x \in E$ tel que $m_A(x) >^x m_{A'}(x)$. Un couplage est Pareto-optimal s'il est stable et n'est pas Pareto dominé sur $V \cup U$.*

En économie, la Pareto-optimalité d'un système exprime le fait que l'on ne peut améliorer la satisfaction d'un individu du système sans diminuer la satisfaction d'un autre.

III.1 - Recherche de couplages stables

L'algorithme de Gale-Shapley (**algorithme 3**) permet de montrer qu'il existe toujours au moins un couplage parfait stable et d'en trouver un de façon efficace. L'idée de l'algorithme est tout d'abord de choisir un des deux ensembles (ici pour illustration V). Les éléments de V font des propositions aux éléments de U. Quand un élément de V fait une proposition à un élément de U, celui-ci peut soit la refuser définitivement, soit l'accepter provisoirement en attendant une éventuelle meilleure offre. Toute nouvelle acceptation vaut rejet définitif des propositions précédemment acceptées.

```
Algorithme 3 - Algorithme de Gale-Shapley
Entrées : n = |V| = |U|, les listes de préférences des éléments de V et U
Sorties : Un couplage parfait stable A
début
    A := ∅
    tant que il existe un élément de V non apparié faire
        Choisir un tel v
        u := élément de U que v préfère parmi ceux à qui il n'a pas encore ↩
           proposé
        si u n'est pas apparié dans A alors
            A := A ∪ {(v, u)}
        sinon
            si u est apparié dans A à v' ET v >^u v' alors
                A := A \ {(v', u)}
                A := A ∪ {(v, u)}
```

`retourner` A

Soit le jeu de données $\mathcal{D}$ suivant : $V = \{v_1, v_2, v_3\}, U = \{u_1, u_2, u_3\}$ et :

V	U
$u_2 >^{v_1} u_1 >^{v_1} u_3$	$v_1 >^{u_1} v_3 >^{u_1} v_2$
$u_1 >^{v_2} u_2 >^{v_2} u_3$	$v_2 >^{u_2} v_1 >^{u_1} v_3$
$u_1 >^{v_3} u_2 >^{v_3} u_3$	$v_2 >^{u_3} v_1 >^{u_3} v_3$

Q17. Donner la trace de l'**algorithme 3** sur ces données. Le choix de v dans l'algorithme se fera par ordre croissant des indices.

correction

On choisit v_1.
$u := u_2$
$A := \{(v_1, u_2)\}$
On choisit v_2.
$u := u_1$
$A := \{(v_1, u_2),\ (v_2, u_1)\}$
On choisit v_3.
$u := u_1$, déjà apparié à v_2, mais qui préfère v_3.
$A := \{(v_1, u_2),\ (v_3, u_1)\}$
On choisit v_2.
$u := u_2$, déjà apparié à v_1, mais qui préfère v_2.
$A := \{(v_2, u_2),\ (v_3, u_1)\}$
On choisit v_1.
$u := u_1$, déjà apparié à v_3, mais qui préfère v_1
$A := \{(v_2, u_2),\ (v_1, u_1)\}$
On choisit v_3.
$u := u_2$, déjà apparié à v_2, et qui préfère v_2 à v_3.
On choisit v_3.
$u := u_3$
$A := \{(v_2, u_2),\ (v_1, u_1),\ (v_3, u_3)\}$

Q18. Montrer que l'**algorithme 3** termine.

correction

On considère comme variant le nombre de couples (v, u), où v a fait une proposition à u.
Ce nombre augmente strictement à chaque passage dans la boucle `tant que`, et est majoré par le nombre de couples (n^2), donc l'algorithme termine en au plus n^2 itérations.

Q19. Donner, sans le prouver, un invariant de boucle permettant de prouver que l'**algorithme 3** retourne un couplage parfait.

correction

On considère l'invariant «à la fin de chaque itération, A est un couplage».

Q20. Montrer que le couplage parfait calculé est stable.

correction

Raisonnons par l'absurde. Supposons qu'il existe deux couples (v_1, u_1) et (v_2, u_2) renvoyés par l'algorithme tels que $u_2 >^{v_1} u_1$ et $v_1 >^{u_2} v_2$.

— Si v_1 choisit u_2 avec que v_2 ne le fasse. Lorsque u_2 quittera v_1, se sera pour un v_i tel que $v_i >^{u_2} > v_1 >^{u_2} v_2$, et donc u_2 ne pourra jamais être apparié avec v_2 : absurde.

