Union des Professeurs de classes Préparatoires Scientifiques

INFORMATIQUE

2024

Union des Professeurs
de classes préparatoires
Scientifiques

Sommaire

Informatique commune

X – ENS B (XULSR) (2h) - MP, PC, PSI [i224m2e]
Logimage 1
Mines-Ponts (2h) - MP, PC, PSI [i24mmce]
Problèmes de communication numérique 9
CCINP (3h) - PC, PSI [i24psue]
Jeu de l'awalé 22
CCINP (3h) - TPC, TSI [i24piue]
Colorer un graphe 50
Banque PT (4h) [i24bpt]
Voitures électriques 70

MPI et MP option informatique

X – ENS C (XULSR) (4h) - MPI [i224mpiec]
Égalités et différences 94
X – ENS A (XULSR) (4h) - MPI, MP [i224m1ea]
Arbres équilibrés et géométrie algorithmique 113
X – ENS Info fondamentale (XULSR) (4h) - MPI, MP [i24m3eif]
Couverture minimale 124
Mines-Ponts (3h) - MPI [i24mmmpi1e]
Sous-graphe le plus dense 138
Mines-Ponts (3h) - MPI [i24mmmpi2e]
Autour des flots 148
Mines-Ponts (3h) - MP [i24mmoe]
Relation d'ordre 161
Centrale-Supélec (4h) - MPI [i24cmpie]
Le jeu de go 169
Centrale-Supélec (4h) - MP [i24cmoe]
Arbre de Steiner 183
CCINP (4h) - MPI [i24pmpie]
Partie I : algorithme Single Pass - Partie II : langages réguliers - Partie III : correspondance de Burge 191

CCINP (4h) - MP [i24pmoe]
Partie I (informatique commune) : coloration de graphes - Partie II (option informatique) : satisfiabilité d'une formule propositionnelle - Partie III (option informatique) : automates et reconnaissance de motifs 203

CAPES et agrégation d'informatique

CAPES externe d'informatique - Épreuve 1 [i24c31e]
B-arbres 211
CAPES externe d'informatique - Épreuve 2 [i24c32e]
Multiples thèmes 221
Agrégation externe d'informatique - Épreuve 1 [i24ag1e]
Partie 1 : robot de dépôt de solutions chimiques - Partie 2 : tableaux flexibles - Partie 3 : gestion d'un concours de recrutement 236
Agrégation externe d'informatique - Épreuve 2 [i24ag2e]
Modèles de Markov cachés 256
Agrégation externe d'informatique - Épreuve 3 [i24ag3e]
Option A : gestion de compétitions d'ultimate - Option B : déduction naturelle 273

En guise d'introduction

L'informatique, ça fait gagner beaucoup de temps... à condition d'en avoir beaucoup devant soi !
Mireille Sitbon

Le recueil 2024 comporte trois parties. La première contient les épreuves d'informatique de l'enseignement commun à toutes nos filières hors MPI. La deuxième contient les épreuves de MPI et de l'option informatique de MP. Enfin, la troisième contient les épreuves des CAPES et agrégation d'informatique.

Le bulletin des concours Informatique n'est proposé aux adhérents que sous forme électronique. Dans ce recueil, le nom du fichier contenant chaque sujet figure dans le sommaire et en tête de chaque page. Les fichiers sont disponibles sur le site de l'UPS à l'adresse `https://ups-cpge.fr/index.php?module=Sujets&voir=accueil`. Ce bulletin et ses prédécesseurs sont à votre disposition sur le site de l'UPS à l'adresse `https://ups-cpge.fr/index.php?rubrique=7`.

Amélie Stainer - Septembre 2024
`amelie.stainer@prepas.org`

ECOLE POLYTECHNIQUE - ESPCI
ECOLES NORMALES SUPERIEURES

CONCOURS D'ADMISSION 2024

JEUDI 18 AVRIL 2024
16h30 - 18h30

FILIERES MP-MPI-PC-PSI
Epreuve n° 8

INFORMATIQUE B (XELSR)

Durée : 2 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve

Logimage

Le sujet comporte 7 pages, numérotées de 1 à 7.

Un logimage consiste en une grille de `nl` lignes et `nc` colonnes, dans laquelle il faut retrouver une image en noir et blanc à partir de **clés**. Chaque ligne (et chaque colonne) est codée par une liste de clés entières. Un entier m dans une liste de clés correspond à un **bloc** de m cases consécutives à colorier en noir. Par exemple, pour une ligne donnée, la liste de clés $[2, 4]$ signifie qu'en regardant les cases de cette ligne de gauche à droite, on trouve d'abord un certain nombre (éventuellement nul) de cases blanches, puis un bloc de 2 cases noires consécutives, suivi d'au moins une case blanche, puis un bloc de 4 cases noires, puis éventuellement des cases blanches. De même, la liste de clés pour une colonne décrit la taille des blocs rencontrés depuis le haut jusqu'au bas de la colonne. On veut s'assurer que la solution d'un logimage existe et est unique. La figure 1 présente un exemple de logimage et sa solution : les listes en regard des lignes et des colonnes sont les clés codant la solution. Par exemple, la deuxième ligne est codée par une liste de deux clés : $[3, 1]$, alors que la première ligne est codée par une liste contenant une seule clé : $[2]$.

	[1]	[5]	[4]	[2]	[4]
[2]					
[3,1]					
[4]					
[4]					
[1,1]					

(a) Exemple de logimage

	[1]	[5]	[4]	[2]	[4]
[2]		■	■		
[3,1]	■	■	■		■
[4]		■	■	■	■
[4]		■	■	■	■
[1,1]		■			■

(b) Solution

FIGURE 1: Exemple de logimage et sa solution.

Dans ce sujet, nous représenterons les clés des lignes par une liste de listes d'entiers, de même que pour les clés des colonnes. Dans la suite du sujet, `cle_l` représentera la liste des listes de clés par lignes, et `cle_c` la liste des listes de clés par colonnes. Dans l'exemple ci-dessus, on aura donc `cle_l=[[2],[3,1],[4],[4],[1,1]]`. Une **solution** `sol` sera représentée par une liste de listes, telle que `sol[i][j]` représente l'état de la case de la grille sur la ligne `i` et la colonne `j`. Un état sera codé par un entier : 0 pour une case blanche et 1 pour une case noire.

Complexité. La complexité, ou le temps d'exécution, d'une fonction F est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, ...) nécessaires à l'exécution de F. Lorsque cette complexité dépend de plusieurs paramètres n et m, on dira que F a une complexité en $O(\phi(n,m))$ lorsqu'il existe trois constantes A, n_0 et m_0 telles que la complexité de F est inférieure ou égale à $A \cdot \phi(n,m)$, pour tout $n \geq n_0$ et $m \geq m_0$. Lorsqu'il est demandé de donner la complexité d'un programme, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Rappels concernant le langage Python. L'utilisation de toute fonction Python sur les listes autre que celles mentionnées dans ce paragraphe est interdite. Si `a` désigne une liste en Python de longueur n :

- `len(a)` renvoie la longueur de cette liste, c'est-à-dire le nombre d'éléments qu'elle contient ; la complexité de `len` est en $O(1)$.
- `a[i]` désigne le `i`-ème élément de la liste, où l'indice `i` est compris entre 0 et `len(a) - 1` inclus ; la complexité de cette opération est en $O(1)$.
- `a.append(e)` ajoute l'élément `e` à la fin de la liste `a` ; on supposera que la complexité de cette opération est en $O(1)$.
- `a.pop()` renvoie la valeur du dernier élément de la liste `a` et l'élimine de la liste `a` ; on supposera la complexité de cette opération en $O(1)$.
- `a.copy()` fait une copie de la liste `a` ; la complexité de cette opération est en $O(n)$.
- Si `f` est une fonction, la syntaxe `[f(x) for x in a]` permet de créer une nouvelle liste, ayant le même contenu que la liste `b` résultant de l'exécution du code suivant (et avec la même complexité) :

```
b = []
for x in a:
    b.append(f(x))
```

Manipulation de solutions. On pourra utiliser dans la suite les fonctions suivantes pour manipuler les solutions :

- `init_sol(nl,nc,v)` renvoie une nouvelle solution de `nl` lignes et `nc` colonnes dont toutes les cases sont initialisées à `v`.
- `copy_sol(sol)` renvoie une copie de la solution `sol` obtenue en créant une nouvelle solution de même taille et en copiant le contenu de chaque case.

La complexité de chacune de ces fonction est en $O(\texttt{nl} \times \texttt{nc})$.

Convention. Dans le sujet, on supposera que `nl` et `nc` (qui décrivent le nombre de lignes et le nombre de colonnes de la grille) sont des variables globales accessibles à l'intérieur de toutes les fonctions.

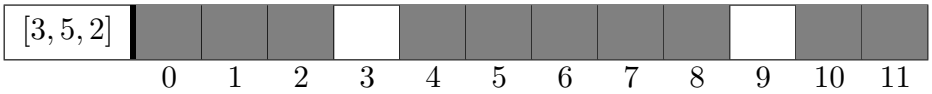
Organisation du sujet. Ce sujet comporte quatre parties, de difficulté croissante. La partie IV dépend des résultats de la partie III, autrement les parties sont indépendantes.

Partie I – Préliminaires et vérification d'une solution

Question 1. *Écrire une fonction* `cases_noires` *qui prend en argument une liste de listes d'entiers* `cle_l` *représentant les clés des lignes d'un logimage et qui renvoie le nombre de cases à colorier en noir dans la solution. Quelle est la complexité de cette fonction ?*

Question 2. *À partir de* `cle_l`*, on peut obtenir le nombre de cases noires de la solution, et de même pour* `cle_c`*. Écrire une fonction* `compatibles` *qui prend ces deux listes de listes d'entiers en arguments et qui renvoie* `True` *si et seulement si elles codent le même nombre de cases noires.*

Les règles de construction des logimages imposent au moins une case blanche entre deux blocs dans une solution valide. La clé d'une ligne permet de calculer la taille minimale de la partie de la ligne occupée par les blocs codés. Par exemple, la clé de ligne `[3,5,2]` nécessite une taille minimale de 12 cases :



Question 3. *Écrire la fonction* `taille_minimale` *qui prend en argument une liste (supposée non vide) d'entiers représentant les clés d'une ligne et renvoie cette taille minimale.*

On souhaite maintenant vérifier si une solution est valide. Pour ceci, on considère la fonction `verif_ligne` ci-dessous, qui a pour but de vérifier que la ligne d'indice `i` de la solution `sol` ne contient que des 0 et des 1 et contient tous les blocs décrits par `cle_l`. On peut s'apercevoir que cette implémentation contient une erreur, qui sera corrigée à la question suivante.

```
1  def verif_ligne(sol, cle_l, i):
2      i_bloc=0
3      taille=0
4      for j in range(nc):
5          couleur_case=sol[i][j]
6          if couleur_case==1:
7              taille=taille+1
8          if taille>0 and (couleur_case==0 or j+1==nc):
9              if i_bloc<len(cle_l[i]) and taille==cle_l[i][i_bloc]:
10                 taille=0
11                 i_bloc=i_bloc+1
12             else:
13                 return False
14     return True
```

On rappelle que l'expression `for j in range(nc)` permet d'itérer pour des valeurs de `j` allant de 0 à `nc` -1.

Question 4. *Pour la fonction* `verif_ligne` *ci-dessus, donner des exemples d'arguments* `sol` *et* `cle_l` *suivant les spécifications ci-dessous. Ces exemples doivent comporter une seule ligne* ($\texttt{nl} = 1$) *et au plus 5 colonnes* ($\texttt{nc} \leq 5$).

1. *Donner et expliciter deux exemples pour lesquels la fonction renvoie* `False` *à la ligne 13 (pour deux raisons différentes).*
2. *Donner un exemple pour lequel le résultat de la fonction* `verif_ligne` *est incorrect. Expliquer comment la modifier pour obtenir un résultat correct.*

Partie II – Résolution systématique

On cherche maintenant à construire une solution du logimage. On considère une **solution partielle** `sol_p`, représentée par une liste de listes (comme précédemment, telle que `sol_p[i][j]` représente l'état de la case de la grille sur la ligne `i` et la colonne `j`), mais où toutes les cases ne sont pas encore déterminées à 0 ou à 1. Pour ceci, on codera par l'entier -1 la valeur initiale représentant une case non encore déterminée. Dans les schémas suivants, nous représenterons une telle case de couleur indéterminée par un point d'interrogation, comme illustré ci-dessous :

valeur -1 : ? valeur 0 : valeur 1 :

Une première méthode pour chercher une solution à partir des clés de lignes et de colonnes consiste à tester toutes les grilles possibles une par une. On suppose ici qu'on dispose d'une fonction `verif(sol_p,cle_l,cle_c)` qui renvoie `True` si et seulement si la solution `sol_p` est une solution complète et valide du logimage décrit par `cle_l` et `cle_c`, et dont la complexité est en $O(\texttt{nl} \times \texttt{nc})$. Nous allons remplir les cases ligne après ligne de haut en bas et de gauche à droite.

Question 5. *On suppose que les n premières cases de la grille sont déjà fixées à 0 ou à 1 ; ce sont les cases des k premières lignes (le plus en haut), ainsi que les ℓ cases les plus à gauche de la $(k+1)^{\text{ème}}$ ligne. Exprimer k et ℓ à partir de la valeur de n.*

Question 6. *Écrire une fonction* `liste_solutions` *qui prend pour argument les clés des lignes et des colonnes* `cle_l` *et* `cle_c` *et qui teste systématiquement toutes les combinaisons possibles de 0 et 1 des cases de la grille, en utilisant la fonction* `verif` *sur chacune d'entre elles, pour renvoyer la liste de toutes les solutions valides. On utilisera une fonction auxiliaire récursive* `liste_solutions_aux` *qui prend comme arguments un entier* `n`*, une solution partielle* `sol_p` *ainsi qu'une liste de solutions* `liste` *et qui ajoute à* `liste` *toutes les solutions valides dont les* `n` *premières cases (au sens de la question précédente) sont celles dont la valeur est fixée dans* `sol_p`*. Donner la complexité de la fonction* `liste_solutions`*.*

On peut diminuer le nombre de grilles testées en arrêtant l'exploration quand une ligne (ou une colonne) contient plus de cases noires que ce qui est dicté par la liste de ses clés.

Question 7. *Comment modifier la fonction de la question 6 pour ne pas explorer les grilles contenant trop de cases noires sur au moins une ligne ou une colonne ? On pourra se contenter d'indiquer les modifications à faire dans le code.*

Partie III - Placements possibles d'un bloc

Les algorithmes de la partie précédente ont une complexité trop élevée pour permettre de résoudre des grilles de grande taille. Dans cette partie, on considère un bloc particulier et on cherche à savoir où il peut être placé dans une ligne, en fonction des cases déjà déterminées dans cette ligne (cases blanches ou noires). Dans toute cette partie, on s'intéresse à une seule ligne de la grille, d'indice `i_ligne` fixé. Cet indice et la solution partielle `sol_p` pourront être considérés comme des variables globales accessibles à toutes les fonctions.

On cherche à placer un nouveau bloc de taille `s` dans la ligne d'indice `i_ligne`. On dit qu'un bloc de `s` cases **commence à l'indice** `j` s'il occupe les cases d'indice `j`, `j+1`,...,`j+s-1` de la ligne. Soit `c` l'indice d'une case dans la ligne considérée ($0 \leq \texttt{c} < \texttt{nc}$). On cherche d'abord

à savoir si le bloc de taille `s` que l'on souhaite placer peut commencer à l'indice `c`, en fonction des décisions déjà prises, c'est-à-dire des valeurs de la liste `sol_p[i_ligne]` qui sont déjà fixées à 0 ou à 1. On dit qu'**une case** `i` de la ligne **est en conflit avec le placement du bloc en position** `c` si la valeur de `sol_p[i_ligne][i]` empêche de faire commencer le bloc en position `c`, c'est-à-dire si :

— (a) $\texttt{i} = \texttt{c} - 1$ et `sol_p[i_ligne][i]=1` (`i` est une case noire juste avant le bloc),

— (b) $\texttt{c} \leq \texttt{i} < \texttt{c} + \texttt{s}$ et `sol_p[i_ligne][i]=0` (`i` est une case blanche au milieu du bloc),

— (c) ou $\texttt{i} = \texttt{c} + \texttt{s}$ et `sol_p[i_ligne][i]=1` (`i` est une case noire juste après le bloc).

Considérons par exemple la ligne de 10 cases représentée ci-dessous, avec deux cases prédéterminées (la case 4 est noire et la case 2 est blanche), dans laquelle on veut placer un bloc de taille $\texttt{s} = 3$. La case $\texttt{i} = 4$ est en conflit avec le placement du bloc commençant en position $\texttt{c} = 5$ à cause de (a). Pour un placement commençant en position $\texttt{c} = 1$, les cases $\texttt{i} = 2$ et $\texttt{i} = 4$ sont en conflit, respectivement à cause de (b) et (c). Pour un placement commençant en position $\texttt{c} = 3$, aucune case n'est en conflit.

?	?		?		?	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Question 8. *Écrire une fonction* `conflit(c,s)` *qui, pour un bloc de taille* `s` *que l'on souhaite faire commencer à la case d'indice* `c`*, renvoie la plus petite position en conflit avec ce placement. S'il n'y a aucun conflit pour ce placement, la fonction doit renvoyer* `nc`*. On supposera que* $\texttt{c} + \texttt{s} \leq \texttt{nc}$*. La complexité doit être en* $O(\texttt{s})$ *et sera justifiée.*

On veut maintenant trouver la **première position possible** d'un bloc, c'est-à-dire l'indice le plus petit auquel un bloc peut commencer. On cherche ici la première position possible d'un bloc **à partir d'un indice** `c` (donc supérieure ou égale à `c`). Dans l'exemple précédent, la première position possible pour placer un bloc de taille $\texttt{s} = 3$ à partir de l'indice $\texttt{c} = 1$ est 3. On pourrait utiliser la fonction `conflit` sur tous les indices à partir de `c`, avec une complexité $O(\texttt{nc} \times \texttt{s})$, mais on veut une solution plus efficace.

Question 9. *Écrire une fonction* `prochain(c,s)` *qui renvoie la première position valide supérieure ou égale à* `c` *à laquelle peut commencer un bloc de taille* `s`*, et* -1 *si une telle position n'existe pas. La complexité de la fonction devra être en* $O(\texttt{nc})$ *et sera justifiée.*

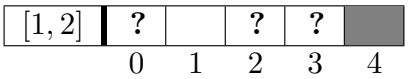
Partie IV - Placements possibles de tous les blocs d'une ligne

Comme dans la partie précédente, on se concentre ici sur le placement des blocs sur la ligne d'indice `i_ligne`. On pourra considérer que `i_ligne`, `sol_p` et `cle_l` sont des variables globales accessibles à toutes les fonctions.

On cherche maintenant à prendre en compte toutes les cases déjà coloriées en noir : si comme dans l'exemple ci-dessous, une case noire a été fixée en toute fin de ligne, alors cette case doit nécessairement être incluse dans le dernier bloc. Dans l'exemple ci-dessous, le dernier bloc (de taille 3) peut commencer uniquement à l'indice 7 afin qu'il contienne la case noire en position 9.

[1, 2, 3]	?		?	?		?	?	?	?	
	0	1	2	3	4	5	6	7	8	9

Nous allons procéder par programmation dynamique, en construisant une matrice `M` de taille $\texttt{nc} \times \texttt{B}$, où $\texttt{B} = \texttt{len(cle_l[i_ligne])}$ est le nombre de blocs sur la ligne considérée. Pour calculer la valeur de `M[c][b]`, on considère le sous-problème où il faut placer les blocs d'indice 0 à `b` dans la portion de ligne contenant les cases d'indice `0` à `c` de la ligne, en prenant en compte toutes les cases noires parmi elles. Pour l'exemple ci-dessus, `M[4][1]` correspond ainsi au sous-problème qui considère uniquement les cases 0 à 4 de la ligne, et cherche à y placer les blocs d'indice 0 et 1. Ce sous-problème est représenté ci-dessous :



Dans ce sous-problème, `M[c][b]` indique la **première position possible** pour le bloc d'indice `b`, **en prenant en compte toutes les cases noires** dans la portion de ligne considérée. Dans l'exemple ci-dessus, le bloc d'indice 1 (de taille 2) devra nécessairement inclure la case d'indice 4 car celle-ci est déjà coloriée en noir. On aura donc $\texttt{M[4][1]} = 3$. Lorsque les blocs d'indice 0 à `b` ne peuvent pas être placés dans les cases d'indice 0 à `c`, on pose par convention $\texttt{M[c][b]} = -1$.

On note `s` la taille du bloc d'indice `b` dont on veut calculer la première position possible : $\texttt{s} = \texttt{cle_l[i_ligne][b]}$. On considère tout d'abord le cas d'un bloc qui n'est pas le premier de la ligne ($\texttt{b} > 0$). On peut exprimer la valeur de `M[c][b]` de façon récursive, en considérant les deux cas suivants :

1. Si le dernier bloc pouvait être placé sans la dernière case ($\texttt{M[c-1][b]} \geq 0$) et que la dernière case n'est pas noire ($\texttt{sol_p[i_ligne][c]} ! = 1$) alors on peut placer ce bloc de la même façon que sans la dernière case, c'est-à-dire $\texttt{M[c][b]} = \texttt{M[c-1][b]}$.
2. Sinon, on regarde si on peut placer le bloc en toute fin de ligne (donc le faire commencer à la case d'indice $\texttt{c} - \texttt{s} + 1$). Pour ceci, il faut que les deux conditions ci-dessous soient réunies :
   (a) $\texttt{conflit(c-s+1,s)} > \texttt{c}$ (il n'y a pas de conflit avec les cases de la ligne jusqu'à `c`)
   (b) $\texttt{M[c-s-1][b-1]} \geq 0$ (il y a suffisamment de place dans les cases d'indice 0 à $\texttt{c} - \texttt{s} - 1$ pour placer les blocs précédents, afin de laisser une case blanche à l'indice $\texttt{c} - \texttt{s}$)

   Si ces deux conditions sont vérifiées, alors on peut placer le bloc en position $\texttt{c} - \texttt{s} + 1$, donc $\texttt{M[c][b]} = \texttt{c} - \texttt{s} + 1$. Sinon, le dernier bloc ne peut pas être placé : $\texttt{M[c][b]} = -1$.

Question 10. *On suppose que les valeurs de* `M[c][0]` *ont été précalculées pour tout* `c`*. Écrire un algorithme* `calcul_matrice` *qui prend* `M` *en argument et qui calcule le reste de la matrice par programmation dynamique. La complexité de la fonction devra être en* $O(\texttt{nc}^2)$ *et sera justifiée.*

Dans le cas du premier bloc ($\texttt{b} = 0$), une attention particulière doit être portée à la première case noire de la ligne, si elle existe. On note `p` la position de la première case noire si elle existe ($\texttt{sol_p[i_ligne][p]} = 1$ et $\texttt{sol_p[i_ligne][j]} \neq 1$ pour tout $\texttt{j} < \texttt{p}$) ou $\texttt{p} = -1$ s'il n'y a pas de case noire dans `sol_p`.

Question 11. *Expliquer en quoi la position de la première case noire contraint la valeur de* `M[c][0]`*. En prenant en compte cette contrainte, modifier le calcul de* `M[c][b]` *détaillé ci-dessus pour le cas* $\texttt{b} = 0$*. On exprimera* `M[c][0]` *en fonction de* `sol_p[i_ligne]`*, des valeurs de* `M[c'][0]` *pour* $\texttt{c'} < \texttt{c}$*, de* `p` *et du résultat de* `conflit(c-s+1,s)`*.*

Question 12. *Écrire une fonction* `premiere_case` *qui prend en argument la matrice* `M` *et qui renvoie une liste de la même taille que* `cle_l[i_ligne]` *dont l'élément d'indice* `b` *est la*

première position possible du bloc décrit par `cle_l[i_ligne][b]`*, étant donnés toutes les cases prédéterminées et tous les autres blocs de la ligne. S'il n'existe pas de solution, alors la fonction renverra une liste vide. On pourra déterminer les premières positions possibles des blocs en commençant par ceux les plus à droite (donc d'indices les plus grands).*

On appelle `liste_pp` la **liste des premières positions** calculée dans la question précédente. De façon similaire, on peut calculer `liste_dp`, la **liste des dernières positions** possibles pour le **début de chaque bloc**, c'est-à-dire le plus grand indice `c` auquel peut **commencer** chaque bloc. Lorsque la première et la dernière position d'un bloc sont proches, on peut déterminer avec certitude que certaines cases appartiennent à ce bloc et doivent donc être coloriées en noir. Par exemple, pour la ligne représentée ci-dessous avec ses clés, nous obtenons `liste_pp=[0,5,10]` et `liste_dp=[5,8,13]`. Le second bloc commence donc au plus tôt à la case 5, et au plus tard à la case 8. Comme ce bloc est de taille 4, il contiendra nécessairement la case 8, que l'on peut donc colorier en noir.

$[2,4,1]$	?	?		?		?	?	?	?	?	?	?	?	?
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Question 13. *Écrire une fonction* `remplissage` *qui prend en arguments les listes* `liste_pp` *et* `liste_dp` *et colorie en noir toutes les cases de la ligne* `sol_p[i_ligne]` *qui peuvent être déduites de ces deux listes par le critère précédent. On suppose que les listes* `liste_pp` *et* `liste_dp` *ne sont pas vides, c'est-à-dire qu'il existe une solution et que la ligne contient au moins un bloc.*

On peut déduire d'autres propriétés des listes `liste_pp` et `liste_dp`. Si on reprend l'exemple précédent, la case 3 est entourée de deux cases à 0 (blanches), donc ne pourrait participer qu'à un bloc de taille 1. Les deux listes `liste_pp` et `liste_dp` nous disent cependant que le seul bloc dont l'intervalle de positions contient cette case est le bloc 0 qui est de taille 2. On peut donc en déduire que la case 3 est blanche dans la solution. Plus généralement, pour une case d'indice `i` donné, on peut calculer la taille maximale `m` d'un bloc qui contiendrait cette case à partir des cases blanches les plus proches qui encadrent `i`. À l'aide des listes `liste_pp` et `liste_dp`, on sait quels blocs sont susceptibles de couvrir la case d'indice `i`. Si la taille minimale de ces blocs est plus grande que `m`, alors aucun bloc ne pourra couvrir la case d'indice `i`, qui est donc nécessairement blanche.

Question 14. *Écrire une fonction* `cases_blanches` *qui prend en arguments les listes* `liste_pp` *et* `liste_dp` *et colorie en blanc les cases de* `sol_p[i_ligne]` *qui ne peuvent faire partie d'aucun bloc d'après le critère précédent, avec les mêmes hypothèses sur les listes* `liste_pp` *et* `liste_dp` *que dans la question précédente.*

Fin du sujet.

A2024 – IC

ÉCOLE DES PONTS PARISTECH,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARIS,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.

Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).

CONCOURS 2024

ÉPREUVE D'INFORMATIQUE COMMUNE

Durée de l'épreuve : 2 heures

L'usage de la calculatrice ou de tout autre dispositif électronique est interdit.

Cette épreuve est commune aux candidats des filières MP, PC et PSI.

Les candidats sont priés de mentionner de façon apparente
sur la première page de la copie :

INFORMATIQUE COMMUNE

L'énoncé de cette épreuve comporte 12 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé,
il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives
qu'il est amené à prendre.

Les sujets sont la propriété du GIP CCMP. Ils sont publiés sous les termes de la licence
Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 France.
Tout autre usage est soumis à une autorisation préalable du Concours commun Mines Ponts.

Rappels concernant le langage Python : il n'est pas possible d'utiliser des fonctions internes à Python sur les listes ou les chaînes de caractères telles que **min**, **max**, `count`, `remove` ... Seules les instructions basiques telles que **len**`(liste)`, `liste.append(e)` sont autorisées. Le tranchage ou slicing ainsi que la concaténation sont également permis. Les programmes doivent être commentés lorsque c'est nécessaire pour justifier les choix. Il n'est pas utile de rappeler la signature des fonctions demandées dans les différentes questions.

Introduction à deux problèmes en communication numérique

Introduction

On s'intéresse au problème de communication entre deux personnes, nommées Alice et Bob qui cherchent à s'envoyer un message au travers d'un canal de communication (une bande de fréquences radio par exemple).

Avant d'être lu par Bob, le message original d'Alice passe par plusieurs étapes que nous allons séparer de la manière suivante :

- une phase de **compression**, durant laquelle Alice cherche à trouver la représentation la plus compacte possible du message,
- une phase d'**encodage** durant laquelle le message compressé est transformé en une succession de symboles transmissibles au travers du canal de communication utilisé,
- une phase de **transmission** durant laquelle le message encodé circule sur le canal de communication et est susceptible de subir une altération,
- une phase de **décodage** durant laquelle Bob décode le message qu'il a reçu, le message lui apparaît alors sous la forme compressée,
- une phase de **décompression** durant laquelle Bob applique l'opération réciproque de la compression opérée par Alice.

Ce modèle est décrit par le schéma de la Figure 1 :

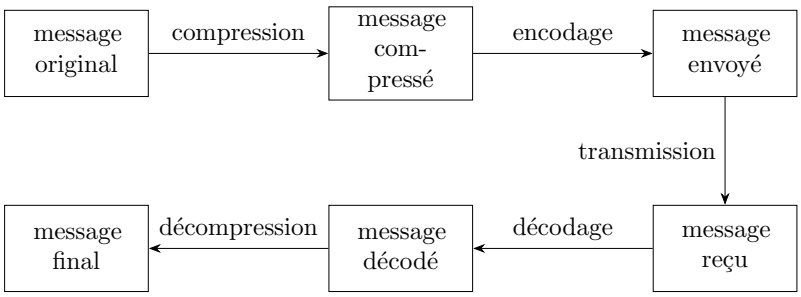


FIGURE 1 – Schématisation du modèle de communication considéré ici.

Dans cette épreuve, nous allons nous intéresser uniquement à deux phases : la compression du message d'origine par Alice en un message compact et le décodage par Bob d'un message transmis, potentiellement entaché d'erreurs.

1 Compression du message d'Alice : codage arithmétique

Le codage ASCII (American Standard Code for Information Interchange) définit une norme où 128 caractères sont codés sur 7 bits. Ce codage est illustré par les quelques lignes suivantes :

Caractère	Codage binaire	Équivalent numérique
`'a'`	110 0001	97
`'b'`	110 0010	98
`'z'`	111 1010	122

Lorsqu'une chaîne de caractères n'utilise pas l'intégralité des 128 caractères proposés par le codage ASCII, il est possible de convenir d'une représentation différente, plus économique en nombre de symboles. Considérons la chaîne de caractères `s='abaabaca'`. On peut proposer de coder celle-ci à l'aide du tableau suivant :

Caractère	Code
`'a'`	`00`
`'b'`	`01`
`'c'`	`10`

Dans ce cas, la chaîne de caractères `s` est codée sur 16 bits par :

```
00 01 00 00 01 00 10 00
```

(où des espaces ont été introduits pour faciliter la lecture).

Dans un souci de compression de l'information, il est intéressant de représenter les caractères les plus fréquents par des expressions courtes et de ne plus nécessairement coder avec des codes de longueur constante chaque caractère. Dans l'exemple précédent, il est possible de coder le caractère `'a'` avec 1 bit, et les caractères `'b'` et `'c'` avec 2 bits afin de coder la chaîne `s` sur seulement 11 bits en tout.

❑ **Q1** – Proposer une telle représentation en expliquant pourquoi celle-ci pourra être décodée sans ambiguïté. Vous ferez en sorte que la représentation binaire de `'a'` soit inférieure à celle de `'b'`, elle-même inférieure à celle de `'c'`.

La représentation précédente emploie la même longueur pour coder les caractères `'b'` et `'c'` alors que le caractère `'b'` est deux fois plus présent que le caractère `'c'` dans la chaîne `s`.

Il est possible d'aller un cran plus loin et le codage arithmétique présenté dans cette étude permet un gain de compression comme s'il parvenait à représenter un caractère avec un nombre non entier de bits au prorata de sa fréquence d'apparition.
Ce principe de compression est notamment utilisé par la norme *JPEG2000* de compression des images. Nous ne le présenterons cependant ici que dans le cadre de l'étude de chaînes de caractères.

1.1 Analyse du texte source

L'objet de cette partie est d'analyser le contenu d'une chaîne de caractères `s` afin de déterminer :

— les caractères utilisés par la chaîne `s` ;
— le nombre d'occurrences de chacun.

❏ **Q2** – Écrire une fonction nommée `nbCaracteres(c:str,s:str)->int` qui prend comme argument un caractère `c`, une chaîne `s` et qui renvoie le nombre d'occurrences (c'est-à-dire le nombre d'apparitions) de `c` dans `s`. La fonction doit avoir une complexité linéaire en `n`, la longueur de la chaîne `s`.

❏ **Q3** – Pour déterminer la liste des caractères utilisés à l'intérieur d'une chaîne `s` on utilise la fonction définie ci-dessous :

```
1 def listeCaracteres(s:str):
2     listeCar = []
3     n = len(s)
4     for i in range(n):
5         c = s[i]
6         if not(c in listeCar):
7             listeCar.append(c)
8     return listeCar
```

Que renvoie cette fonction lorsque `s='abaabaca'` ? Expliquer succinctement le principe de fonctionnement de cette fonction.

❏ **Q4** – En fonction de la longueur `n` de la chaîne et du nombre `k` de caractères distincts dans celle-ci, déterminer la complexité asymptotique dans le pire des cas de la fonction de la question **Q3**. Par exemple pour `s='abaabaca'`, on a $n = 8$ et $k = 3$. On négligera la complexité des `append` mais pas celle des tests d'appartenance de la forme `i in L`. Autrement dit, la ligne 7 est considérée comme étant de complexité constante et la ligne 6 de complexité linéaire en la longueur de la liste `listeCar`.

❏ **Q5** – On définit alors une fonction `analyseTexte(s:str)->list`

```
1 def analyseTexte(s:str):
2     R = []
3     l = listeCaracteres(s)
4     for i in range(len(l)):
5         c = l[i]
6         R.append((c, nbCaracteres(c, s)))
7     return R
```

Expliquer ce que fait cette fonction et donner la valeur renvoyée par la commande `analyseTexte('babaaaabca')`.

❏ **Q6** – En fonction de la longueur n de `s` et du nombre k de caractères distincts présents

dans `s`, (autrement dit k est la longueur de `listeCaracteres(s)`), donner une estimation de la complexité asymptotique dans le pire des cas de la fonction `analyseTexte`.

❏ **Q7** – Adapter la fonction de la question **Q5** pour qu'elle utilise (et renvoie) un dictionnaire. Elle devra avoir une complexité,

i. linéaire en la longueur n de `s`,

ii. indépendante de k nombre de caractères distincts présents dans `s`.

De plus, cette fonction devra impérativement ne parcourir qu'une seule fois la chaîne de caractères. On admettra qu'un test d'appartenance d'une clé à un dictionnaire se fait à coût constant. Par exemple, `analyseTexte('abracadabra')` renverra `{'a':5, 'b':2, 'r':2, 'c':1, 'd':1}`.

1.2 Exploitation d'analyses existantes

On peut éviter de réaliser une analyse de la chaîne de caractères à coder en exploitant des données statistiques disponibles. La numérisation de larges corpus littéraires a permis la création de bases de données contenant des informations relatives au nombre d'occurrences des différents caractères alphanumériques dans diverses langues. Nous allons supposer disposer d'une base de données constituée de trois tables (les clefs primaires sont soulignées) :

— caractere(<u>idCar</u>, symbole, typeCaractere, codeHTML) ;

— corpus(<u>idLivre</u>, titre, auteur, annee, nombreCaracteres, langue) ;

— occurrences(idCar, idLivre, nombreOccurrences) ;

dont voici quelques extraits :

Table caractere :

idCar	symbole	typeCaractere	codeHTML
65	`'A'`	`'lettre'`	`'A'`
48	`'0'`	`'nombre'`	`'0'`

Table corpus :

idLivre	titre	auteur	annee	nombreCaracteres	langue
1	`'Germinal'`	`'Zola'`	1885	1152365	`'Français'`
2	`'Les Misérables'`	`'Hugo'`	1862	2245300	`'Français'`

Table occurrences :

idCar	idLivre	nombreOccurrences
62	31	155
37	21	1550

❏ **Q8** – Écrire en langage SQL une requête permettant d'obtenir la liste sans doublon des auteurs présents dans la table corpus.

❏ **Q9** – Écrire une requête SQL permettant d'obtenir la fréquence d'occurrences (donc comprise entre 0 et 1) de chaque caractère en `'Français'`. Plus précisément, la requête

devra renvoyer une table à deux attributs : une colonne contenant le symbole de chaque caractère, et une autre colonne contenant le rapport entre le nombre total d'occurrences du caractère et le nombre total de caractères des textes de tout le corpus.

1.3 Compression

La compression par codage arithmétique consiste à représenter une chaîne de caractères `s` par un nombre réel déterminé à l'intérieur de l'intervalle $[0; 1[$.

Initialement, on attribue à chaque caractère utile une portion de l'intervalle $[0; 1[$ proportionnelle à sa fréquence d'occurrences. Par exemple, pour un alphabet à 5 lettres `'abcde'`, on pourrait avoir un tableau comme ci-dessous :

Caractère	`'a'`	`'b'`	`'c'`	`'d'`	`'e'`
Fréquence	0.2	0.1	0.2	0.4	0.1
Intervalle	$[0; 0.2[$	$[0.2; 0.3[$	$[0.3; 0.5[$	$[0.5; 0.9[$	$[0.9; 1[$

La chaîne de caractères `s` est codée en partant de l'intervalle $[0; 1[$. À chaque caractère successif de celle-ci, on affine cet intervalle en ne considérant que la portion correspondant au caractère lu.

Si par exemple la chaîne à coder est `s = 'dac'` :

— on obtient d'abord l'intervalle $[0.5; 0.9[$ correspondant au caractère `'d'` ;

— le caractère `'a'` détermine alors le sous-intervalle $[0.50; 0.58[$ de $[0.5; 0.9[$ correspondant à la portion associée au caractère `'a'`.

— le caractère `'c'` détermine enfin l'intervalle $[0.524; 0.540[$.

La figure qui suit illustre ce processus.

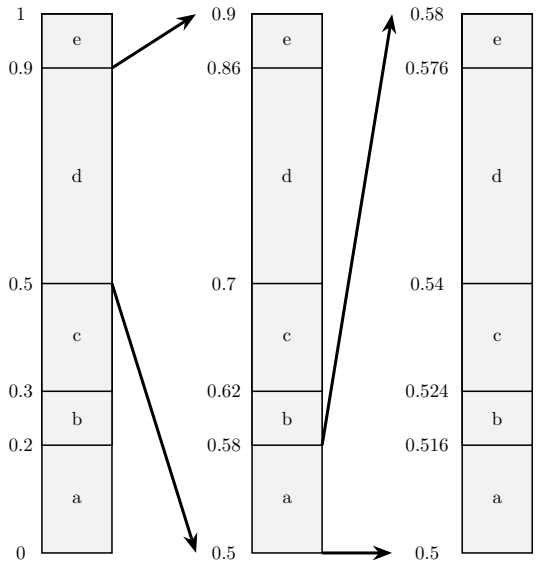


FIGURE 2 – Encodage de `s = 'dac'`

❏ **Q10** – En considérant la table des fréquences précédente, proposer l'intervalle correspondant à la chaîne `s='bac'`.

On suppose disposer d'une fonction `codeCar(car:str,g:float,d:float)->(float,float)` qui prend en argument un caractère `car` et les deux extrémités d'un intervalle $[g, d[$ et qui renvoie un tuple composé des extrémités du sous-intervalle de $[g, d[$ déterminé par le caractère `car`. En reprenant l'illustration précédente `codeCar('b',0,1)` produit `(0.2,0.3)` et `codeCar('a',0.5,0.9)` produit `(0.5,0.58)`.

❏ **Q11** – Écrire une fonction `codage(s:str)->(float,float)` prenant en argument la chaîne `s` et fournissant en réponse le tuple `(g,d)` constitué des deux extrémités de l'intervalle $[g, d[$ produit par l'algorithme de codage précédent.

Le codage arithmétique consiste alors à coder la chaîne `s` par un flottant `x` choisi arbitrairement à l'intérieur de l'intervalle $[g, d[$.

1.4 Décodage

Pour effectuer le décodage d'un flottant `x`, il suffit de repérer dans quelle succession d'intervalles celui-ci se trouve.

