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.*
- *Écrire 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 :

$$\begin{aligned} S &\rightarrow SaS \mid A \\ A &\rightarrow AbA \mid B \\ B &\rightarrow BcB \mid \varepsilon \end{aligned}$$

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 ?

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

Une grammaire est dite *récursive gauche directe* si elle possède une règle de la forme $A \rightarrow A\alpha$, 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.

Pour éliminer la récursivité gauche, on utilise l'algorithme suivant :
Soit $A \rightarrow A\alpha \mid \beta$ 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 : $A \rightarrow \beta A'$,
(ii). une règle pour une nouvelle variable A' : $A' \rightarrow \alpha A' \mid \varepsilon$.

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

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

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 $\mathcal{R} : [\![1, n]\!] \rightarrow [\![1, k]\!]$. 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 $\sum_{i \in B_j} a_i \leq 1$.

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.

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.

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 : $n \in \mathbb{N}, (a_1 \cdots a_n) \in (\mathbb{Q} \cap]0,1])^n$
Sorties : Une solution $(k, \mathcal{R})$
début
  $\mathcal{L}$: liste de boîtes ouvertes
  $\mathcal{L} = \emptyset$
  pour *i=1 à n* **faire**
    Rechercher la première boîte de $\mathcal{L}$ 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 $\mathcal{L}$ et l'objet i y est placé
  k := longueur($\mathcal{L}$)
  Déduire $\mathcal{R}$ de $\mathcal{L}$.

Algorithme 2 - Algorithme Premier Casier Décroissant

Entrées : $n \in \mathbb{N}, (a_1 \cdots a_n) \in (\mathbb{Q} \cap]0,1])^n$
Sorties : Une solution $(k, \mathcal{R})$
début
  σ = permutation telle que $a_{\sigma(1)} \geq a_{\sigma(2)} \cdots \geq a_{\sigma(n)}$
  $k, \mathcal{R}_1$:= `Premier Casier`$(n, a_{\sigma(1)}, \cdots, a_{\sigma(n)})$
  pour *i = 1 à n* **faire**
    $\mathcal{R}(i) := \mathcal{R}_1(\sigma(i))$

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-BIN-PACKING) 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 $k^* \geq k_1 \geq \frac{2}{3}k$.

Q9. 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 $B_1 \cdots B_{k_1 - 1}$.

Q10. Justifier que $\sum_{i=1}^{n} a_i > min(k_1 - 1, 2(k - k_1) + 1)$. En admettant que pour $k \in \mathbb{N}$, $\frac{2}{3}k + \frac{2}{3} \geq \lceil 2k/3 \rceil$, en déduire que $k^* \geq \left\lceil \frac{2}{3}k \right\rceil \geq \frac{2}{3}k$.

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

```
type objet = {
  id : int; (*identifiant de la boite*)
  taille : float; (*taille de la boite*)
}

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.

Q12. Écrire une fonction de signature

`ajoute_boite : objet -> boite -> boite`

qui renvoie la boîte obtenue en ajoutant un objet à une boîte donnée.

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`.

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 x`. Si `i`est plus grand que la taille de la liste, `f d` est ajouté à la fin.

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.

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.

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 :
- $\left|E\right|$ 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 $\geq^x$.
- $m_A(x)$ 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 $x \in E$, $m_A(x) \geq^x m_{A'}(x)$ 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
  $A := \emptyset$
  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**
      $A := A \cup \{(v, u)\}$
    sinon
      si *u est apparié dans A à v' ET $v >^u v'$* **alors**
        $A := A \setminus \{(v', u)\}$
        $A := A \cup \{(v, u)\}$
  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.

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

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

Q20. Montrer que le couplage parfait calculé est stable.

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

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'.

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.

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.

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^*.

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.

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

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:=$\emptyset$
  tant que *il existe un élément de* V *non apparié* **faire**
    Construire le graphe $\mathcal{G}$ sur les données courantes
    pour *chaque arc* (v,u) *présent dans un circuit de* $\mathcal{G}$ **faire**
      $A = A \cup \{(v,u)\}$
    Supprimer du problème tous les $v \in V$ et les $u \in U$ 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 $\mathcal{D}$.

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

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.

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

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 éléments 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`.

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.

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.

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 :

```
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 $\mathcal{G}$.

Q36. Écrire une fonction de prototype

```
void supprime_arc(graphe *g, int i,int j)
```

qui supprime l'arc (i, j) du graphe $\mathcal{G}$.

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.

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 **Q26**. On prendra bien garde à la numérotation des éléments de U.

FIN