— Si v_2 choisit u_2 avant que v_1 ne le fasse. Mais alors, v_1 (ou un v encore plus intéressant pour u_2) choisira u_2 après, et v_2 ne sera pas apparié avec u_2 : absurde.

Q21. Montrer que cet algorithme effectue moins de n^2 itérations de boucles.

correction

Déjà prouvé ci-dessus lors de la réponse sur la terminaison.

Pour des préférences données, il peut exister plus d'un couplage parfait stable. Soient A et A' deux tels couplages pour un problème donné. On dira qu'un élément de $V \cup U$ *préfère* A à A' s'il n'a pas $'$ le même partenaire dans les deux couplages et s'il préfère celui de A à celui de A'.

Q22. Montrer que si v préfère le couplage A, alors son partenaire u dans A préfère A'.

correction

Supposons que $(v, u) \in A$, $(v, u') \in A'$, $(v', u) \in A'$, $u \neq u'$, $v \neq v'$ et $u >^v u'$ (i.e. v préfère A à A').
On souhaite montrer que u préfère A', i.e. $v' >^u v$.
Si ça n'était pas le cas, on aurait $v >^u v'$, et les 4 propriétés suivantes forment une contradiction avec la définition de la stabilité de A' :

— $(v, u') \in A'$
— $(v', u) \in A'$
— $v >^u v'$
— $u >^v u'$

Donc u préfère bien A' à A.

On peut montrer qu'en même temps le partenaire u' de v dans A' préfère A.

La correction de l'algorithme de Gale-Shapley ne dépend pas de l'élément $v \in V$ qui fait sa proposition à chaque tour. En fait, on montre dans la suite que, quel que soit l'élément $v \in V$ non encore apparié qui fait une proposition à chaque tour, l'algorithme calcule toujours le même couplage parfait stable. Dans la suite, on dira qu'un élément $u \in U$ est *réalisable* pour un élément $v \in V$ s'il existe un couplage parfait stable qui contient (v, u).

Q23. Montrer qu'au cours de l'**algorithme 3**, chaque $v \in V$ n'est rejeté que par les éléments $u \in U$ qui ne sont pas réalisables pour v.

correction

TODO

Soit $M(v) \in U$ l'élément réalisable le mieux classé sur la liste des préférences de v et $P(u) \in V$ l'élément réalisable le moins bien classé sur la liste des préférences de u. On note enfin $A^* = \{(v, M(v)), v \in V\}$.

Q24. Montrer que lors de l'exécution de l'**algorithme 3** :

1. Tout $v \in V$, s'il est apparié, l'est avec $u \in U$ supérieur ou égal à $M(v)$ dans sa liste de préférences.
2. Tout $v \in V$, s'il n'est pas apparié, n'a fait des propositions qu'à des $u \in U$ strictement supérieurs à $M(v)$ dans sa liste de préférences.

correction

On montre que les 2 propriétés forment un invariant.

Elles sont trivialement vraies avant le premier passage dans le `tant que` – aucun v n'est apparié et il n'y a eu aucune proposition.

Montrer que les propriétés sont préservées par un passage dans la boucle. Distinguons des cas :

— Si v fait une proposition à u qui n'est pas apparié, l'appariement a lieu. En fin d'exécution, si v sera apparié avec u' (éventuellement égal à u) – à qui il aura fait une proposition après (au sens large) u. u' sera donc réalisable, et u est après u' donc après $M(v)$ dans sa liste de préférence.

— Si v fait une proposition à u mais que u refuse : en fin d'exécution, si v sera apparié avec $u' \neq u$ – à qui il aura fait une proposition après u. u' sera donc réalisable, et u est strictement après u' donc strictement après $M(v)$ dans sa liste de préférence.
La propriété reste vraie trivialement pour les précédents u_i auquel v a fait une proposition précédemment, qui sont encore après dans la liste.

— Si v fait une proposition à u, déjà apparié à v', et que u accepte. Alors
  — Par l'argument du premier point, u est supérieur ou égal à $M(v)$ dans la liste des préférences de u.
  — v' n'est plus apparié. Par hypothèse, il n'a fait que des proposition à des éléments supérieures ou égaux à $M(v')$. Reste à montrer que u n'est pas égal à $M(v')$ – i.e. que (v', u) n'est pas réalisable. C'est le cas d'après la question précédente, car en considérant une exécution de l'algorithme qui aurait apparié v à u avant que v' ne fasse une proposition à u, u aurait rejeté v'.
  NDMP. Argument qui mériterait d'être précisé, je suis preneur de preuves alternatives.