À titre d'exemple, reprenons le tableau précédent et considérons le nombre `x=0.123`

Caractère	'a'	'b'	'c'	'd'	'e'
Fréquence	0.2	0.1	0.2	0.4	0.1
Intervalle	$[0; 0.2[$	$[0.2; 0.3[$	$[0.3; 0.5[$	$[0.5; 0.9[$	$[0.9; 1[$

Puisque `x` appartient à l'intervalle $[0; 0.2[$, le premier caractère est un `'a'`. Puisque `x` appartient au sous-intervalle $[0.10; 0.18[$, le caractère suivant est un `'d'`, *etc.*

❏ **Q12** – Déterminer le caractère qui suit `'ad'` dans la chaîne codée par `x=0.123` en spécifiant le sous-intervalle qui a permis de décoder ce caractère.

❏ **Q13** – Dans le cadre de l'exemple de cette partie, indiquer deux chaînes qui peuvent correspondre au flottant `0.2`. Expliquer par une phrase ce qui est à l'origine de cette ambiguïté.

Une solution possible pour résoudre le problème précédent consiste à introduire un caractère nouveau signifiant la fin de la chaîne de caractères.

Nous conviendrons de désigner ce caractère par `'#'`. Ce caractère se voit attribuer une plage non vide au voisinage de 0. Dans la suite, on suppose que la table des fréquences est adaptée de sorte à prendre en compte la présence de ce nouveau caractère.

On suppose disposer, en plus de la fonction `codeCar(car,g,d)` précédente, d'une fonction `decodeCar(x:float,g:float,d:float)->str` qui détermine le caractère correspondant à la valeur `x` quand celle-ci est comprise dans la plage de $[g, d[$.
Par exemple `decodeCar(0.123,0,1)` donne `'a'` tandis que `decodeCar(0.123,0,0.2)` donne le caractère `'d'`.

❑ **Q14** – Écrire une fonction `decodage(x:float)->str` produisant la chaîne de caractères `s` déterminée par la valeur de `x` (avec le caractère `'#'` compris).

Le codage précédent ne fonctionne que sous l'hypothèse illusoire d'exactitude des calculs jusqu'à une très grande précision. Cette précision est cependant limitée par la taille de la mantisse représentant un réel et il ne peut être mis en œuvre sous la forme précédente qu'avec des chaînes relativement courtes. Pour résoudre cette difficulté, on adopte un codage en arithmétique entière : au lieu de considérer un intervalle flottant $[0, 1[$, c'est un intervalle entier $[\![000; 999]\!]$ qui est utilisé. Cette technique ne sera pas abordée dans le cadre de cette épreuve de durée limitée.

2 Décodage du message reçu par Bob à l'aide de l'algorithme de Viterbi

2.1 Modélisation du canal de communication par un graphe

Dans cette partie, nous allons désormais considérer que le message compressé par Alice a été envoyé au travers d'un canal de communication. À cette fin, et indépendamment de la phase de compression étudiée dans la première partie, le message a subi une deuxième phase de transformation, dite d'encodage (cf Figure 1).

Le message envoyé sur le canal est une suite de **symboles** à valeurs dans un **alphabet**, noté Σ, comportant K symboles. Le choix d'un alphabet efficace n'est pas l'objet de notre étude et constitue un sujet à part entière.

Suite au passage dans le canal de communication, le message envoyé subit une **altération** de sorte que Bob reçoit une séquence de symboles de Σ qui ne correspond pas nécessairement à celle qui a été émise.

Dans cette partie, nous allons voir une approche permettant à Bob de décoder le message reçu et de potentiellement corriger quelques erreurs liées à la transmission du message et à la connaissance *a priori* de la propension du canal de communication à altérer les symboles du message lors de la transmission.

La modélisation proposée est la suivante :

— Bob observe une suite de N symboles $obs_0, \ldots, obs_{N-1}$, que nous allons représenter par une liste Python $Obs = [obs_0, \ldots obs_{N-1}]$,

— pour simplifier, on supposera que l'alphabet Σ est un ensemble de K entiers consécutifs commençant à 0, de sorte que $\Sigma = [\![0, K-1]\!]$. Par exemple si $K = 3$ et $N = 8$, un message valide reçu par Bob pourrait être `[2,0,0,2,1,1,0,0]`.

— chacun des symboles observés obs_t correspond à l'altération d'un symbole s_t envoyé par Alice. On note $[s_0, \ldots s_{N-1}]$ le message original ; pour reprendre l'exemple précédent, Alice pourrait avoir envoyé `[2,0,0,2,1,1,2,0]`.

— on connaît, pour chaque paire $(i, j) \in \Sigma^2$, la probabilité $E_{i,j}$ que le canal altère le symbole j en un symbole i. On stocke ces probabilités dans une liste de listes `E` ; autrement dit, `E[i][j]` est la probabilité conditionnelle d'observer le symbole i sachant que le symbole j a été émis.

Ici (par exemple) on pourrait considérer : $E = \begin{pmatrix} 0.7 & 0.2 & 0.3 \\ 0.2 & 0.7 & 0.1 \\ 0.1 & 0.1 & 0.6 \end{pmatrix}$, représentée par la liste de listes : `E = [[0.7,0.2,0.3],[0.2,0.7,0.1],[0.1,0.1,0.6]]`.

Le fait que `E`[2][0] vaille 0.1 signifie donc que la probabilité que le symbole observé par Bob soit un 2 sachant qu'Alice a émis un 0 est de 0.1.

— on suppose également que le symbole courant s_t envoyé par Alice a une incidence sur le symbole suivant s_{t+1} qu'elle peut envoyer, au même titre que dans une langue comme le français, la probabilité d'observer un `'t'` dans un mot, n'est pas la même suivant que le caractère précédent est un `'e'` ou un `'z'`.

Ainsi pour chaque couple de symboles $(i, j) \in \Sigma^2$, on suppose que l'on connaît la probabilité d'émettre le symbole j à l'instant $t + 1$ sachant que symbole i a été émis à l'instant t. On suppose également que cette probabilité ne dépend pas de t.

L'information concernant ces probabilités de transition d'un symbole à l'autre peut se stocker dans une matrice P de taille $K \times K$, que l'on représente informatiquement par une liste de listes `P`. Chaque entrée `P[i][j]` donne la probabilité qu'Alice émette le symbole j à l'instant $t + 1$ sachant qu'elle a émis le symbole i à l'instant t. En d'autres termes, `P[i][j]` $= P_{s_t=i}(s_{t+1} = j)$.

On prendra ici à titre d'exemple : $P = \begin{pmatrix} 0.3 & 0.2 & 0.5 \\ 0.4 & 0.4 & 0.2 \\ 0.2 & 0.3 & 0.5 \end{pmatrix}$, représentée par la liste de listes : `P=[[0.3,0.2,0.5],[0.4,0.4,0.2],[0.2,0.3,0.5]]`.

Le fait que `P`[2][0] vaille 0.2 signifie donc que la probabilité que le symbole envoyé par Alice à l'instant $t + 1$ soit un 0 sachant que celui envoyé à l'instant t est un 2 vaut 0.2.

Nous allons désormais nous intéresser au problème du décodage : étant donné la liste $Obs = [obs_0, \ldots, obs_{N-1}]$ des symboles observés par Bob, quelle séquence $\tilde{s}_0, \ldots, \tilde{s}_{N-1}$ est la plus probable ?

En d'autres termes, $\tilde{s}_0, \ldots, \tilde{s}_{N-1}$ est l'*estimation* la plus probable faite par Bob du message d'origine $s_0, \ldots, s_{N-1}$, étant donné les observations $obs_0, \ldots, obs_{N-1}$.

La modélisation précédente peut se représenter à l'aide d'un graphe défini comme suit (voir Figure 3 pour un exemple) :

— on crée un sommet $S_{i,j}$ pour chaque symbole possible $0 \leqslant i \leqslant K-1$ et chaque indice d'observation $0 \leqslant j \leqslant N - 1$. Chaque couche verticale dans le graphe correspond à un caractère dans le message. Chaque strate horizontale correspond à un symbole.

— au niveau de la j-ème couche verticale, les sommets $S_{i,j}$ pour $j < N - 1$ ont pour successeurs les états $S_{k,j+1}$ pour tous les symboles k possibles,

— par commodité, on ajoute un état source σ correspondant au début du message décodé et un état cible τ correspondant à la fin du message, ces états étant respectivement reliés à la première et la dernière couche,

— le décodage du message envoyé par Alice correspond à un chemin entre σ et τ dans ce graphe. A chaque sommet du chemin correspond une lettre décodée. Par exemple, le chemin passant par $S_{0,0}, S_{2,1}, S_{0,2}, S_{1,3}$, correspond au décodage de $[0, 2, 0, 1]$.

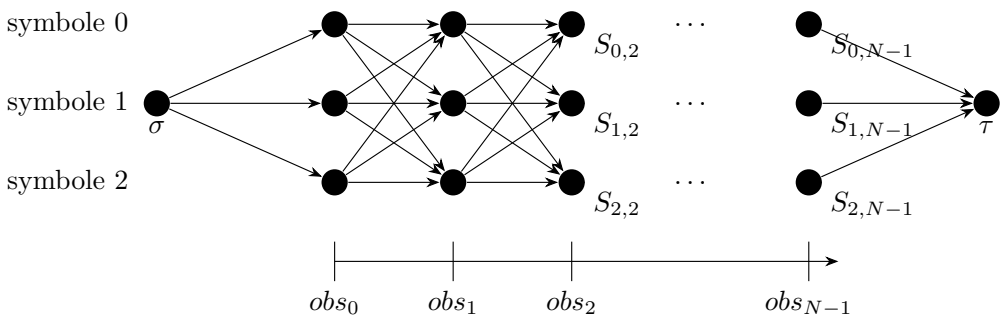


FIGURE 3 – Illustration du modèle de décodage considéré ici.

À chacune des observations obs_i on peut faire correspondre K états. Chacun de ces états correspond à l'un des symboles potentiellement émis par Alice (ici $K = 3$). Les états sont organisés en N couches successives. Chaque état est noté $S_{i,j}$; j correspond à l'indice dans la suite des symboles émis et i au symbole correspondant. Les états de la couche j et les états de la couche $j + 1$ sont reliés par un arc de j vers $j + 1$. On ajoute également, par commodité, un état source σ et un état cible τ respectivement reliés à la première et à la dernière couche.

❏ **Q15** – En fonction de N et de K, donner le nombre de sommets et d'arcs du graphe illustré par la Figure 3. On ne comptera pas les sommets source σ et cible τ, ni les arcs partant du sommet source σ ni ceux arrivant à la cible τ.

On choisit désormais de pondérer chaque arc par la probabilité de transiter par cet arc. Autrement dit :

— les arcs issus de la source σ vers $S_{i,0}$ sont pondérés par $E_{obs_0,i}$ la probabilité d'observer le symbole obs_0 sachant que le symbole i a été émis par Alice ;
— les arcs arrivant à la cible τ sont pondérés par 1 (en fin de message, on transite forcément vers l'état final),
— les arcs internes entre $S_{i,j}$ et $S_{k,j+1}$ sont pondérés par la probabilité $E_{obs_{j+1},k}P_{i,k}$.

La probabilité d'un chemin $\sigma S_{i_0,0}S_{i_1,1}\dots S_{i_{N-1},N-1}\tau$ entre σ et τ est le **produit** des probabilités des arcs qui le composent.

L'objectif va être de trouver le chemin de **probabilité maximale** dans ce graphe entre le sommet source σ et le sommet cible τ.

❏ **Q16** – On suppose que Bob a observé la séquence `[2,0]`. En utilisant les matrices `E` et `P` données dans l'énoncé (avec $K = 3$), construire le graphe pondéré associé à ce message de longueur $N = 2$. Les arcs entre les sommets devront être pondérés par les probabilités correspondantes.

❏ **Q17** – On revient dans le cas général, N et K sont désormais quelconques. Indiquer combien il existe de chemins entre σ et τ (un ordre de grandeur utilisant la notation O ou Θ est accepté). Préciser si un algorithme d'exploration exhaustive est envisageable dans ce cas.

2.2 Stratégie gloutonne

Pour pouvoir implémenter correctement la recherche du chemin de probabilité maximale, il est utile de disposer d'une fonction auxiliaire qui sera utilisée dès que nécessaire.

❏ **Q18** – Pour une liste `liste`, on appelle argument du maximum et on note `argMax` tout indice i tel que `liste[i]` soit maximal. Proposer une fonction `maximumListe(liste:[float])->(float,int)` qui prend en entrée une liste de nombres et qui renvoie la valeur du maximum de la liste ainsi que le plus petit argument du maximum, *i.e.* le premier indice auquel cette valeur maximale apparaît.

On souhaite appliquer un algorithme glouton pour trouver le chemin de probabilité maximale entre le sommet source σ et le sommet cible τ. On rappelle qu'un algorithme glouton cherche, à chaque étape, à faire le choix localement optimal. Ici, si à une étape on se retrouve au sommet $S_{i,j}$, il s'agit de choisir l'arc de plus forte probabilité partant de ce sommet.

Dans un premier temps on écrit une fonction `initialiserGlouton(Obs:[[int]], E:[[float]], K:int)->int` qui permet d'initialiser l'algorithme glouton en trouvant le sommet le plus probable parmi $S_{i,0}$ pour i variant entre 0 et $K-1$. Pour cela il faut regarder la colonne `Obs[0]` de `E` et relever l'indice de la plus grande valeur.

```
1 def initialiserGlouton(Obs, E, K):
2     probasInitiales = [E[Obs[0][i]] for i in range(K)]
3     s, symbole = maximumListe(probasInitiales)
4     return symbole
```

❏ **Q19** – Proposer une fonction `glouton(Obs:[int],P:[[float]],E:[[float]],K:int,N:int)->[int]` qui renvoie la liste d'états obtenue par l'approche gloutonne. Même si cela n'est pas nécessaire, K, N seront des arguments de cette fonction.

❏ **Q20** – En fonction de K et de N, quelle est, en ordre de grandeur, la complexité temporelle asymptotique de l'approche gloutonne ?

❏ **Q21** – Indiquer le chemin renvoyé par l'algorithme glouton appliqué à la Figure 4. Conclure quant à l'optimalité de l'approche.

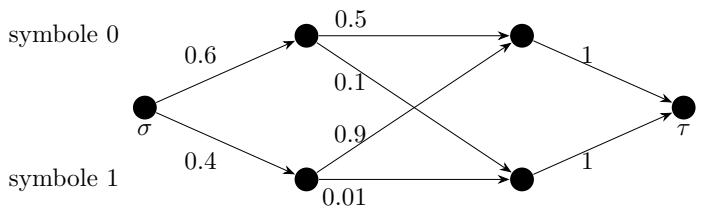


FIGURE 4 – Que donne l'algorithme glouton ici ?

2.3 Stratégie de programmation dynamique

❑ **Q22** – Expliquer en quoi rechercher un chemin de probabilité maximale pourrait se transformer en un problème de recherche de plus court chemin dans un graphe pondéré à poids positifs. Préciser alors quel algorithme pourrait être utilisé.

Les algorithmes évoqués à la question précédente ne sont cependant pas optimaux dans ce cas. L'algorithme optimal est dû à Andrew Viterbi et date de 1967. Il repose sur le paradigme de la programmation dynamique.

On appelle $T_{i,j}$ la valeur de probabilité maximale entre la source et l'état $S_{i,j}$. On peut alors établir l'équation de programmation dynamique suivante :

$$\begin{cases} T_{i,j} = \max\limits_{k\in[\![0,K-1]\!]} \left\{T_{k,j-1} \times P_{k,i} \times E_{obs_j,i}\right\} \text{ si } N-1 \geqslant j > 0 \\ T_{i,0} = E_{obs_0,i} \end{cases}$$

Comme on cherche également à obtenir la valeur des états correspondant au chemin optimal, on maintient également le tableau des prédécesseurs suivant :

$$\begin{cases} argT_{i,j} = \arg\max\limits_{k\in[\![0,K-1]\!]} \left\{T_{k,j-1} \times P_{k,i} \times E_{obs_j,i}\right\} \text{ si } N-1 \geqslant j > 0 \\ argT_{i,0} = -1 \end{cases}$$

La valeur -1 du deuxième tableau est purement conventionnelle et ne sert qu'à représenter l'état source σ qui ne correspond pas à une observation.

On suppose avoir codé une fonction `initialiserViterbi(E:[[float]], Obs0:int, K:int, N:int)->([[float]],[[int]])` qui prend en entrée la matrice `E` d'émission, la valeur de la première observation `Obs0`, le nombre d'éléments `K` de Σ, le nombre d'observations `N` et qui renvoie deux tableaux (liste de listes) `T` et `argT` vérifiant les caractéristiques suivantes :

— `T` et `argT` sont de dimensions K lignes par N colonnes,
— `T[i][0]` contient la valeur de $E_{obs_0,i}$,
— `argT[i][0]` contient la valeur -1,
— les autres valeurs de `T` et `argT` sont à 0.

En voici une implémentation possible :

```
def initialiserViterbi(E, Obs0, K, N):
    probasInitiales = [E[Obs0][i] for i in range(K)]
    T = [[0 for j in range(N)] for i in range(K)]
    argT = [[0 for j in range(N)] for i in range(K)]
    for i in range(K):
        T[i][0] = probasInitiales[i]
        argT[i][0] = -1
    return T, argT
```

❑ **Q23** – Proposer une fonction (méthode de bas en haut de programmation dynamique) `construireTableauViterbi(Obs:[int], P:[[float]], E:[[float]], K:int, N:int) ->([[float]],[[int]])` qui prend comme arguments la liste des observations `Obs`, la matrice des probabilités de transition `P` et la matrice des probabilités `E` et renvoie les deux listes de listes `T` et `argT` de taille $K \times N$.

❑ **Q24** – L'algorithme de Viterbi codé en Python et appliqué à un message en entrée

donne les tableaux `T` et `argT` suivants. Indiquer la séquence d'états la plus probable.

$$T = \begin{pmatrix} 0.1 & 0.084 & 0.018 & 0.00053 & 0.00021 & 8.9e-05 & 8.7e-05 & 1.8e-05 \\ 0.1 & 0.036 & 0.0054 & 0.00041 & 0.0011 & 0.00031 & 2.5e-05 & 3.5e-06 \\ 0.6 & 0.09 & 0.014 & 0.0053 & 0.00026 & 2.2e-05 & 1.9e-05 & 1.3e-05 \end{pmatrix}$$

$$argT = \begin{pmatrix} -1 & 2 & 0 & 0 & 2 & 1 & 1 & 0 \\ -1 & 2 & 2 & 2 & 2 & 1 & 1 & 0 \\ -1 & 2 & 0 & 0 & 2 & 1 & 1 & 0 \end{pmatrix}$$

❑ **Q25** – En fonction de K et de N, donner l'ordre de grandeur de la complexité temporelle de l'approche de programmation dynamique, ainsi que la complexité spatiale.

Fin de l'épreuve.

SESSION 2024 **PC5IN**

ÉPREUVE MUTUALISÉE AVEC E3A-POLYTECH

ÉPREUVE SPÉCIFIQUE - FILIÈRE PC

INFORMATIQUE

Durée : 3 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.

L'épreuve est à traiter en langage **Python** sauf pour les bases de données.
Les différents algorithmes doivent être rendus dans leur forme définitive sur le **Document Réponse** dans l'espace réservé à cet effet en respectant les éléments de syntaxe du langage (les brouillons ne sont pas acceptés).
La réponse ne doit pas se limiter à la rédaction de l'algorithme sans explication, les programmes doivent être expliqués et commentés de manière raisonnable.

Énoncé et Annexe : 16 pages
Document Réponse : 12 pages

Seul le Document Réponse doit être rendu dans son intégralité (le QR Code doit être collé sur la première page de ce Document Réponse).

Le jeu de l'awalé

Les fonctions seront définies avec leur signature dans le sujet :

$$\textbf{ma_fonction(arg1:type1, arg2:type2)} \rightarrow \textbf{type3}.$$

Cette notation permet de définir une fonction qui se nomme **ma_fonction** qui prend deux arguments en entrée **arg1** de type **type1** et **arg2** de type **type2**. Cette fonction renvoie une valeur de type **type3**.
Il ne faut pas recopier les signatures des fonctions dans le **Document Réponse (DR)**, il faut écrire directement :

```
def ma_fonction(arg1, arg2):
    # liste d'instructions
```

Partie I - Présentation et règles

Le sujet porte sur l'étude de l'awalé, un jeu de stratégie très ancien qui fait partie des jeux de semailles. En effet, à son origine, il était pratiqué à l'aide de graines qui étaient semées dans deux rangées de 6 trous creusés dans un plateau en bois ou à même le sol. Ce jeu est très répandu en Afrique et à partir du XVIIe siècle des indices de sa pratique sont également trouvés en Amérique du Sud et en Asie.

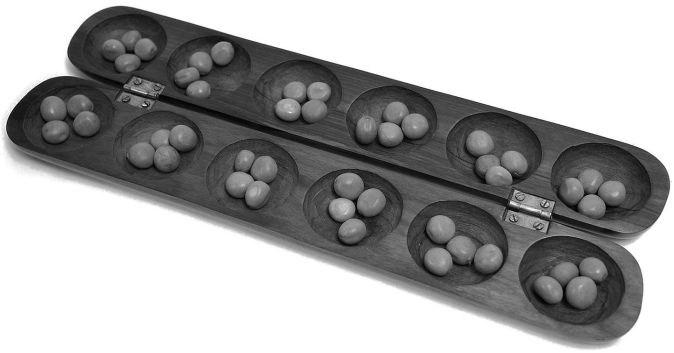


Figure 1 - Plateau d'awalé en bois

Les règles de ce jeu sont particulièrement simples et s'apprennent rapidement. Il en existe plusieurs variantes mais ce sujet n'en exposera qu'une seule. L'awalé se joue à deux. À tour de rôle, les joueurs prennent les graines situées dans un trou de leur rangée pour ensuite les déplacer dans les autres trous. Des graines peuvent ensuite être récoltées pour que les joueurs se constituent une réserve personnelle.

I.1 - But du jeu

L'objectif est de récolter plus de graines que son adversaire. Au départ, le plateau est composé de 2 rangées de 6 trous contenant chacun 4 graines comme le montre la **figure 2**. Le jeu s'arrête si l'un des joueurs obtient dans sa réserve personnelle à côté du plateau plus de la moitié des graines en jeu, c'est-à-dire au moins 25 graines ou jusqu'à une situation empêchant le gain de nouvelles graines.

I.2 - Déroulement de la partie

Les 2 participants jouent à tour de rôle. Les joueurs sont appelés par la suite Alice et Bob ; Alice joue en premier. Les joueurs sont face à face. La rangée de 6 trous située juste devant le joueur est appelée son camp.

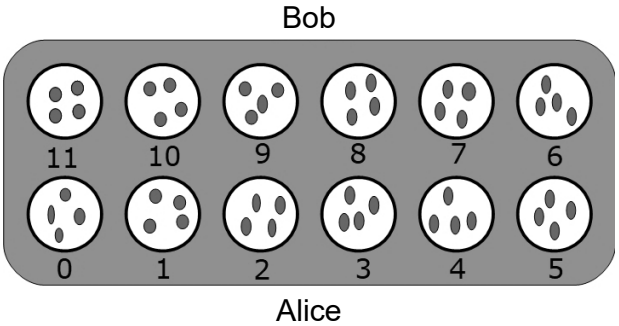


Figure 2 - Situation de départ

Chaque coup consiste à choisir une case non vide de son camp, à prendre toutes les graines de cette case en main et à les semer à raison d'une graine par case en suivant le sens direct, c'est-à-dire le sens inverse des aiguilles d'une montre (**figure 3**). On ne sème jamais de graine dans la case d'origine choisie. En effet, si la case de départ choisie contient plus de 11 graines, le joueur va semer sur un tour de plateau complet et revenir à la case d'origine des graines ; il faut alors sauter cette case pour continuer de semer les graines dans les autres cases. À la fin de son tour de jeu, la case de départ choisie par le joueur est nécessairement vide.

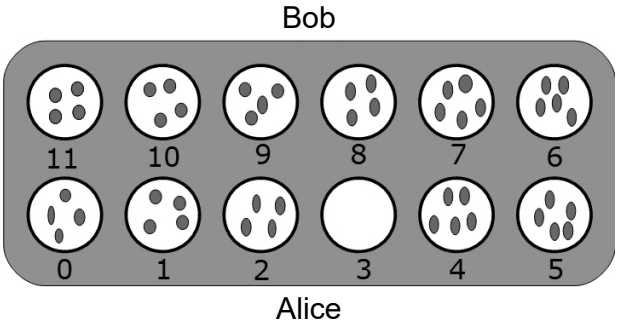


Figure 3 - Situation obtenue après qu'Alice ait semé la case d'indice 3

Une règle fondamentale de l'awalé est l'interdiction d'affamer son adversaire (**règle de la famine**). Lorsqu'un joueur n'a plus de graines dans son camp, son adversaire est obligé de jouer un coup qui lui en apporte au moins une.
Par ailleurs, il est interdit de jouer un coup qui ôte, après récolte, toutes les graines du camp adverse.

Une fois qu'un joueur a terminé de semer, il peut récolter les graines du plateau de jeu (en respectant la règle précédente). La récolte consiste à retirer les graines du plateau pour les stocker dans sa réserve personnelle sur la table à côté du plateau de jeu. Le joueur récolte les graines disponibles après son tour de semence, en commençant par la dernière case dans laquelle il a semé et sous les conditions suivantes :

- la case appartient au camp adverse (condition 1) ;
- cette case contient exactement 2 ou 3 graines (condition 2) ;
- s'il vient de ramasser les graines de la case, le joueur doit continuer la récolte dans le sens inverse de la semence, si la case respecte les deux premières conditions ;
- il est interdit d'affamer son adversaire, on ne peut donc pas prendre toutes les graines du camp adverse. Si la phase de récolte se termine ainsi, alors la récolte est annulée (condition 3 liée à la règle de la famine).

I.3 - Conditions de fin

La partie s'arrête sous certaines conditions :

- un joueur obtient au moins 25 graines dans sa réserve ;
- 3 graines ou moins restent sur le plateau ;
- un joueur est dans l'incapacité de jouer car aucun coup ne permet de respecter les différentes règles.

À la fin de la partie, chaque joueur ramasse les graines de son camp pour les transférer dans sa réserve personnelle. Le décompte des points peut alors se faire. Le joueur ayant obtenu au moins 25 graines est alors déclaré vainqueur. Une situation de nullité est possible si les deux joueurs obtiennent autant de graines chacun à la fin de la partie.

I.4 - Compréhension des règles

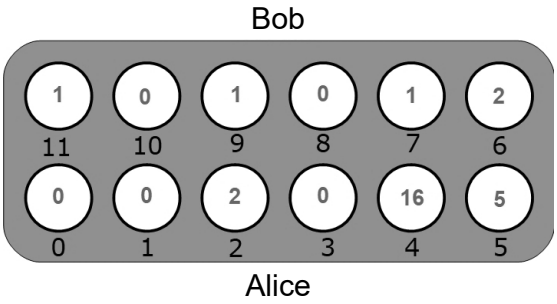


Figure 4 - Exemple de situation - Alice doit jouer

On considère la situation de jeu indiquée par la **figure 4**. C'est au tour d'Alice de jouer.

Q1. Indiquer, en justifiant, les indices des cases qu'Alice peut choisir de jouer.

Q2. Sur le **DR**, pour chacun des choix de cases possibles, renseigner la situation possible du plateau de jeu après le coup d'Alice (c'est-à-dire après avoir semé et récolté les graines). Indiquer également le gain éventuel pour chaque coup possible de la **figure 4**.

On considère les deux situations de jeu indiquées par la **figure 5**. C'est au tour d'Alice de jouer.

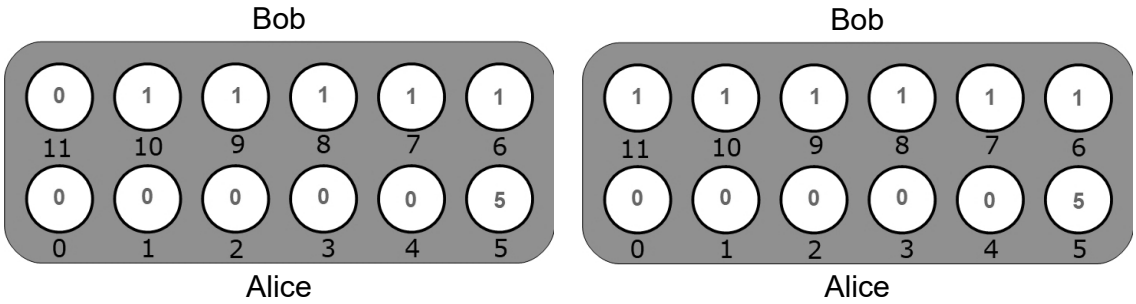


Figure 5 - Autres exemples de situation - Alice doit jouer

Q3. Pour chacune des deux situations de jeu de la **figure 5**, donner et justifier la situation du plateau après le tour de jeu d'Alice.

Partie II - Programmation de la structure de jeu

Cette partie aborde la modélisation et la structure du jeu dans le langage Python.

II.1 - Représentation du jeu

Le choix d'un dictionnaire a été fait pour stocker l'ensemble des données du jeu. De plus, l'appel de la fonction **intialisation(nom_joueur1:str,nom_joueur2:str)** → **dict** permet de créer la structure du jeu dans les conditions de départ.

```
def initialisation(nom_joueur1, nom_joueur2) :
    jeu = {}
    jeu['joueur1'] = nom_joueur1 # Nom du premier joueur
    jeu['joueur2'] = nom_joueur2 # Nom du second joueur
    jeu['score'] = [0,0]         # Réserve du joueur1, puis du joueur2
    jeu['n'] = 0                 # Nombre de tours déjà effectués
    jeu['plateau'] = [4]*12      # Plateau de jeu initial
    return jeu
```

Le compteur de tours détermine le tour de jeu des joueurs et commence donc à zéro. Le joueur1 commence la partie.
Le plateau de jeu est séparé en deux parties égales. Les six premières cases correspondent à celles du joueur dont c'est le tour ; les six dernières cases sont celles de l'adversaire. À la fin d'un tour de jeu, il faut échanger les deux ensembles de six cases. Ainsi les cases d'indices 0 à 5 correspondent toujours à celles du joueur dont c'est le tour (joueur actif) et les cases d'indices 6 à 11 à celles de son adversaire.
L'argument **jeu:dict** fait référence à un dictionnaire représentant le jeu de structure identique à celui renvoyé par **initialisation**.

Q4. Donner la parité de **jeu['n']** lorsque c'est au tour du joueur1 de jouer. Écrire une fonction **tour_joueur1(jeu:dict)** → **bool** qui renvoie **True** si c'est le tour de jeu du joueur1 et **False** sinon.

Q5. Écrire une fonction **tourner_plateau(jeu:dict)** → **None** qui modifie l'entrée **"plateau"** du dictionnaire **jeu** en échangeant les cases d'indices 0 à 5 avec celles de 6 à 11 (**figure 6**).

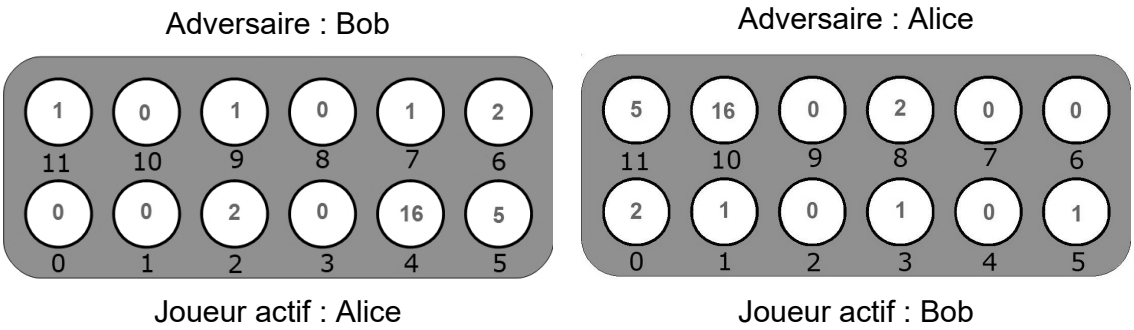


Figure 6 - Exemple d'inversion de plateau à la fin d'un tour entre Alice et Bob

Q6. Donner le maximum de graines que peut contenir une case. Déterminer alors le nombre de bits nécessaires pour coder les entiers représentants le nombre de graines par case.

Dans la suite du sujet, nous aurons besoin d'une fonction permettant de copier le jeu (dictionnaire) en entier dont certains éléments sont des listes.

Q7. Écrire une fonction **copie(jeu:dict)** → **dict** qui renvoie une copie profonde de la structure de dictionnaire retenue. Il est interdit d'utiliser la fonction **deepcopy** du module **copy**. L'opérateur **copy** des listes est autorisé.

Le déroulement d'un jeu d'awalé entre deux joueurs "humains" a la structure suivante :

```
1 def awale_jcj(nom_joueur1, nom_joueur2):
2     jeu = initialisation(nom_joueur1, nom_joueur2)
3     jeu_continue = True
4     while jeu_continue :
5         affiche(jeu['plateau'])
6         case_choisie = int(input("Choisir une case : "))
7         jeu_continue = tour_jeu(jeu, case_choisie)
8     return gagnant(jeu)
```

Remarque : la fonction **input** permet de récupérer une chaîne de caractères fournie par le joueur dans la console.
La fonction **affiche** permet d'afficher le plateau de jeu à l'écran.
Dans le code proposé, nous considérons que le joueur ne commet pas d'erreur de frappe et rentre toujours dans la console l'indice de la case choisie.

II.2 - Programmation d'un tour de jeu

Une fois la case de début de tour choisie par le joueur, un tour de jeu a la structure suivante :

1. tester si la case où l'on prend les graines est valide ou non (les conditions de validité sont explicitées ensuite) ;
2. si le choix est valide, semer les graines puis récolter des graines. On incrémente alors le nombre de tours, on ajoute au score du joueur le nombre de graines récoltées et on échange les deux parties du plateau ;
3. tester si la partie est finie ou non (les conditions sont décrites dans la première partie).

Les pré-conditions des fonctions de cette partie et de la partie suivante sont répertoriées ci-dessous :

- l'argument **jeu:dict** fait référence à un dictionnaire représentant le jeu comme présenté au début de cette partie ;
- l'argument **plateau:[int]** fait référence à une structure de données similaire à ce que contient **jeu['plateau']** ;
- l'argument **case:int** fait référence à un entier compris entre 0 inclus et 12 exclu.

On s'intéresse tout d'abord à l'étape 2 de manière à écrire deux fonctions :

- **deplacer_graines(plateau:[int], case:int) → int** ;
- **ramasser_graines(plateau:[int], case:int) → int**.

La fonction **deplacer_graines(plateau:[int], case:int) → int** prend comme argument le plateau de jeu ayant la structure choisie précédemment et la case non vide où le joueur actuel prend les graines. Cette fonction réalise la prise des graines de la **case** choisie et les sème une par une dans le sens direct (sens inverse des aiguilles d'une montre). Elle renvoie l'indice de la case où la dernière graine a été semée.

Q8. Proposer une fonction **deplacer_graines(plateau:[int],case:int) → int** modifiant en place l'argument **plateau**. On rappelle que l'on ne sème pas de graine dans la case choisie en début du tour.

Q9. Écrire une fonction auxiliaire **case_ramassable(plateau:[int], case:int) → bool** qui renverra **True** si le joueur dont c'est le tour a le droit de ramasser les graines de la case proposée et **False** sinon. On ne testera pas la question de la famine pour simplifier le problème. Pour rappel, le joueur peut ramasser le contenu de la case si :
  - la case appartient au camp de l'adversaire, soit toujours dans la deuxième moitié du plateau ;
  - la case contient 2 ou 3 graines.

La fonction **ramasser_graines(plateau:[int], case:int) → int** prend comme argument le plateau de jeu après déplacement des graines et l'indice de la case où la dernière graine a été semée. Cette fonction procède au ramassage des graines.

Si le joueur peut ramasser le contenu de la case, alors il ramasse le contenu et passe à la case précédente puis ramasse les graines de cette case si ces mêmes conditions sont respectées et ainsi de suite. Pour simplifier, la fonction **ramasser_graines** ne testera pas la condition de famine citée précédemment. Elle renverra le nombre de graines récoltées (c'est-à-dire le nombre de points gagnés).

Q10. Écrire la fonction **ramasser_graines(plateau:[int], case:int) → int**. On demande que la fonction **ramasser_graines** soit une fonction récursive. Cette fonction utilisera la fonction précédente **case_ramassable** et devra modifier le plateau en place et renvoyer le résultat de la récolte des graines.

Il faut vérifier à chaque tour de jeu si le choix d'une case est autorisé ou non. Si c'est un joueur humain qui joue, son choix peut se porter sur une case "interdite", c'est-à-dire dont il ne peut pas prendre les graines. Si c'est un joueur virtuel, celui-ci doit pouvoir faire la liste des cases "acceptables". Une case est "acceptable" si :

- condition 1 : elle est du côté du joueur dont c'est le tour ;
- condition 2 : elle est non vide ;
- condition 3 : à la fin du tour de jeu, les cases de l'adversaire ne sont pas complètement vides (condition de famine).

Q11. Écrire une fonction **test_famine(plateau:[int], case:int) → bool** qui vérifie que la case **case** choisie vérifie la condition 3 (renvoie **True** si la condition 3 est vérifiée et **False** sinon). Il n'est pas nécessaire qu'elle vérifie les deux conditions 1 et 2. Il est possible d'utiliser les fonctions demandées précédemment quelle que soit leur implémentation. *Remarque : les arguments d'entrée ne doivent pas être modifiés par la fonction.*

Q12. On propose ci-dessous une fonction qui vérifie que la case choisie vérifie bien les trois conditions précédentes, connaissant l'état actuel du jeu (le plateau). Compléter la condition de valeur **test** afin de déterminer si la case est acceptable ou non.

```
1 def test_case(plateau, case):
2     """ Vérifie si la case choisie par le joueur est acceptable
3     renvoie True si la case est acceptable, False sinon """
4     condition3 = test_famine(plateau, case)
5     # Case acceptable
6     test =  # à compléter
7     return test
```

Q13. Écrire la fonction **cases_possibles(jeu:dict) → [int]** qui renvoie la liste des indices de toutes les cases jouables par le joueur actif. Le dictionnaire **jeu** ainsi que sa clé **plateau** ne devront pas être modifiés.

Après un tour, il faut vérifier si le jeu est terminé ou si les joueurs peuvent continuer à jouer. Le jeu se termine si l'une des conditions suivantes est vérifiée :

- un des joueurs possède plus de la moitié des graines (soit au moins 25) ;
- le nombre de tours joués est supérieur ou égal à 100 (pour éviter un jeu infini lorsqu'il y a peu de graines) ;
- il reste 3 graines ou moins sur le plateau ;
- le joueur qui va jouer ne possède plus de case jouable. On suppose que le plateau a été échangé avant de faire les tests, donc le joueur qui doit jouer a ses graines dans les cases d'indices 0 à 5.

Q14. Écrire une fonction **tour_suivant(jeu:dict) → bool**. Cette fonction renvoie **True** si le jeu peut continuer et **False** sinon.

L'ébauche de la fonction **tour_jeu(jeu:dict, case:int) → bool** est donnée ci-dessous. Si le test de la case n'est pas valide, la fonction affiche un message et renvoie **True** pour permettre au joueur de proposer une nouvelle case. Sinon, elle renvoie **True** si le jeu peut continuer et **False** si le jeu ne peut pas continuer.

Q15. Donner l'instruction 1, l'instruction 2, la condition 1 et l'instruction 3 permettant à la fonction de répondre à la description précédente.

```
def tour_jeu(jeu, case):
    plateau = jeu['plateau']
    if test_case(plateau, case):  # La case jouée est acceptable
        # Instruction 1 : deplacer les graines
        # Instruction 2 : ramasser les graines
        """Pour augmenter le score, il faut savoir qui joue grâce à la
           parité du nombre de tours"""
        if # Condition 1:
            jeu['score'][0] = jeu['score'][0] + graines_gagnees
        else:
            jeu['score'][1] = jeu['score'][1] + graines_gagnees
        # Instruction 3  # On incrémente le nombre de tours
        tourner_plateau(jeu)  # Echanger les plateaux
        return tour_suivant(jeu)
    else:
        print("La case choisie n'est pas valable")
        return True
```

Q16. Écrire la fonction **gagnant(jeu:dict)** → **str** prenant comme argument le dictionnaire **jeu** contenant l'état actuel du jeu. La fonction doit procéder au ramassage des graines de chaque côté du plateau et les affecter au score, puis renvoyer le nom du joueur qui a gagné, c'est-à-dire qui a le plus de graines à la fin du jeu. S'il y a match nul, la fonction devra renvoyer la chaîne de caractère **"égalité"**.

Partie III - Programmation de l'Intelligence Artificielle (IA)

Cette partie aborde la programmation de l'Intelligence Artificielle (IA) si l'on désire jouer contre l'ordinateur. La structure du déroulement du jeu reste la même que précédemment, seul le choix de la case de jeu est différent. Il s'agit ici de programmer l'IA de manière à ce qu'elle choisisse la meilleure case pour elle. Nous allons pour cela utiliser un algorithme MinMax ou plus précisément sa version appelée Negamax.

III.1 - Arbre des configurations

On peut décrire l'ensemble des configurations possibles du plateau par un arbre orienté où chaque nœud correspond à un état du jeu (il peut donc être représenté par le dictionnaire **jeu**). Chaque arête orientée correspond à un tour de jeu. Un arbre est un graphe qui ne possède pas de cycle.

On rappelle le rôle des fonctions utiles créées dans les parties précédentes :

- **copie(jeu:dict)** → **dict** : réalise la copie profonde du dictionnaire **jeu** passé en argument ;
- **deplacer_graines(plateau:[int], case:int)** → **int** : réalise le déplacement des graines sur le **plateau** depuis la **case** choisie et renvoie la case (entier) où la dernière graine a été déposée ;
- **ramasser_graines(plateau:[int], case:int)** → **int** : réalise le ramassage des graines sur le **plateau** depuis la **case** où la dernière graine a été déposée et renvoie le nombre de graines récoltées (nombre de points gagnés) ;
- **tour_suivant(jeu:dict)** → **bool** : teste si la configuration de jeu permet de continuer, renvoie **True** si c'est le cas et **False** sinon ;
- **test_case(plateau:[int], case:int)** → **bool** : teste si le joueur dont c'est le tour a le droit de jouer la **case** choisie, renvoie **True** si c'est le cas et **False** sinon ;
- **cases_possibles(jeu:dict)** → **[int]** : renvoie la liste des cases jouables par le joueur actif.

Le grand nombre de possibilités de coups rend impossible la description complète de l'arbre. Pour choisir le coup à jouer, l'IA ne va parcourir l'arbre que sur une profondeur choisie à l'avance à partir de la configuration de jeu au moment où c'est à elle de jouer. Lors du parcours, il est important de déterminer trois caractéristiques :

- est-ce que le nœud est une feuille, c'est-à-dire une configuration où le jeu se termine ? La fonction **tour_suivant(jeu:dict)** écrite dans la partie précédente sert à cet effet ;
- combien d'enfants possèdent un nœud, c'est-à-dire, quels coups sont possibles à partir d'une configuration de jeu ?
- quel est le nombre de graines gagnées quand on passe d'un nœud à un autre ?

Q17. Grâce aux fonctions proposées précédemment et en s'inspirant de la fonction **tour_jeu**, écrire une fonction **gain(jeu:dict, case:int)** → **int,dict** qui renvoie le nombre de graines gagnées et un **nouveau dictionnaire** contenant l'état du jeu après le coup (toutes les grandeurs seront mises à jour). On considère que la case est valide, ce n'est pas la peine de le vérifier ici.
Remarque : les arguments d'entrée ne doivent pas être modifiés par la fonction.

III.2 - Algorithme MinMax

L'algorithme MinMax est un algorithme de théorie des jeux consistant à minimiser la perte maximale pour des jeux à deux joueurs. Le principe de l'algorithme est de visiter l'arbre sur une profondeur donnée et de remonter une "valeur de jeu" estimée pour chaque coup possible par une fonction d'utilité. Pour un nœud donné :

- si c'est au joueur dont c'est le tour de jeu, on remonte la valeur de jeu maximale (la plus favorable au joueur) ;
- si c'est à l'adversaire dont c'est le tour de jeu, on remonte la valeur de jeu minimale (la plus favorable à l'adversaire).

La valeur de jeu est calculée récursivement à partir des valeurs de jeu remontées depuis les nœuds enfants. Le calcul dépend du jeu considéré.

Le jeu d'awalé se prête particulièrement bien à une variante de l'algorithme appelée Negamax car la façon d'évaluer la valeur de jeu est symétrique par rapport à 0 entre le joueur et l'adversaire. Ainsi, au lieu de différencier le cas "Joueur" et "Adversaire", il suffit à chaque nœud p dont les enfants sont notés p_i de remarquer que :

$$\min_i(NegaMax(p_i)) = \max_i(-NegaMax(p_i)).$$

Comment est estimée la valeur de jeu dans le cas de l'awalé ?

Le cas de l'awalé est assez simple car la valeur de jeu est assez évidente : il s'agit de la différence de graines gagnées par chaque camp, positive si le joueur en ramasse plus et négative si c'est l'adversaire. Pour la calculer, on procéde de cette manière :

- si le nœud est une feuille, on connaît alors qui est le gagnant. On va donc tester qui est le gagnant et :
  - si c'est le joueur actif, on renvoie une valeur de jeu très grande (500) qui ne pourra être dépassée que par une autre configuration gagnante ;
  - si c'est l'adversaire, on renvoie une valeur de jeu très petite (-500) qui sera forcément dépassée par une autre configuration non perdante ;
- si l'on a atteint la profondeur maximale fixée, il n'y a pas de gain supplémentaire, donc cette valeur est nulle $NegaAwale(p_{term}) = 0$;
- sinon, pour chaque nœud enfant p_i, on calcule le gain g_i (nombre de graines ramassées) pour passer du nœud p au nœud p_i, puis on **lui retranche** la valeur de jeu calculée au nœud p_i . La valeur de jeu renvoyée par $NegaAwale(p)$ correspond alors au maximum des différences, soit $NegaAwale(p) = \max_i(g_i - NegaAwale(p_i))$.

Pour la question suivante, on choisit une profondeur de 2 et pour faciliter la lecture du plateau, **le plateau n'a pas été inversé entre deux tours de jeu**. On donne dans le **DR** un arbre indiquant la valeur du gain G obtenue en passant d'un nœud supérieur à un nœud inférieur, puis la valeur de jeu VJ du nœud inférieur, égale à $NegaAwale(p)$. On suppose que, pour le sommet (a), Bob vient tout juste de jouer.

À titre d'exemple, la branche de gauche, où Alice joue sa deuxième case, a été remplie :

- en appliquant les règles de jeu, on obtient facilement que le gain de (a) vers (b) vaut 4, le gain de (b) vers (c) vaut 4 et le gain de (b) vers (d) vaut 0 ;
- les valeurs de jeu de (c) et (d) sont nulles car on a atteint la profondeur de 2 ;
- enfin la valeur de jeu de (b) est égale au $\max(4-0; 0-0) = 4$.

Q18. Compléter le reste de l'arbre du **DR**.
Donner la case à jouer avec cette profondeur de recherche pour optimiser le gain d'Alice.

On propose la fonction **NegaAwale (jeu:dict, profondeur_max:int, profondeur: int) → int** incomplète suivante. Cette fonction applique l'algorithme *NegaMax* décrit précédemment. La fonction **max_vals** sera explicitée plus tard, c'est elle qui permet la détermination du maximum et du coup à jouer.

```
def NegaAwale (jeu, profondeur_max, profondeur):
    if                    : # Condition 1
        if (tour_joueur1(jeu) and gagnant(jeu) == jeu['joueur1']) or
           (not(tour_joueur1(jeu)) and gagnant(jeu) == jeu['joueur2']):
            return 500
        elif (tour_joueur1(jeu) and gagnant(jeu) == jeu['joueur2']) or
           (not(tour_joueur1(jeu)) and gagnant(jeu) == jeu['joueur1']):
            return -500
        else: # Egalité
            return 0
    elif                  : # Condition 2
        return 0
    else:
        choix_cases = cases_possibles(jeu)
        vals_jeu = []
        for case in choix_cases :
            # Instruction 1 : Détermination du gain et du nouveau jeu
            # Instruction 2 : Remontée de la valeur de jeu du noeud
               enfant
            vals_jeu.append([case, g-p])
return max_vals(vals_jeu, profondeur)
```

Q19. Compléter les lignes Condition 1, Condition 2, Instruction 1 et Instruction 2 de la fonction **NegaAwale** proposée pour réaliser l'algorithme MinMax.

La fonction **max_vals** ne doit pas se contenter de renvoyer la valeur de jeu maximale parmi les valeurs estimées. En effet, ce n'est pas la valeur de jeu maximale qu'on recherche mais l'indice de la case correspondant à celle-ci. On doit donc distinguer deux cas :

- soit le nœud père est le nœud de départ : dans ce cas, on renvoie l'indice (un entier) de la case correspondant à la valeur de jeu maximale ;
- soit le nœud père est un nœud intermédiaire : dans ce cas, on renvoie la valeur de jeu maximale (un entier).

Q20. Proposer une fonction **max_vals(vals_jeu:list, profondeur:int) → int** qui réalise ce qui est demandé précédemment. *On supposera pour simplifier qu'en cas de maximum multiple, c'est le premier maximum trouvé qui est conservé.*

On rappelle le programme de jeu dans le cas d'une partie entre deux joueurs humains :

```
1 def awale_jcj(nom_joueur1, nom_joueur2)
2     jeu = initialisation(nom_joueur1, nom_joueur2)
3     jeu_continue = True
4     while jeu_continue :
5         affiche( jeu['plateau'] )
6         case_choisie = int(input("Choisir une case : "))
7         jeu_continue = tour_jeu(jeu, case_choisie)
8     return gagnant(jeu)
```

Q21. Indiquer le numéro de la (ou des) ligne(s) qui doit (ou doivent) être modifiée(s) pour une partie entre deux joueurs IA et proposer une modification en prenant une profondeur maximale de 6.

III.3 - Bibliothèque d'ouverture

Afin d'améliorer l'efficacité de l'IA, il est possible pour les premiers coups d'effectuer une recherche dans une base de données relationnelle. En effet, l'exploration en profondeur de l'ensemble des coups possibles est très coûteuse et on préfère s'appuyer sur l'historique des parties pour déterminer les configurations du plateau qui seront les plus avantageuses.

La base de données contient des informations sur chaque joueur, ainsi que les parties effectuées entre les joueurs.

Joueur				
id_Joueur	nom	prenom	niveau	naissance
18571	Martin	Jean	2048	23/02/1958
18572	Dupond	Marie	2103	03/01/1972
18573	Develion	Théo	1857	05/10/2004

Partie					
id_Partie	id_joueur1	id_joueur2	resultat	jour	jeu
1	1547	1568	0.5	08/01/2001	'egai...'
2	1204	3	0	12/07/1998	'egaj...'
3	4	2	1	15/07/2018	'egbi...'

La table Joueur contient les attributs suivants :

- **id_Joueur** : identifiant d'un joueur (entier, clé primaire) ;
- nom : nom du joueur (chaîne de caractères) ;
- prenom : prénom du joueur (chaîne de caractères) ;
- niveau : niveau maximal atteint par le joueur au cours de sa carrière (entier) ;
- naissance : date de naissance du joueur (date).

La table Partie contient les attributs suivants :

- **id_Partie** : identifiant de la partie (entier, clé primaire) ;
- id_joueur1 : identifiant du joueur débutant la partie (entier) ;
- id_joueur2 : identifiant du second joueur (entier) ;
- resultat : 1 est une victoire du joueur1, 0.5 une égalité et 0 une victoire du joueur2 (flottant) ;
- jour : date du jour de la partie (date) ;
- jeu : liste des coups successifs de la partie, sans inversion du plateau, stockée sous forme d'une chaîne de caractères. 'egai...' signifie que le joueur1 a joué la 5e case (d'indice 4) représentée par la lettre 'e', puis le joueur2 a joué la 7e case (d'indice 6) représentée par la lettre 'g', puis le joueur1 a joué la 1re case et le joueur2 a répondu par la 9e case et ainsi de suite.

Remarque : l'opérateur LIKE est utilisé pour comparer des chaînes de caractères dans la clause WHERE des requêtes SQL. Ce mot-clé permet d'effectuer une recherche sur un modèle particulier. Il est par exemple possible de rechercher les enregistrements dont la valeur d'une colonne commence par telle ou telle lettre. Le caractère _ (underscore) représente n'importe quel caractère, mais un seul caractère uniquement alors que le caractère pourcentage % peut être remplacé par un nombre quelconque (et possiblement nul) de caractères. Par exemple parmi une recherche dans les communes de France, nom LIKE '_ff%f%' ne renvoie que Offendorf alors que remplacer le _ par un % renvoie Pfaffenhoffen et Staffelfelden en plus de Offendorf.

Q22. Écrire une requête SQL permettant d'extraire les identifiants des joueurs ayant un niveau strictement supérieur au score 1900.

Q23. Écrire une requête SQL permettant de déterminer le pourcentage de victoires du joueur1 pour les parties où la case d'indice 0 a été jouée en premier.

Q24. Écrire une requête SQL permettant d'afficher le nom et le prénom des 3 joueurs ayant le niveau le plus élevé.

Q25. Écrire une requête SQL permettant de déterminer les joueurs ayant plus de cent victoires lorsqu'ils commencent la partie. La requête doit renvoyer le nom, le prénom et le nombre de victoires de ces joueurs classés par ordre décroissant du nombre de victoires.

ANNEXE

Rappels des syntaxes en Python

Seules les commandes listées ici peuvent être utilisées. Il n'est pas autorisé de faire appel à certaines fonctions Python déjà implémentées (min, max, sort, ...).

Fonctionnalités	Commandes Python
définir une liste	L = [1,2,3]
définir un dictionnaire	dic = {'a':0,'b':1,'c':2}
accéder à un élément	L[0] renvoie 1 dic['a'] renvoie 0
extraire une sous-liste	L[1:2] renvoie [2]
vérifier si une clé est dans un dictionnaire	'a' **in** dic renvoie True
ajouter un élément à une liste	L.append(5)
supprimer le dernier élément d'une liste et le renvoyer	a = L.pop()
copier une liste	L2 = L.copy()
ajouter un élément à un dictionnaire	dic['d'] = 4
définir une chaîne de caractères	mot = 'Python'
taille d'une chaîne, d'une liste ou d'un dictionnaire	len(mot)
extraire des caractères	mot[2:6]
concaténer des chaînes ou des listes	'cc'+'inp' donne 'ccinp'
dupliquer des chaînes ou des listes	'c'*3 donne 'ccc'
convertir en flottant, en entier, en chaîne, en liste	float(s), int(s), str(L), list(s)
définir une chaîne de caractères contenant une tabulation	chaine = 'a\t b' >>> print(chaine) a b
parcours en valeur d'un dictionnaire	**for** v **in** dic.values(): print(v)
parcours des clés d'un dictionnaire	**for** c **in** dic : print(c)
parcours des clés et des valeurs d'un dictionnaire	**for** c, v **in** dic.items() : print(c, v)

FIN

IMPRIMERIE NATIONALE – 24 1099 – D'après documents fournis

CONCOURS COMMUN INP

Numéro d'inscription

Numéro de table

Né(e) le

Nom :

Prénom :

Emplacement QR Code

Filière : PC

Session : 2024

Épreuve de : INFORMATIQUE

Consignes
- *Remplir soigneusement l'en-tête de chaque feuille avant de commencer à composer*
- *Rédiger avec un stylo non effaçable bleu ou noir*
- *Ne rien écrire dans les marges (gauche et droite)*
- *Numéroter chaque page (cadre en bas à droite)*
- *Placer les feuilles A3 ouvertes, dans le même sens et dans l'ordre*

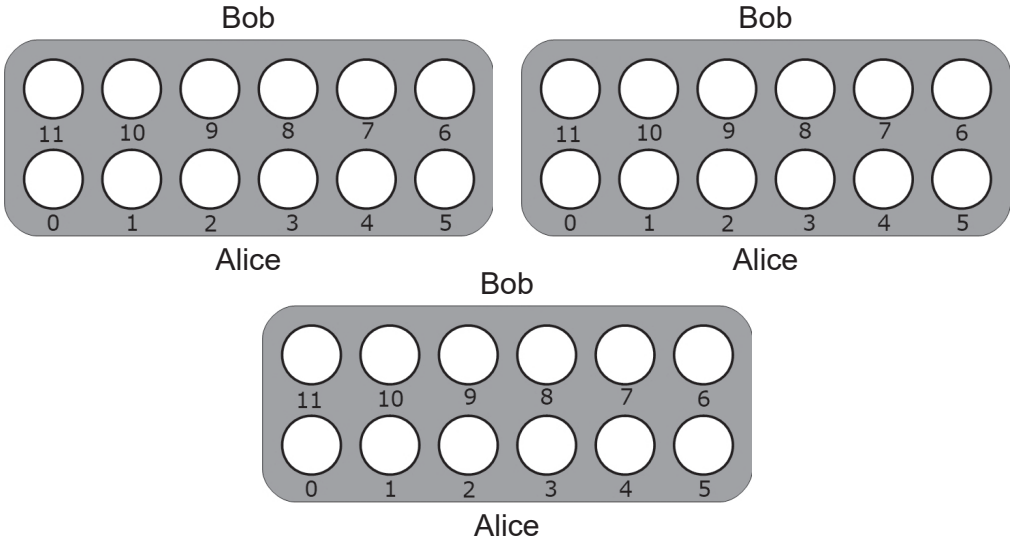
PC5IN

DOCUMENT RÉPONSE

Ce Document Réponse doit être rendu dans son intégralité.

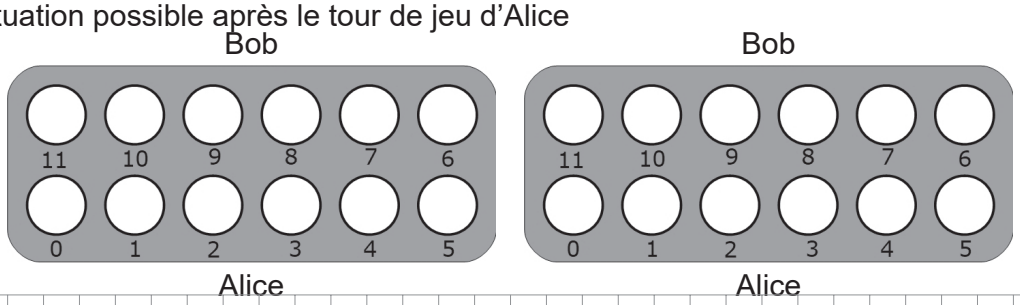
Q1 - Indice et justification

Q2 - Situation possible après le tour d'Alice



NE RIEN ÉCRIRE DANS CE CADRE

Q3 - Situation possible après le tour de jeu d'Alice



Q4 - Parité de **jeu['n']**. Fonction **tour_joueur1(jeu:dict)** → **bool**

Q5 - Fonction **tourner_plateau(jeu:dict)** → **None**

Q6 - Maximum de graines. Bits nécessaires

Q7 - Fonction **copie(jeu:dict)** → **dict**

Q8 - Fonction **deplacer_graines(plateau:[int], case:int) → int**

Q9 - Fonction **case_ramassable(plateau:[int], case:int) → bool**

CONCOURS COMMUN INP

Numéro d'inscription

Numéro de table

Né(e) le

Nom :

Prénom :

Emplacement QR Code

Filière : PC

Session : 2024

Épreuve de : INFORMATIQUE

Consignes

- *Remplir soigneusement l'en-tête de chaque feuille avant de commencer à composer*
- *Rédiger avec un stylo non effaçable bleu ou noir*
- *Ne rien écrire dans les marges (gauche et droite)*
- *Numéroter chaque page (cadre en bas à droite)*
- *Placer les feuilles A3 ouvertes, dans le même sens et dans l'ordre*

PC5IN

Q10 - Fonction récursive **ramasser_graines(plateau:[int], case:int) → int**

NE RIEN ÉCRIRE DANS CE CADRE

Q11 - Fonction **test_famine(plateau:[int], case:int) → bool**

Q12 - Expression de la condition test

Q13 - Fonction **cases_possibles(jeu:dict) → [int]**

Q14 - Fonction **tour_suivant(jeu:dict) → bool**

Q15 - Instruction 1, instruction 2, condition 1 et instruction 3

Instruction 1

Instruction 2

Condition 1

Instruction 3

Q16 - Fonction **gagnant(jeu:dict) → str**

CONCOURS COMMUN INP

Numéro d'inscription

Numéro de table

Né(e) le

Nom :

Prénom :

Emplacement QR Code

Filière : PC

Session : 2024

Épreuve de : INFORMATIQUE

Consignes

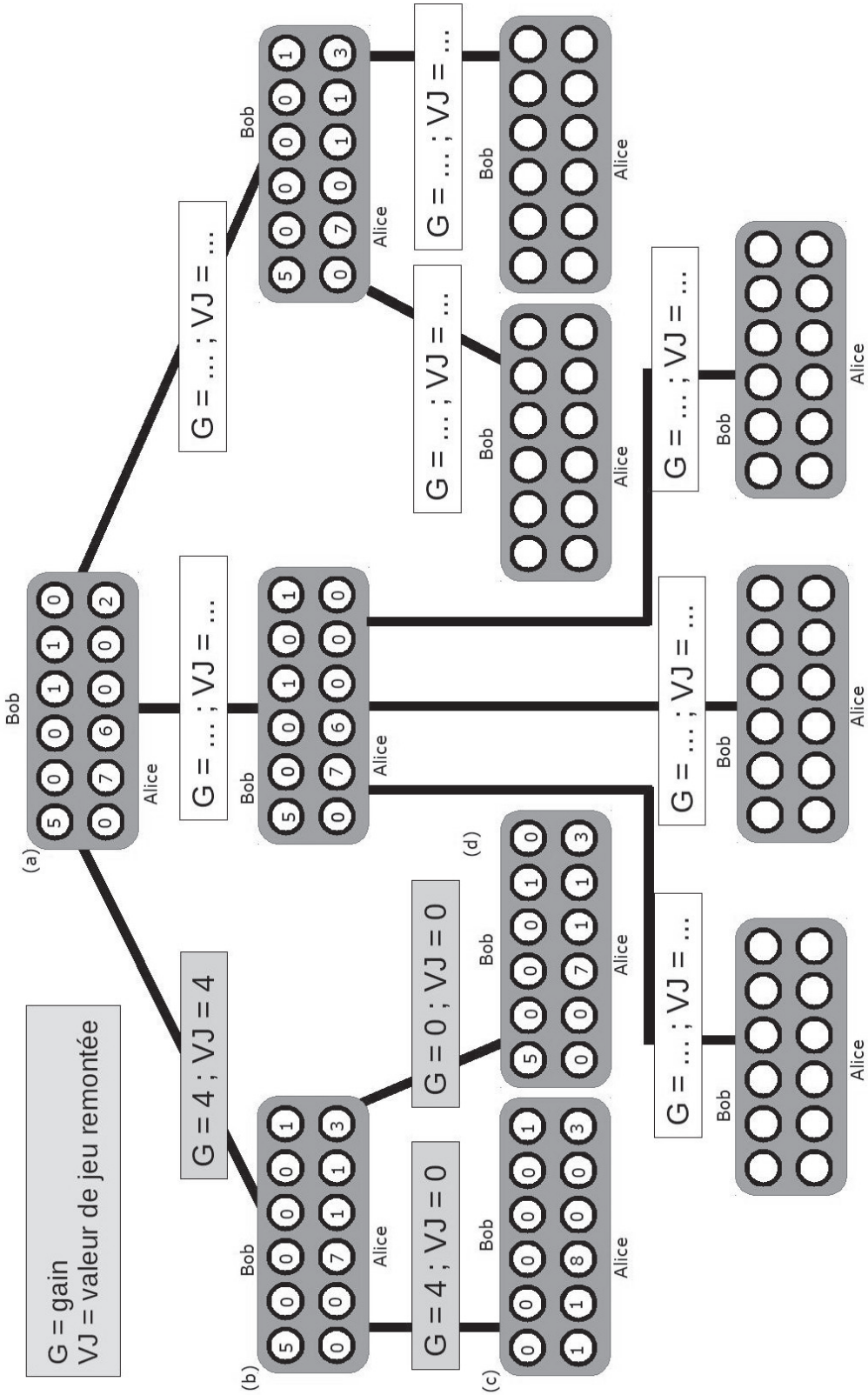
- *Remplir soigneusement l'en-tête de chaque feuille avant de commencer à composer*
- *Rédiger avec un stylo non effaçable bleu ou noir*
- *Ne rien écrire dans les marges (gauche et droite)*
- *Numéroter chaque page (cadre en bas à droite)*
- *Placer les feuilles A3 ouvertes, dans le même sens et dans l'ordre*

PC5IN

Q17 - Fonction **gain(jeu:dict, case:int) → int,dict**

NE RIEN ÉCRIRE DANS CE CADRE

Q18 - Arbre des jeux possibles à compléter



Q18 - Numéro de la case à jouer avec cette profondeur de recherche

Q19 - Compléter NegaAwale

Condition 1

Condition 2

Instruction 1

Instruction 2

Q20 - Fonction **max_vals(vals_jeu:list, profondeur:int) → int**

Q21 - Numéro(s) de(s) ligne(s) à modifier et modification(s) à apporter

Q22 - Identifiants des joueurs ayant un niveau strictement supérieur au score 1900

Q23 - Pourcentage de victoires en commençant la partie par la première case

Q24 - Nom et prénom des 3 joueurs ayant le niveau le plus élevé

Q25 - Les joueurs ayant plus de cent victoires en commençant la partie

FIN

SESSION 2024 **TSI5IN**

CONCOURS COMMUN INP

ÉPREUVE SPÉCIFIQUE - FILIÈRE TSI

INFORMATIQUE

Durée : 3 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 deux parties indépendantes.

Important : vous pouvez utiliser les fonctions des questions précédentes, même si vous ne les avez pas toutes implémentées.

Vous devez répondre directement sur le Document Réponse, soit à l'emplacement prévu pour la réponse lorsque celle-ci implique une rédaction, soit en complétant les différents programmes en langage Python.

- Texte du sujet : page 2 à page 7
- Annexes : page 7 à page 8
- Document Réponse : 12 pages

Seul le Document Réponse (DR) doit être rendu dans son intégralité
(le QR Code doit être collé sur la première page du Document Réponse).

A

Colorer un graphe

La coloration d'une carte de pays consiste à attribuer une couleur à chacun des pays de manière à ce que deux pays voisins soient de couleurs différentes.
Étudier ces techniques de coloration revient de façon plus abstraite à travailler sur des graphes.
Le champ d'applications de la coloration de graphes est très vaste et couvre des domaines aussi variés que le problème de l'attribution de fréquences dans les télécommunications, la conception de puces électroniques ou l'allocation de registres en compilation.

Soulignons que tous les graphes considérés dans ce sujet sont non-orientés.
Quelques rappels de syntaxe Python figurent dans l'**annexe 2**.

Partie I - Des algorithmes pour colorer un graphe

I.1 - Introduction sur un exemple

On cherche à colorer une carte de pays avec comme seule contrainte que deux pays ayant une frontière commune ne peuvent être de la même couleur.
Comme les pays, les couleurs sont numérotées à partir de zéro.

À titre d'exemple, on considèrera la carte suivante (**figure 1**), comportant 8 pays numérotés de 0 à 7,

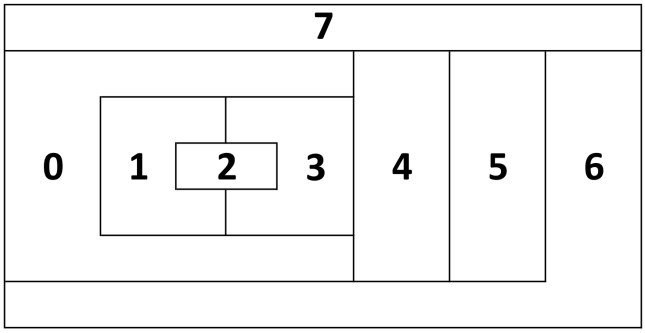


Figure 1 - Exemple d'une carte de pays

que l'on représentera par le graphe G_{ex} donné en **figure 2** :

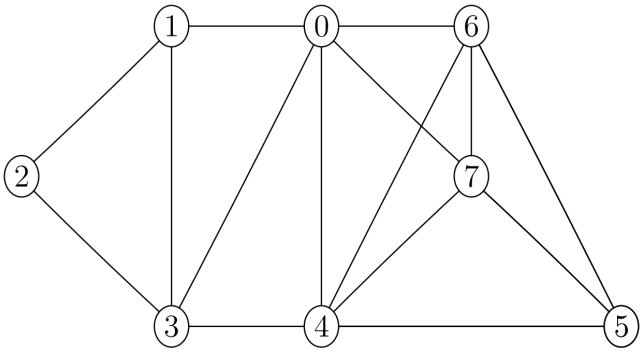


Figure 2 - Graphe G_{ex} associé à la carte de pays de la **figure 1**

Ainsi, sur le graphe de la **figure 2**, deux pays sont voisins si et seulement si les sommets correspondants sont reliés par une arête.

Q1. Un graphe peut être représenté par une matrice d'adjacence.
Le **DR** contient la matrice d'adjacence pré-remplie du graphe G_{ex}.
La compléter et expliquer le processus de construction.

Q2. Une autre manière de représenter en mémoire un graphe est d'utiliser une liste d'adjacence.

La liste d'adjacence d'un graphe (constitué de sommets numérotés de 0 à $n-1$) est une liste `LA`, telle que `LA[i]` est la liste des sommets voisins du sommet `i`. Par convention, les voisins énumérés dans `LA[i]` seront listés dans l'ordre croissant.

Donner la liste d'adjacence du graphe G_{ex} présenté en exemple.

Q3. Donner un avantage et un inconvénient d'une représentation par matrice d'adjacence.

Donner un avantage et un inconvénient d'une représentation par liste d'adjacence.

Q4. On rappelle que le degré d'un sommet `s` d'un graphe `G` est le nombre de voisins du sommet `s`, c'est-à-dire le nombre de sommets reliés à `s`.

Le **DR** fournit le tableau des degrés des différents sommets du graphe G_{ex}.

Le compléter.

I.2 - Tester si une coloration est valide

Q5. Écrire une fonction `voisins` avec trois arguments, deux nombres entiers distincts i et j et une liste d'adjacence `LA` représentant un graphe, qui renvoie `True` si les sommets numérotés i et j sont reliés par une arête et `False` sinon.

On considère une carte de pays, représentée par un graphe `G` pour laquelle on a une proposition de coloration donnée par une liste `C`. Ainsi, `C[i]` donne le numéro de la couleur attribuée au sommet `i`. On souhaite déterminer si la coloration proposée est valide (deux sommets du graphe reliés par une arête ne peuvent pas être de la même couleur).

Q6. Écrire la fonction `coloration_valide` avec pour arguments une liste d'adjacence `LA` et une liste de couleurs `C`, qui renvoie `True` si la coloration est valide, `False` sinon.

Q7. Pour un graphe comportant `n` sommets, quelle est la complexité temporelle dans le pire des cas de la fonction `coloration_valide` ?

I.3 - Un algorithme intuitif de coloration

L'attribution des couleurs à chaque sommet est caractérisée par une liste `C` où `C[i]` est la couleur attribuée au sommet `i` ; `C[i]` vaut -1 si la couleur n'est pas encore attribuée. La liste `C` ne contient que des -1 au départ et ses valeurs sont modifiées progressivement au fur et à mesure que les couleurs sont attribuées.

Q8. On suppose (dans cette question seulement) que `n` est une constante déjà définie. Écrire la ou les instruction(s) permettant de créer une liste initiale `C` composée de `n` éléments valant -1.

Q9. Compléter la fonction `colore_sommet` ayant trois arguments, la liste `C` des couleurs attribuées, le numéro `s` du sommet à colorer et la liste d'adjacence `LA` caractérisant le graphe.

Cette fonction ne renvoie rien mais modifie la liste `C` en donnant à `C[s]` la plus petite couleur possible, en fonction des couleurs des sommets voisins qui sont déjà colorés.

Par exemple, pour le graphe G_{ex}, prenons `C=[0,1,-1,-1,-1,-1,-1,-1]`. Les sommets 0 et 1 ont donc déjà été colorés avec les couleurs `C[0]=0` et `C[1]=1`.

L'appel `colore_sommet(C,2,LA)` modifie la liste `C` en `C=[0,1,0,-1,-1,-1,-1,-1]`. Cela veut dire que le sommet 2, adjacent au sommet 1 mais pas au sommet 0, a reçu la même couleur que le sommet 0.

Q10. À l'aide de **Q9**, écrire une fonction `colorer1` avec pour argument une liste `LA` caractérisant un graphe, qui crée et renvoie la liste `C` des numéros des couleurs attribuées en colorant les sommets un par un par ordre croissant de leurs numéros.

Par exemple, l'application de la fonction `colorer1` au graphe G_{ex} renverra la liste de couleurs `[0,1,0,2,1,0,2,3]`.

Q11. L'ordre de coloration imposé à la question précédente est arbitraire. On souhaite maintenant colorer le graphe en traitant les sommets selon un ordre arbitraire donné en argument.

Écrire une fonction `colorer2` analogue à `colorer1` et avec un argument supplémentaire, une liste `ordre` fixant l'ordre de coloration des pays.

Par exemple `colorer2([0,2,4,6,1,3,5,7],LA)` colorera le graphe G_{ex} en commençant d'abord par le sommet `0`, puis en continuant par les sommets `2,4,6`...

Q12. Donner la liste des couleurs renvoyée par `colorer2` pour colorer le graphe G_{ex} donné en exemple en prenant `ordre=[7,6,5,4,3,2,1,0]`.

Combien de couleurs ont-elles été utilisées ?

La méthode que nous venons de décrire est rapide et fonctionne plutôt bien. Cependant, si on cherche à utiliser le nombre minimum de couleurs, l'efficacité de l'algorithme proposé ci-dessus dépend en grande partie de l'ordre dans lequel on choisit de colorer les sommets du graphe.

L'objectif des sous-parties suivantes est d'affiner la stratégie pour mieux choisir cet ordre de coloration.

I.4 - Variante de Welsh-Powell

Une alternative est donnée par la variante de Welsh-Powell. L'idée est de parcourir l'ensemble des sommets du graphe par ordre décroissant de leurs degrés.
Comme le degré d'un sommet est un entier positif, il est possible d'écrire un algorithme de tri efficace (dit par répartition).

Q13. Écrire une fonction `degre` avec pour argument la liste d'adjacence `LA` d'un graphe quelconque, qui renvoie la liste des degrés des sommets du graphe.

Par exemple, pour un graphe de liste d'adjacence `LA=[[1,2],[0,2,3],[0,1,3],[1,2,4],[3]]`, la fonction `degre` renverra la liste `[2,3,3,3,1]`.

Q14. Écrire une fonction `init` avec pour argument un entier `n`, qui renvoie une liste de listes `R` de taille `n`, telle que `R[i]` soit une liste vide.

Par exemple, `init(3)` renverra `[[],[],[]]`.

Q15. Écrire une fonction `ranger` avec pour argument une liste d'adjacence `LA`, qui renvoie une liste `R` de même taille que `LA`, telle que `R[i]` soit la liste des sommets de degré `i`.

Ainsi, pour l'exemple de la question **Q13**, l'appel `ranger(LA)` renverra la liste `[[],[4],[0],[1,2,3],[]]`.

Q16. Écrire une fonction `renverse` avec pour argument une liste `L`, qui crée et renvoie une nouvelle liste obtenue en lisant `L` dans l'ordre inverse.

Par exemple, `renverse([1,2,3,4])` renverra `[4,3,2,1]`.

Q17. Écrire une fonction `trier_sommets` avec pour argument une liste d'adjacence `LA`, qui renvoie la liste des sommets triés dans l'ordre décroissant de leur degré.

Par exemple, pour un graphe de liste d'adjacence `LA=[[1,2],[0,2,3],[0,1,3],[1,2,4],[3]]`, la fonction `trier_sommets` renverra la liste de sommets `[1,2,3,0,4]`.

Q18. Pour un graphe à `n` sommets, quelle est la complexité temporelle de la fonction `trier_sommets` dans le pire des cas ?

Q19. Écrire la fonction `colorer3` avec pour argument une liste d'adjacence `LA`, qui crée et renvoie une liste de couleurs `C`, telle que `C[i]` soit la couleur à attribuer au sommet numéro `i`, les sommets étant colorés dans l'ordre décroissant de leur degré.

Quelle est la complexité de `colorer3` dans le pire des cas pour un graphe à n sommets ?

Q20. Pour le graphe G_{ex} de la **figure 2**, donner la liste `C` des couleurs renvoyée par la fonction `colorer3`.

L'amélioration proposée donne de bons résultats dans un grand nombre de cas. Cependant, il reste quelques cas où la coloration obtenue utilise trop de couleurs.

La méthode suivante, proposée en 1979 par Danier Brélaz de l'École Polytechnique Fédérale de Lausanne, raffine la détermination de l'ordre de coloration. La priorité de coloration est ainsi recalculée après chaque traitement d'un sommet et non plus une fois pour toute au départ. Au final, cette approche fournit rapidement une coloration optimale dans un très grand nombre de cas.

I.5 - Algorithme DSATUR

Lorsque l'on colore un graphe, le degré de saturation d'un sommet est le nombre de couleurs différentes déjà attribuées à ses différents sommets voisins. Évidemment, ce degré de saturation est susceptible d'évoluer à chaque fois qu'un nouveau sommet est coloré.

Q21. Écrire une fonction `degre_satur` avec 3 arguments, une liste d'adjacence `LA`, un sommet `s` du graphe, une liste `C` de couleurs. Cette fonction renvoie le degré de saturation du sommet `s`. On rappelle que le sommet `i` est coloré si et seulement si `C[i]` est différent de -1.

Q22. Écrire une fonction `liste_satur` avec deux arguments, une liste d'adjacence `LA`, la liste `C` des couleurs des sommets, qui renvoie la liste des sommets non colorés du graphe ayant un degré de saturation maximum parmi les sommets non colorés.

On notera qu'il s'agit d'une liste car plusieurs sommets peuvent avoir le même degré de saturation. On supposera de plus qu'il reste au moins un sommet non coloré.

Q23. Écrire une fonction `pas_fini` avec pour argument une liste `C`, qui renvoie `True` si cette liste contient la valeur `-1`, `False` sinon.

Q24. Compléter la fonction `colorer4` ayant pour argument une liste d'adjacence `LA`, qui renvoie une liste `C` constituant une coloration du graphe. Cette fonction procède de la façon suivante.

Tant qu'il reste un sommet non coloré :

- déterminer parmi les sommets non colorés ceux de degré de saturation maximale ;
- si plusieurs sommets non colorés ont un degré de saturation maximale, en choisir un parmi ceux-ci qui soit de degré maximal ;
- colorer le sommet choisi en lui attribuant la couleur disponible ayant la plus petite valeur.

Il n'est pas facile d'être certain d'avoir utilisé le nombre minimum de couleurs. La sous-partie suivante propose une piste.

I.6 - Un minorant du nombre de couleurs nécessaires

Considérons un graphe `G` et un sous-ensemble `K` de sommets de ce graphe. On dit que `K` forme une clique si et seulement si pour chaque paire de sommets de `K`, il existe une arête les reliant.
Par exemple, sur le graphe G_{ex} de la **figure 2**, `(4, 5, 6, 7)` constitue une clique de 4 sommets et `(4, 5, 6)` une clique de 3 sommets. Par contre, `(0, 4, 5, 6, 7)` ne forme pas une clique puisque le sommet 0 n'est pas relié au sommet 5.
Enfin, on note n_c le cardinal de la plus grande clique de `G`.

Q25. Justifier que le nombre minimum de couleurs pour colorer un graphe est supérieur ou égal au nombre n_c.
Montrer également que : $n_c \leqslant 1 + \max\{\deg s,\ s \in G\}$ où $\deg s$ désigne le degré du sommet s dans le graphe G.

Dans ce qui suit, on suppose toujours que les sommets d'un graphe à n sommets sont numérotés de 0 à $n-1$.

Q26. Écrire une fonction `est_clique` avec deux arguments, la liste d'adjacence `LA` d'un graphe et un tuple `K` de sommets de ce graphe, qui renvoie `True` si `K` est une clique, `False` sinon.

Q27. La fonction `combinations` du module `itertools`, présentée à l'**annexe 2**, fournit toutes les combinaisons de taille fixée d'une liste.
Sur le **DR**, compléter la fonction `minoration_nb_couleurs` ayant pour argument la liste d'adjacence `LA` d'un graphe, qui renvoie le cardinal de la plus grande clique du graphe considéré.