Q25. En déduire que quel que soit l'élément $v \in V$ choisi dans la boucle Tant que de l'**algorithme 3**, l'algorithme termine en produisant l'ensemble A^*.

correction

À la fin de l'algorithme, on obtient un couplage stable – donc par définition, v est apparié avec un u réalisable.
Or d'après la question précédente, il est avec un u supérieur ou égal à $M(v)$ – qui est l'élément réalisable le mieux classé.
Il est donc nécessairement apparié avec $M(v)$.
Ceci est vrai pour tout v, donc l'algorithme termine en produisant A^*.

Ainsi, l'algorithme calcule le meilleur couplage parfait stable possible du point de vue des éléments de V. Il s'avère également que l'algorithme de Gale-Shapley fait correspondre u avec $P(u)$, pour tout $u \in U$. L'algorithme avantage donc de manière claire le groupe ayant l'initiative (ici V) : le couplage stable produit est dit *Pareto-V-optimal* en ce sens qu'il n'est pas Pareto-dominé sur V.

III.2 - Couplages Pareto-optimaux

Dans cette sous-partie, on utilise également le jeu de données $\mathcal{D}$.
L'algorithme de Gale-Shapley ne retourne pas nécessairement un couplage Pareto-optimal. On s'intéresse ici à un algorithme susceptible de produire cette propriété.
Pour construire un couplage Pareto-optimal, on représente les données à l'aide d'un graphe orienté $\mathcal{G} = (S, E)$ où :

— les sommets S sont les éléments de $V \cup U$,

— les arcs E représentent les premiers choix : $(v, u) \in E$ si et seulement si u est le premier choix de v dans sa liste de préférences et $(u, v) \in E$ si et seulement si v est le premier choix de u.

Q26. Montrer que $\mathcal{G}$ contient au moins un circuit, c'est-à-dire un chemin dont les deux sommets extrémités sont identiques.

correction

On part d'un sommet v arbitraire, on suit l'unique arc (v, u) partant de v, puis on recommence à partir de u.
On créé ainsi un chemin infini (car il y a un arc sortant de tout sommet), qui passe donc deux fois par un même sommet – ce qui prouve l'existence d'un circuit dans le graphe.

Q27. Justifier par l'absurde que tout $x \in V \cup U$ est dans au plus un circuit.

correction

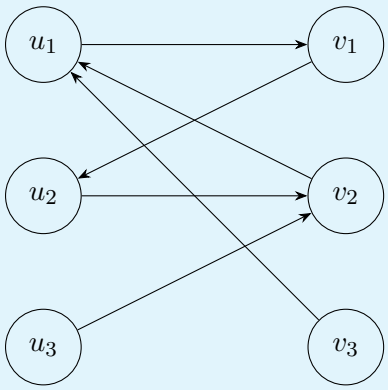
Il ne paraît pas utile de raisonner par l'absurde. Pour tout $n \in \mathbb{N}$, une récurrence immédiate montre qu'il existe une unique chemin de longueur n commençant par u (car il y a exactement un arc sortant de chaque sommet).
Quitte à changer son point de départ, on peut toujours considérer qu'un circuit contenant u part de u.
Il existe donc un unique circuit contenant u.

On propose alors l'**algorithme 4** pour calculer un couplage, dont on admettra qu'il est Pareto-optimal pour V.

```
Algorithme 4 - Algorithme de couplage par graphe orienté
Entrées : n = |V| = |U|, les listes de préférences des éléments de V et U
Sorties : Un couplage A
début
    A := ∅
    tant que il existe un élément de V non apparié faire
        Construire le graphe G sur les données courantes
        pour chaque arc ( v,u ) présent dans un circuit de G faire
            A = A ∪ {(v,u)}
        Supprimer du problème tous les v ∈ V et lest u ∈ U appariés
        Mettre à jour les listes de préférences des individus restants
    retourner A
```

Q28. Donner les graphes et la construction itérative de A produits par l'**algorithme 4** sur $\mathcal{D}$.

correction



On peut considérer le circuit $u_2 \to v_2 \to u_1 \to v_1 \to u_2$, d'où les appariements (v_2, u_1) et (v_1, u_2).
À l'étape suivante, le graphe est réduit aux sommets u_3 et v_3, et a deux arcs les reliant. On aura donc l'appariement (v_3, u_3).