Ainsi, si le nombre de couleurs utilisées pour colorer le graphe est égal à n_c, on est sûr d'avoir utilisé le nombre minimum de couleurs.

Partie II - Interrogation d'une base de données géographiques

On dispose d'une base de données composée de quatre tables (**annexe 1**) :
- La table `Continents` constituée des champs suivants :
  - *nom :* nom du continent (chaîne de caractères) ;
  - *surface :* surface du continent en kilomètres carrés (entier).
- La table `Pays` constituée des champs suivants :
  - *nom :* nom du pays (chaîne de caractères) ;
  - *code_pays :* identifiant unique du pays (chaîne de caractères) ;
  - *capitale :* capitale administrative du pays (chaîne de caractères) ;
  - *population :* nombre d'habitants du pays (entier).
- La table `Inclusion` constituée des champs suivants :
  - *code_pays :* identifiant unique du pays (chaîne de caractères) ;
  - *continent :* nom du continent auquel appartient le pays (chaîne de caractères).
- La table `Frontieres` constituée des champs suivants :
  - *code_pays1 :* identifiant unique du premier pays (chaîne de caractères) ;
  - *code_pays2 :* identifiant unique du second pays (chaîne de caractères) ;
  - *longueur :* longueur en kilomètres de la frontière entre *pays1* et *pays2* (nombre flottant strictement positif).

On a toujours *code_pays1<code_pays2* pour l'ordre lexicographique, ce qui assure que chaque frontière n'apparaît qu'une fois dans la table `Frontieres`.

Q28. Écrire une requête SQL permettant de récupérer le code du pays nommé 'France'.

Q29. Écrire une requête SQL permettant d'obtenir les noms des pays appartenant au continent 'Europe'.

Dans les deux dernières questions, on admet que le code du pays nommé 'France' est 'F' et on peut utiliser librement cette information.

Q30. Écrire une requête SQL permettant de récupérer la longueur totale de la frontière du pays nommé 'France'.

Q31. Écrire une requête SQL permettant de récupérer les noms des pays frontaliers du pays nommé 'France'.

ANNEXE 1 - Base de données géographiques

Table `Continents`

nom	surface
'Asie'	44579000
'Europe'	9938000
...	...

Table `Pays`

nom	code_pays	capitale	population
'Albanie'	'AL'	'Tirana'	3088385
'Algerie'	'DZ'	'Alger'	44487616
...	...	...	...

Table `Inclusion`

code_pays	continent
'AL'	'Europe'
'DZ'	'Afrique'
...	...

Table `Frontieres`

code_pays1	code_pays2	longueur
'AL'	'GR'	282.8
'AL'	'MK'	151.5
...	...	...

ANNEXE 2 - Rappels de syntaxe Python

Test d'appartenance	`>>> 2 in [1,2,3,4]`
	`True`
	`>>> 2 in [1,3,4]`
	`False`

Définir une liste	`>>> L=[1,2,3]`
	`>>> L[0]`
	`1`

Ajouter un élément à la fin d'une liste	`>>> L=[1,2,3]`
	`>>> L`
	`[1,2,3]`
	`>>> L.append(5)`
	`>>> L`
	`[1,2,3,5]`

Ajouter tous les éléments d'une liste `L1` à la fin d'une liste `L`	`>>> L=[6,8,7]`
	`>>> L1=[-1,-2]`
	`>>> L.extend(L1)`
	`>>> L`
	`[6,8,7,-1,-2]`

Obtenir les combinaisons d'une taille donnée d'une liste	`>>> from itertools import combinations`
	`>>> for C in combinations([0,1,2,3,4],3):`
	`  print(C)`
	`(0, 1, 2)`
	`(0, 1, 3)`
	`(0, 1, 4)`
	`(0, 2, 3)`
	`(0, 2, 4)`
	`(0, 3, 4)`
	`(1, 2, 3)`
	`(1, 2, 4)`
	`(1, 3, 4)`
	`(2, 3, 4)`

FIN

IMPRIMERIE NATIONALE – 24 1114 – D'après documents fournis

CONCOURS COMMUN INP

Numéro d'inscription

Numéro de table

Né(e) le

Nom : _____

Prénom : _____

Emplacement QR Code

Filière : TSI

Session : 2024

Épreuve de : INFORMATIQUE

Consignes

- *Remplir soigneusement l'en-tête de chaque feuille avant de commencer à composer*
- *Rédiger avec un stylo non effaçable bleu ou noir*
- *Ne rien écrire dans les marges (gauche et droite)*
- *Numéroter chaque page (cadre en bas à droite)*
- *Placer les feuilles A3 ouvertes, dans le même sens et dans l'ordre*

TSI5IN

DOCUMENT RÉPONSE

Seul ce Document Réponse doit être rendu dans son intégralité.

Q1.

$$M = \begin{pmatrix} & & & & 1 & 0 & 1 & 1 \\ & & & & 0 & 0 & 0 & 0 \\ & & & & 0 & 0 & 0 & 0 \\ & & & & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Mode de construction de la matrice d'adjacence

.....

.....

.....

.....

.....

.....

.....

.....

NE RIEN ÉCRIRE DANS CE CADRE

Q2. Liste d'adjacence du graphe G_{ex}

Q3. Un avantage et un inconvénient d'une matrice d'adjacence, d'une liste d'adjacence

Q4.

Sommet	0	1	2	3	4	5	6	7
Degré								

Q5. Fonction `voisins`

Q6. Fonction `coloration_valide`

Q7. Complexité temporelle dans le pire des cas de `coloration_valide`

Q8. Liste composée de `n` éléments valant `-1`

Q9. Compléter la fonction selon les instructions de l'énoncé

```
def colore_sommet(C,s,LA) :
    # on détermine la liste des couleurs des voisins de s déjà colorés
    coul_vois = []









    # coul_vois est maintenant déterminée et on recherche la
    # plus petite couleur, notée num_coul, absente de coul_vois:

    # la valeur num_coul trouvée devient la couleur du sommet s:
```

Q10. Fonction `colorer1`

CONCOURS COMMUN INP

Numéro d'inscription

Numéro de table

Né(e) le

Nom :

Prénom :

Emplacement QR Code

Filière : TSI

Session : 2024

Épreuve de : INFORMATIQUE

Consignes
- *Remplir soigneusement l'en-tête de chaque feuille avant de commencer à composer*
- *Rédiger avec un stylo non effaçable bleu ou noir*
- *Ne rien écrire dans les marges (gauche et droite)*
- *Numéroter chaque page (cadre en bas à droite)*
- *Placer les feuilles A3 ouvertes, dans le même sens et dans l'ordre*

TSI5IN

Q11. Fonction `colorer2`

Q12. Liste des couleurs renvoyée par `colorer2` pour G_{ex}. Nombre de couleurs utilisées.

NE RIEN ÉCRIRE DANS CE CADRE

Q13. Fonction `degre`

Q14. Fonction `init`

Q15. Fonction `ranger`

Q16. Fonction `renverse`

Q17. Fonction `trier_sommets`

Q18. Complexité temporelle dans le pire des cas de la fonction `trier_sommets`

Q19. Fonction `colorer3`

Complexité de la fonction `colorer3`

Q20. Liste `C` des couleurs renvoyée par `colorer3` pour le graphe G_{ex}

CONCOURS COMMUN INP

Numéro d'inscription

Numéro de table

Né(e) le

Nom :

Prénom :

Emplacement QR Code

Filière : TSI

Session : 2024

Épreuve de : INFORMATIQUE

Consignes
- *Remplir soigneusement l'en-tête de chaque feuille avant de commencer à composer*
- *Rédiger avec un stylo non effaçable bleu ou noir*
- *Ne rien écrire dans les marges (gauche et droite)*
- *Numéroter chaque page (cadre en bas à droite)*
- *Placer les feuilles A3 ouvertes, dans le même sens et dans l'ordre*

TSI5IN

Q21. Fonction `degre_satur`

Q22. Fonction `liste_satur`

NE RIEN ÉCRIRE DANS CE CADRE

Q23. Fonction `pas_fini`

Q24. Compléter la fonction selon les instructions de l'énoncé

```
def colorer4(LA):
    n =            # nombre de sommets du graphe
    D =            # liste des degrés des sommets du graphe
    C =            # initialisation de la liste des couleurs
    while                  :
        # liste des sommets non colorés de degré de saturation maximal

        Ls =

        # en cas d'égalité,  recherche du sommet de degré maximal









        # coloration du sommet prioritaire



    return C
```

Q25. Relations entre n_c et le nombre minimum de couleurs, entre n_c et le degré maximum.

Q26. Fonction `est_clique`

Q27. Compléter la fonction selon les instructions de l'énoncé

```
from itertools import combinations
def minoration_nb_couleurs(LA):
    n =                          # nombre de sommets du graphe
    S = [ k for k in range(n) ] # liste des sommets du graphe
    i =
    test = True
    while test:
        for K in combinations(S,i):









        i=i-1
    return
```

Q28. Requête SQL

Q29. Requête SQL

Q30. Requête SQL

Q31. Requête SQL

FIN

014

Epreuve d'Informatique et Modélisation de Systèmes Physiques

Durée 4 h

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, d'une part il le signale au chef de salle, d'autre part il le signale sur sa copie et poursuit sa composition en indiquant les raisons des initiatives qu'il est amené à prendre.

L'usage de calculatrices est interdit.

AVERTISSEMENT

La **présentation**, la lisibilité, l'orthographe, la qualité de la **rédaction, la clarté et la précision** des raisonnements entreront pour une **part importante** dans **l'appréciation des copies**. En particulier, les résultats non justifiés ne seront pas pris en compte. Les candidats sont invités à encadrer les résultats de leurs calculs.

CONSIGNES :

- Composer lisiblement sur les copies avec un stylo à bille à encre foncée : bleue ou noire.
- L'usage de stylo à friction, stylo plume, stylo feutre, liquide de correction et dérouleur de ruban correcteur est interdit.
- Remplir sur chaque copie en MAJUSCULES toutes vos informations d'identification : nom, prénom, numéro inscription, date de naissance, le libellé du concours, le libellé de l'épreuve et la session.
- Une feuille, dont l'entête n'a pas été intégralement renseigné, ne sera pas prise en compte.
- Il est interdit aux candidats de signer leur composition ou d'y mettre un signe quelconque pouvant indiquer sa provenance.

Tournez la page S.V.P

PARTIE MODELISATION

Des applications industrielles, comme la fabrication de batteries, réclament un environnement précisément contrôlé. En particulier, la température doit être finement régulée pour permettre un déroulement fiable des procédés.

Dans cette partie, deux dispositifs qui participent à cette régulation sont modélisés, un climatiseur puis une ailette de refroidissement.

A. Etude d'un climatiseur

1. Transferts thermiques

On envisage la construction d'un climatiseur monotherme, c'est-à-dire une machine thermique cyclique capable de recevoir ou de fournir un travail, et d'absorber de l'énergie thermique d'une source de chaleur, thermostat à la température T.

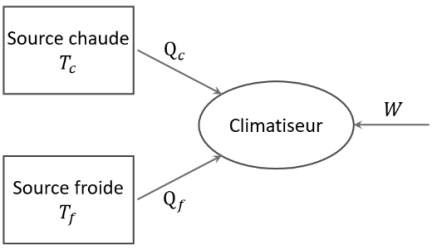
W et Q correspondent aux grandeurs algébriquement reçues sur un cycle

Q1. Exprimer la variation d'énergie interne et d'entropie pour cette machine, en fonction de W, Q, T et de l'entropie $S_{créée}$ créée sur un cycle. Simplifier les expressions en considérant un fonctionnement cyclique.

Q2. Déterminer les signes du travail W et du transfert thermique Q.

Q3. Conclure sur la possibilité de concevoir un tel climatiseur monotherme.

2. Efficacité d'un cycle réversible



On s'intéresse désormais à un climatiseur ditherme, c'est-à-dire à une machine thermique cyclique capable de recevoir ou de fournir un travail, et d'échanger de l'énergie thermique avec deux thermostats aux températures T_f (pour la source « froide ») et T_c (pour la source « chaude »).

Q4. Ecrire deux relations entre le travail W, les transferts thermiques Q_f et Q_c, et les températures T_f et T_c.

Q5. Préciser le signe du transfert thermique Q_f (considérer que la machine est un climatiseur).

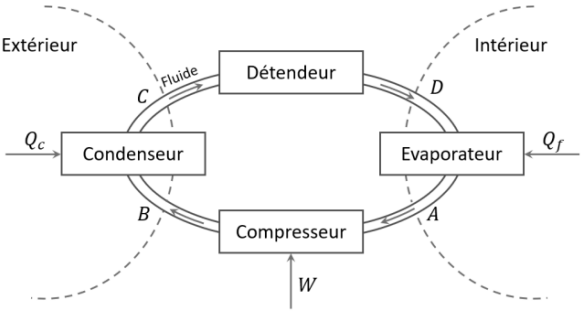
Q6. En déduire le signe du travail W.

Q7. Définir l'efficacité du climatiseur ditherme, et déterminer son expression en fonction des températures T_f et T_c dans l'hypothèse d'un fonctionnement réversible.

Q8. La pièce à climatiser doit conserver une température constante. Commenter l'évolution de l'efficacité dans l'hypothèse d'un fonctionnement réversible, en fonction de la température extérieure.

3. Machine réelle

Le cycle $ABCD$ d'un climatiseur réel peut être représenté par un schéma :



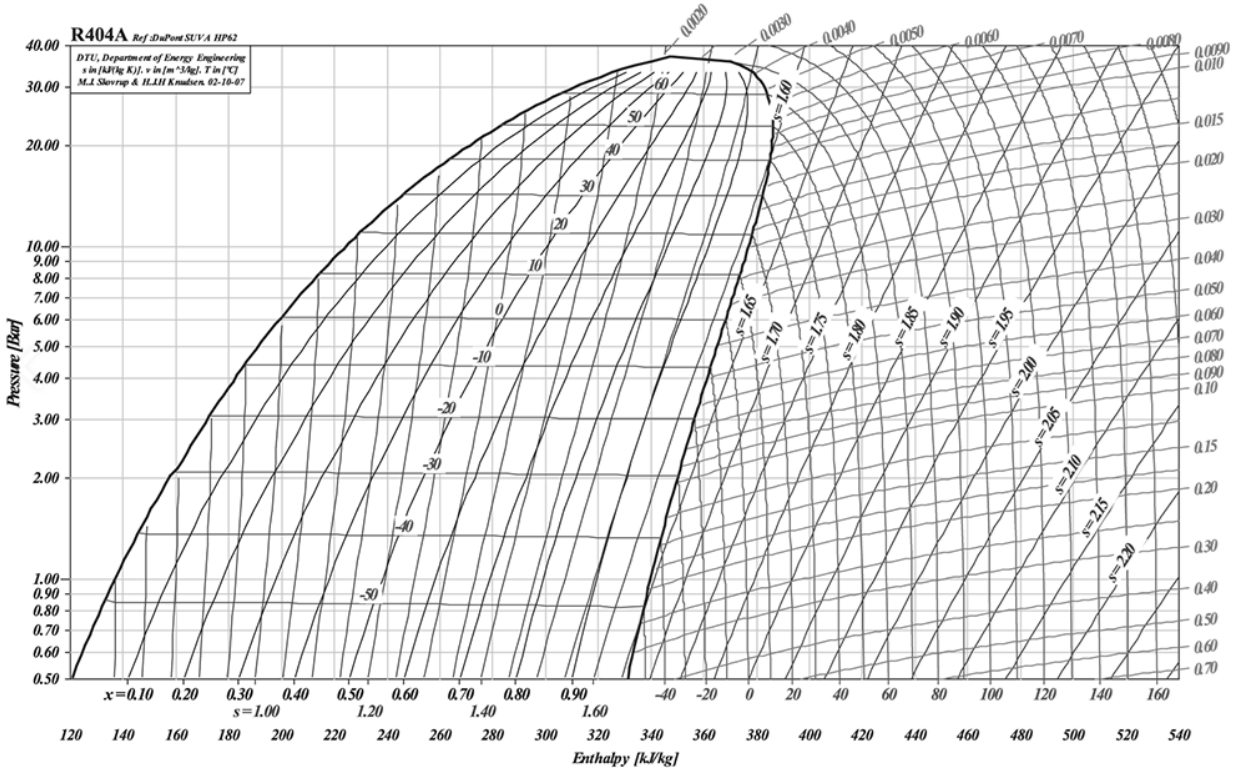
Sur le diagramme enthalpique (P, h) du fluide R404A, sont représentés :

- la courbe de saturation de l'équilibre liquide-vapeur (en trait fort) ;
- les isothermes pour des températures comprises entre $-50\ °C$ et $160\ °$;
- les isentropiques pour des entropies massiques comprises entre $1{,}00\ kJ \cdot K^{-1} \cdot kg^{-1}$ et $2{,}20\ kJ \cdot K^{-1} \cdot kg^{-1}$;
- les isotitres en vapeur sous la courbe de saturation pour des titres massiques en vapeur x variant de 0 à 1.

Dans le domaine du liquide seul, les isothermes sont verticales.

P est exprimée en bar et h en $kJ \cdot kg^{-1}$.

Il est conseillé de placer les points particuliers sur le diagramme ci-dessous pour déterminer les valeurs demandées, mais le diagramme n'est pas à rendre avec la copie.



Tournez la page S.V.P

On étudie l'évolution du fluide au cours d'un cycle en régime permanent.

Q9. A la sortie de l'évaporateur, le fluide est à l'état de vapeur sèche à la température $T_1 = 10\ °C$ et à la pression $P_1 = 4\ bar$. Repérer le point A sur le diagramme. Relever la valeur de l'enthalpie massique h_A du fluide.

Q10. Le compresseur comprime cette vapeur de façon adiabatique et réversible jusqu'à la pression $P_B = 20\ bar$. Repérer le point B sur le diagramme. Relever la valeur de la température T_B et celle de l'enthalpie massique h_B en sortie du compresseur.

Q11. Déterminer la valeur du travail mécanique massique w reçu par le fluide lors de son passage dans le compresseur.

Q12. Le fluide entre ensuite dans le condenseur dans lequel il est refroidi et liquéfié de manière isobare jusqu'à la température $T_C = 43\ °C$.Repérer le point C sur le diagramme. Relever la valeur de l'enthalpie massique h_C en sortie du condenseur.

Q13. A partir de l'état C, le fluide est détendu de manière adiabatique, jusqu'à la pression de l'évaporateur P_1. Le détendeur ne comprend aucune pièce mobile. Montrer que la transformation dans le détendeur est isenthalpique.

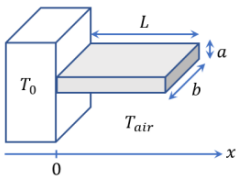
Q14. Repérer le point D sur le diagramme. Relever la valeur de la température T_D et le titre massique en vapeur x_D en sortie du détendeur.

Q15. En déduire le transfert thermique massique q_f échangé par le fluide lors de son passage à travers l'évaporateur (transformation isobare), entre D et A. Commenter son signe.

Q16. Calculer la valeur de l'efficacité e du climatiseur.

Q17. Comparer cette valeur à celle d'un climatiseur réversible fonctionnant entre la température de l'évaporateur et la température de liquéfaction du fluide sous la pression P_2. Commenter le résultat obtenu.

B. Efficacité du refroidissement



On étudie une ailette de longueur L, de section rectangulaire $S = ab$, dont la face en $x = 0$ est en contact avec le dispositif à refroidir, à la température $T_0 = 50\ °C$.

On note $T(x)$ la température dans l'ailette, en la considérant uniforme sur toute section d'abscisse x, et le régime est stationnaire. La température de l'air ambiant, constante et uniforme, est T_{air}.

Le transfert thermique de l'ailette vers l'air ambiant est modélisé, sur la surface latérale de l'ailette, par la relation : $dP = h(T - T_{air})dS$ où dP est la puissance thermique échangée par la surface dS.

h est une constante caractéristique de cet échange thermique, la conductivité thermique de l'ailette est notée λ.

Q18. A l'aide d'un bilan d'énergie réalisé sur une tranche de l'ailette comprise entre les abscisses x et $x + dx$, déterminer l'équation différentielle vérifiée par la température dans l'ailette.

Q19. Introduire une distance caractéristique δ, dont on donnera l'expression en fonction de λ, a, b et h.

Q20. Rappeler la forme générale de la solution de l'équation différentielle, puis préciser à quelle condition portant sur δ il est possible de considérer l'ailette comme infinie.

Q21. En supposant la condition sur δ vérifiée, déterminer complètement l'expression de $T(x)$.

Q22. Exprimer la puissance thermique totale évacuée par l'ailette en fonction de T_0, T_{air}, a, b, λ et h.

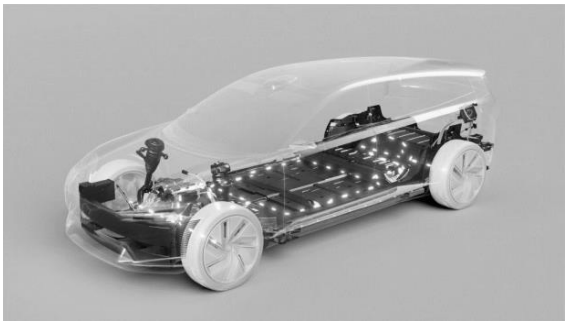
PARTIE INFORMATIQUE

Dans cette partie, les fonctions et programmes attendus devront être écrits en langage Python (sauf, bien entendu, lorsqu'une requête SQL est demandée). On prendra garde à la lisibilité du code, et notamment aux indentations. Toute fonction définie dans le sujet pourra être utilisée, même sans avoir été codée dans les questions précédentes.

L'objectif de développement de la voiture électrique et des hybrides rechargeables en Europe devait suivre le planning suivant :

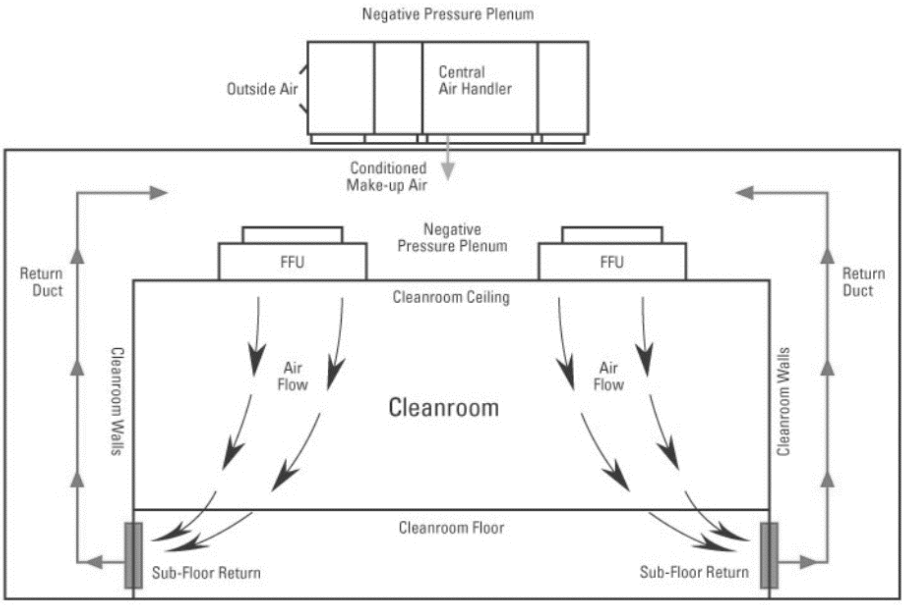
- 9 % en 2020 ;
- 38 % en 2025 ;
- 70 % en 2030 ;
- 100 % en 2035.

En effet, l'objectif fixé par le gouvernement consiste à mettre fin à la voiture thermique d'ici à 2035. Toutefois, pour cela, les pays doivent s'équiper de suffisamment de sites de production de batteries pour véhicules électriques afin d'alimenter l'ensemble des constructeurs automobiles européens.



Ainsi la première gigafactory européenne de fabrication de batteries Lithium-ion pour véhicules automobiles voit le jour en Suède et les premières batteries intègreront les véhicules européens dès 2024. Les salles de fabrication d'une telle usine sont des salles dites *salles blanches* avec climatisation et contrôle environnemental précis :

- taux d'empoussièrement classe ISO5
- taux d'humidité réduit : -40° de point de rosée (air très sec comme au pôle Nord)
- température régulée à 20°C +/- 1°C
- pression de +40 Pa (surpression)



Process de contrôle environnemental d'une salle blanche

C. Déterminer le nombre de FFU, leur répartition et ajustement de la vitesse de rotation

Le plafond de la salle blanche est muni de FFU (Fan filter unit) qui sont des ventilateurs de 1 mètre de diamètre. Ils aspirent l'air et le poussent vers le bas. Ils assurent un taux de renouvellement de l'air de 50 volumes / heure.

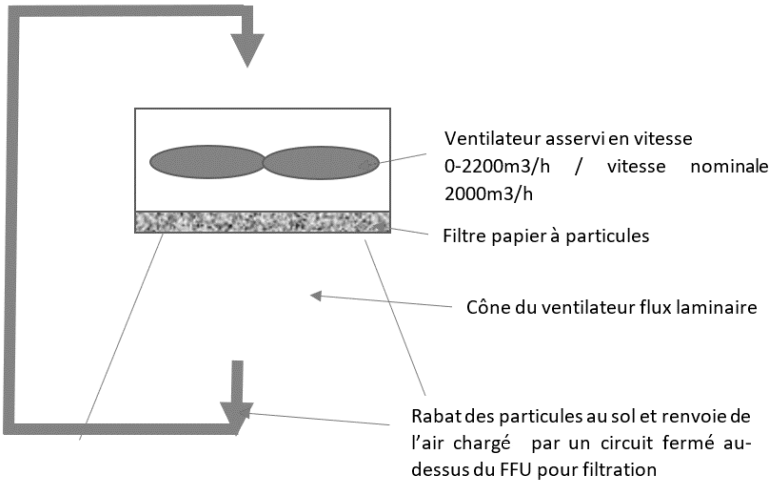


Schéma de principe d'un FFU

L'objectif de cette partie est de déterminer d'une part le nombre de FFU, leur répartition spatiale afin qu'ils soient équirépartis dans la salle et d'autre part la vitesse de rotation proportionnelle au débit à régler sur chacun des FFU afin de respecter une exigence de taux d'empoussièrement imposé par la norme noté tx_renew

On note :
L : longueur de la salle blanche en m ;
ℓ : largeur de la salle blanche en m ;
h : hauteur de la salle blanche en m ;
V : volume de la salle blanche en m^3 ;
Q : débit d'air d'un seul FFU en m^3/h ;
tx_renew : taux de renouvellement d'air indique le nombre de fois à l'heure que l'air total d'une pièce doit être renouvelé ;
nx : nombre de FFU sur une ligne horizontale ;
ny : nombre de FFU sur une ligne verticale ;
n : nombre total de FFU dans la salle.

Q23. Ecrire une fonction **`volume`** qui prend en argument les trois dimensions de la pièce et qui fournit le volume de la pièce en sortie.

Q24. Ecrire une fonction **`nbre_ffu`** qui prend en argument V, Q et tx_renew et qui retourne le nombre n de FFU à prévoir pour répondre à l'exigence du renouvellement d'air du cahier des charges.

On souhaite déterminer le nombre de FFU positionnables le long de l'horizontale noté nx, le long de la verticale noté ny ainsi que l'espacement entre chaque appareil noté u en horizontal et v en vertical afin d'avoir une répartition équirépartie des ventilateurs dans la salle blanche.

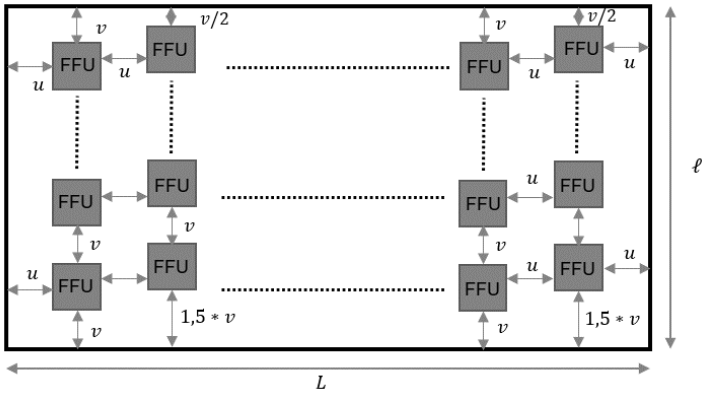


Schéma de positionnement des FFU dans une salle blanche de dimension $L \times \ell$

L'encombrement des FFU est simplifié et modélisé par un carré de 1 m de côté.

L'espacement u ne peut être inférieur à 1 m et l'espacement v ne peut être inférieur à zéro car aucun chevauchement n'est acceptable techniquement.

On souhaite écrire une fonction **`repartition(L,ℓ,n)`** qui prend en argument la longueur L, la largeur ℓ de la salle blanche, le nombre n de FFU calculé précédemment et qui renvoie les valeurs nx, ny, u et v afin d'avoir une répartition équirépartie des ventilateurs dans la salle.

Q25. Justifier que le système d'équations qui relie nx, ny, u et v est le suivant :

$$\begin{cases} L = (nx+1) * u + nx \\ \ell = (ny+1) * v + ny \\ (nx \times ny) < n \ \ avec\ nx\ et\ ny\ des\ entiers\ naturels \end{cases}$$

Q26. Ecrire la situation d'initialisation de la boucle de la fonction **repartition** qui est la situation où tous les FFU sont alignés sur une seule ligne. Déterminer les expressions de nx, ny, u et v dans cette situation initiale.

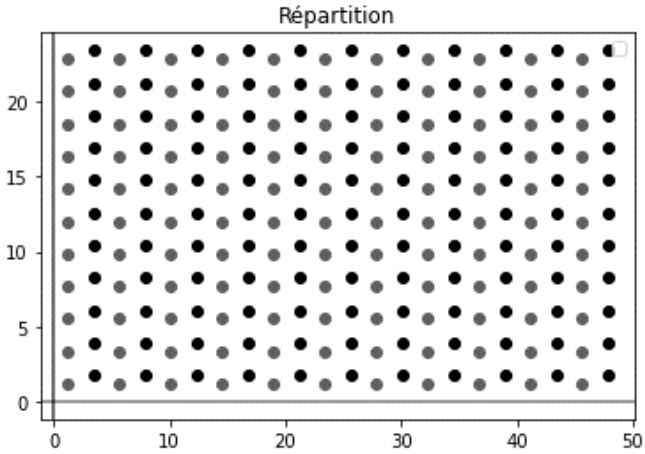
```
def repartition(l,L,n):
    initialisation de nx, ny, u et v

    while COND1 :
        ny=ny+1
        nx=n//ny
        u=EXP2
        v=EXP3
    return nx,ny,a,b
```

script fonction ***repartition(L,ℓ,n)***

Q27. Après l'initialisation des paramètres, la fonction **repartition(L,ℓ,n)** utilise une boucle while pour calculer les nouvelles grandeurs nx, ny, u et v dans la situation d'une répartition des FFU sur une ligne supplémentaire. Une proposition de la boucle while de cette fonction est fournie ci-dessus. Définir la condition d'arrêt de la boucle while COND1 et les expressions EXP2, EXP3 définies dans le script précédent.

La solution obtenue en appliquant la fonction **repartition(L,ℓ,n)** précédente peut être dessinée sous python :



Représentation de la solution de répartition équirépartie des FFU pour une salle blanche de 50x25 m

Comme nx et ny doivent être des entiers naturels, la fonction **repartition(L,ℓ,n)** renvoie une solution telle que $nx * ny < n$.

Pour maintenir le même taux de renouvellement d'air avec moins de ventilateurs, une possibilité est d'augmenter légèrement le débit des ventilateurs par simple augmentation de la vitesse de rotation des ventilateurs sachant que débit et vitesse de rotation sont proportionnels. Le coefficient multiplicateur est calculable par la fonction :

```
def coeff(Q,V,nx,ny,tx_renew):
    Qreel=V*tx_renew/(nx*ny)
    return Qreel/Q
```

D. Etude de résultats expérimentaux afin de modéliser le couple résistant ramené sur l'arbre moteur

L'objectif de cette partie est de dépouiller de manière automatique les fichiers .txt contenant les résultats de 18 essais expérimentaux réalisés sur un FFU afin d'en extraire les données exploitables en vue de construire un modèle de comportement du couple résistant.

1. Extraction des résultats expérimentaux d'un essai

Les fichiers des essais sont tous enregistrés sur le cloud sous le même format .txt. Par exemple les résultats expérimentaux de l'essai 11 nommé "essai11.txt" sont les suivants :

Les 12 premières lignes contiennent les informations relatives à l'essai.

- Ligne 1 : date de l'essai ;
- Ligne 2 : heure de l'essai ;
- Ligne 3 : fréquence d'échantillonnage de la mesure ici 1 000 Hz ;
- Ligne 4 : Système FFU ;
- Ligne 5 : Loi de pilotage : ici la consigne est une loi de vitesse en trapèze avec une phase1 d'accélération – une phase2 à vitesse constante et une phase3 de décélération ;
- Ligne 6 : Nombre de cycles effectués. Ici 1 seul cycle de consigne de vitesse en trapèze ;
- Ligne 7 : durée de la phase d'accélération 30 s ;
- Ligne 8 : consigne de vitesse pour la phase à vitesse constante (différente pour chacun des essais expérimentaux) ;
- Ligne 9 : Durée de la phase à vitesse constante : 50 s ;
- Ligne 10 : : durée de la phase de décélération 30 s ;
- Ligne 11 : indique le nombre de points en lequel est découpé le trapèze de consigne ici 360 points ;
- Ligne 12 : indique l'intervalle de temps entre deux points du trapèze « numérique » précédent ici 0,3 s. Comme la fréquence d'échantillonnage est de 10 Hz = 0,1 s, cela signifie que la consigne est identique pour 3 points de mesure consécutifs.

Les lignes suivantes contiennent les informations relatives aux mesures

- Ligne 13 : titres des grandeurs mesurées séparés par des tabulations : le temps (s) – la consigne (tr/min) – la vitesse du moteur mesurée (tr/min) – le courant moteur mesuré (Ampère) ;
- Ligne 14 et suivantes : les valeurs mesurées à chaque instant. Chaque ligne est une liste de 4 chaines de caractères séparées par des tabulations /t.

Le répertoire courant étant celui qui contient l'ensemble des fichiers résultats, on travaillera par la suite avec les chemins relatifs simples comme "essai11.txt".

Le script suivant permet d'afficher les 26 premières lignes de l'essai 11 :

```
nom_fichier="essai11.txt"
fichier=open(nom_fichier,"r")
for i in range(0,26):
    ligne=fichier.readline()
    print(ligne)
fichier.close()
```

L'instruction **`fichier.readline()`** permet d'obtenir une ligne d'un fichier puis de passer à la ligne suivante.

```
Date Essai  07/02/2023
Heure Essai 14:45
Frequence (Hz)    10
Fonctionnement    ffu
Type Pilotage     Trapeze
Nb cycles   1
Duree acceleration (s)  30
Vitesse (tr/min)  2200
Temps maintien (s)      50
 Duree deceleration (s) 30
Nb points de consigne   365
dt consigne 0,3
Temps (s)   Consigne (tr/min) Vitesse du moteur (tr/min)    Courant (A)
0     22    -0,884      0
0,1   22    -9,96 0,041
0,2   22    -12,924     0,057
0,3   44    -12,924     0,079
4     44    -9,96 0,144
0,5   44    -0,884      0,151
0,6   66    3,744 0,255
0,7   66    3,744 0,349
0,8   66    13,748      0,349
```

Les résultats expérimentaux relevés sont des chaines de caractères type string nombres à virgule alors que Python utilise des nombres à point.

Q28. Écrire une fonction python nommée **`remplace_virgule_par_point(chaine)`** qui prend en argument une chaine de caractères appelée `chaine` et qui renvoie une nouvelle chaine de caractères obtenue en remplaçant les virgules "**,**" par des points "**.**" On rappelle que le type string est un conteneur numéroté comme une liste mais non mutable. Ainsi on fabriquera la nouvelle chaine, caractère par caractère.

Pour pouvoir étudier le comportement du système, il est nécessaire pour chacun des essais d'extraire des données numériques de temps, de consigne, de vitesse et de courant d'un fichier .txt issu d'un essai sur le système FFU, puis de stocker les données extraites de chaque ligne de mesures du fichier dans une liste des temps intitulée **`Ltemps`**, une liste des valeurs de consigne intitulée **`Lconsigne`**, une liste des vitesses moteur intitulée **`Lvitesse`** et une liste des courant intitulée **`Lcourant`**.

On rappelle que :

- L'instruction **`ligne.split("\t")`** renvoie une liste de chaînes de caractères obtenue en divisant la chaîne de caractères **`ligne`** à chaque apparition du séparateur tabulation : **`"\t"`**.

Par exemple :

```
fichier=open("essai11.txt","r")
    for i in range(13):
        ligne=fichier.readline()
    for ligne in fichier:
        ligne2=remplace_virgule_par_point(ligne)
        liste=ligne2.split("\t")
        print(liste)
```

renvoie pour les 6 premières lignes :

```
['0', '22', '-0.884', '0\n']
['0.1', '22', '-9.96', '0.041\n']
['0.2', '22', '-12.924', '0.057\n']
['0.3', '44', '-12.924', '0.079\n']
['0.4', '44', '-9.96', '0.144\n']
['0.5', '44', '-0.884', '0.151\n'] etc…
```

- L'instruction **`fichier.readline()`** permet d'obtenir une ligne d'un fichier puis de passer à la ligne suivante, ce qui va permettre de passer les 13 premières lignes du fichier essai11.txt
- L'instruction **`float("3.14")`** permet d'obtenir le flottant 3.14 correspondant à la chaine de caractères "3.14".Par contre, **`float("3,14")`** renvoie un message d'erreur, car Python ne gère pas les flottants avec des virgules.

```
def extraction_resultats_essai(chemin_fichier):
    #initilisation des listes vides
    Ltemps=[]
    Lconsigne=[]
    Lvitesse=[]
    Lcourant=[]
    fichier=open(chemin_fichier,"r")
    #pour passer les 13 premières lignes
    for i in range(13):
        ligne=fichier.readline()
    #pour traiter les lignes suivantes
    for ligne in fichier:
        ligne2=remplace_virgule_par_point(ARGS1)
        liste=ligne2.split("\t")
        #remplissage des données en transformant les str en float
        Ltemps.append(ARGS2)
        Lconsigne.append(ARGS3)
        Lvitesse.append(ARGS4)
        Lcourant.append(ARGS5)
    fichier.close()
      return Ltemps, Lconsigne, Lvitesse, Lcourant
```

Script incomplet de la fonction ***`extraction_resultats_essai(nom_fichier)`***

Q29. La capture précédente fournit le script incomplet de la fonction **`extraction_resultats_essai(nom_fichier)`** qui prend en argument le nom du fichier de l'essai analysé et qui renvoie les quatre listes des grandeurs physiques numériques type flottants de l'essai étudié **`Ltemps`**, **`Lconsigne`**, **`Lvitesse`**, **`Lcourant`**, soit les données présentes dans la première, la deuxième, la troisième et la quatrième colonne du fichier à partir de la 14e ligne. Définir les arguments ARGS1, ARGS2, ARGS3 et ARGS4 définis dans le script précédent.

La fonction précédente **`extraction_resultats_essai(nom_fichier)`** est ensuite complétée par les instructions suivantes :

```
dico={"temps":Ltemps,"consigne":Lconsigne,"vitesse":Lvitesse,"courant
":Lcourant}
return dico
```

Q30. Que signifient ces 2 instructions et décrire la structure de données choisie ? Préciser son intérêt.

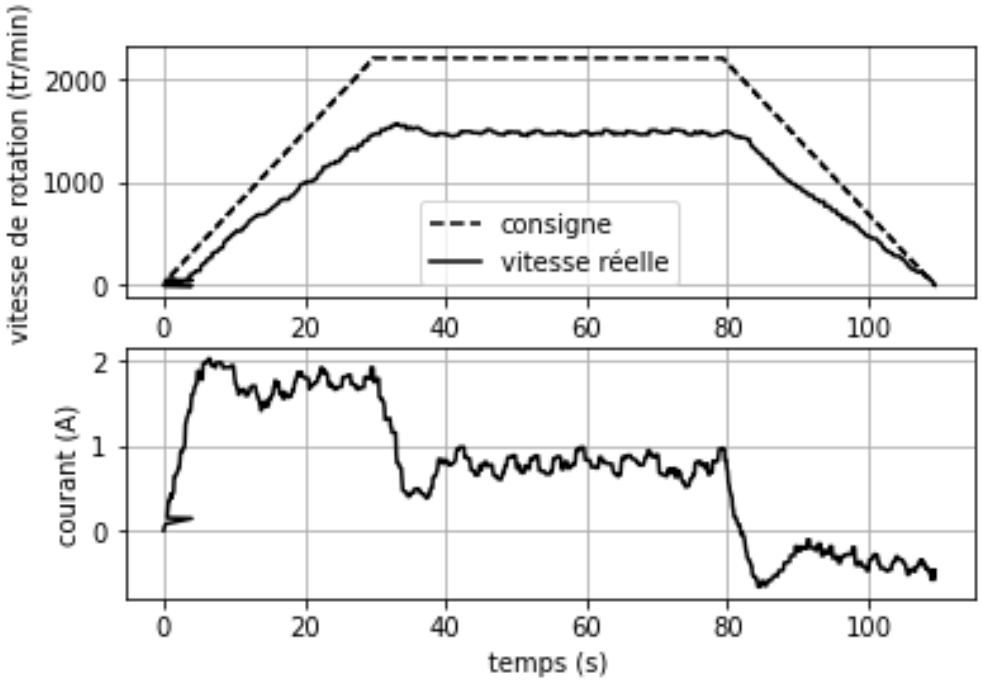
Q31. Ecrire un script python qui permet d'extraire les résultats de l'essai 1 qui seront stockés dans un dictionnaire nommé **donnees**, puis qui affiche la liste des temps.

La bibliothèque matplot est intégrée dans le programme :

```
import matplotlib.pyplot as plt
```

Q32. Ecrire une fonction **`affichage(donnees)`** qui prend en argument le dictionnaire stockant les résultats d'un essai et qui renvoie la courbe des vitesses mesurées d'un essai en fonction du temps.

Les courbes obtenues des résultats expérimentaux pour l'essai 11 sont les suivantes :



Représentation des résultats expérimentaux extrait du fichier « essai11.txt »

2. Détermination du couple résistant

En régime permanent (vitesse de rotation constante, pas d'accélération), le couple moteur $C_m(t)$ sert uniquement à vaincre le couple résistant $C_r(t)$ dans un système FFU où l'axe moteur est fixe par rapport au repère galiléen. De plus pour un moteur à courant continu, le couple moteur est proportionnel au courant $i(t)$ selon la constante de couple K, le couple résistant peut donc se déterminer par la relation suivante (en régime permanent) :

$$C_r(t) = C_m(t) = K \times i(t) \text{ (si la vitesse de rotation est constante)}$$
$$\text{Avec } K = 0{,}13\ Nm/A \text{ (Newton mètre par ampère)}$$

Q33. Écrire la fonction **`moyenne_sur_intervalle(Liste,deb,fin)`** qui permet de déterminer la valeur moyenne des nombres compris entre les indices deb (inclus) et fin (exclu) de la liste **`Liste`**.

Le régime permanent est atteint pour $t \in [50\,;70]$ en secondes correspondant aux indices $i \in [500\,;700]$ avec la fréquence d'échantillonnage de 10 Hz.

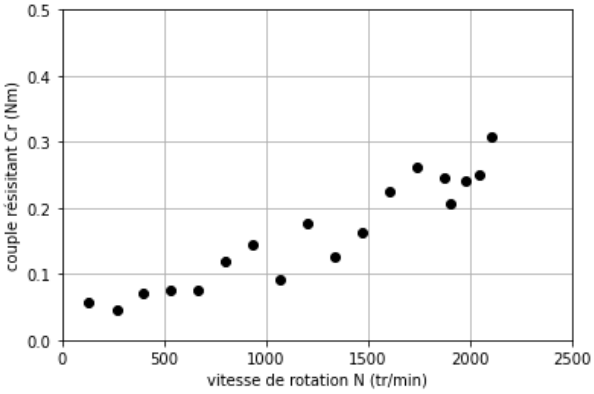
Q34. Écrire une fonction **`point_de_fonctionnement(donnees)`** qui prend en argument le dictionnaire stockant les résultats de l'essai étudié et qui renvoie la valeur de la vitesse de rotation N en régime permanent et du couple résistant Cr de l'essai étudié en tenant compte uniquement du courant entre 50 et 70 s.

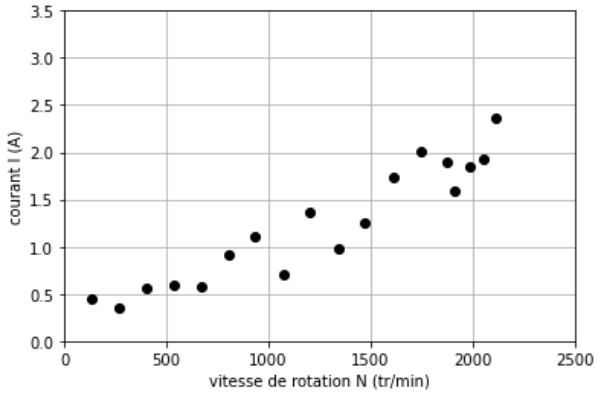
3. Analyse de l'ensemble des essais

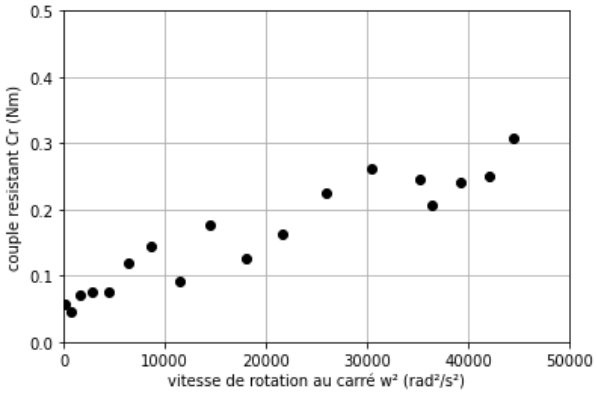
On souhaite maintenant déterminer le point de fonctionnement (N, Cr) de chacun des 18 essais expérimentaux dont les mesures sont stockées dans le même dossier dans 18 fichiers .txt intitulés "essai1.txt" à "essai18.txt". Chacun des points de fonctionnement est mémorisé dans une liste **`L_vit`** et une liste **`L_Cr`**.

Q35. Ecrire un script python qui pour chaque essai présent dans le dossier, détermine le couple résistant Cr et la vitesse de rotation N sur l'intervalle de temps compris entre 50 et 70 s et mémorise Cr et N dans deux listes L_vit et L_Cr.

Les allures des 18 points de fonctionnement expérimentaux du couple résistant (Nm) en fonction de la vitesse de rotation du moteur (tr/min), du courant (A) en fonction de la vitesse de rotation du moteur (tr/min) et du couple résistant (Nm) en fonction du carré de la vitesse de rotation du moteur (rad²/s²) sont représentées sur les trois figures suivantes :







4. Modélisation du comportement couple résistant Cr en fonction de la vitesse du moteur N par une régression linéaire.

L'objectif est de construire un modèle de comportement du couple résistant agissant sur l'arbre moteur dans l'ensemble FFU.

L'analyse des points de fonctionnement expérimentaux nous permet de proposer le modèle de comportement type quadratique :

$$Cr = a_1 * \omega^2 + a_0$$

Il s'agit donc dans la suite de déterminer les constantes a_1 et a_0 par régression linéaire.

La méthode utilisée pour construire la régression est la méthode de descente de gradient utilisée en intelligence artificielle. Celle-ci est utilisée dans notre étude simplifiée pour réaliser une régression linéaire simple mais son champ d'utilisation est bien plus large et permet de réaliser une régression linéaire multiple sur une base de données à plusieurs paramètres. Par exemple on aurait pu ajouter dans le modèle les paramètres : température, humidité et nombre d'heures d'utilisation.

Q36. Justifier que la méthode de descente de gradient est une méthode d'apprentissage supervisé. Expliquer la différence entre un apprentissage supervisé avec classification et avec régression.

Q37. Qu'est-ce qu'une régression linéaire multiple ? Quel serait dans ce cas l'expression du modèle de comportement de Cr à rechercher ?

Description de la base de données résultats de l'étude de la partie précédente:

- la base contient m observations ;

Chaque observation contient :

- une valeur explicative $x^{(k)}$, carré de la vitesse de rotation ω^2 en rad²/s² stockée dans un vecteur V_N2 de dimension $(m \times 1)$;
- une réponse $y^{(k)}$, couple résistant en Nm, stockée dans un vecteur Y de dimension $(m \times 1)$.

$$V_N2 = \begin{bmatrix} x^{(1)} \\ \vdots \\ x^{(m)} \end{bmatrix}, \quad Y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

La bibliothèque numpy est intégrée dans le programme :

```
import numpy as np
```

On rappelle :

- La fonction **`np.hstack((M,N))`** permet de concaténer deux tableaux pour former une matrice ;
- La fonction **`np.ones([i,j])`** permet de créer un tableau rempli de 1 de i lignes et j colonnes ;
- La fonction **`np.dot(M,N)`** renvoie le produit de deux matrices M et N ;
- La fonction **`np.transpose(M)`** renvoie la matrice transposée de M ;
- La fonction **`np.sum(V)`** renvoie la somme des composantes du vecteur V.

Q38. Ecrire un script python qui permet de construire la matrice $X = \begin{bmatrix} x^{(1)} & 1 \\ \vdots & \vdots \\ x^{(m)} & 1 \end{bmatrix}$ afin de prendre en compte le biais du modèle affine.

Pour déterminer le modèle, un vecteur F de dimension $(m \times 1)$ est créé : $F = \begin{bmatrix} f(x^{(1)}) \\ \vdots \\ f(x^{(m)}) \end{bmatrix}$

Pour calculer ce vecteur, on utilise la relation : $F = X \cdot \theta$ où θ est le vecteur θ qui contient tous les paramètres du modèle : $\theta = \begin{bmatrix} a_1 \\ a_0 \end{bmatrix}$

Q39. Ecrire la fonction **`modele`** qui prend en argument (X et $theta$) et retourne la matrice F.

La fonction de coût $Q(a_0, a_1)$ caractérise la somme des erreurs au carré entre le modèle $f\left(x^{(k)}\right)$ et la réponse $y^{(k)}$.

Q40. Ecrire la fonction coût $Q(a_0, a_1)$ en fonction de a_0, a_1, les $y^{(k)}$ et $x^{(k)}$ et m.

Q41. Ecrire la fonction coût $Q(\theta)$ sous forme matricielle en fonction de X, θ et Y.

Q42. Ecrire en python la fonction **`cout`** qui prend en argument trois vecteurs X,Y et theta et qui renvoie la valeur du coût $Q(\theta)$.

Q43. Ecrire les dérivées partielles de la fonction $Q(a_0, a_1)$: $\frac{\partial Q}{\partial a_1}$ et $\frac{\partial Q}{\partial a_0}$.

Q44. Ecrire le vecteur gradient de la fonction coût $\overrightarrow{grad}(Q)$ sous forme matricielle en fonction de X, θ et Y et montrer que : $\overrightarrow{grad}(Q) = 2 \cdot X^T(X \cdot \theta - Y)$ avec X^T la transposée de la matrice X.

Q45. Ecrire en python une fonction **`gradient`** qui prend en argument trois vecteurs X,Y et theta et qui renvoie le vecteur gradient du cout Q, $\overrightarrow{grad}(Q)$. On pourra utiliser la fonction `np.transpose(M)` qui transpose la matrice M.

La descente de gradient est une méthode itérative qui repose sur le fait que, dans le voisinage d'un point, la fonction de coût décroît moins que son gradient. Et au voisinage du minimum l'écart entre la pente de la fonction coût et le gradient tend vers zéro.

Cet algorithme permet ainsi de trouver le minimum de la fonction de coût en partant de paramètres a_0, a_1 initiaux et aléatoires. A chaque itération les paramètres du modèle sont mis à jour en leur soustrayant la plus grande pente possible multipliée par la vitesse de convergence (le pas) α.

$$a_{1,k+1} = a_{1,k} - \alpha \cdot \frac{\partial Q(a_{1,k})}{\partial a_1}$$

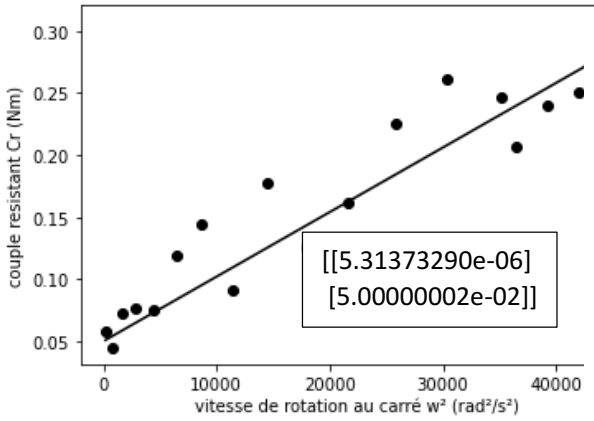
$$a_{0,k+1} = a_{0,k} - \alpha \cdot \frac{\partial Q(a_{0,k})}{\partial a_0}$$

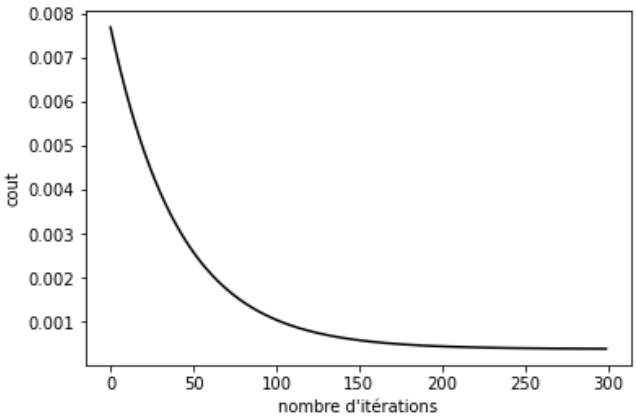
La vitesse de convergence est déterminée par l'hyperparamètre α, appelé vitesse d'apprentissage.

Q46. Ecrire la fonction **`descente_gradient(X,Y,theta_init,alpha,n)`** qui prend en argument les données **X** et **Y**, le vecteur initial noté **`theta_init`**, la vitesse d'apprentissage **`alpha`** choisi, et **n** le nombre d'itérations choisi et qui renvoie le vecteur **`theta`** obtenu par la méthode de descente de gradient après les **n** itérations.

Le programme précédent permet d'obtenir le modèle de couple résistant suivant après 300 itérations:

$$Cr = a_1 * \omega^2 + a_0 \quad avec\ a_0 = 0.05\ et\ a_1 = 5.313e-6$$





Tournez la page S.V.P

E. Etude du comportement du moteur électrique entraînant le ventilateur

Dans la suite de l'étude, l'objectif est d'étudier les caractéristiques du comportement d'un moteur électrique à courant continu à excitation séparée entraînant le ventilateur dont on désire régler la vitesse de rotation.

1. Etude du régime établi : recherche de la tension de commande moteur

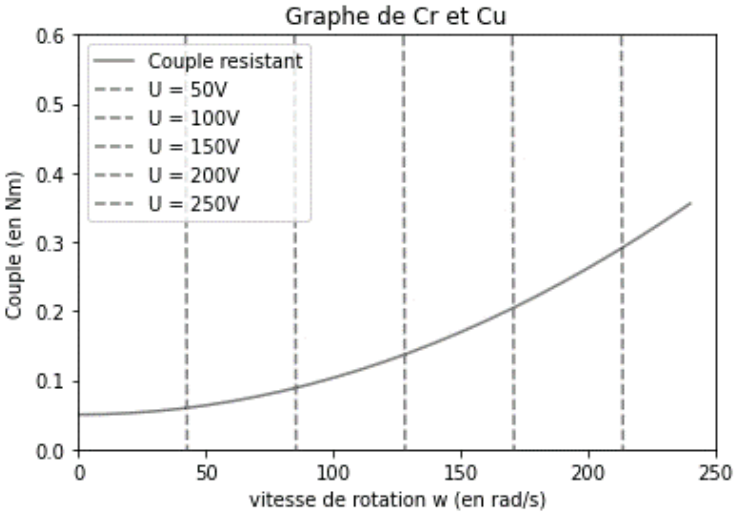
Le ventilateur est entrainé directement par le moteur.

Le couple résistant Cr est lié à la fréquence de rotation ω par la loi : $Cr = a_1 * \omega^2 + a_0$ (avec Cr en Nm, ω en rad/s, a_0=0,05 Nm et a_1 =5,3*10^{-6} Nm/(rad/s)²

Q47. Ecrire une fonction **affichage** qui prend en argument les valeurs de a_0 et a_1 et qui permet de tracer l'allure $Cr(\omega)$ pour ω compris entre 0 et 250 rad/s par pas de 10.

Les caractéristiques du moteur du FFU permettent de déterminer l'expression du couple utile Cu (en Nm) sur l'arbre moteur en fonction de ω (en rad/s) et de la tension de commande U (en V) pour I=1,5A=cste dont l'expression est la suivante :

$$C_u = \frac{k}{R}(U - k.\omega) = 0{,}839 \cdot (U - 1{,}17 \cdot \omega)$$



On souhaite déterminer les différentes valeurs de ω pour une tension d'alimentation U donnée. Cela revient à chercher le point d'intersection entre :

$$Cr = a_1 * \omega^2 + a_0$$

$$C_u = 0{,}839 \cdot (U - 1{,}17 \cdot \omega)$$

Et donc à chercher une solution de l'équation :

$$a_1 * \omega^2 + a_0 - 0{,}839 \cdot (U - 1{,}17 \cdot \omega) = 0$$

On définit la fonction f par

```
def f(w,U):
    a1=5.3e-6
    a0=0.05
    return a1*w*w+a0-0.839*(U-1.17*w)
```

Q48. Ecrire une **`fonction dichotomie(U,deb,fin,e)`** qui prend en argument la valeur de la tension d'alimentation imposée U (en V), les bornes de l'intervalle de $\omega \in [deb, fin]$, la précision du résultat ε et qui retourne les coordonnées (ω, U) solution de l'équation $f(\omega, U) = 0$ à ε près.

Ainsi grâce à la partie C et la partie E.1 : on connait la tension à imposer à l'ensemble moteur + ventilateur soit 146 V en régime permanent pour assurer la vitesse de rotation nécessaire de 1250 tr/min et donc le taux de renouvellement d'air imposé par le cahier des charges.

2. Etude en régime dynamique : prévision des caractéristiques de la réponse du FFU à un échelon de courant.

Dans cette partie, le courant I dans l'inducteur reste constant.

On néglige désormais la résistance de l'induit R.

On note J le moment d'inertie de l'ensemble aéraulique ramené à l'arbre moteur.

Le moteur est couplé directement au ventilateur, on lui applique un échelon de courant I. On souhaite déterminer l'évolution de la vitesse de rotation $\omega(t)$ au cours du temps sachant que $\omega(0) = \omega_0$.

L'équation de la dynamique s'écrit : $C_m - C_r = J \cdot \frac{d\omega}{dt} \Rightarrow k \cdot I - a_1 * \omega^2 - a_0 = J \cdot \frac{d\omega}{dt}$

Il est donc nécessaire de résoudre l'équation différentielle non linéaire suivante :

$$J \cdot \frac{d\omega}{dt} + a \cdot \omega^2 + b = k \cdot I$$

On utilisera la méthode d'Euler-explicite.

Q49. Ecrire l'équation différentielle régissant la vitesse de rotation du rotor sous la forme $\frac{d\omega}{dt} = g(\omega(t), t)$. On précise que la fonction g ne dépend pas explicitement du temps t.

Q50. A l'aide d'un développement limité de la variable $\omega(t+T)$ à l'ordre 1, montrer que l'approximation de la dérivée peut s'écrire :

$$\frac{d\omega(t)}{dt} \approx \frac{\omega(t+T) - \omega(t)}{T} \quad avec\, T << t$$

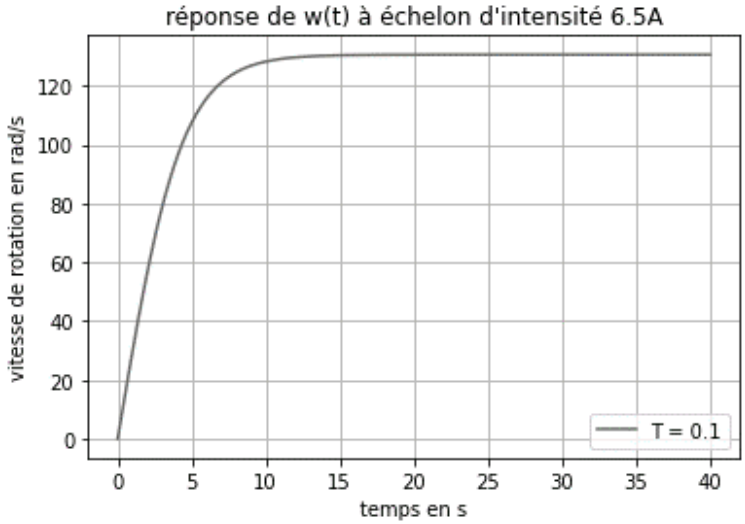
Q51. Déterminer l'expression de $\omega(t+T)$ en fonction de T, $g(\omega(t), t)$ et $\omega(t)$

On note $\boldsymbol{LT}$ la liste des temps qui démarre à t0 et s'arrête à t1 et de pas de temps T.

On note $\boldsymbol{L\omega}$ la liste des valeurs de la vitesse de rotation ω .

Q52. Proposer une fonction **`Euler_Explicite`** dont les arguments sont $g, \omega 0, t0, t1, T$ permettant de résoudre l'équation différentielle $\frac{d\omega}{dt} = g(\omega(t), t)$ avec comme condition initiale $\omega(0) = \omega 0$, par itérations successives de pas de temps T sur l'intervalle $[\boldsymbol{t0}, \boldsymbol{t1}]$ en utilisant la méthode d'Euler explicite, et renvoyant la liste des temps $\boldsymbol{LT}$ et la liste $\boldsymbol{L\omega}$ des solutions approchées associées. On impose d'utiliser une boucle while.

Le tracé de la solution numérique est représenté ci-dessous :



Q53. Les listes $\boldsymbol{LT}$ et $\boldsymbol{L\omega}$ étant calculées, écrire un script python permettant de déterminer puis d'afficher le temps de réponse et la vitesse de rotation en régime permanent du système.

Il sera alors par la suite possible d'analyser les écarts entre les performances simulées et les performances attendues.

IMPRIMERIE NATIONALE – 24 1014 – D'après documents fournis

ECOLE POLYTECHNIQUE
ECOLES NORMALES SUPERIEURES

CONCOURS D'ADMISSION 2024

JEUDI 18 AVRIL 2024
08h00 - 12h00

FILIERE MPI - Epreuve n° 7

INFORMATIQUE C (XULSR)

Durée : 4 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve

Égalités et différences

Le sujet comporte 17 pages, numérotées de 1 à 17 et une annexe.

Vue d'ensemble du sujet.

Ce sujet traite du problème de satisfiabilité de formules construites par la conjonction de contraintes d'égalité et de différence (dis-égalité) entre expressions. Ce problème se présente, par exemple, dans la compilation de programmes, quand on cherche à réduire des expressions complexes (nécessitant beaucoup de calculs) vers des expressions plus simples, en utilisant les propriétés des opérateurs apparaissant dans ces expressions. Par exemple, si `x` est une variable entière, `(x * 2)/2 == x` est une contrainte d'égalité entre deux expressions écrites en C. De même, si `y` et `z` sont des variables booléennes, l'égalité `(y || (y && z)) == y` permet de simplifier l'expression C écrite à gauche de l'égalité.

Il est important de savoir si un ensemble donné de contraintes d'égalité ou de différence est satisfiable. Dans la suite, P_L ***est le problème suivant : « Étant donnée une formule F, conjonction de contraintes de la classe L, décider si F est satisfiable ».***

Ce sujet propose d'étudier la complexité du problème P_L et les algorithmes pour le résoudre dans le cas de quatre classes de contraintes :

La partie I considère la classe L_1 où les contraintes ont la forme $x \leftrightarrow y$ ou $x \leftrightarrow \neg z$ avec x, y et z des variables propositionnelles (c'est-à-dire, ayant des valeurs booléennes).

La partie II concerne la classe L_2 incluant les contraintes de la classe L_1 et celles de la forme $x \leftrightarrow (y \wedge z)$ (entre autres) avec x, y et z des variables propositionnelles. Les algorithmes des deux premières parties seront codés en OCaml.

La partie III s'intéresse à la classe L_3 des contraintes de la forme $x = y$ ou $x \neq y$ avec x et y des variables entières.

Enfin, la partie IV concerne les contraintes d'égalité et de différence entre les expressions construites à partir de variables entières et de fonctions à deux arguments. Les algorithmes des deux dernières parties seront codés en C.

Les parties III et IV peuvent être traitées indépendamment de parties I et II. Il est recommandé de traiter les parties I et II de manière séquentielle. De même pour les parties III et IV.

Notations

Les contraintes seront construites en utilisant un ensemble $\mathbb{X}$ de n variables notées X_i avec $1 \leq i \leq n$. Une valeur de vérité est un élément de $\mathbb{B} \stackrel{\text{def}}{=} \{0, 1\}$ où 1 représente le vrai et 0 le faux. Une valeur entière k est un élément de $\mathbb{Z}$ et sa valeur absolue est notée $|k|$. Étant donnés $k, m \in \mathbb{Z}$ avec $k \leq m$, l'intervalle $[k; m]$ est l'ensemble des $i \in \mathbb{Z}$ tels que $k \leq i \leq m$.

Logique propositionnelle. On utilisera les formules de la logique propositionnelle construites à partir de l'ensemble de variables propositionnelles $\mathbb{X}$, les constantes propositionnelles $\bot$ (faux) et $\top$ (vrai), les connecteurs logiques classiques $\neg$ (négation), $\wedge$ (conjonction), $\vee$ (disjonction). On définit $F \to G$ (implication) par $(\neg F \vee G)$, et $F \leftrightarrow G$ (équivalence) par $(F \to G) \wedge (G \to F)$. Pour les connecteurs $\wedge$ et $\vee$, on utilise leur forme k-aire avec $k \geq 0$. Par exemple, $X_1 \wedge X_2 \wedge \ldots \wedge X_k$ est une formule utilisant une conjonction k-aire. Par convention, une conjonction 0-aire est $\top$, et une disjonction 0-aire est $\bot$; une conjonction (resp. disjonction) unaire est identique à son opérande.

Un ***littéral*** ℓ est soit une variable X_i soit sa négation $\neg X_i$. Une formule est en ***forme normale conjonctive*** (CNF) si c'est une conjonction de disjonctions de littéraux. Une formule est en forme k-CNF avec $k > 0$, si elle est en forme CNF et le nombre de littéraux différents dans chaque disjonction est au plus k. Par exemple, la formule $(X_2 \vee X_3 \vee \neg X_4) \wedge (\neg X_3 \vee X_4) \wedge X_1$ est en forme 3-CNF.

On rappelle qu'une formule propositionnelle F peut être ***satisfaite*** ou non par une valuation $v : \mathbb{X} \to \mathbb{B}$, qui assigne une valeur de vérité à chaque variable. On note $v \models F$ la relation de satisfaction. Une formule F' est ***conséquence logique*** de F, noté $F \models F'$, quand, pour toute valuation v telle que $v \models F$, on a aussi $v \models F'$. Deux formules sont ***logiquement équivalentes*** lorsque chacune est conséquence logique de l'autre (ou, de façon équivalente, quand elles sont satisfaites par les mêmes valuations). Une formule est une ***tautologie*** si elle est satisfaite par toutes les valuations. Une formule est une ***contradiction*** (antilogie) ou contradictoire si elle n'est satisfaite par aucune valuation. Une formule F est ***satisfiable*** s'il existe une valuation v telle que $v \models F$.

L'annexe rappelle certaines règles de la déduction naturelle.

Complexité. Par la complexité d'un algorithme A on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaires à l'exécution de A ***dans le pire cas***. Lorsque la complexité dépend d'un ou plusieurs paramètres $k_0, \ldots, k_r$, on dit que A a une complexité en $O(f(k_0, \ldots, k_r))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de $k_0, \ldots, k_r$ suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres $k_0, \ldots, k_r$ la complexité est au plus $C \times f(k_0, \ldots, k_r)$. De même, on dit que A a une complexité en $\Theta(f(k_0, \ldots, k_r))$ s'il existe deux constantes $C_1 > 0$ et $C_2 > 0$ telles que, pour toutes les valeurs de $k_0, \ldots, k_r$ suffisamment grandes, la complexité de A est au moins $C_1 \times f(k_0, \ldots, k_r)$ et au plus $C_2 \times f(k_0, \ldots, k_r)$. Dans tout ce sujet, on note « log » le logarithme à base 2, et on utilisera exclusivement ce logarithme.

Programmation. Dans les questions de programmation en C et OCaml, on n'utilise que les fonctions qui sont incluses dans la bibliothèque standard du langage.

Pour le code écrit en C, on suppose que les en-têtes `stdbool.h`, `stdio.h`, `stdlib.h` et `assert.h` ont été inclus.

On rappelle quelques opérations sur les listes en OCaml.

— `List.iter f lst` applique la fonction `f: 'a -> unit` à chaque élément de la liste `lst` de type `'a list`. Si la complexité de `f` est en $O(1)$, alors la complexité de `List.iter` est en $O(k)$, où k est la longueur de `lst`.

— `List.map f lst` renvoie la liste de type `'b list` obtenue en appliquant la fonction `f: 'a -> 'b` à chaque élément de la liste `lst` de type `'a list`. Si la complexité de `f` est en $O(1)$, alors la complexité de `List.map` est en $O(k)$, où k est la longueur de `lst`.

— `List.mem e lst` renvoie `true` si et seulement si `e` est un élément de `lst`. Si la complexité du test d'égalité est en $O(1)$, la complexité de `List.mem` est en $O(k)$, où k est la longueur de `lst`.

— `List.exists f lst` renvoie `true` si et seulement s'il existe au moins un élément `e` de `lst` (liste de type `'a list`) tel que `f e` renvoie `true` avec `f: 'a -> bool`. Si la complexité de `f` est en $O(1)$, alors la complexité de `List.exists` est en $O(k)$, où k est la longueur de `lst`.

— `List.for_all f lst` renvoie `true` si et seulement si tous les éléments de `lst` (de type `'a list`) satisfont le prédicat `f: 'a -> bool`. Si la complexité de `f` est en $O(1)$, alors la complexité de `List.for_all` est en $O(k)$, où k est la longueur de `lst`.

— `List.filter f lst` renvoie la liste de tous les éléments de `lst` (de type `'a list`) qui satisfont le prédicat `f: 'a -> bool`. Si la complexité de `f` est en $O(1)$, alors la complexité de `List.filter` est en $O(k)$, où k est la longueur de `lst`.

On rappelle quelques opérations sur les tableaux en OCaml.

— `Array.length tab` renvoie la longueur du tableau `tab` en temps $O(1)$.

— `Array.make k v` crée un tableau de `k` éléments, tous initialisés à `v`, en temps $O(\mathtt{k})$.

— La valeur à la case numéro `i` du tableau `tab` est `tab.(i)`. Les cases sont numérotées à partir de zéro et l'accès est fait en temps $O(1)$.

— L'affectation de la case `i` du tableau `tab` avec la valeur `v` s'écrit `tab.(i) <- v`; sa complexité est en $O(1)$.

— `Array.copy tab` crée une copie du tableau `tab` en temps $O(k)$ avec k la longueur du tableau en argument.

Partie I. Équivalences de littéraux

Dans cette partie, on se place dans le cadre de la logique propositionnelle. On définit L_1 comme la classe des contraintes de la forme $\ell \leftrightarrow \ell'$ avec ℓ et ℓ' des littéraux. Une formule F est dite construite sur L_1, notation $F \in \mathcal{F}(L_1)$, si elle est une conjonction de contraintes de L_1. ***Dans cette partie, on considère que la variable X_1 est toujours interprétée à 1, donc qu'elle représente la valeur de vérité vrai.*** La formule F_0 définie ci-dessous est dans $\mathcal{F}(L_1)$:

$$F_0 \quad \stackrel{\text{def}}{=} \quad (X_3 \leftrightarrow \neg X_2) \wedge (X_4 \leftrightarrow X_3) \wedge (X_3 \leftrightarrow X_4) \wedge (X_1 \leftrightarrow \neg X_5)$$

Question 1. Donner une formule logiquement équivalente à F_0 en forme 2-CNF. ⌟

Question 2. Donner une réduction en temps linéaire du problème P_{L_1} au problème de satisfiabilité des formules en 2-CNF. ⌟

Mise en œuvre en OCaml. Une contrainte de L_1 est représentée par un couple d'entiers (i, j) avec $i, j \in [1; 2n]$. Pour tout $i \in [1; 2n]$, le littéral représenté par i, noté $lit(i)$, est X_i si $i \in [1; n]$ et $\neg X_{i-n}$ si $i \in [n+1; 2n]$. Par la convention sur l'interprétation de X_1, l'entier $i = 1$ (resp. $i = n+1$) représente la valeur de vérité 1 (resp. 0). Enfin, une formule dans $\mathcal{F}(L_1)$ est représentée par la liste des couples correspondant aux contraintes dont la formule est conjonction. Par convention, la liste vide représente $\top$. Le nombre de variables, n, est une variable globale entière. Ce codage est donné par les déclarations OCaml ci-dessous :

```
1 type contr1 = int * int    (* contraintes L1 *)
2 type eform1 = contr1 list (* formules L1 *)
3 exception Bad_literal of int
4 exception Exn_unsat
```

Les exceptions déclarées ci-dessus seront utilisées dans les fonctions de cette partie et de la partie suivante.

Question 3. Écrire les fonctions suivantes :

- `var_of_lit: int -> int` renvoie l'index de la variable apparaissant dans le littéral représenté par l'argument i. Par exemple, pour $n = 5$, la fonction renvoie 3 si l'argument est 3, et 2 si l'argument est 7. Si $i \notin [1; 2n]$ alors la fonction exécute `raise (Bad_literal i)`.
- `neg_lit: int -> int` renvoie, pour un argument i, l'entier j tel que $lit(j)$ est logiquement équivalent à $\neg lit(i)$; si $i \notin [1; 2n]$ alors la fonction exécute `raise (Bad_literal i)`.
- `eform1_is_wf: eform1 -> bool` renvoie `true` si et seulement si l'argument est « bien formé », c'est-à-dire qu'il contient uniquement des entiers qui représentent des littéraux sur l'ensemble de variables $\mathbb{X}$.

⌟

Dans la suite de ce sujet, on suppose que tous les entiers utilisés par les valeurs de type `contr1` ***et*** `eform1` ***représentent des littéraux sur l'ensemble de variables*** $\mathbb{X}$***.*** Par conséquent, l'exception `Bad_literal` ne doit pas être levée.

Graphe d'une formule. Le ***graphe associé*** à $F \in \mathcal{F}(L_1)$, noté $G_F = (V_F, E_F)$, est un graphe non orienté dont les sommets V_F sont les entiers dans l'intervalle $[1; 2n]$. Les arêtes E_F sont les ensembles $\{i, j\}$ tels que $i, j \in V_F$, $i \neq j$ et F contient soit la contrainte $\mathit{lit}(i) \leftrightarrow \mathit{lit}(j)$, soit la contrainte $\mathit{lit}(i') \leftrightarrow \mathit{lit}(j')$ avec i' et j' égaux respectivement à `neg_lit`(i) et `neg_lit`(j). Par exemple, pour $n = 5$, la contrainte $(X_3 \leftrightarrow X_4)$ est représentée par les arêtes $\{3, 4\}$ et $\{8, 9\}$. On note que les contraintes de F de la forme $\ell \leftrightarrow \ell$ ne sont pas représentées par E_F.

Question 4. On suppose $n = 5$. Pour chacune des formules suivantes, donner le graphe associé :

$$F_1 \stackrel{\text{def}}{=} (X_3 \leftrightarrow X_4) \wedge (X_4 \leftrightarrow X_2) \wedge (X_3 \leftrightarrow \neg X_2) \wedge (X_5 \leftrightarrow X_2)$$
$$F_2 \stackrel{\text{def}}{=} (\neg X_4 \leftrightarrow X_2) \wedge (X_2 \leftrightarrow \neg X_3) \wedge (X_4 \leftrightarrow \neg X_5) \wedge (X_3 \leftrightarrow X_5)$$

⌟

On suppose disposer d'une bibliothèque munie d'un type `graph` et qui code des graphes non orientés. Les sommets sont représentés par des entiers. Les opérations suivantes sont disponibles pour la construction et l'exploration des graphes :

— `g_create : int -> graph` renvoie une valeur représentant un graphe sans arêtes et ayant k sommets, où k est donné en argument, supposé strictement positif. Les sommets sont numérotés de 1 à k. La complexité de cette opération est en $O(k)$.

— `g_add_edge : graph -> int -> int -> unit` ajoute au graphe donné comme premier argument l'arête entre les sommets donnés en 2ème et 3ème arguments. Si les sommets en argument ne sont pas déclarés dans le graphe ou si l'arête existe déjà, alors cette fonction ne change pas le graphe. Sinon, le graphe est modifié en place. La complexité de cette opération est en $O(k)$, avec k le nombre de sommets du graphe.

— `g_nb_vertex : graph -> int` renvoie le nombre de sommets du graphe en argument. La complexité de cette opération est en $O(1)$.

— `g_succ : graph -> int -> int list` renvoie la liste des sommets adjacents du sommet en second argument dans le graphe donné en premier argument. Si le sommet en argument n'existe pas, la liste renvoyée est vide. La complexité de cette opération est en $O(1)$.

Question 5. Écrire une fonction `eform1_to_graph: eform1 -> graph` qui renvoie le graphe associé à la formule F représentée par l'argument. Si F contient une contrainte de la forme $\ell \leftrightarrow \neg\ell$, alors la fonction lève l'exception **`Exn_unsat`**. ⌟

Lorsqu'un appel à `eform1_to_graph lst` lève l'exception **`Exn_unsat`**, la formule représentée par `lst` contient toujours une contrainte contradictoire. Cependant, une formule peut être une contradiction sans que l'exception soit levée, comme c'est le cas de la formule $(X_1 \leftrightarrow X_2) \wedge (X_2 \leftrightarrow \neg X_1)$.

Question 6. Donner et justifier soigneusement la complexité de `eform1_to_graph` en fonction de n et de la longueur p de la liste en argument. ⌟

Soient $F \in \mathcal{F}(L_1)$ une formule et `g` la représentation de G_F, le graphe associé à F. On note par $CC(G_F)$ l'ensemble des ensembles de sommets des composantes connexes de G_F. Soit m la taille de $CC(G_F)$, c'est-à-dire, le nombre de composantes connexes de G_F. On représente $CC(G_F)$ par un tableau `c` de $2n + 1$ entiers, qui associe à chaque `i` $\in [1; 2n]$, sommet de `g`, un entier dans l'intervalle $[1; m]$. Ainsi, `c.(i)` identifie de manière unique la composante connexe du sommet `i`. La case de `c` à la position 0 ne sera pas utilisée, mais elle permet d'utiliser directement les indices des variables comme positions de `c`. Par convention, `c.(0)` est 0.

Question 7. Écrire une fonction `g_cc_array: graph -> (int array * int)` qui renvoie le couple $(\mathtt{c}, m)$, où `c` est le tableau des composantes connexes du graphe en argument et où m est le nombre de composantes connexes. ⌟

Question 8. Démontrer que si i et j sont des sommets distincts appartenant à la même composante connexe de G_F, alors $F \models \mathit{lit}(i) \leftrightarrow \mathit{lit}(j)$. ⌟

Question 9. Soit $C = \{i_1, \ldots, i_q\} \in CC(G_F)$ l'ensemble des sommets d'une composante connexe de G_F. Soit C' l'ensemble de sommets $\{j_1, \ldots, j_q\}$ avec $j_k = \mathtt{neg_lit}(i_k)$ pour tout $k \in [1; q]$. Démontrer que C' est élément de $CC(G_F)$. ⌟

Question 10. Démontrer que F est une contradiction si et seulement s'il existe une composante connexe C de G_F et un sommet i de C tels que $\mathtt{neg_lit}(i)$ est un sommet de C. ⌟

Question 11. Écrire une fonction `cc_is_sat: int array -> bool` qui a comme argument `c`, le tableau représentant les composantes connexes de G_F et qui renvoie `true` si et seulement si F est satisfiable. ⌟

Représentation d'une valuation. Une valuation v de $\mathbb{X}$ est représentée par un tableau `v` de $n+1$ valeurs 0 ou 1 tel que `v.(`i`)` est 1 si et seulement si $v(X_i) = 1$. En particulier, `v.(1)=1` par convention sur X_1. La case à la position `0` n'est pas utilisée, mais elle permet d'utiliser les indices des variables comme positions du tableau ; par convention, `v.(0)` est 0.