Q29. En utilisant les données $\mathcal{D}$, montrer que l'**algorithme 4** ne produit pas toujours un couplage stable.

correction

L'appariement obtenu à la question précédente n'est pas stable, car $u_1 >^{v_3} u_3$, et $v_3 >^{u_1} v_2$.

On s'intéresse alors à une autre forme de stabilité pour assurer à la fois une propriété de Pareto et une stabilité dans les couplages.

Définition 4 (P-stabilité)
Un couplage A est P-stable *s'il n'existe aucune paire $(v, v') \in V^2$ telle que v préfère $m_A(v')$ et v' préfère $m_A(v)$.*

Q30. Montrer par l'absurde que l'**algorithme 4** produit un couplage P-stable.

correction

Raisonnons par l'absurde, et raionnons par l'absurde, en supposant que l'algorithme 4 peut produire un couplage P-stable, i.e. dans lequel il existe une paire (v, v') telle que v préfère $m_A(v')$ à $m_A(v)$, et v' préfère $m_A(v)$ à $m_A(v')$.
Distinguons plusieurs cas :
— $(v, m_A(v))$ est apparu dans un cycle strictement avant $(v', m_A(v'))$. Mais alors, lors du choix de $(v, m_A(v))$, $m_A(v')$ était encore présent dans le graphe, et comme v préfère $m_A(v')$ à $m_A(v)$, l'arc $(v, m_A(v))$ ne pouvait être présent à ce moment là : absurde.
— De même dans le cas symétrique.
— Si les deux appariements sont apparus à la même itération, c'est que les 4 sommets $v, v', m_A(v)$ et $m_A(v')$ étaient présents dans le même graphe, ce qui est à nouveau absurde pour la même raison que précédemment.

Q31. Montrer que l'**algorithme 3** ne produit pas nécessairement un couplage P-stable.

correction

On reprend le résultat renvoyé pour $\mathcal{D}$ par cet algorithme : $\{(v_1, u_1), (v_2, u_2), (v_3, u_3)\}$. Ce couplage n'est pas P-stable car v_1 préfère $u_2 = m_A(v_2)$ à $u_1 = m_A(v_1)$, et réciproquement.

III.3 - Implémentation

Dans la suite, les individus sont numérotés de 0 à $n-1$. Ainsi $V = \{0, \cdots n-1\}$ et $U = \{0, \cdots n-1\}$. On suppose disposer de deux matrices de taille $n \times n$ `Lv` et `Lu` donnant respectivement les listes de préférences des éléments de V et U : ainsi, `Lu[i][j]` est le j-ème élément de V préféré par l'élément $i \in U$.
On définit ces tableaux en C par l'intermédiaire du code suivant :

```
/* La directive #define est utilisée pour définir une valeur pour la ↩
   constante n qui
    servira à déclarer des tableaux de taille fixe.*/
#define n 4
int main(void) {
    int Lv[n][n],Lu[n][n] ;
(...)
}
```

III.3.1 - Algorithme de Gale-Shapley

Pour connaître rapidement le classement des éléments de V dans les listes de préférences des élements de U, on définit un tableau `Rang` de taille $n \times n$, tel que `Rang[i][j]` donne le rang de $j \in V$ dans la liste de préférences de l'élément $i \in U$. Par convention, les rangs commencent à 0 .

Q32. Écrire une fonction de prototype
`void calcule_rang (int Lu[n][n], int Rang[n][n])` qui construit le tableau `Rang`.

correction

```
void calcule_rang(int Lu[n][n], int Rang[n][n]) {
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      {
  Rang[i][Lu[i][j]] = j;
      }
}
```

Afin de pouvoir ajouter ou supprimer rapidement des couples (v, u) au couplage parfait stable A, on définit deux tableaux `CoupleV` et `CoupleU` tels que `CoupleV[i]` ou `CoupleU[i]` est le partenaire actuel de i. On définit enfin un dernier tableau `USuiv` de taille n stockant pour chaque élément de V le rang du prochain élément de U à qui il peut faire une proposition.
Q33. Proposer des valeurs d'initialisation pour ces trois tableaux.

correction

`CoupleV[i] = CoupleU[i] = -1` – convention possible pour dire que les sommets correspondant ne sont pas appariés.
`USuiv[i] = 0`.