Question 12. Écrire une fonction `eform1_sat: eform1 -> (int array) option` qui a comme argument une liste `lst` représentant une formule $F \in \mathcal{F}(L_1)$. L'appel `eform1_sat lst` renvoie `None` si F est une contradiction. Si F est satisfiable, l'appel renvoie `Some(v)` avec `v` un tableau qui représente une valuation qui satisfait F. La complexité en temps de l'algorithme doit être en $O(p \times n)$ où p est la longueur de la liste `lst` en argument. Justifier que votre algorithme a cette complexité en temps. ⌟

Partie II. Équivalences avec une conjonction de deux littéraux

On reste dans le cadre de la logique propositionnelle, mais on considère une nouvelle classe de contraintes. La classe L_2 contient les contraintes de la forme $\ell \leftrightarrow (\ell' \wedge \ell'')$ où ℓ, ℓ', ℓ'' sont des littéraux. Pour identifier les contraintes des classes L_1 et L_2, on appelle ***i*-contrainte** une contrainte de la classe L_i avec $i \in \{1, 2\}$. Une formule F est dite construite sur L_2, notation $F \in \mathcal{F}(L_2)$, si elle est une conjonction de 2-contraintes. ***On garde l'hypothèse que*** X_1 ***est toujours interprétée à*** 1. Un exemple de formule de $\mathcal{F}(L_2)$ est :

$$F_3 \quad \stackrel{\text{def}}{=} \quad (X_1 \leftrightarrow (X_2 \wedge \neg X_4)) \wedge (\neg X_2 \leftrightarrow (X_2 \wedge X_3)) \wedge (X_4 \leftrightarrow (\neg X_1 \wedge X_5))$$

Question 13. Donner une réduction en temps polynomial du problème P_{L_2} au problème de satisfiabilité des formules en 3-CNF. ⌟

Question 14. Donner une dérivation formelle (preuve en déduction naturelle) du séquent suivant :

$$X_2 \to (\neg X_2 \wedge X_3), (\neg X_2 \wedge X_3) \to X_2 \quad \vdash \quad \neg X_2.$$

Les règles de la déduction naturelle à utiliser dans cette preuve sont rappelées dans l'annexe. ⌟

En utilisant la convention sur l'interprétation de X_1, toute 1-contrainte $\ell \leftrightarrow \ell'$ est logiquement équivalente à la 2-contrainte $\ell \leftrightarrow (\ell' \wedge X_1)$. À l'inverse, certaines 2-contraintes sont logiquement équivalentes à des formules de $\mathcal{F}(L_1)$. Par exemple, les équivalences logiques suivantes sont vraies pour tout couple de littéraux ℓ et ℓ' (les preuves ne sont pas demandées) :

$\ell \leftrightarrow (X_1 \wedge \ell')$ équivalent à $\ell \leftrightarrow \ell'$	par 1 élément neutre de $\wedge$	(1)
$\ell \leftrightarrow (\neg X_1 \wedge \ell')$ équivalent à $\ell \leftrightarrow \neg X_1$	par 0 élément absorbant de $\wedge$	(2)
$\ell \leftrightarrow (\ell' \wedge \ell')$ équivalent à $\ell \leftrightarrow \ell'$	par idempotence de $\wedge$	(3)
$\ell \leftrightarrow (\ell' \wedge \neg \ell')$ équivalent à $\ell \leftrightarrow \neg X_1$	par contradiction	(4)
$X_1 \leftrightarrow (\ell \wedge \ell')$ équivalent à $(X_1 \leftrightarrow \ell) \wedge (X_1 \leftrightarrow \ell')$	par définition de $\wedge$	(5)
$\ell \leftrightarrow (\neg \ell \wedge \ell')$ équivalent à $(\neg X_1 \leftrightarrow \ell) \wedge (\neg X_1 \leftrightarrow \ell')$	admis	(6)

Dans la suite de cette partie, on dénote par $\mathbb{E}(\mathbb{X})$ toutes les équivalences définies sur l'ensemble des littéraux par les équivalences de (1) à (6) et par leur variantes obtenues en utilisant la symétrie de la conjonction. Par exemple, « $\ell \leftrightarrow (\ell' \wedge X_1)$ équivalent à $\ell \leftrightarrow \ell'$ » est également une équivalence de $\mathbb{E}(\mathbb{X})$, grâce à la symétrie de la conjonction appliquée à (1).

Mise en œuvre en OCaml. Une 2-contrainte $\ell \leftrightarrow (\ell' \wedge \ell'')$ est représentée par un enregistrement avec trois champs entiers. Les entiers de ces enregistrements prennent des valeurs dans $[1; 2n]$ et représentent des littéraux avec la même convention de codage qu'en première partie. Une formule dans $\mathcal{F}(L_2)$ est représentée par une liste d'enregistrements ; la liste vide représente $\top$. Ce codage est donné par les types OCaml suivants :

```
1 type contr2 = { ll: int; lr1: int; lr2: int } (* contraintes L2 *)
2 type eform2 = contr2 list                     (* formules L2 *)
```

Une valeur `e` de type `contr2` représente la 2-contrainte `e.ll` $\leftrightarrow$ (`e.lr1` $\wedge$ `e.lr2`).

Dans la suite de cette partie, on suppose que toutes les valeurs des types `contr`i et `eform`i avec $i \in \{1, 2\}$ sont bien formées, c'est-à-dire que les entiers qu'elles contiennent représentent des littéraux sur $\mathbb{X}$.

Question 15. Écrire la fonction `contr2_simplify_eq5: contr2 -> eform1 option` qui applique l'équivalence (5) pour simplifier la 2-contrainte en argument. L'appel `contr2_simplify_eq5 e2` :

— renvoie **`None`** si la simplification ne peut pas être faite avec (5) ;

— lève l'exception **`Exn_unsat`** (déclarée en partie I) si la simplification peut être faite mais la 2-contrainte représentée par `e2` est une contradiction ;

— enfin, renvoie **`Some`** `[(1,e2.lr1);(1,e2.lr2)]` si la simplification peut être effectuée sans détecter de contradiction. ⌟

Valuation partielle. Une valuation partielle est une fonction $v : \mathbb{X} \to \{-1, 0, 1\}$ telle que $v(X_1) = 1$. Pour tout $i \neq 1$, si $v(X_i) = b \in \mathbb{B}$ alors on dit que v assigne X_i à b. Si $v(X_i) = -1$ alors on dit que X_i n'est pas assignée par v. On étend la notion de valuation partielle aux littéraux et aux formules comme suit :

— $v(\neg X_i) = \neg b$ si $v(X_i) = b$ avec $b \in \mathbb{B}$, et $v(\neg X_i) = -1$ si $v(X_i) = -1$.

— Soit la 1-contrainte $\ell \leftrightarrow \ell'$. Si $v(\ell), v(\ell') \in \mathbb{B}$ alors $v(\ell \leftrightarrow \ell') = 1$ si $v(\ell) = v(\ell')$ et 0 sinon. Si au moins une des valeurs $v(\ell)$ ou $v(\ell')$ est -1, alors $v(\ell \leftrightarrow \ell') = -1$.

— Pour la conjonction de deux littéraux $\ell \wedge \ell'$, $v(\ell \wedge \ell') = -1$ si au moins une des valeurs $v(\ell)$ ou $v(\ell')$ est -1 ; sinon la valeur de $v(\ell \wedge \ell')$ est 1 si et seulement si $v(\ell) = 1$ et $v(\ell') = 1$.

— Soit la 2-contrainte $\ell \leftrightarrow (\ell' \wedge \ell'')$. Si $v(\ell), v(\ell' \wedge \ell'') \in \mathbb{B}$ alors $v(\ell \leftrightarrow (\ell' \wedge \ell'')) = 1$ si $v(\ell) = v(\ell' \wedge \ell'')$ et 0 sinon. Si au moins une des valeurs $v(\ell)$ ou $v(\ell' \wedge \ell'')$ est -1, alors $v(\ell \leftrightarrow (\ell' \wedge \ell'')) = -1$.

La ***valeur d'une formule*** $F_i \in \mathcal{F}(L_i)$ où $i \in \{1, 2\}$ pour une valuation partielle v, notation $v(F_i)$, est -1 s'il existe une contrainte e de F_i telle que $v(e) = -1$; sinon, $v(F_i) = 1$ si pour toute contrainte e de F_i on a $v(e) = 1$, et $v(F_i) = 0$ sinon.

Une valuation partielle v est ***compatible*** avec une valuation v' si pour tout $X_i \in \mathbb{X}$ tel que $v(X_i) \in \mathbb{B}$ on a $v'(X_i) = v(X_i)$.

Une valuation partielle v est représentée en OCaml par un tableau `v` de taille $n + 1$ dont les éléments appartiennent à $\{-1, 0, 1\}$. Comme pour les valuations, la case de position `0` n'est pas utilisée et, par convention, `v.(0)=0`.

Question 16. Écrire la fonction `eform1_of_valp: int array -> eform1` qui renvoie une formule F de $\mathcal{F}(L_1)$ telle que F encode les assignations de la valuation partielle `v` en argument. Ainsi, un appel à `eform1_of_valp v` renvoie la liste de couples (i, j) tels que $i > 0$, `v.(`i`)` $\neq -1$ et si `v.(`i`)` $= 1$ alors $j = 1$ et si `v.(`i`)` $= 0$ alors $j = n + 1$.

Soit F la formule représentée par le résultat de l'appel à `eform1_of_valp v`. Démontrez que, pour toute valuation v' on a $v' \models F$ si et seulement si `v` est compatible avec v'. ⌟

Une 2-contrainte peut être simplifiée en présence d'une valuation partielle v en une formule de $\mathcal{F}(L_1)$. Par exemple, si $v(X_2) = 1$ alors la 2-contrainte $X_2 \leftrightarrow (\ell' \wedge \ell'')$ a la même valeur que la 1-contrainte $(X_1 \leftrightarrow \ell') \wedge (X_1 \leftrightarrow \ell'')$ pour la valuation partielle v. On suppose donnée une fonction `contr2_simplify_valp: contr2 -> int array -> eform1 option` qui applique les équivalences de $\mathbb{E}(\mathbb{X})$ pour simplifier la 2-contrainte `e2` en premier argument en tenant compte de

la valuation partielle `vp` donnée comme second argument. Plus précisement, si `vp(e2)=0`, alors la fonction lève l'exception **`Exn_unsat`**. Si aucune simplification ne s'applique, alors le résultat est **`None`**. Si `e2` est simplifiable, le résultat est **`Some`**`(lst)` avec `lst` une liste représentant une formule F_1' de $\mathcal{F}(L_1)$ obtenue par simplification de `e2` telle que `vp(e2) = vp(`F_1'`)`. La complexité de `contr2_simplify_valp` est en $O(1)$.

Algorithme. Les questions suivantes vous guident pour prouver la correction et la terminaison de l'algorithme de décision pour P_{L_2} qui est mis en œuvre dans la fonction `eform2_sat` de la figure 1.

On cherche à démontrer que la fonction `eform2_sat` est correcte, c'est-à-dire qu'un appel à `eform2_sat lst0` doit renvoyer **`None`** si la formule $F_0 \in \mathcal{F}(L_2)$ représentée par `lst0` est une contradiction. Sinon, l'appel doit renvoyer **`Some`**`(v)` où `v` est un tableau qui représente une valuation qui satisfait F_0. L'algorithme mis en œuvre pour `eform2_sat` est un algorithme de type retour sur trace (***backtracking***) dont le principe est de réduire la formule $F_0 \in \mathcal{F}(L_2)$ en une formule de $\mathcal{F}(L_1)$ afin d'appliquer `eform1_sat`, l'algorithme polynomial de la partie I. Pour effectuer cette réduction, `eform2_sat` appelle à la ligne 5 la fonction récursive `eform2_sat_aux` dont les arguments sont une liste `l2` représentant une formule $F_2 \in \mathcal{F}(L_2)$, une liste `l1` représentant une formule $F_1 \in \mathcal{F}(L_1)$ et une valuation partielle `vp`.

La correction de `eform2_sat_aux` est un lemme important de la preuve de correction de `eform2_sat`. Un appel à `eform2_sat_aux l2 l1 vp` est correct s'il renvoie un tableau qui représente une valuation v telle que $v \models F_2 \wedge F_1$ et `vp` est compatible avec v, si une telle valuation existe ; sinon, l'appel doit lever l'exception **`Exn_unsat`**.

La mise en œuvre de `eform2_sat_aux` cherche à compléter la valuation partielle `vp` en assignant des valeurs à certaines variables (non assignées) qui apparaissent dans les littéraux de F_2 et qui permettent de simplifier F_2 en une formule de $\mathcal{F}(L_1)$. Pour savoir si une 2-contrainte `e2` de `l2` peut être simplifiée en présence de `vp`, on appelle `contr2_simplify_valp e2 vp` (ligne 15). Si `e2` se simplifie, l'argument `l1` accumule les formules de $\mathcal{F}(L_1)$ obtenues par la simplification et `eform2_sat_aux` continue la simplification du reste des contraintes de `l2` (ligne 17). Si la contrainte `e2` est une contradiction en présence de `vp`, l'exception **`Exn_unsat`** est levée par `contr2_simplify_valp e2 vp`. Si `e2` ne peut pas être simplifiée (ligne 19), alors `eform2_sat_aux` explore les simplifications de la contrainte `e2` obtenues en assignant la variable du littéral `e2.lr1` à chaque valeur de $\mathbb{B}$. L'exploration d'une assignation est faite grâce à un appel récursif à `eform2_sat_aux` avec le paramètre représentant la valuation partielle mis à jour avec l'assignation choisie.

Quand toutes les 2-contraintes de F_2 ont été simplifiées, c'est-à-dire quand la liste `l2` est vide (ligne 10), `eform2_sat_aux` appelle `eform1_sat` avec l'argument `(l1'@l1)` où la liste `l1'` est obtenue par l'appel à `eform1_of_valp vp`. Si l'appel à `eform1_sat (l1'@l1)` renvoie **`Some`**`(v)`, alors le résultat de `eform2_sat_aux` est `v`. Sinon, `eform2_sat_aux` lève l'exception **`Exn_unsat`**. Ceci a comme effet de revenir dans un appel précédent (sur la pile d'appels) de `eform2_sat_aux` pour explorer une autre assignation (lignes 25–28) ou, si toutes les assignations ont été explorées, de terminer l'appel à `eform2_sat_aux l2 l1 vp` par l'exception **`Exn_unsat`**.

Question 17. Démontrer que tout appel `eform2_sat lst0` termine. ⌟

Question 18. Démontrer que la fonction `eform2_sat` est correcte en supposant que la fonction `eform2_sat_aux` est correcte. ⌟

Dans les trois questions suivantes, on cherche à démontrer les principaux cas de la correction de `eform2_sat_aux`.

```
1  let rec eform2_sat (lst:eform2) : (int array) option =
2    try
3      let vinit = Array.make (n+1) (-1) in (
4        vinit.(0) <- 0; vinit.(1) <- 1;
5        Some (eform2_sat_aux lst [] vinit) )
6    with Exn_unsat -> None
7
8  and eform2_sat_aux (l2:eform2) (l1:eform1) (vp:int array) : int array =
9    match l2 with
10   | [] -> let l1' = eform1_of_valp vp in
11           let  r  = eform1_sat (l1'@l1) in
12           (match r with None -> raise Exn_unsat | Some v -> v)
13
14   | e2::l2' ->
15     match contr2_simplify_valp e2 vp with
16     | Some l1' -> (* simplification *)
17         eform2_sat_aux l2' (l1'@l1) vp
18
19     | None -> (* branchement sur valeur de e2.lr1 *)
20       let xlr1 = var_of_lit e2.lr1 in
21       try
22         let vp1 = Array.copy vp in (
23           vp1.(xlr1) <- 1;
24           eform2_sat_aux l2 l1 vp1 )
25       with Exn_unsat ->
26         let vp0 = Array.copy vp in (
27           vp0.(xlr1) <- 0;
28           eform2_sat_aux l2 l1 vp0 )
```

FIGURE 1 – Mise en œuvre OCaml de l'algorithme de décision pour P_{L_2}

Question 19. Démontrez la correction de `eform2_sat_aux` quand son premier argument est la liste vide, on supposant que les fonctions `eform1_of_valp` et `eform1_sat` sont correctes. ⌟

Question 20. Soit un appel `eform2_sat_aux l2 l1 vp` tel que `l2` est `e2::l2'` et `contr2_simplify_vp e2 vp` renvoie `Some l1'`. Démontrez la correction d'un tel appel en supposant que l'appel récursif à `eform2_sat_aux` de la ligne 17 est correct. ⌟

Question 21. Soit un appel `eform2_sat_aux l2 l1 vp` tel que `l2` est `e2::l2'` et `contr2_simplify_vp e2 vp` renvoie `None`. Démontrez la correction d'un tel appel en supposant que les appels récursifs aux lignes 24 et 28 sont corrects. ⌟

Question 22. Donner la complexité de `eform2_sat` en fonction de la longueur k de la liste en argument et du nombre de variables n. ⌟

Partie III. Égalités et différences entre entiers

Dans cette partie, le domaine d'interprétation des variables est $\mathbb{Z}$**.** La classe L_3 est constituée de contraintes de la forme $X_i = X_j$ et $X_i \neq X_j$, avec $X_i, X_j \in \mathbb{X}$. Une formule F est dite construite sur L_3, notation $F \in \mathcal{F}(L_3)$, si elle est une conjonction de contraintes de L_3. Une valuation est une fonction $v : \mathbb{X} \to \mathbb{Z}$ qui assigne une valeur dans $\mathbb{Z}$ à chaque variable. Une valuation v ***satisfait*** une contrainte $X_i = X_j$ (resp. $X_i \neq X_j$) si et seulement si $v(X_i) = v(X_j)$ (resp. $v(X_i) \neq v(X_j)$). Une valuation v ***satisfait*** une formule $F \in \mathcal{F}(L_3)$ si et seulement si v satisfait chaque contrainte de F. Les notions de conséquence logique, équivalence logique, tautologie et antilogie sont définies de manière similaire à la logique propositionnelle.

Le problème P_{L_3} peut être réduit (en temps polynomial en n et en la taille de la formule) au problème de satisfiabilité d'une formule propositionnelle. Dans cette partie, on étudie un algorithme polynomial en n et en la taille de la formule pour P_{L_3}.

Mise en œuvre en langage C. On propose de représenter les formules de $\mathcal{F}(L_3)$ par des listes simplement chaînées acycliques de type `eform3_t*`. Chaque cellule `c` (de type `eform3_t`) d'une telle liste représente une contrainte $e \in L_3$ grâce à trois données : un booléen `c.is_eq` qui est vrai si et seulement si e est une égalité, et deux entiers positifs `c.xi` et `c.xj` qui représentent les indices des variables utilisées par e. La formule $\top$ est représentée par la liste vide de valeur `NULL`. On suppose que le nombre n de variables dans $\mathbb{X}$ est déclaré comme une variable globale entière. Enfin, on suppose que toute structure `eform3_t` est bien formée, c'est-à-dire que tous les entiers qu'elle contient appartiennent à l'intervalle $[1; n]$.

Question 23. Donner la déclaration du type `eform3_t`. ⌟

Question 24. Écrire la fonction `void eform3_print(eform3_t *lst)` qui affiche sur la sortie standard chaque contrainte (par exemple, sous la forme `X_1 != X_4`) de la formule représentée par `lst`, une contrainte par ligne ; si `lst` est `NULL`, la fonction affiche `true` sur une ligne. ⌟

Classes d'équivalence avec « unir et trouver ». On utilise une structure unir et trouver (***union-find***) pour résoudre le problème P_{L_3}. On suppose disposer d'une bibliothèque C qui fournit une implémentation d'une structure unir et trouver sur des ensembles d'entiers. Cette bibliothèque exporte les déclarations de types et de fonctions suivantes :

```
1  /* Rappel: definition dans stdlib.h
2  typedef unsigned int size_t; */
3
4  struct uf_s {       /* type unir et trouver (union-find) */
5      size_t nelem;   /* nombre d'elements de l'ensemble */
6      int   *parent;  /* tableau de taille nelem = relation parent */
7      ...             /* champs utiles pour operations en O(log n) */
8  };
9  typedef struct uf_s uf_t;
10 uf_t *uf_create(size_t sz);
11 int   uf_find(uf_t *cs, int i);
12 void  uf_union(uf_t *cs, int i1, int i2);
```

Dans ce qui suit, on supposera les propriétés suivantes de cette implémentation :

— `uf_create(s)` renvoie un pointeur `cs` vers une structure unir et trouver allouée dans le tas et qui représente une partition de l'ensemble $\{0,\ldots,\mathtt{s}-1\}$ où chaque élément de l'ensemble est seul dans sa classe d'équivalence ; `s` donne la valeur du champ `cs->nelem`.

— `uf_find(cs,i)` renvoie la classe de `i`, c'est-à-dire un entier entre 0 et `cs->nelem-1` qui est le représentant de cette classe.

— `uf_union(cs,i1,i2)` fusionne les classes de `i1` et `i2` en modifiant la structure `*cs`.

Si `cs` est un pointeur valide vers une structure `uf_t` avec `cs->nelem==s`, le champ `cs->parent` pointe vers un tableau de `s` entiers ayant des valeurs dans l'intervalle $[0;\mathtt{s}-1]$; ce tableau encode la relation représentant d'une classe d'équivalence. Si `cs->parent[i]==j` alors `i` est dans la classe de `j`. Le tableau `cs->parent` est initialisé par `uf_create` de sorte que `cs->parent[i]==i`. Les opérations ci-dessus sur la structure unir et trouver ont une complexité en $\Theta(\mathtt{s})$ pour `uf_create` et en $O(\log \mathtt{s})$ pour `uf_find` et `uf_union`.

Pour notre problème, ***l'ensemble de valeurs à partitionner sera*** $[0;n]$, c'est-à-dire l'ensemble d'indices de variables dans $\mathbb{X}$ auquel on ajoute la valeur 0. La valeur 0 ne sera pas utilisée, mais elle permet d'utiliser directement les indices des variables comme positions du tableau `cs->parent`.

Question 25. Écrire la fonction `uf_t *eform3_to_uf(eform3_t *lst)` qui renvoie une valeur de type `uf_t*` représentant les classes d'équivalence sur $[0;n]$ définies par les ***contraintes d'égalité*** de la formule représentée par `lst`. On suppose que `lst` n'est pas vide. Les contraintes de différence de `lst` seront ignorées dans cette fonction. ⌟

Algorithme de décision. D'après la question précédente, la structure unir et trouver permet donc de représenter les contraintes d'égalité d'une formule de $\mathcal{F}(L_3)$, mais pas les contraintes de différence.

Question 26. Écrire une fonction `bool eform3_sat_int(eform3_t *lst)` qui renvoie `true` si et seulement si toutes les contraintes de différence de la formule représentée par `lst` peuvent être satisfaites en respectant les contraintes d'égalité de `lst`. Si `lst` est `NULL`, la fonction renvoie `true`. Justifier la correction de la fonction `eform3_sat_int`, c'est-à-dire, que tout appel à cette fonction renvoie `true` si et seulement si la formule représentée par son argument est satisfiable. ⌟

Question 27. On suppose que `lst` (de type `eform3_t*`) représente une formule avec un nombre de contraintes d'égalité égal à e et un nombre de contraintes de différence égal à d. Donner la complexité de l'appel à `eform3_sat_int(lst)` en fonction de n, e et d. ⌟

Calculer une valuation. Dans la suite de cette partie, on se propose de calculer, si elle existe, une valuation $v : \mathbb{X} \to \mathbb{Z}$ qui satisfait une formule de $\mathcal{F}(L_3)$.

Question 28. Écrire une fonction `int uf_classes(uf_t *cs)` qui renvoie le nombre de classes d'équivalence de la structure unir et trouver pointée par `cs` ; si `cs` est `NULL` alors la fonction renvoie 0. Cette fonction doit avoir une complexité en $O(n)$. ⌟

Question 29. Écrire une fonction `int *uf_valuation(uf_t *cs)` qui renvoie un pointeur `v` vers un tableau de $n+1$ entiers, alloué dans le tas. Soit k le résultat de l'appel de `uf_classes(cs)`. On suppose $k \geq 2$. Pour tout $i \in [1;n]$, `v[`i`]` est une valeur entière dans l'intervalle $[1;k]$ qui

est associée ***de façon unique*** à (chaque élément de) la classe d'équivalence de i. La case à la position 0 de `v` n'est pas utilisée, mais elle permet d'utiliser les indices des variables comme positions de `v` ; par convention, `v[0]==0`. La fonction `uf_valuation` doit avoir une complexité en $O(n \log n)$. ⌟

Soit $F \in \mathcal{F}(L_3)$ une formule représentée par une liste `lst` différente de `NULL` telle que `eform3_sat_int(lst)` renvoie `true`. Soient `cs` le résultat de l'appel à `eform3_to_uf(lst)` et `v` le résultat de l'appel à `uf_valuation(cs)`.

Question 30. Démontrer que `v` pointe vers un tableau qui représente une valuation qui satisfait la formule F représentée par `lst`. ⌟

Question 31. Si F ne contient pas de contraintes de différence, alors quel est le nombre minimal de valeurs que peut avoir l'image d'une valuation v qui satisfait F ? Donner un exemple d'une telle valuation.

La même question si F contient une seule contrainte de différence. ⌟

Question 32. Lorsque cela est possible, on souhaite construire une valuation v telle que v satisfait la formule F et l'image de v est un ensemble de taille 2. Quel algorithme sur les graphes peut être utilisé pour obtenir ce résultat et sur quel graphe ? Vous devez définir formellement le graphe en entrée de l'algorithme et donner la complexité de l'algorithme. ⌟

Partie IV. Égalités et différences d'expressions

Soit $\mathbb{F}$ un ensemble de m symboles de fonction notés f_i $(1 \leq i \leq m)$. Chaque symbole $f_i \in \mathbb{F}$ représente une fonction de $\mathbb{Z} \times \mathbb{Z}$ dans $\mathbb{Z}$. Comme dans la partie III, on interprète les variables X_i de $\mathbb{X}$ dans l'ensemble $\mathbb{Z}$. Les symboles de fonctions de $\mathbb{F}$ ***ne sont pas interprétés***, c'est-à-dire qu'ils n'ont pas de définition. Les seules propriétés connues de ces fonctions sont données sous la forme de contraintes d'égalité ou de différence. Ces contraintes sont celles de la classe L_4 définie ci-dessous.

Les contraintes de la classe L_4 utilisent des termes. Un ***terme*** t est défini de manière inductive : t est

— soit X_i, une variable de $\mathbb{X}$,

— soit $f_i(t_1, t_2)$, l'application d'une fonction $f_i \in \mathbb{F}$ sur deux termes t_1 et t_2, appelés aussi ***sous-termes directs*** de t.

Le ***symbole racine*** du terme X_i (resp. $f_j(t_1, t_2)$) est X_i (resp. f_j). On note $\mathbb{T}(\mathbb{X}, \mathbb{F})$ l'ensemble des termes construits avec les variables de $\mathbb{X}$ et les symboles de fonctions $\mathbb{F}$.

Ces termes peuvent être représentés par des arbres binaires. La figure 2 représente les arbres binaires correspondants aux termes :

$$t_1 \stackrel{\text{def}}{=} f_1(f_2(X_1, X_2), X_2) \quad \text{et} \quad t_2 \stackrel{\text{def}}{=} f_3(X_1, f_2(X_1, X_2)).$$

Chaque noeud de ces arbres est étiqueté en partie haute par un nom unique u_i appelé identifiant, et en partie basse par le symbole racine du terme. L'arc à gauche (resp. droite) d'un noeud correspond au premier (resp. second) sous-terme direct et est dessiné avec un trait en pointillé (resp. plein).

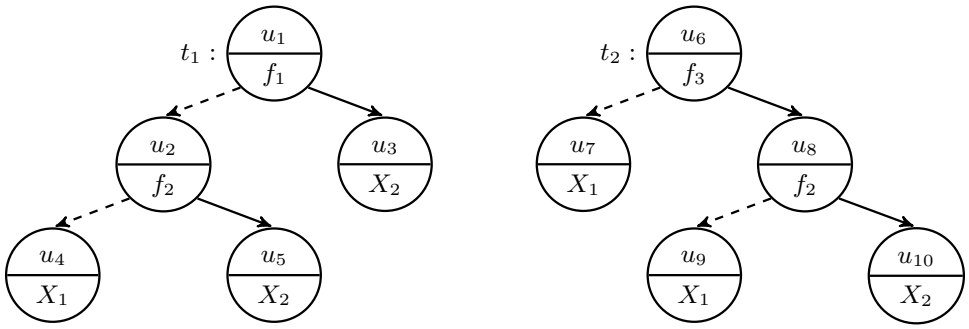


FIGURE 2 – Arbres des termes t_1 (à gauche) et t_2 (à droite)

L'ensemble des sous-termes d'un terme t, noté $\mathcal{T}(t)$, est défini de manière inductive par

$$\mathcal{T}(X_i) \stackrel{\text{def}}{=} \{X_i\} \quad \text{et} \quad \mathcal{T}(f_i(t_1, t_2)) \stackrel{\text{def}}{=} \{f_i(t_1, t_2)\} \cup \mathcal{T}(t_1) \cup \mathcal{T}(t_2).$$

Question 33. Donner $\mathcal{T}(t_1)$ et $\mathcal{T}(t_2)$ pour les termes t_1 et t_2 définis ci-dessus. ⌟

La classe L_4 contient les contraintes de la forme $t_1 = t_2$ ou $t_1 \neq t_2$ avec t_1 et t_2 des termes. Une formule F de $\mathcal{F}(L_4)$ est une conjonction de contraintes L_4. L'ensemble des sous-termes d'une contrainte e de la forme $t_1 = t_2$ ou $t_1 \neq t_2$ est $\mathcal{T}(e) \stackrel{\text{def}}{=} \mathcal{T}(t_1) \cup \mathcal{T}(t_2)$. L'ensemble des sous-termes d'une formule F de $\mathcal{F}(L_4)$, noté $\mathcal{T}(F)$, est défini comme l'union des ensembles des sous-termes de toutes les contraintes de F.

Mise en œuvre en langage C. Afin de concevoir un algorithme de décision pour $P(L_4)$, nous représentons ***les contraintes d'égalité*** d'une formule de $\mathcal{F}(L_4)$ par une structure de données appelée ***e-graphe***. Cette structure combine un graphe orienté acyclique avec une structure unir et trouver. Les définitions de types C suivantes codent une structure de e-graphe :

```
1  struct tnode_s {  /* type des noeuds des arbres representant les termes */
2      char *label;  /* symbole racine */
3      int   li;     /* position dans tset du premier sous-terme direct ou 0 */
4      int   ri;     /* position dans tset du second sous-terme direct ou 0 */
5  };
6  typedef struct tnode_s tnode_t;
7
8  struct egraph_s {     /* type e-graphe representant une formule L4 */
9      size_t   tset_sz; /* longueur de tset */
10     tnode_t *tset;    /* debut tableau des termes de la formule */
11     int     *pre;     /* relation sous-terme -- terme */
12     uf_t    *uf;      /* unir et trouver pour les classes des termes */
13 };
14 typedef struct egraph_s egraph_t;
```

Dans ce qui suit, on supposera les propriétés suivantes de la mise en œuvre de la structure `egraph_t`. Si `g` est une variable de type `egraph_t` représentant les contraintes d'égalité d'une formule $F \in \mathcal{F}(L_4)$, `g.tset` code l'ensemble des sous-termes $\mathcal{T}(F)$ comme suit :

- `g.tset` est un pointeur vers le debut d'un tableau de taille `g.tset_sz` ;
- `g.tset[0]` est initialisée à la valeur `{.label=NULL, .li=0, .ri=0}` (premier champ à `NULL` et les suivants à 0) de `tnode_t`, valeur qu'on appelle aussi ***terme vide*** ;
- un terme est représenté par une valeur `t` de type `tnode_t` tel que `t.label` est une chaîne de caractères qui décrit le symbole racine du terme, et `t.li` (resp. `t.ri`) est la position du premier (resp. second) sous-terme direct dans le tableau `g.tset`.

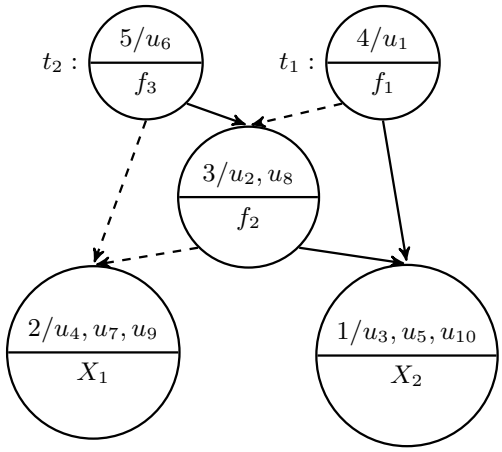
Un invariant du tableau `g.tset` est le suivant :

> ***tout terme de $\mathcal{T}(F)$ est représenté par une unique position du tableau*** `g.tset`.

Cet invariant correspond à une factorisation des termes communs et donc à la transformation de l'ensemble des arbres correspondant aux termes en un graphe orienté acyclique.

Par exemple, considérons la formule $t_1 = t_2$ utilisant les termes représentés dans la figure 2. La partie droite de la figure 3 représente le tableau `g.tset` de l'ensemble des sous-termes de la formule (une valeur `tnode_t` par ligne) et le terme vide. La partie gauche de la figure 3 représente le tableau `g.tset` sous la forme d'un graphe orienté acyclique : les noeuds contiennent en partie haute la position du terme dans le tableau `g.tset` et, pour la lisibilité de la transformation, l'ensemble des identifiants u_i des nœuds correspondants dans les arbres de la figure 2. La partie basse d'un noeud donne le symbole racine.

La composante `g.pre` de l'e-graphe contient la relation prédécesseur dans le graphe dirigé acyclique représenté par `g.tset`. Cette relation est donnée sous la forme d'une matrice linéarisée en un tableau de `g.tset_sz`^{2} valeurs dans l'ensemble $\{0, 1\}$. Si `g.pre[i*g.tset_sz + j]` vaut 1 alors le terme `g.tset[i]` a comme sous-terme direct le terme `g.tset[j]`. La valeur 0 dénote que `g.tset[j]` n'est pas sous-terme direct de `g.tset[i]`.



0 :	NULL	0	0
1 :	"X2"	0	0
2 :	"X1"	0	0
3 :	"f2"	2	1
4 :	"f1"	3	1
5 :	"f3"	2	3

FIGURE 3 – Codage de $\mathcal{T}(t_1 = t_2)$ sous forme de graphe (à gauche) et de tableau (à droite)

La composante `g.uf` de l'e-graphe `g` code les classes d'équivalence définies par les contraintes d'égalités entre les termes. La définition du type `uf_t` est la même que celle donnée dans la partie III. Chaque terme est représenté dans l'ensemble des éléments de `g.uf` par la position du noeud racine du terme dans le tableau `g.tset`. Par conséquent, un autre invariant de la structure d'e-graphe est le suivant :

> **`g.uf->nelem` *est égal à* `g.tset_sz`.**

Question 34. Écrire la fonction `bool term_is_diff(egraph_t *g, int tid1, int tid2)` qui teste si la contrainte $t_1 \neq t_2$ peut être satisfaite simultanément aux contraintes d'égalité représentées par `g`. Les paramètres `tid1` et `tid2` sont les positions dans `g.tset` des noeuds racine représentant respectivement t_1 et t_2. ⌟

Congruence. Soient $F \in \mathcal{F}(L_4)$ et `g` un e-graphe tel que `g.tset` représente $\mathcal{T}(F)$, l'ensemble des sous-termes de F. On définit la relation d'équivalence $\equiv_{\mathtt{g}}$ sur $\mathcal{T}(F)$ par : $t \equiv_{\mathtt{g}} t'$ si et seulement si t et t' sont représentés par des positions de `g.tset` qui sont dans la même classe d'équivalence de `g.uf`. On définit la relation $\mathcal{C}_{\mathtt{g}}$ sur $\mathcal{T}(F)$ comme suit :

> ***pour tout*** $f_i \in \mathbb{F}$ ***et pour tout quadruplet de termes*** $(t_1, t_2, t_3, t_4) \in \mathcal{T}(F)^4$***,***
> ***si*** $t_1 \equiv_{\mathtt{g}} t_3$ ***et*** $t_2 \equiv_{\mathtt{g}} t_4$***, alors*** $(f_i(t_1, t_2), f_i(t_3, t_4)) \in \mathcal{C}_{\mathtt{g}}$***.***

Question 35. Écrire la fonction `bool term_is_congruent(egraph_t *g, int tid1, int tid2)` qui renvoie `true` si et seulement si les termes représentés aux positions `tid1` et `tid2` dans `g.tset` sont dans la relation $\mathcal{C}_{\mathtt{g}}$. Donner et justifier soigneusement la complexité de `term_is_congruent` en fonction de `g.tset_sz`, si on suppose que le test d'égalité d'étiquettes (symboles de $\mathbb{X}$ ou $\mathbb{F}$) se fait en temps constant. ⌟

Algorithme de décision. Soient $F \in \mathcal{F}(L_4)$ et `g` un e-graphe tel que `g` représente les termes et les contraintes d'égalité de F ainsi que deux termes t_1 et t_2, pas nécessairement dans $\mathcal{T}(F)$. On cherche à modifier `g` pour qu'il représente $F \wedge (t_1 = t_2)$. On appelle cette opération `merge(g, t1, t2)` pour faire la différence avec `uf_union`. En effet, cette opération ne peut pas se limiter à appeler `uf_union` pour unir les classes d'équivalence des positions représentant les termes t_1 et t_2. Elle doit en plus tester si $(t'_1, t'_2) \in \mathcal{C}_{\mathtt{g}}$ pour tous termes t'_1 et t'_2 qui incluent respectivement t_1 et t_2 comme sous-termes directs mais ***ne sont pas déjà dans la même***

classe d'équivalence. Si $(t'_1, t'_2) \in \mathcal{C}_{\mathtt{g}}$, alors t'_1 et t'_2 doivent être mis dans la même classe d'équivalence. Cette dernière relation est propagée aux termes qui contiennent t'_1 et t'_2 et ainsi de suite, tant qu'on trouve des couples de termes dans la relation $\mathcal{C}_{\mathtt{g}}$ mais pas dans $\equiv_{\mathtt{g}}$.

Question 36. Écrire une fonction récursive `void merge(egraph_t *g, int tid1, int tid2)` qui ajoute la contrainte d'égalité entre les termes aux positions `tid1` et `tid2` dans `g.tset`, selon le processus décrit ci-dessus. Si `tid1` et `tid2` sont déjà dans la même classe d'équivalence, `merge` termine immédiatement. ⌟

Question 37. Donner le nombre maximum d'appels récursifs qu'un appel à `merge` peut engendrer en fonction de `g.tset_sz`. ⌟

Question 38. On suppose que le graphe orienté acyclique représenté par `g.tset` contient a arêtes. Donner la complexité en temps de l'opération `merge` en fonction de `g.tset_sz` et a. ⌟

Fin du sujet.

Annexe : Preuves en déduction naturelle

Un séquent $\Gamma \vdash F$ est composé d'un ensemble de formules Γ et d'une formule F.

Le sous-ensemble suivant de règles de la déduction naturelle peut être utilisé dans la preuve demandée à la question 14.

Introduction	**Elimination**
$\dfrac{}{\Gamma, F \vdash F}\ \mathrm{ax}$	
$\dfrac{}{\Gamma \vdash \top}\ \top_{\mathrm{I}}$	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash F}\ \bot_{\mathrm{E}}$
$\dfrac{\Gamma \vdash F_1' \quad \Gamma \vdash F_2'}{\Gamma \vdash F_1' \wedge F_2'}\ \wedge_{\mathrm{I}}$	$\dfrac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_1}\ \wedge_{\mathrm{E}}^{1} \qquad \dfrac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_2}\ \wedge_{\mathrm{E}}^{2}$
$\dfrac{\Gamma, F' \vdash F}{\Gamma \vdash F' \to F}\ \to_{\mathrm{I}}$	$\dfrac{\Gamma \vdash F' \to F \quad \Gamma \vdash F'}{\Gamma \vdash F}\ \to_{\mathrm{E}}$
$\dfrac{\Gamma, F \vdash \bot}{\Gamma \vdash \neg F}\ \neg_{\mathrm{I}}$	$\dfrac{\Gamma \vdash \neg F \quad \Gamma \vdash F}{\Gamma \vdash \bot}\ \neg_{\mathrm{E}}$

ECOLE POLYTECHNIQUE
ECOLES NORMALES SUPERIEURES

CONCOURS D'ADMISSION 2024

MARDI 16 AVRIL 2024
14h00 - 18h00

FILIERES MP-MPI - Epreuve n° 4

INFORMATIQUE A (XULSR)

Durée : 4 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve

Cette composition ne concerne qu'une partie des candidats de la filière MP, les autres candidats effectuant simultanément la composition de Physique et Sciences de l'Ingénieur.
Pour la filière MP, il y a donc deux enveloppes de Sujets pour cette séance.

Arbres équilibrés et géométrie algorithmique

Ce sujet traite de l'implémentation d'ensembles finis par des arbres équilibrés dits AVL et de leur application à un problème de géométrie algorithmique classique, la localisation d'un point dans le plan.

Ce sujet est conçu pour être traité linéairement. Les parties I et II introduisent les arbres AVL et leur construction et la partie III les utilise pour la localisation d'un point dans le plan. Pour traiter une question, on peut admettre le résultat de toute question précédente.

Langage OCaml. Ce sujet utilise notamment les tableaux d'OCaml. On peut créer un tableau avec la fonction `Array.make`. L'appel de `Array.make n x` crée un tableau de taille `n` dont toutes les cases contiennent la valeur `x`. Les cases d'un tableau sont numérotées à partir de 0. La fonction `Array.length` renvoie la taille d'un tableau. Pour un tableau `tab`, on accède à l'élément d'indice `i` avec `tab.(i)` et on le modifie avec `tab.(i) <- v`.

Dans le code OCaml fourni dans le sujet, on utilise la construction **`assert`** `false` pour signaler un point de code inatteignable, c'est-à-dire un cas de figure qui n'est pas censé se produire si les préconditions de la fonction sont respectées.

Pour les questions de programmation, il n'est pas demandé de justifier la correction du code donné en réponse.

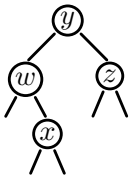
Complexité. Par *complexité en temps* d'un algorithme A on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaires à l'exécution de A dans le cas le pire. Par *complexité en espace* d'un algorithme A on entend l'espace mémoire minimal nécessaire à l'exécution de A dans le cas le pire. Lorsque la complexité dépend d'un ou plusieurs paramètres $\kappa_0, \cdots, \kappa_{r-1}$, on dit que A a une complexité en $\mathcal{O}(f(\kappa_0, \cdots, \kappa_{r-1}))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de $\kappa_0, \cdots, \kappa_{r-1}$ suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres $\kappa_0, \cdots, \kappa_{r-1}$, la complexité est au plus $C \cdot f(\kappa_0, \cdots, \kappa_{r-1})$. De même, on dit que A a une complexité en $\Theta(f(\kappa_0, \cdots, \kappa_{r-1}))$ s'il existe deux constantes $C_1 > 0$ et $C_2 > 0$ telles que, pour toutes les valeurs de $\kappa_0, \cdots, \kappa_{r-1}$ suffisamment grandes, la complexité de A est au moins $C_1 \cdot f(\kappa_0, \cdots, \kappa_{r-1})$ et au plus $C_2 \cdot f(\kappa_0, \cdots, \kappa_{r-1})$.

Dans tout ce sujet, on note « log » le logarithme à base 2. On utilisera exclusivement ce logarithme.

Partie I. Préliminaires

On cherche à construire une structure de données pour représenter des sous-ensembles finis d'un ensemble U donné. L'ensemble U est supposé muni d'un ordre strict total noté $<$, c'est-à-dire que, pour tous $x, y \in U$, on a $x < y$ ou $x = y$ ou $y < x$. On va utiliser des arbres binaires de recherche équilibrés pour représenter des sous-ensembles de U.

Arbres binaires. Un *arbre binaire* est défini récursivement de la manière suivante : soit comme l'arbre vide, noté $\langle\rangle$, qui ne contient aucun nœud ; soit comme un *nœud*, noté $\langle \ell, x, r\rangle$ avec un sous-arbre gauche ℓ, un élément $x \in U$ à la racine et un sous-arbre droit r. On dessine un arbre tel que $\langle\langle\langle\rangle, w, \langle\langle\rangle, x, \langle\rangle\rangle\rangle, y, \langle\langle\rangle, z, \langle\rangle\rangle\rangle$ de la manière suivante



(A)

avec la racine (ici y) en haut, reliée aux sous-arbres gauche et droit dessinés en-dessous. Les sous-arbres vides ne sont pas dessinés, mais les liaisons vers les sous-arbres vides le sont.

On note $n(t)$ le nombre de nœuds et $h(t)$ la hauteur d'un arbre binaire t, définis par

$$n(\langle\rangle) = h(\langle\rangle) = 0, \quad n(\langle \ell, x, r\rangle) = 1 + n(\ell) + n(r) \quad \text{et} \quad h(\langle \ell, x, r\rangle) = 1 + \max(h(\ell), h(r))$$

On notera ici que, pour des raisons purement techniques, la hauteur de l'arbre vide est fixée à 0 contrairement au programme qui la fixe à -1. Ainsi, pour l'arbre A pris en exemple ci-dessus, on a $n(A) = 4$ et $h(A) = 3$.

Le *parcours infixe* d'un arbre binaire est une séquence d'éléments de U définie récursivement de la manière suivante :

— le parcours infixe de l'arbre vide $\langle\rangle$ est la séquence vide ;

— le parcours infixe d'un arbre $\langle \ell, x, r\rangle$ est la concaténation, dans cet ordre, du parcours infixe de l'arbre ℓ, de l'élément x et du parcours infixe de l'arbre r.

Ainsi, le parcours infixe de l'arbre précédent est la séquence w, x, y, z.

Un arbre binaire t est un *arbre binaire de recherche* si la séquence de ses éléments donnée par un parcours infixe est triée par ordre strictement croissant. En particulier, un élément n'apparaît qu'au plus une fois dans cette séquence. Tous les arbres binaires manipulés dans ce sujet seront toujours des arbres binaires de recherche.

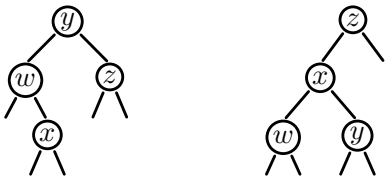
Question 1. Combien y a-t-il d'arbres binaires de recherche contenant exactement les trois éléments x, y et z, avec $x < y < z$? Les dessiner.

Question 2. Montrer qu'un arbre t est un arbre binaire de recherche si et seulement si t est l'arbre vide ou si t est un arbre $\langle \ell, x, r\rangle$ avec ℓ et r des arbres binaires de recherche où tous les éléments de ℓ sont strictement inférieurs à x et tous les éléments de r sont strictement supérieurs à x.

Arbres AVL. Pour équilibrer nos arbres binaires de recherche, on va imposer une condition supplémentaire : pour tout nœud $\langle \ell, x, r\rangle$, on a

$$|h(\ell) - h(r)| \leq 1 \tag{1}$$

De tels arbres sont dits AVL (du nom de leurs auteurs, Adelson–Velsky et Landis). Ainsi, dans les deux arbres ci-dessous,



celui de gauche est un AVL mais celui de droite n'en est pas un (en supposant $w < x < y < z$ par ailleurs).

Question 3. Combien y a-t-il d'arbres AVL contenant exactement les trois éléments x, y et z, avec $x < y < z$? Les dessiner. Même question avec les quatre éléments $w < x < y < z$. Justifier les réponses.

Question 4. Montrer que pour tout AVL t, on a $h(t) \leq 2\log(n(t) + 1)$. Indication : on pourra considérer N_h, le nombre minimal de nœuds d'un AVL de hauteur h, et minorer cette quantité en fonction de h.

Mise en œuvre en OCaml. On suppose que les éléments de U sont représentés par un type OCaml `elt`, qui n'est pas précisé pour l'instant. Pour deux éléments x et y donnés, on peut tester si $x = y$ avec `eq` x y et tester si $x < y$ avec `lt` x y.

```
val eq: elt -> elt -> bool
val lt: elt -> elt -> bool
```

Un AVL est représenté par une valeur du type OCaml suivant :

```
type tree = E | N of int * tree * elt * tree
```

Le constructeur `E` représente l'arbre binaire vide $\langle\rangle$. Une valeur `N(`z, ℓ, x, r`)` représente l'arbre binaire non vide $\langle \ell, x, r\rangle$, avec l'*invariant* que la valeur z du premier champ est toujours égale à la hauteur de l'arbre, *i.e.* $z = h(\langle \ell, x, r\rangle)$. On se donne une fonction `height` pour obtenir la hauteur d'un arbre :

```
let height t = match t with E -> 0 | N(h, _, _, _) -> h
```

On note que cette fonction s'évalue en temps constant. Pour maintenir l'invariant, on se donne une fonction `node` pour construire un nouveau nœud :

```
let node l x r = N(1 + max (height l) (height r), l, x, r)
```

On note que cette fonction s'évalue également en temps constant. Dans tout ce qui suit, on utilisera toujours la fonction `node` pour construire un nœud et jamais le constructeur `N` directement. On utilisera le constructeur `N` uniquement dans les motifs de filtrage.

Question 5. Écrire une fonction `mem: elt -> tree -> bool` qui prend en arguments un élément x et un arbre AVL t et qui détermine si x apparaît dans t. La complexité doit être en $O(\log n(t))$ et on demande de la justifier.

Partie II. Construire des arbres AVL

Maintenir la propriété d'arbre AVL pendant la construction des arbres binaires de recherche demande un certain effort. On propose ici de concentrer cet effort dans une unique fonction, `join`, dont le code est donné figure 1. La fonction `join` reçoit en paramètres deux arbres AVL `l` et `r` et un élément `x`. Elle suppose que les éléments de `l` (resp. `r`) sont strictement plus petits (resp. plus grands) que `x`. Elle renvoie un arbre AVL contenant l'union des éléments de `l`, de `r` et du singleton {`x`}.

Si les hauteurs de `l` et `r` satisfont déjà à la propriété d'AVL, alors il suffit d'appeler `node` (ligne 4). Sinon, on appelle `join_right` (resp. `join_left`) selon que le déséquilibre vient de `l` (ligne 2) ou de `r` (ligne 3). La fonction `join_right` procède récursivement, en descendant le long de la branche droite de l'arbre `l` (lignes 8–11) jusqu'à trouver un sous-arbre de hauteur

acceptable (lignes 4–7). Si besoin, des rotations sont effectuées localement avec les deux fonctions `rotate_left` et `rotate_right` (en haut de la figure 1). Ces deux fonctions implémentent la notion usuelle de rotation dans les arbres binaires de recherche, illustrée juste en-dessous de leur code. La fonction `join_left` est symétrique et son code est omis. Dans les raisonnements sur la fonction `join`, on pourra se contenter de considérer seulement le cas où `join_right` est appelée, en considérant l'autre cas symétrique.

On prendra le temps de bien lire et de bien comprendre le code de la fonction `join`. L'objectif des questions suivantes est de se persuader que `join` fonctionne correctement, ce qui est plus subtile qu'il n'y paraît.

Question 6. Montrer que tout appel à `join`, avec deux arguments `l` et `r` qui sont des AVL, termine et ne peut pas « planter », au sens où la ligne 3 de `rotate_left`, la ligne 3 de `rotate_right` et la ligne 12 de `join_right` ne sont jamais atteintes.

Question 7. Montrer que, lorsque la fonction `join_right` est appelée avec un AVL `l` de hauteur h et un AVL `r` de hauteur $\leq h-2$, alors elle renvoie

— soit un arbre de hauteur h ;

— soit un arbre de hauteur $h+1$ avec un sous-arbre gauche de hauteur $h-1$ et un sous-arbre droit de hauteur h.

Indication : pour chaque cas qu'on choisit de distinguer dans la preuve, on pourra se contenter d'un schéma donnant la forme de l'arbre renvoyé annoté avec les hauteurs pertinentes.

Question 8. Montrer que la fonction `join`, appliquée à deux AVL `l` et `r`, renvoie bien un arbre AVL dont la hauteur est au plus $1 + \max(h(\texttt{l}), h(\texttt{r}))$.

Question 9. Montrer que la fonction `join` a une complexité en $O(|h(\texttt{l}) - h(\texttt{r})|)$.

Insertion d'un élément. Pour insérer un nouvel élément dans un arbre AVL, on se donne la fonction `insert` (voir figure 2). Elle utilise une fonction auxiliaire, `split`, dont le code est également donné.

Question 10. Expliquer ce que fait la fonction `split` appliquée à un AVL `t` et un élément `y`.

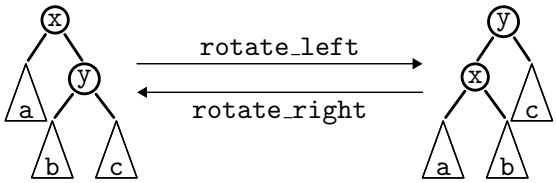
Question 11. Illustrer les étapes de l'insertion de l'élément c dans un AVL contenant les éléments a, b et d, avec $a < b < c < d$.

Question 12. Montrer que la fonction `split`, lorsqu'elle est appliquée à un AVL `t` et un élément `y`, renvoie deux arbres AVL dont la hauteur n'excède pas celle de `t`.

Question 13. Montrer que la fonction `insert`, lorsqu'elle est appliquée à un élément `x` et un arbre AVL `t`, renvoie bien un arbre AVL et a une complexité en $O(\log n(\texttt{t}))$.

```
let rotate_left t = match t with
  | N(_, a, x, N(_, b, y, c)) -> node (node a x b) y c
  | _ -> assert false
```

```
let rotate_right t = match t with
  | N(_, N(_, a, x, b), y, c) -> node a x (node b y c)
  | _ -> assert false
```




```
let rec join_right (l: tree) (x: elt) (r: tree) : tree =
  match l with
  | N(_, ll, lx, lr) ->
      if height lr <= height r + 1 then
        let t = node lr x r in
        if height t <= height ll + 1 then node ll lx t
        else rotate_left (node ll lx (rotate_right t))
      else
        let t = join_right lr x r in
        let t' = node ll lx t in
        if height t <= height ll + 1 then t' else rotate_left t'
  | E -> assert false
```

```
let rec join_left (l: tree) (x: elt) (r: tree) : tree =
  ...
```

```
let join (l: tree) (x: elt) (r: tree) : tree =
  if height l > height r + 1 then join_right l x r
  else if height r > height l + 1 then join_left l x r
  else node l x r
```

FIGURE 1 – Fonction `join` sur les AVL.

```
let rec split (t: tree) (y: elt) : tree * bool * tree =
  match t with
  | E -> E, false, E
  | N(_, l, x, r) ->
      if eq y x then l, true, r
      else if lt y x then let ll, b, lr = split l y in ll, b, join lr x r
      else let rl, b, rr = split r y in join l x rl, b, rr
```

```
let insert (x: elt) (t: tree) : tree =
  let l, _, r = split t x in
  join l x r
```

FIGURE 2 – Fonctions `split` et `insert` sur les AVL.

Question 14. Soit `n` un entier positif ou nul. On construit un tableau `a` de `n` arbres AVL de la manière suivante :

```
let a = Array.make n E
let () = for i = 1 to n - 1 do a.(i) <- insert (i-1) a.(i-1) done
```

Calculer le nombre total $f(\mathtt{n})$ d'entiers (possiblement répétés) stockés dans l'ensemble des arbres du tableau `a`. Montrer que la complexité en espace de la construction de `a` est en $O(g(\mathtt{n}))$ pour une fonction g telle que $g(n) = o(f(n))$. Expliquer pourquoi il n'y a pas de contradiction.

Supprimer un élément. On souhaite maintenant écrire une fonction `remove` pour supprimer un élément dans un arbre AVL, qui ait une complexité logarithmique comme la fonction `insert`.

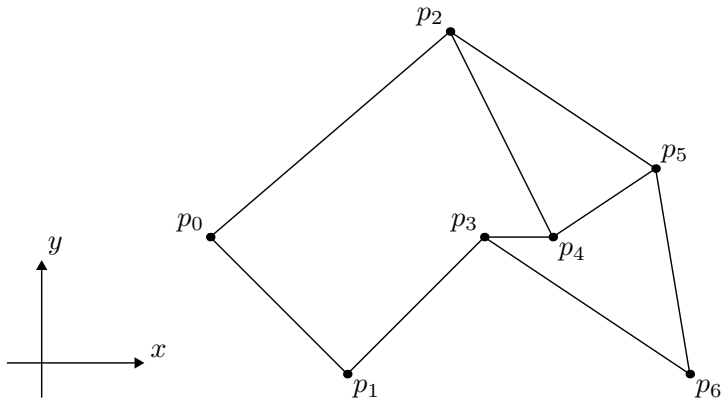
Question 15. Écrire une fonction `split_last: tree -> tree * elt` qui prend en argument un arbre AVL t, supposé non vide, et qui renvoie une paire (u, x) où x est le plus grand élément de t et u un arbre AVL contenant les éléments de t autres que x. La complexité doit être $O(h(t))$. On ne demande pas de la justifier.

Question 16. Écrire une fonction `join2: tree -> tree -> tree` qui prend en arguments deux arbres AVL ℓ et r, en supposant les éléments de ℓ strictement plus petits que ceux de r, et qui renvoie un arbre AVL t contenant l'union des éléments de ℓ et de r. La complexité doit être $O(h(\ell) + h(r))$. On ne demande pas de la justifier.

Question 17. Écrire une fonction `remove: elt -> tree -> tree` qui prend en arguments un élément x et un AVL t, et qui renvoie un arbre AVL contenant les éléments de t privés de x. (Si x n'apparaît pas dans t, le résultat contient les mêmes éléments que t.) La complexité doit être $O(\log n(t))$. On ne demande pas de la justifier.

Partie III. Application : localisation d'un point dans le plan

On considère une subdivision du plan définie par un ensemble fini de polygones. Ces polygones peuvent partager des sommets et des arêtes. Les arêtes ne se croisent pas. Voici un exemple avec 7 points et trois polygones :



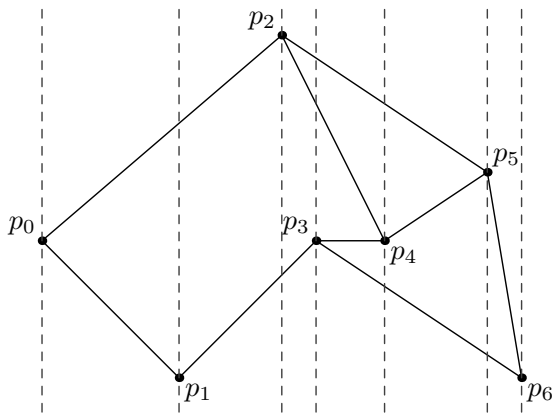
Le plan est ici divisé en quatre zones : l'intérieur de chaque polygone et la zone extérieure. Notre objectif est de construire une structure de données pour représenter cette subdivision, afin de répondre ensuite efficacement à la question « étant donné un point, dans quelle zone se trouve-t-il ? ». On s'intéresse notamment à la complexité de cette structure (temps et espace pour sa construction) et à la complexité de son utilisation ensuite. Ces complexités seront calculées en fonction du nombre total de sommets, noté N dans la suite. On admet que le nombre d'arêtes est en $O(N)$.

Notations. Pour un point p, on note $x(p)$ son abscisse et $y(p)$ son ordonnée. Les N sommets sont notés $p_0, p_1, \ldots, p_{N-1}$ et ordonnés de la gauche vers la droite. On suppose qu'il n'y a pas deux sommets sur la même abscisse, c'est-à-dire

$$x(p_0) < x(p_1) < \cdots < x(p_{N-1})$$

Une arrête e est un couple (p_i, p_j) avec $0 \leq i < j < N$. On note $start(e)$ son extrémité gauche, c'est-à-dire le point p_i, et $end(e)$ son extrémité droite, c'est-à-dire le point p_j. On note $xmin(e)$ l'abscisse de p_i et $xmax(e)$ l'abscisse de p_j. L'arrête e s'étend donc de l'abscisse $xmin(e)$ à l'abscisse $xmax(e)$.

Approche. Notre approche consiste à diviser le plan en $N + 1$ tranches verticales, délimitées par les abscisses des N points. Sur notre exemple, on a donc 8 tranches :



La tranche 0 correspond aux abscisses $]-\infty, x(p_0)]$, la tranche 1 aux abscisses $]x(p_0), x(p_1)]$, la tranche 2 aux abscisses $]x(p_1), x(p_2)]$, etc., et la tranche N aux abscisses $]x(p_{N-1}), +\infty[$. Chaque tranche est traversée par un certain nombre d'arêtes. On note que, dans chaque tranche, les segments d'arêtes qui traversent cette tranche peuvent être ordonnés verticalement (car, on le rappelle, les arêtes ne se croisent pas). Ainsi, dans la tranche 3 ci-dessus, on a, de bas en haut, l'arête (p_1, p_3), l'arête (p_2, p_4) puis enfin l'arête (p_2, p_5).

L'idée est alors de trier les tranches selon les abscisses, puis, dans chaque tranche, de trier les arêtes la traversant selon les ordonnées. Ainsi, étant un point du plan à localiser, il suffira d'une recherche dichotomique pour déterminer la tranche contenant le point puis d'une seconde recherche dichotomique pour déterminer entre quelles arêtes se trouve le point (et, par conséquent, dans quel polygone se trouve le point, le cas échéant).

Question 18. On note T_i le nombre d'arêtes qui traversent la tranche i. (Dans l'exemple ci-dessus, on a $T_0 = 0$, $T_1 = T_2 = 2$, $T_3 = 3$, etc.) Montrer que la quantité $\sum T_i$ peut être en $\Theta(N^2)$.

Mise en œuvre en OCaml. On utilise des nombres flottants pour les coordonnées des points. On introduit les deux types suivants pour les points et les arêtes, respectivement.

```
type point = float * float
type edge = point * point (* avec x1 < x2 *)
```

On se donne deux fonctions pour récupérer l'abscisse gauche et l'abscisse droite d'une arête :

```
let xmin ((x, _), _) = x
let xmax (_, (x, _)) = x
```

Pour représenter chaque tranche, on va utiliser un arbre AVL dont les éléments sont des arêtes, c'est-à-dire le type `tree` introduit plus haut dans le sujet avec

```
type elt = edge
```

Une tranche est alors représentée par le type suivant,

```
type slab = { xleft: float; xright: float; tree: tree }
```

où `xleft` est l'abscisse du bord gauche de la tranche, `xright` l'abscisse du bord droit et `tree` l'arbre AVL contenant les arêtes qui traversent cette tranche, ordonnées de bas en haut.

Question 19. Écrire une fonction `lt: edge -> edge -> bool` qui prend en arguments deux arêtes e_1 et e_2, supposées traversant une même tranche, et qui détermine si e_1 est strictement en dessous de e_2.

Construction des tranches. On représente l'ensemble des tranches par un tableau `slabs: slab array` de taille $N + 1$. Il est trié selon les abscisses, c'est-à-dire

$$\begin{aligned} -\infty = \texttt{slabs.(0).xleft} &< \texttt{slabs.(0).xright} = x(p_0) = \texttt{slabs.(1).xleft} \\ &< \texttt{slabs.(1).xright} = x(p_1) = \texttt{slabs.(2).xleft} \\ &< \ldots \\ &< \texttt{slabs.(}N\texttt{).xright} = +\infty \end{aligned}$$

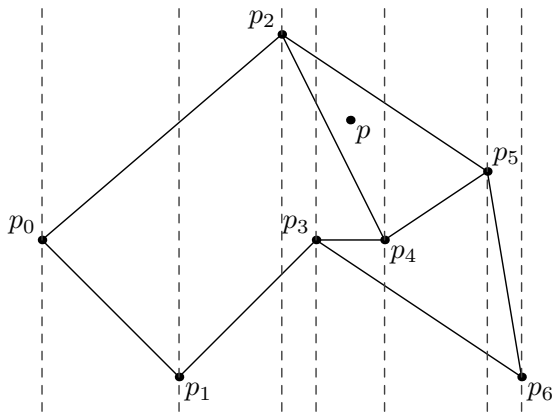
(Le type `float` inclut des valeurs `neg_infinity` pour $-\infty$ et `infinity` pour $+\infty$.) On construit les arbres AVL de chaque tranche de la gauche vers la droite, avec l'algorithme suivant :

— L'arbre de la tranche 0 est vide.
— Pour construire l'arbre de la tranche i, pour $1 \leq i \leq N$,
  — on retire les arêtes qui se terminent au point p_{i-1} ;
  — on ajoute les arêtes qui démarrent au point p_{i-1}.

En particulier, l'arbre de la dernière tranche est vide.

Question 20. Justifier que l'on peut construire, à partir de la liste des arêtes, le tableau `slabs` en temps et en espace $O(N \log N)$. On ne demande pas d'écrire le code, mais on demande d'être précis quant aux structures de données et algorithmes utilisés et on demande d'être convaincant quant à la complexité. On ne demande pas de vérifier que tous les sommets ont des abscisses différentes.

Localisation d'un point. Enfin, on peut utiliser le tableau `slabs` pour localiser un point p donné. On fait l'hypothèse que ce point n'est sur aucune arête, et est donc en particulier différent des N points p_i. Si on prend l'exemple suivant



alors le point p est d'abord localisé dans la tranche 4 puis, dans cette tranche, localisé entre les arêtes (p_2, p_4) et (p_2, p_5).

Question 21. Écrire une fonction `find_slab: slab array -> float -> slab` qui prend en arguments les tranches et l'abscisse du point p recherché et renvoie la tranche dans laquelle se trouve le point p. La complexité doit être $O(\log N)$. On ne demande pas de la justifier.

Question 22. Écrire une fonction `find: slab array -> point -> answer` qui prend en arguments les tranches et un point p et renvoie la zone dans laquelle le point p se trouve, sous la forme d'une valeur du type

```
type answer = Outside | Between of edge * edge
```

où `Outside` signifie que le point p est à l'extérieur de tout polygone et `Between`(e_1, e_2) signifie que le point p se trouve à l'intérieur d'un polygone, entre les arêtes e_1 et e_2. La complexité doit être $O(\log N)$ et on demande de la justifier.

* *
*

ECOLES NORMALES SUPERIEURES

CONCOURS D'ADMISSION 2024

VENDREDI 19 AVRIL 2024
14h00 - 18h00

FILIERES MP et MPI

Epreuve n° 10

INFO-FONDAMENTALE (ULSR)

Durée : 4 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve

Couverture minimale d'un graphe

Le sujet comporte 12 pages, numérotées de 1 à 12.

Début de l'épreuve.

Ce sujet est consacré au calcul d'un paramètre de graphe appelé ***couverture minimale***, défini comme étant le plus petit nombre de sommets d'un ***ensemble couvrant***. On étudiera plusieurs façon de le calculer, ce qui nous permettra de faire un tour d'horizon de quelques techniques algorithmiques modernes.

Ce sujet est constitué de 6 parties qui peuvent être traitées relativement indépendamment ; cependant, les concepts introduits dans une partie peuvent être réutilisés dans des parties ultérieures.

— Dans la première partie (page 3) on découvre progressivement certaines propriétés des couvertures minimales, leurs valeurs sur quelques familles de graphes simples et leurs liens avec d'autres paramètres classiques de la théorie des graphes.

— Dans la deuxième partie (page 4) on développe deux algorithmes efficaces qui calculent un ensemble couvrant minimum d'un arbre.

— Dans la troisième partie (page 5) on étudie deux heuristiques naturelles calculant un ensemble couvrant s'avérant loin d'être minimum, pour finalement découvrir un algorithme simple calculant une approximation de la couverture minimale.

— Dans la quatrième partie (page 7) on découvre la complexité paramétrée.

— Dans la cinquième partie (page 9) on continue notre exploration de la complexité paramétrée en introduisant la notion de noyau.

— Dans la sixième partie (page 10) on découvre la puissance de l'optimisation linéaire, permettant une autre approximation de la couverture minimale ainsi qu'un noyau plus petit que celui de la partie V.

Notations et définitions

— Un ***graphe*** (non-orienté) G est la donnée d'un ensemble fini $\mathrm{V}(G)$ de ***sommets*** et d'un ensemble $\mathrm{E}(G)$ dont les éléments, appelés ***arêtes***, sont des paires de sommets de V. On notera le fait que deux sommets u et v sont reliés par une arête par $\{u, v\} \in \mathrm{E}(G)$. On dit que u et v sont les ***extrémités*** de l'arête $\{u, v\}$ et que u et v sont ***adjacents*** dans G.

— Une conséquence de cette définition est que tous les graphes considérés sont ***simples***, c'est à dire sans boucle ($\{u, u\} = \{u\}$ n'est pas une arête), et avec au plus une arête entre deux sommets.

— Soit v un sommet d'un graphe G. Le ***voisinage*** $\mathrm{N}(v)$ de v est l'ensemble des sommets adjacents à v, c'est-à-dire $\{\, u \in \mathrm{V}(G) \mid \{u, v\} \in \mathrm{E}(G) \,\}$. Le ***degré*** $\mathrm{d}(v)$ est le cardinal de $\mathrm{N}(v)$.

— Un ***chemin*** $P = (v_1, \ldots, v_n)$ dans un graphe G est une liste de sommets deux à deux distincts tels que pour tout $1 \leqslant i < n$, v_i et v_{i+1} sont adjacents dans G.

— Un ***ensemble couvrant*** d'un graphe G est un ensemble de sommets X tel que toute arête de G a au moins une extrémité dans X. Le nombre de sommets d'un ***ensemble couvrant minimum*** (c'est-à-dire, un ensemble couvrant minimisant le nombre de sommets) de G est appelé ***couverture minimale*** de G et noté $\tau(G)$.

— Soit G un graphe et X un ensemble de sommets de G. On dénote par $G - X$ le graphe sur les sommets $\mathrm{V}(G) \setminus X$ avec comme arêtes $\{\, \{u, v\} \in \mathrm{E}(G) \mid u \notin X \text{ et } v \notin X \,\}$.

— Par ***complexité*** (en temps) d'un algorithme ALG, on entend le nombre d'opérations élémentaires nécessaires à l'exécution de ALG dans le pire des cas.

— Lorsque la complexité en temps dépend d'un ou plusieurs paramètres $x_1, \ldots, x_p$, on dit qu'elle est en $O(f(x_1, \ldots, x_p))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de $x_1, \ldots, x_p$, suffisamment grandes (c'est-à-dire, plus grandes qu'un certain seuil), la complexité est au plus $C \times f(x_1, \ldots, x_p)$. La complexité est dite ***linéaire*** si $f(x_1, \ldots, x_p) = x_1 + \cdots + x_p$.

— Dans tout l'énoncé (sauf dans la partie II où on travaille sur des arbres et dans la partie III où on ne s'intéresse pas à la complexité précise des algorithmes), on fera l'hypothèse que les graphes sont encodés par leur matrice d'adjacence.

Partie I

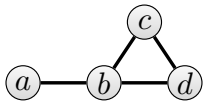
Un graphe est ***complet*** si tous les sommets sont deux à deux adjacents. On dénote par K_n le graphe complet sur n sommets $\{1, \ldots, n\}$.

Un ***ensemble indépendant*** d'un graphe est un ensemble de sommets du graphe dont les sommets sont deux à deux non-adjacents. On note $\alpha(G)$ le nombre maximum de sommets d'un ensemble indépendant de G. Un ***couplage*** est un ensemble d'arêtes deux à deux disjointes, c'est à dire ne partageant aucun sommet. Le nombre maximum d'arêtes dans un couplage de G est noté $\mu(G)$.

Un graphe est ***biparti*** si ses sommets peuvent être partitionnés en deux ensembles indépendants. Il est ***biparti complet*** s'il admet une partition en deux ensembles indépendants telle que chaque sommet d'une partie est adjacent à chaque sommet de l'autre partie. Étant donné deux entiers naturels a et b, on dénote par $K_{a,b}$ le graphe biparti complet dont les parties de la partition en deux ensembles indépendants ont respectivement a et b sommets (disons, les sommets étant numérotés de 1 à a dans la première partie et de $a+1$ à $a+b$ dans la deuxième).

On dénote par P_n le ***graphe chemin*** sur $n \geqslant 1$ sommets, c'est-à-dire le graphe dont les sommets sont $\{1, \ldots, n\}$ et les arêtes $\{\, \{1,2\}, \{2,3\}, \ldots, \{n-1, n\} \,\}$. Un ***graphe cycle*** est le graphe obtenu en ajoutant une arête entre les deux extrémités d'un graphe chemin. On dénote par C_n le graphe cycle sur $n \geqslant 2$ sommets.

Question I.1. Donner les valeurs de $\tau(G)$, $\alpha(G)$ et $\mu(G)$ pour le graphe G représenté ci-dessous. Justifier.



Question I.2. Donner la valeur de $\tau(K_n)$, $\tau(P_n)$, $\tau(C_n)$ et $\tau(K_{a,b})$ pour tous entiers naturels non nuls a, b, n. Justifier soigneusement les réponses.

Question I.3. Montrer que pour tout graphe G, $\tau(G) = |\mathrm{V}(G)| - \alpha(G)$.

Question I.4. Montrer que pour tout graphe G, $\tau(G) \geqslant \mu(G)$.

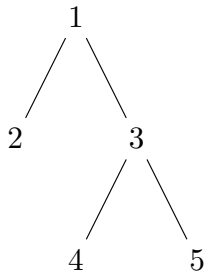
Aucun algorithme polynomial n'est connu pour calculer un ensemble couvrant minimum d'un graphe. Dans la question suivante, on demande un algorithme exponentiel.

Question I.5. Donner une description à haut niveau (sans détailler les lignes de pseudo-code) d'un algorithme calculant un ensemble couvrant minimum d'un graphe G en temps $O\left(2^{|\mathrm{V}(G)|} \cdot |\mathrm{V}(G)|^2\right)$.

Partie II

Un ***arbre*** est un graphe connexe sans cycle. Un sommet de degré $\leqslant 1$ d'un arbre est appelé une ***feuille***. Dans cette partie, on va développer deux algorithmes efficaces pour calculer un ensemble couvrant minimum d'un arbre.

Dans cette partie, contrairement au reste du sujet, les arbres ne sont pas représentés par leur matrice d'adjacence mais par un tableau T dont les éléments sont des ensembles de sommets : pour chaque sommet $1 \leqslant i \leqslant n$ (où n est le nombre de sommets de l'arbre), $T[i]$ est une représentation de l'ensemble des sommets adjacents à i. On supposera que la structure de données utilisée pour représenter l'ensemble $T[i]$ permet en temps $O(1)$ les opérations suivantes : renvoyer le nombre d'éléments de l'ensemble ; rechercher si un élément donné appartient à l'ensemble ; ajouter un nouvel élément à l'ensemble ; retirer un élément existant de l'ensemble. Par exemple, voici un arbre sur les sommets $\{1, \ldots, 5\}$ et sa représentation sous forme de tableau dont les éléments sont les ensembles de sommets adjacents :



1	{2, 3}
2	{1}
3	{1, 4, 5}
4	{3}
5	{3}

Question II.1. Soient T un arbre et $\{u, v\}$ une arête de T telle que v est une feuille de T. Montrer qu'il existe un ensemble couvrant minimum de T qui contient le sommet u, mais pas v.

Question II.2. En utilisant la question précédente, proposer un algorithme (en pseudo-code) qui reçoit un arbre T en entrée, et renvoie un ensemble couvrant minimum de T, en enlevant progressivement des arêtes à l'arbre. La complexité de cet algorithme doit être linéaire en le nombre n de sommets de l'arbre. Justifier de la complexité de l'algorithme.

Question II.3. Proposer un autre algorithme (en pseudo-code) qui renvoie le nombre de sommets d'un ensemble couvrant minimum d'un arbre, cette fois basé sur la programmation dynamique. On pourra commencer par ***enraciner*** l'arbre, c'est-à-dire choisir un sommet arbitraire qui jouera le rôle de racine de l'arbre. Pour chaque sommet $v \in \mathrm{V}(G)$, on note T_v le sous-arbre de T enraciné en v ; pour calculer le résultat final, l'algorithme devra d'abord calculer en programmation dynamique pour chaque $v \in \mathrm{V}(G)$ les trois valeurs suivantes :

— $D_{\in}(v)$ = nombre de sommets d'un ensemble couvrant minimum de T_v contenant v ;

— $D_{\notin}(v)$ = nombre de sommets d'un ensemble couvrant minimum de T_v ne contenant pas v ;

— $D(v)$ = nombre de sommets d'un ensemble couvrant minimum de T_v.

On ne demande pas la preuve de la correction, mais il faut donner et prouver sa complexité en fonction du nombre n de sommets de l'arbre.

Partie III

Dans cette partie, plutôt que d'essayer de calculer un ensemble couvrant minimum, on va chercher à en calculer un dont le cardinal est proche du minimum.

Formalisons cette idée. Soit ALG un algorithme prenant un graphe en entrée et renvoyant un ensemble couvrant (pas forcément minimum). On note $ALG(G)$ le nombre de sommets de l'ensemble couvrant renvoyé par ALG appliqué à G. On note $\mathrm{OPT}(G)$ le nombre de sommets d'un ensemble couvrant minimum de G. On dit que ALG est une ***c-approximation*** (pour un nombre réel $c \geqslant 1$) si pour tout graphe G, on a :

$$\frac{ALG(G)}{\mathrm{OPT}(G)} \leqslant c.$$

Noter que les algorithmes que nous allons considérer n'ont pas une façon unique de s'exécuter sur un graphe donné. Typiquement, à la ligne 3 de l'algorithme Heuristique ci-dessous, il peut y avoir beaucoup de façon différentes de choisir le sommet v. Dans ce cas, on considère que Heursitique(G) est le résultat de la pire exécution possible, c'est à dire celle renvoyant le plus grand ensemble couvrant parmi tous les ensembles couvrants qu'elle peut potentiellement renvoyer.

On considère un premier algorithme calculant un ensemble couvrant :

Algorithme 1 Heuristique(G)

1: $S \leftarrow \varnothing$
2: **while** G a des arêtes **do**
3: Choisir un sommet v de G tel que $\mathrm{d}(v) \geqslant 1$
4: $S \leftarrow S \cup \{v\}$
5: $G \leftarrow G - \{v\}$
6: **return** S

Question III.1. Montrer que l'ensemble S calculé par Heuristique est bien un ensemble couvrant.

Question III.2. Montrer que, pour tout entier $n \geqslant 2$, il existe un graphe G à n sommets tel que

$$\frac{\text{Heuristique}(G)}{\mathrm{OPT}(G)} \geqslant n - 1.$$

L'heuristique ci-dessus pouvant donner des résultats catastrophiques, on propose de l'améliorer en choisissant le sommet de degré maximum, donc celui couvrant le plus d'arêtes.

Algorithme 2 Glouton(G)

1: $S \leftarrow \varnothing$
2: **while** G a des arêtes **do**
3: Choisir un sommet v de degré maximum dans G
4: $S \leftarrow S \cup \{v\}$
5: $G \leftarrow G - \{v\}$
6: **return** S

Glouton renvoie un ensemble couvrant pour les mêmes raisons que Heuristique.

On va définir une série de graphes bipartis G_n avec bipartition A_n et B_n comme suit. A_n est un ensemble de n sommets. B_n est partitionné en n sous-ensembles $B_n^1, \ldots, B_n^n$ ayant les propriétés suivantes pour $k = 1, \ldots, n$:

(i) $|B_n^k| = \lfloor \frac{n}{k} \rfloor$,

(ii) chaque sommet de B_n^k a exactement k voisins dans A_n,

(iii) deux sommets de B_n^k n'ont pas de voisins communs dans A_n.

Question III.3. Dessiner (une réalisation possible de) G_4 et donner les valeurs de $\mathrm{OPT}(G_4)$ et de $\textsc{Glouton}(G_4)$. On ne demande pas de justifier.

Question III.4. En utilisant la construction ci-dessus, montrer que pour toute constante c, Glouton n'est pas une c-approximation.

Ainsi, Glouton ne donne pas non plus une bonne approximation. On va maintenant considérer l'algorithme suivant.

Algorithme 3 Approx(G)

```
1: S ← ∅
2: while G a des arêtes do
3:     Choisir une arête {u, v} de G
4:     S ← (S ∪ {u}) ∪ {v}
5:     G ← (G − {u}) − {v}
6: return S
```

Question III.5. Montrer que Approx renvoie un ensemble couvrant, en s'appuyant sur la question I.3.