Q34. Avec ces structures de données, écrire une fonction de prototype `void gale_shapley(int Lv[n][n], int Lu[n][n], int CoupleU[n], int CoupleV[n])` qui réalise l'**algorithme 3**. On ne retournera pas le couplage parfait stable, on remplira seulement les variables `CoupleU` et `CoupleV` passées en argument. Pour le choix de v, on prendra les éléments de V par ordre croissant de leur numéro en choisissant à chaque fois le premier libre.

correction

```c
void gale_shapley(int Lv[n][n], int Lu[n][n], int CoupleU[n], int ↩
    CoupleV[n]) {
  int Rang[n][n];
  calcule_rang(Lu, Rang);

  int Usuiv[n];
  for (int i = 0; i < n; i++) {
    CoupleV[i] = CoupleU[i] = -1;
    Usuiv[i] = 0;
  }
  int nb_app = 0; // nombre de v appariés
  while (nb_app < n) {
    int v = 0;
    while (CoupleV[v] != -1) v++;
    // v contient le premier v non apparié
    int u = Usuiv[v];
    Usuiv[v]++;
    if (CoupleU[u] == -1) {
      CoupleU[u] = v;
      CoupleV[v] = u;
      nb_app++;
    }
    else if (Rang[u][v] < Rang[u][CoupleU[u]]) {
      CoupleV[CoupleU[u]] = -1; // on dés-apparie v' = CoupleU[u]
      CoupleU[u] = v;
      CoupleU[v] = u;
    }
  }
}
```

III.3.2 - Algorithme de couplage par graphe orienté

Sans explicitement coder tout l'**algorithme 4** qui requiert de rechercher les circuits dans $\mathcal{G}$, on se propose de coder quelques fonctions utiles à la réalisation de l'algorithme.
Pour pouvoir numéroter les sommets du graphe de manière continue, les individus sont maintenant numérotés de 0 à $2n-1$ de la manière suivante : $V = \{0, \cdots n-1\}$ et $U = \{n, \cdots 2n-1\}$.
On code le graphe orienté $\mathcal{G}$ par listes d'adjacence. On propose les types suivants :

```c
struct noeud {
    int individu; // Element de U union V
    struct noeud* suivant;
};
typedef struct noeud *liste;
struct graphe_s {
    int nb_sommets;
    liste *liste_adjacence;
};
typedef struct graphe_s graphe;
```

Q35. Écrire une fonction de prototype

`void ajoute_arc (graphe *g, int i,int j)` qui ajoute l'arc (i, j) au graphe $\mathcal{G}$.

correction

Le type proposé «overshoote» largement dans la mesure où chaque sommet du graphe considéré a exactement un sommet sortant...

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

Q36. Écrire une fonction de prototype
`void supprime_arc (graphe *g, int i,int j)` qui supprime l'arc (i, j) du graphe $\mathcal{G}$.

correction

```
void supprime_arc (graphe *g, int i, int j) {
  liste cell = g->liste_adjacence[i];
  if (cell -> individu == j) g->liste_adjacence[i] = g->↩
     liste_adjacence[i] -> suivant;
  else {
    while (cell != NULL && cell -> suivant -> individu != j) cell = ↩
       cell -> suivant;
    if (cell != NULL) cell -> suivant = cell -> suivant -> suivant;
  }
}
```

Q37. Écrire une fonction de prototype
`void supprime_sommet (graphe * g, int s)`
qui supprime le sommet s du graphe $\mathcal{G}$. Pour ce faire, on supprimera uniquement tous les arcs dont ce sommet est origine ou destination.

correction

```
void supprime_sommet(graphe *g, int s) {
  g->liste_adjacence[s] = NULL;
  for (int i = 0; i < n; i++) {
    supprime_arc(g, i, s); // supprime_arc prend bien en compte le ↩
       cas où l'arc est absent
  }
}
```

Q38. Écrire une fonction de prototype
`graphe *construit_graphe (int Lv[n][n], int Lu[n][n])`
qui construit le graphe $\mathcal{G}$ utilisé dans l'**algorithme 4** et dont la définition est donnée juste avant la Q 26. On

prendra bien garde à la numérotation des éléments de U.

correction

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

FIN