Question III.6. Montrer que Approx est une 2-approximation.

Question III.7. Donner une suite infinie de graphes justifiant que, pour tout $c < 2$, Approx n'est pas une c-approximation.

Partie IV

Dans cette partie, on cherche à répondre à la question suivante : *Étant donné un graphe G et un entier k, est-ce que $\tau(G) \leqslant k$?* On rappelle que G est représenté par sa matrice d'adjacence.

Let but est de trouver un algorithme qui, étant donné un graphe G et un entier k, décide si $\tau(G) \leqslant k$ en temps $O(f(k) \times |\mathrm{V}(G)|^c)$ où f est une fonction (arbitraire, potentiellement exponentielle en son argument) et $c \geqslant 0$ une constante. Un algorithme avec une telle complexité est dit ***FPT*** pour ***fixed-parameter tractable*** ou ***tractable à paramètre fixé***.

On va commencer modestement. Noter que l'algorithme demandé à la question ci-dessous n'est pas FPT.

Question IV.1. Donner une description à haut niveau (sans détailler les lignes de pseudo-code) d'un algorithme qui, étant donné un graphe G et un entier k, décide si $\tau(G) \leqslant k$ en temps $O(|\mathrm{V}(G)|^{k+2})$.

Le reste de cette partie est dédié à la conception de notre premier algorithme FPT.

Question IV.2. Soit G un graphe et $\{u, v\} \in \mathrm{E}(G)$. Montrer l'équivalence suivante :

$$\tau(G) \leqslant k \quad \Longleftrightarrow \quad \tau(G - \{u\}) \leqslant k - 1 \text{ ou } \tau(G - \{v\}) \leqslant k - 1.$$

Question IV.3. En s'aidant de la question précédente, donner un algorithme récursif (en pseudo-code) qui, étant donné un graphe G et un entier k, décide si $\tau(G) \leqslant k$ en temps $O(2^k \times |\mathrm{V}(G)|^2)$. Justifier la correction et la complexité de l'algorithme. On rappelle qu'avec la représentation en matrice d'adjacence, il est possible de supprimer l'ensemble des arêtes dont un sommet u est une extrémité en $O(|V(G)|)$.

On va maintenant légèrement modifier l'algorithme de la question précédente pour améliorer sa complexité.

Question IV.4. Donner une description précise des graphes de degré maximum 2. En déduire une description à haut niveau (sans détailler les lignes de pseudo-code) d'un algorithme en $O(|\mathrm{V}(G)|)$ pour décider si un graphe de degré maximum 2 a un ensemble couvrant avec au plus k sommets.

On rappelle que, étant donné un sommet v, $\mathrm{N}(v)$ est l'ensemble des voisins de v et on définit $\mathrm{N}[v] = \mathrm{N}(v) \cup \{v\}$, le ***voisinage fermé*** de v.

Question IV.5. Soit $v \in V$. Montrer que si $\mathrm{d}(v) \geqslant 3$, alors

$$\tau(G) \leqslant k \quad \Longleftrightarrow \quad \tau(G - \{v\}) \leqslant k - 1 \text{ ou } \tau(G - \mathrm{N}[v]) \leqslant k - \mathrm{d}(v).$$

Question IV.6. En s'aidant de la question précédente, donner un algorithme récursif (en pseudo-code) qui décide si $\tau(G) \leqslant k$ en temps $O(1.4656^k \times |\mathrm{V}(G)|^2)$. Justifier la correction et la complexité de l'algorithme. On utilisera le fait que la fonction $f(k)$ définie récursivement comme suit :

$$f(i) = \begin{cases} f(i-1) + f(i-3) & \text{si } i \geqslant 3 \\ 1 & \text{sinon.} \end{cases}$$

satisfait $f(k) \leqslant 1.4656^k$.

Partie V

Comme dans la partie précédente, on cherche à répondre à la question suivante : *Étant donné un graphe G et un entier k, est-ce que $\tau(G) \leqslant k$?*

On va s'intéresser à la notion de noyau. On dit que le problème de l'ensemble couvrant a un ***noyau de taille*** $f(k)$ (pour une certaine fonction croissante f) s'il existe un algorithme qui étant donné une instance (G, k) renvoie en temps polynomial (en $|\mathrm{V}(G)|$) une instance (G', k') tel que :

(i) $\tau(G) \leqslant k$ si et seulement si $\tau(G') \leqslant k'$,
(ii) si $\tau(G) \leqslant k$, alors $|\mathrm{V}(G')| \leqslant f(k)$ et
(iii) $k' \leqslant k$.

Question V.1. Montrer que si le problème de l'ensemble couvrant a un noyau de taille $f(k)$, alors il existe un polynôme P tel qu'on peut décider si $\tau(G) \leqslant k$ en temps $O\left(P(|\mathrm{V}(G)|) + f(k)^{k+2}\right)$.

Le reste de la partie est dédié à la conception d'un algorithme montrant que le problème de l'ensemble couvrant a un noyau de taille $k^2 + k$.

Question V.2. Montrer que si un sommet v de G a degré 0, alors $\tau(G) \leqslant k$ si et seulement si $\tau(G - \{v\}) \leqslant k$.

Question V.3. Montrer que si un sommet v de G a degré au moins $k + 1$, alors $\tau(G) \leqslant k$ si et seulement si $\tau(G - \{v\}) \leqslant k - 1$.

On considère maintenant l'algorithme qui, étant donné (G, k), applique les deux règles suivantes (dans n'importe quel ordre) tant que c'est possible. On note (G', k') le couple obtenu.

Règle 1 : si un sommet v a degré 0, supprimer v.
Règle 2 : si un sommet v a degré au moins $k + 1$, supprimer v, et diminuer k de 1.

On remarque à l'aide des deux questions précédentes que $\tau(G) \leqslant k$ si et seulement si $\tau(G') \leqslant k'$.

Question V.4. Donner la complexité de cet algorithme en fonction de $|\mathrm{V}(G)|$ (on rappelle que G est représenté par sa matrice d'adjacence).

Question V.5. Montrer que pour tout sommet v de G', on a dans G' : $1 \leqslant \mathrm{d}(v) \leqslant k'$. En déduire que si $\tau(G') \leqslant k'$, alors $\mathrm{V}(G') \leqslant k' + k'^2$ et $\mathrm{E}(G') \leqslant k'^2$.

Question V.6. Conclure que le problème de l'ensemble couvrant a un noyau de taille $k^2 + k$.

Partie VI

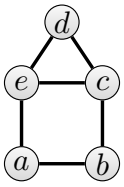
Soit $G = (V, E)$ un graphe avec $V = \{v_1, \ldots, v_n\}$. Un ensemble de sommets S peut être représenté par un vecteur $x = (x_{v_1}, x_{v_2}, \ldots, x_{v_n}) \in \{0,1\}^n$ tel que, pour $i = 1, \ldots, n$, $x_{v_i} = 1$ si $v_i \in S$ et $x_{v_i} = 0$ si $v_i \notin S$.

On introduit une variable x_v pour chaque sommet de G. Observer que les ensembles couvrant de G correspondent précisément aux solutions du système d'équations suivant :

$$\begin{cases} x_u + x_v \geqslant 1 \text{ pour tout } \{u, v\} \in \mathrm{E}(G); \\ x_v \in \{0,1\} \text{ pour tout } v \in \mathrm{V}(G). \end{cases} \tag{1}$$

Ainsi, une solution minimisant $\sum_{v \in \mathrm{V}(G)} x_v$ correspond à un ensemble couvrant minimum. Pour un graphe G donné, on nomme OLNEEC(G) (pour ***optimisation linéaire en nombre entier pour ensemble couvrant***) le système d'équations (1) associé à G.

Question VI.1. Écrire le système d'équations OLNEEC(G) pour le graphe ci-dessous.



En relaxant la contrainte $x_v \in \{0,1\}$ par $0 \leqslant x_v \leqslant 1$, on obtient le système d'équations suivant nommé OLEC(G) (pour ***optimisation linéaire pour ensemble couvrant***) :

$$\begin{cases} x_u + x_v \geqslant 1 \text{ pour tout } \{u, v\} \in \mathrm{E}(G); \\ 0 \leqslant x_v \leqslant 1 \text{ pour tout } v \in \mathrm{V}(G). \end{cases} \tag{2}$$

On peut considérer que $x_u = \frac{1}{3}$ (par exemple) correspond à mettre $\frac{1}{3}$ du sommet u dans la solution. Une solution à ce système d'équations s'appelle un ***ensemble couvrant fractionnaire*** de G et sa ***taille*** vaut $\sum_{v \in \mathrm{V}(G)} x_v$. Une solution minimisant $\sum_{v \in \mathrm{V}(G)} x_v$ est un ***ensemble couvrant fractionnaire minimum*** de G, et sa taille est noté $\tau_{\mathrm{f}}(G)$. Il est clair que pour tout graphe G,

$$\tau_{\mathrm{f}}(G) \leqslant \tau(G)$$

Question VI.2. Parmi les vecteurs $(x_a, x_b, x_c, x_d, x_e)$ suivants, dites lesquels sont des ensembles couvrants fractionnaires du graphe de la question VI.1 et lesquels ne le sont pas. Donner leur taille quand ils le sont, et expliquer précisément la raison quand ils ne le sont pas.

$$\left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right), \quad \left(\frac{1}{3}, \frac{1}{3}, \frac{2}{3}, \frac{2}{3}, \frac{1}{3}\right), \quad \Big(1, 1, 0, 1, 0\Big), \quad \Big(1, 0, 1, 1, 0\Big).$$

Question VI.3. Montrer que pour tout graphe G, on a $\tau_{\mathrm{f}}(G) \leqslant \frac{|\mathrm{V}(G)|}{2}$. En déduire qu'il n'existe pas de constante A telle que $\tau(G) - \tau_{\mathrm{f}}(G) < A$ pour tout graphe G.

Soit G un graphe. Soit $(x_v)_{v\in\mathrm{V}(G)} \in [0,1]^n$ un ensemble couvrant fractionnaire minimum de G. On partitionne les sommets de G de la façon suivante :

$$V_0 = \left\{v : x_v < \frac{1}{2}\right\} \qquad V_{\frac{1}{2}} = \left\{v : x_v = \frac{1}{2}\right\} \qquad V_1 = \left\{v : x_v > \frac{1}{2}\right\}$$

Question VI.4. Montrer que V_0 est un ensemble indépendant (c'est à dire qu'aucune arête n'a ses deux extrémités dans V_0), et qu'il n'y a pas d'arête ayant une extrémité dans V_0 et l'autre dans $V_{\frac{1}{2}}$.

Question VI.5. Montrer que $V_{\frac{1}{2}} \cup V_1$ est un ensemble couvrant de G et que $|V_{\frac{1}{2}} \cup V_1| \leqslant 2\tau(G)$. On admettra qu'il existe un algorithme en temps polynomial permettant de trouver un ensemble couvrant fractionnaire minimum. En déduire un algorithme polynomial qui donne une 2-approximation pour le problème de l'ensemble couvrant minimum.

Le but des 3 prochaines questions est de montrer qu'il existe un ensemble couvrant minimum inclus dans $V_{\frac{1}{2}} \cup V_1$. Soit S^* un ensemble couvrant minimum de G, et soit $S = \left(V_{\frac{1}{2}} \cap S^*\right) \cup V_1$.

Question VI.6. Montrer que S est un ensemble couvrant de G.

On va perturber la solution $(x_v)_{v\in\mathrm{V}(G)}$ comme suit. On pose

$$\varepsilon = 0 \text{ si } V_0 \cup V_1 = \varnothing \qquad \text{et} \qquad \varepsilon = \min\left\{\left|x_v - \frac{1}{2}\right| : v \in V_0 \cup V_1\right\} \text{ sinon}$$

et pour tout $v \in \mathrm{V}(G)$, on définit :

$$y_v = \begin{cases} x_v - \varepsilon & \text{si } v \in V_1 \setminus S^*; \\ x_v + \varepsilon & \text{si } v \in V_0 \cap S^*; \\ x_v & \text{dans tous les autres cas.} \end{cases}$$

Question VI.7. Montrer que $(y_v)_{v\in\mathrm{V}(G)}$ est un ensemble couvrant fractionnaire de G.

Question VI.8. Montrer que S est un ensemble couvrant minimum de G.

On va maintenant étudier l'algorithme suivant pour déterminer, étant donné un graphe G et un entier k, si $\tau(G) \leqslant k$.

Algorithme 4 OLNoyau(G, k)

1: Calculer un ensemble couvrant fractionnaire minimum $(x_v)_{v\in\mathrm{V}(G)}$ à l'aide d'un algorithme en temps polynomial.
2: Définir les ensembles V_0, $V_{\frac{1}{2}}$ et V_1 comme précédemment.
3: **if** $\sum_{v\in\mathrm{V}(G)} x_v > k$ **then return** NON
4: **if** $\left|V_{\frac{1}{2}}\right| > 2k$ **then return** NON
5: Résoudre l'instance $(G - V_0 - V_1, k - |V_1|)$ en utilisant un algorithme par force brute.

Question VI.9. Montrer que si l'algorithme renvoie NON aux lignes 3 ou 4, alors $\tau(G) > k$.

Question VI.10. Montrer que $\tau(G) \leqslant k$ si et seulement si $\tau(G - V_0 - V_1) \leqslant k - |V_1|$. Conclure que le problème de l'ensemble couvrant a un noyau de taille $2k$.

Fin du sujet.

A2024 – INFO I MPI

**ÉCOLE DES PONTS PARISTECH,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARIS,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.**

**Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).**

CONCOURS 2024

PREMIÈRE ÉPREUVE D'INFORMATIQUE

Durée de l'épreuve : 3 heures

L'usage de la calculatrice et de tout dispositif électronique est interdit.

Cette épreuve concerne uniquement las candidats de la filière MPI.

L'énoncé de cette épreuve comporte 9 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Les sujets sont la propriété du GIP CCMP. Ils sont publiés sous les termes de la licence
Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 France.

Tout autre usage est soumis à une autorisation préalable du Concours commun Mines Ponts.

Première épreuve d'informatique MPI 2024

Préliminaires

Présentation du sujet

L'épreuve est composée d'un problème unique comportant 33 questions divisées en trois sections. L'objectif du problème est d'extraire un *sous-graphe le plus dense* d'un graphe non orienté quelconque, c'est-à-dire repérer un sous-ensemble de sommets riche en arêtes. Le calcul d'un sous-graphe dense possède des applications considérables dans le domaine de la fouille des données : qu'il s'agisse de détection de communautés dans des réseaux sociaux, de repérage d'attaques par pollupostage (ou *spamming*) ou d'identification de complexes protéiques au sein de données biologiques massives pour ne citer que quelques exemples.

Dans la première section (page 1), nous résolvons un problème de minimisation de bordure dans un multigraphe orienté à l'aide d'une technique de *chemins augmentants*. Dans la deuxième section (page 5), nous étudions une technique de *réduction du problème* de construction d'un sous-graphe le plus dense au problème de la première section. Dans la troisième section (page 8), nous étudions un *algorithme d'approximation* plus rapide.

Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractères différentes désignera la même entité, mais du point de vue mathématique avec la police en italique (par exemple n, $n_{\max}$ ou n_1) et du point de vue informatique avec celle en romain avec espacement fixe (par exemple `n`, `NMAX` ou `n1`).

Travail attendu

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat.

Selon les consignes, il faudra coder des fonctions à l'aide du langage de programmation C exclusivement, en reprenant le prototype de fonction fourni par le sujet, ou en pseudo-code (c-à-d. dans une syntaxe souple mais conforme aux possibilités offertes par le langage C). Inclure les entêtes tels que `<assert.h>`, `<stdbool.h>`, etc. n'est pas demandé.

Quand l'énoncé demande de coder une fonction, sauf indication explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté et de la concision des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle. Lorsqu'une réponse en pseudo-code est permise, seule la logique de programmation est évaluée, même dans le cas où un code en C a été fourni en guise de réponse.

1 Entourage de plus petite bordure

Dans cette section, nous travaillons dans un *multigraphe orienté*, c'est-à-dire un couple (W, F, μ) où W est un ensemble fini, appelé ensemble de sommets, F est une partie de $W \times W$, appelée ensemble d'arcs, et $\mu : F \to \mathbb{N}^*$ est une application, appelée multiplicité des arcs.

Les démonstrations ou justifications pourront utiliser la notion de *multi-ensemble* sans formalité excessive. L'intersection, notée $\cap$, se calcule en considérant le minimum des multiplicités des éléments ; l'union disjointe, notée $\sqcup$, se calcule en considérant la somme des multiplicités des éléments ; le cardinal se calcule en sommant toutes les multiplicités.

1.1 Réseaux

Définitions : Nous appelons *réseau* tout quintuplet $H = (W, F, \mu, s, t)$ tel que

— W est un ensemble fini, appelé ensemble de *sommets*,

— F est un sous-ensemble de couples de $W \times W$, appelé ensemble d'*arcs*,

— μ est une application $F \to \mathbb{N}^*$, l'entier naturel $\mu(e)$ étant appelé la *multiplicité* de l'arc e,

— s est un sommet particulier de W, appelé la *source*,

— t est un sommet particulier de W, appelé le *puits*.

Indication C : Par convention, nous numérotons les sommets d'un réseau $H = (W, F, \mu, s, t)$ par les entiers entre 0 et $n + 1$, où $n + 2$ est le cardinal de W. La source s porte le numéro 0 ; le puits t porte le numéro $n + 1$.

Nous préparons deux fichiers, `network.h` et `network.c`. Le premier fichier contient les déclarations suivantes.

```
1.  /* Debut du fichier network.h */
2.
3.  #ifndef NETWORK_H
4.  #define NETWORK_H
5.
6.  const unsigned int NMAX = 102;
7.
8.  struct network_s {
9.      unsigned int n;
10.     unsigned mu[NMAX][NMAX];
11. };
12. typedef struct network_s network;
13.
14. void nw_init(unsigned int n);
15. void nw_add(int u, int v);
16. void nw_remove(int u, int v);
17. void nw_disconnect(void);
18.
19. void _arr_init(int *T);
20. bool _bfs_tree(void);
21. void _residue(void);
22.
23. #endif
24.
25. /* Fin du fichier network.h */
```

☐ 1 – Expliquer le rôle des lignes 3, 4 et 23 dans le fichier `network.h`.

Le second fichier contient initialement les définitions suivantes et sera complété par les fonctions de la section 1.

```
26.  /* Debut du fichier network.c */
27.
28.  network H;
29.  int A[NMAX];
30.
31.  /* Fin du fichier network.c */
```

Pour tout couple $(u, v) \in W \times W$, le champ `mu[u][v]` de la variable gobale H vaut 0 si (u, v) n'est pas un arc du réseau et vaut $\mu((u, v))$ si (u, v) est un arc.

☐ 2 – Expliquer comment faire interagir les fichiers `network.h` et `network.c`.

☐ 3 – Écrire une fonction C `void nw_init(unsigned int n)` ainsi spécifiée :
Précondition : L'entier naturel n est strictement inférieur à la constante globale $n_{\max}$.
Effet : La précondition est vérifiée par une assertion.
Postcondition : La variable globale H contient le réseau vide avec $n + 2$ sommets et aucun arc.

☐ 4 – Écrire une fonction C `void nw_add(int u, int v)` qui ajoute une occurence de l'arc (u, v) au réseau (W, F, μ, s, t) stocké dans la variable globale H, autrement dit, qui incrémente la multiplicité $\mu[u][v]$ d'une unité.

☐ 5 – Écrire une fonction C `void nw_remove(int u, int v)` qui retire une occurence de l'arc (u, v) dans le réseau stocké dans la variable globale H et qui se défend, par le truchement d'une assertion, du cas où l'arc à supprimer n'existe pas.

☐ 6 – Écrire une fonction C `void _arr_init(int *T)` ainsi spécifiée :
Précondition : Le pointeur T désigne un tableau d'entiers formé de $n + 2$ cases, où $n + 2$ est le nombre de sommets du réseau stocké dans la variable globale H.
Postcondition : Toutes les cases du tableau désigné par le pointeur T valent -1.

Nous utilisons le terme *chemin* pour signifier systématiquement un chemin simple dans le graphe orienté (W, F), c'est-à-dire une suite d'arcs $(e_1, \ldots, e_t)$ de F distincts qui se succèdent. Le sommet initial d'un chemin s'appelle la *source*, le sommet final s'appelle le *puits*.

☐ 7 – Écrire une fonction C `bool _bfs_tree(void)` ainsi spécifiée :
Précondition : La variable globale H contient un réseau (W, F, μ, s, t) à $n + 2$ sommets.
Effet : La fonction calcule une arborescence $\mathcal{A}$ de plus courts chemins selon un parcours en largeur dans le graphe (W, F) et l'écrit dans la variable globale A.
Postcondition : On a $A[s] = s$ et, pour tout autre sommet $v \in W$ accessible depuis s, la case $A[v]$ contient le prédécesseur du sommet v dans l'arborescence $\mathcal{A}$. Enfin, toute autre case de A vaut -1.
Valeur de retour : Booléen `true` si le puits t est accessible depuis la source s dans l'arborescence $\mathcal{A}$. Booléen `false` sinon.

1.2 Déconnexion de la source et du puits

Définition : Nous disons qu'un ensemble de chemins $\Pi = \{\pi_1, \ldots, \pi_k\}$ de source commune s et de puits commun t est *réalisable* dans le réseau $H = (W, F, \mu, s, t)$ si, pour tout arc $e \in F$, les chemins de Π utilisent moins de $\mu(e)$ fois l'arc e, autrement dit

$$\forall e \in F, \quad \text{Card}\left(\{j \in [\![1, k]\!] \text{ tel que } e \in \pi_j\}\right) \leqslant \mu(e).$$

Définition : Étant donné un ensemble réalisable de chemins $\Pi = \{\pi_1, \ldots, \pi_k\}$ de source commune s et puits commun t dans un réseau $H = (W, F, \mu, s, t)$, nous appelons *réseau résiduel de H par rapport à* Π, et nous notons $\text{Res}(H, \Pi)$, le réseau obtenu à partir de H en effectuant, pour tout indice j compris entre 1 et k et pour tout arc $e = (u, v) \in \pi_j$, la suppression d'une occurence de l'arc e et l'ajout d'une occurence de l'arc opposé $\overline{e} = (v, u)$.

☐ 8 – Écrire une fonction C `void _residue(void)` ainsi spécifiée :
Précondition : La variable globale H contient un réseau (W, F, μ, s, t). L'ensemble des arcs $\{(A[v], v); 0 < v \leqslant n + 1 \text{ avec } A[v] > 0\}$ tiré de la variable globale A forme une arborescence $\mathcal{A}$ de racine $s = 0$ atteignant le puits t dans le graphe (W, F).
Postcondition : La variable H contient le réseau résiduel $\text{Res}\left((W, F, \mu, s, t), \{\alpha\}\right)$ où α est l'unique chemin de source s et de puits t dans l'arborescence $\mathcal{A}$.

☐ 9 – Nous appliquons k fois la fonction `_residue` à un réseau H en mettant à jour la variable globale A et notons α_1, …, α_k les chemins utilisés entre la source s et le puits t. Montrer qu'il est toujours possible de trouver un ensemble réalisable Π de k chemins dans H tels que le réseau obtenu des applications de `_residue` coïncide avec le réseau résiduel par rapport à Π, autrement dit tel que l'on a

$$\text{Res}\left(\ldots \text{Res}\left(\text{Res}\left(H, \{\alpha_1\}\right), \{\alpha_2\}\right), \ldots, \{\alpha_k\}\right) = \text{Res}(H, \Pi).$$

☐ 10 – Écrire une fonction C `void nw_disconnect(void)` ainsi spécifiée :
Précondition : La variable globale H contient un réseau (W, F, μ, s, t).
Effet : Tant qu'il existe un chemin de source s et de puits t, le réseau H est transformé en le réseau résiduel $\text{Res}((W, F, \mu, s, t), \{\alpha\})$, où α est un plus court chemin de s vers t.
Postcondition : Si la fonction se termine, il n'existe pas de chemin de la source vers le puits dans le réseau H. De plus, l'ensemble des arcs $\{(A[v], v); 0 \leqslant v \leqslant n + 1 \text{ avec } A[v] > 0\}$ tiré de la variable globale A forme une arborescence $\mathcal{A}$ dans le réseau résiduel dans laquelle la racine est s et le sommet t n'est pas accessible.

☐ 11 – Démontrer que la fonction `nw_disconnect` se termine en introduisant un variant de boucle inspiré par la question 9.

1.3 Optimalité

Définition : Soit $H = (W, F, \mu, s, t)$ un réseau et S un sous-ensemble de sommets de W. Nous appelons *bordure*, et notons ∂S, l'ensemble d'arcs allant de S vers son complémentaire :

$$\partial S = \{(u, v) \in F \text{ avec } u \in S \text{ et } v \notin S\}.$$

Nous étendons la notion de multiplicité à des ensembles d'arcs en posant :

$$\mu(\partial S) = \sum_{e \in \partial S} \mu(e).$$

Définition : Nous appelons *entourage* dans le réseau $H = (W, F, \mu, s, t)$ tout sous-ensemble de sommets $S \subseteq W$ contenant la source s mais pas le puits t.

☐ 12 – Soient Π un ensemble réalisable de k chemins dans un réseau $H = (W, F, \mu, s, t)$ et $S \subseteq W$ un entourage dans H. Établir la relation

$$k \leqslant \mu(\partial S).$$

Nous fixons pour les quatre prochaines questions un ensemble réalisable Π_0 de k_0 chemins dans un réseau H tel que, dans le réseau résiduel $\mathrm{Res}(H, \Pi_0)$, il n'existe plus aucun chemin entre la source s et le puits t. Nous notons S_0 l'entourage dans H composé des sommets accessibles dans $\mathrm{Res}(H, \Pi_0)$ depuis la source s.

☐ 13 – Montrer que, pour tout arc $e = (u, v) \in F$ tel que $u \in S_0$ et $v \notin S_0$, toutes les $\mu(e)$ occurrences de l'arc e sont utilisées par un chemin de l'ensemble Π_0.

☐ 14 – Montrer qu'aucune occurence d'un arc $(v, u) \in F$ tel que $v \notin S_0$ et $u \in S_0$ n'est utilisée par un chemin de l'ensemble Π_0.

☐ 15 – Démontrer l'égalité

$$k_0 = \mu\left(\partial S_0\right).$$

Définition : Nous disons qu'un entourage S est *de plus petite bordure* s'il minimise la multiplicité $\mu(\partial S)$. Nous notons $\sigma(H)$ la multiplicité de la bordure d'un tel entourage.

☐ 16 – Démontrer la relation

$$k_0 = \sigma(H)$$

et en déduire que, si la variable globale H contient un réseau, après exécution de la fonction `nw_disconnect`, l'ensemble $\{v \in W\ ;\ A[v] \geqslant 0\}$ est un entourage de plus petite bordure de H, à savoir un entourage de multiplicité $\sigma(H)$.

2 Algorithme de Goldberg

2.1 Densité d'un graphe

Pour tout ensemble fini X, la notation $\binom{X}{2}$ désigne l'ensemble des paires (non ordonnées) à valeur dans X, c'est-à-dire l'ensemble

$$\binom{X}{2} = \left\{\{u, v\} \text{ avec } (u, v) \in X^2 \text{ tel que } u \neq v\right\}.$$

Définition : Un *graphe non orienté* est un couple $G = (V, E)$ où V est un ensemble fini, appelé ensemble de sommets, et $E \subseteq \binom{V}{2}$ est un ensemble de paires, appelé ensemble d'arêtes. Pour tout ensemble de sommets $X \subseteq V$, nous notons $G_X = (X, E_X)$ le graphe $G_X = \left(X, E \cap \binom{X}{2}\right)$, appelé *graphe induit par X sur G*.

Indication C : Par convention, nous numérotons les sommets de tout graphe non orienté $G = (V, E)$ par les entiers entre 1 et n, où n est le cardinal de V. Nous représentons un graphe par un tableau de listes d'adjacence doublement chaînées en adoptant les déclarations de type suivantes.

```
32. struct link_s {
33.     int node;
34.     struct link_s *prev;
35.     struct link_s *next;
36. };
37. typedef struct link_s link;
38.
39. struct graph_s {
40.     unsigned int n;
41.     link **neighbours;
42. };
43. typedef struct graph_s graph;
```

Le champ `neighbours` est un tableau alloué dynamiquement de longueur $n + 1$ et dont la première case n'est pas utilisée. Chaque arête est stockée sous forme de deux arcs.

☐ 17 – Écrire une fonction C `graph *gr_create(unsigned int n)` dont la valeur de retour est un graphe non orienté vide à n sommets.

☐ 18 – Écrire une fonction C `void gr_add(graph *G, int u, int v)` ayant pour effet d'insérer une arête $\{u, v\}$ dans le graphe non orienté G.

Définition : Nous notons $\deg_G(v)$ le *degré* d'un sommet v dans le graphe non orienté $G = (V, E)$:

$$\deg_G(v) = |\{u \in V \text{ tel que } \{u, v\} \in E\}| = \sum_{\substack{u \in V \text{ t.q.} \\ \{v,u\} \in E}} 1.$$

☐ 19 – Écrire une fonction C `int degree(graph *G, int v)` dont la valeur de retour est le degré $\deg_G(v)$ du sommet v dans le graphe non orienté G.

☐ 20 – Écrire une fonction C `int edges(graph *G)` dont la valeur de retour est le nombre d'arêtes dans le graphe non orienté G.

Définition : Nous appelons densité du graphe $G = (V, E)$ le rapport

$$\delta(G) = \frac{m}{n}$$

où n est le cardinal de V et m est le cardinal de E.

☐ 21 – Écrire une fonction C `double density(graph *G)` dont la valeur de retour est la densité $\delta(G)$ du graphe non orienté G.

Définition : Le problème d'optimisation du *sous-graphe le plus dense d'un graphe non orienté* G consiste à trouver un ensemble de sommets non vide X tels que la densité $\delta(G_X)$ du sous-graphe induit $G_X = (X, E_X)$ est maximale et calculer cette densité maximale

$$\rho(G) = \max_{\substack{X \subseteq V; \\ X \neq \varnothing}} \delta(G_X).$$

2.2 Oracle pour le problème de décision

Nous nous intéressons au problème de décision suivant :

> Étant donnés un graphe non orienté G à n sommets et un entier naturel r, existe-t-il un sous-graphe induit $G_X = (X, E_X)$ de densité $\delta(G_X)$ strictement supérieure au seuil $\dfrac{r}{n^2}$? Autrement dit, a-t-on $\rho(G) > \dfrac{r}{n^2}$?

Définition : Étant donnés un graphe non orienté $G = (V, E)$ avec n sommets et m arêtes ainsi qu'un entier naturel r, nous appelons *réseau associé au graphe G et au seuil r* le réseau $H = (W, E, \mu, s, t)$ où

(α) L'ensemble W est la réunion $W = V \cup \{s, t\}$, les sommets source s et puits t étant de nouveaux sommets n'appartenant pas à V.

(β) L'ensemble d'arcs F comprend :
- — pour tout sommet $v \in V$, un arc (s, v), de multiplicité mn^2.
- — pour toute arête $\{u, v\} \in E$, un arc (u, v) et un arc (v, u), de multiplicité n^2.
- — pour tout sommet $u \in V$, un arc (u, t), de multiplicité $mn^2 - n^2 \deg_G(u) + 2r$.

☐ 22 – Écrire une fonction C `void reduce(graph *G, int r)` ayant pour effet d'initialiser la constante globale H, de type **`network`**, par le réseau associé au graphe G et au seuil r.

☐ 23 – Soient $G = (V, E)$ un graphe non orienté et $X \subseteq V$ une partie non vide des sommets. Montrer que la densité du graphe induit $G_X = (X, E_X)$ satisfait la relation

$$\delta(G_X) = \frac{\displaystyle\sum_{v \in X} \deg_G(v) - \sum_{\substack{u \in X, v \in V \setminus X \\ \text{t.q.} \{u,v\} \in E}} 1}{2|X|}.$$

☐ 24 – Soient $G = (V, E)$ un graphe non orienté avec n sommets et m arêtes, $X \subseteq V$ une partie non vide des sommets, H le réseau associé au graphe G et à un seuil r et S l'entourage $S = \{s\} \cup X$ dans H. Montrer que la multiplicité de la bordure de S satisfait l'égalité

$$\mu(\partial S) = mn^3 + 2|X| \left(r - n^2\, \delta(G_X)\right)$$

et que

$$\mu(\partial\{s\}) = mn^3.$$

☐ 25 – Soit H le réseau associé au graphe G et au seuil r et soit $\sigma(H)$ la multiplicité de la bordure d'un entourage de plus petite bordure. Montrer les équivalences suivantes :

$$\begin{cases} \sigma(H) = mn^3 & \Leftrightarrow \quad \rho(G) \leqslant \dfrac{r}{n^2}, \\ \sigma(H) < mn^3 & \Leftrightarrow \quad \rho(G) > \dfrac{r}{n^2}. \end{cases}$$

☐ 26 – Écrire une fonction C `bool oracle(graph *G, int r)` dont la valeur de retour est le booléen `true` si $\rho(G) > \dfrac{r}{n^2}$ et `false` si $\rho(G) \leqslant \dfrac{r}{n^2}$. En justifier le principe. On pourra s'intéresser au contenu du tableau global A après exécution de la fonction `nw_disconnect`.

2.3 Recherche dichotomique

Nous considérons l'ensemble de couples d'entiers

$$\Delta = \left\{(m', n') \in \mathbb{N}^2 \text{ avec } 0 \leqslant m' \leqslant m \text{ et } 1 \leqslant n' \leqslant n\right\}.$$

☐ 27 – Soient deux couples (m_1, n_1) et (m_2, n_2) appartenant à Δ. Montrer que,

$$\text{si } \frac{m_1}{n_1} \neq \frac{m_2}{n_2}, \quad \text{alors} \quad \left|\frac{m_1}{n_1} - \frac{m_2}{n_2}\right| \geqslant \frac{1}{n^2}.$$

☐ 28 – Soient $G = (V, E)$ un graphe non orienté à n sommets et X une partie non vide de V. Montrer que s'il n'existe aucun sous-graphe induit de densité supérieure ou égale à $\delta(G_X) + \dfrac{1}{n^2}$, alors le sous-graphe induit G_X est un sous-graphe de densité maximale $\rho(G)$.

☐ 29 – Écrire une fonction C efficace `void binary_search(graph *G)` qui, pour tout graphe $G = (V, E)$, modifie le tableau global A de sorte que l'ensemble de sommets $X = \{v \in V\ ;\ A[v] \geqslant 0\}$ vérifie

$$\delta(G_X) = \rho(G).$$

Justifier brièvement le principe de fonctionnement de la fonction.

☐ 30 – Nous supposons avoir écrit l'ensemble des fonctions de la section 2 dans un fichier `densest_subgraph.c`. Expliquer comment faire interagir le fichier `densest_subgraph.c` avec les fichiers `network.h` et `network.c` de la section 1.

3 Algorithme de Charikar

Dans le cadre d'un compromis entre temps d'exécution et justesse du calcul d'optimisation, nous nous intéressons à l'algorithme suivant.

Algorithme 1 : Recherche gloutonne d'une densité élevée

Entrées : Graphe $G = (V, E)$.
Sorties : Sous-graphe induit à grande densité

1 $\Gamma \leftarrow G$; `/* Plus dense graphe rencontré */`
2 $G_n \leftarrow G$ où $n = |V|$
3 **pour** *i de n à 2 (cas inclus)* **faire**
4 **si** $\delta(G_i) > \delta(\Gamma)$ **alors**
5 $\Gamma \leftarrow G_i$
6 Identifier le sommet v_i qui minimise le degré $\deg_{G_i}$.
7 Former le graphe G_{i-1} en supprimant de G_i le sommet v_i et les arêtes incidentes à v_i.
8 **retourner** Γ

3.1 Analyse d'un facteur d'approximation

Définition : Nous appelons *orientation* d'un graphe non orienté $G = (V, E)$ tout graphe orienté $\widehat{G} = (V, \widehat{E})$ tel qu'il existe une bijection $\omega : E \to \widehat{E}$ avec $\omega(\{u, v\}) = (u, v)$ pour tout arc $(u, v) \in \widehat{E}$. Le *degré entrant* d'un sommet v dans un graphe orienté $\widehat{G} = (V, \widehat{E})$ est

$$\deg^-_{\widehat{G}}(v) = \mathrm{Card}\left(\left\{u \in V \text{ tel que } (u, v) \in \widehat{E}\right\}\right) = \sum_{\substack{u \in V \text{ t.q.} \\ (u,v) \in \widehat{E}}} 1.$$

☐ 31 – Établir la majoration suivante entre la densité d'un graphe non orienté quelconque G et le degré entrant maximum d'une orientation quelconque $\widehat{G}$ de G

$$\delta(G) \leqslant \max_{v \in V} \deg^-_{\widehat{G}}(v).$$

Nous exécutons l'algorithme 1 sur un certain graphe non orienté G. Dans les questions qui suivent, nous numérotons les sommets de G selon la même numérotation que celle de l'algorithme. Nous notons ω_0 la bijection qui oriente l'arête $\{v_i, v_{i'}\}$ dans le sens $(v_{\min(i,i')}, v_{\max(i,i')})$ et $\widehat{G}_0$ l'orientation associée. Autrement dit, nous orientons toute arête, au moment de sa suppression, vers le sommet incident qui a conduit à sa suppression.

☐ 32 – Établir, pour tout i compris entre 1 et n, l'inégalité

$$\deg^-_{\widehat{G}_0}(v_i) \leqslant 2\delta(G_i)$$

et en déduire que l'algorithme 1 est un algorithme d'approximation dont on précisera le facteur d'approximation.

3.2 Implémentation optimale

☐ 33 – Détailler une structure de données qui permette d'exécuter l'algorithme 1 en temps $O(n + m)$ où $n = |V|$ et $m = |E|$. Il est suggéré de nantir la structure de données `graph` d'une table qui maintient, pour tout entier d compris entre 0 et n, la collection des sommets de degré d ainsi que d'autres éléments que l'on jugera opportuns.
Une réponse décrite en pseudo-code ou en langage naturel est admise mais doit être suffisamment précise pour s'assurer que la complexité est belle et bien linéaire.

Fin de l'épreuve

A2024 – INFO II MPI

**ÉCOLE DES PONTS PARISTECH,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARIS,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.**

**Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).**

CONCOURS 2024

DEUXIÈME ÉPREUVE D'INFORMATIQUE

Durée de l'épreuve : 4 heures

L'usage de la calculatrice et de tout dispositif électronique est interdit.

Cette épreuve concerne uniquement las candidats de la filière MPI.

L'énoncé de cette épreuve comporte 12 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Les sujets sont la propriété du GIP CCMP. Ils sont publiés sous les termes de la licence
Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 France.

Tout autre usage est soumis à une autorisation préalable du Concours commun Mines Ponts.

Seconde épreuve d'informatique MPI 2024

Préliminaires

L'épreuve est composée d'un problème unique, comportant 36 questions. Le problème est divisé en quatre sections. Dans la première section (page 1), nous introduisons une forme paresseuse de listes, les *flots de données*, et construisons quelques fonctions de base utiles tout au long du sujet. Dans la deuxième section (page 4), nous nous intéressons à un *algorithme de détermination à la volée du cardinal* d'un flot, dont l'inspiration est probabiliste. Dans la troisième section (page 7), nous nous appuyons sur des raisonnements de logique pour analyser le *chiffrement par flot*. Dans la quatrième section (page 9), nous traitons quelques questions de *combinatoire énumérative* à l'aide d'une grammaire ou simplement d'un procédé manuel de dérécursification.

Les sections 2, 3 et 4 peuvent être vues comme des exercices indépendants qui ne dépendent que de la première section.

Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractères différentes désigne la même entité, mais du point de vue mathématique pour la police en italique (par exemple n) et du point de vue informatique pour celle en romain avec espacement fixe (par exemple `n`).

Des rappels portant sur des extraits du manuel de documentation de OCaml et sur les règles de la déduction naturelle sont fournis en annexe.

Travail attendu

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat.

Il faudra coder des fonctions à l'aide du langage de programmation OCaml exclusivement, en reprenant l'en-tête de fonctions fourni par le sujet, sans s'obliger à recopier la déclaration des types. Il est permis d'utiliser la totalité du langage OCaml mais il est recommandé de s'en tenir aux fonctions les plus courantes afin de rester compréhensible. Quand l'énoncé demande de coder une fonction, sauf demande explicite, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté et de la concision des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle.

1. Flots de données

1.1. Motivation

En langage OCaml, le type natif `'a list` est défini comme point fixe de l'équation

```
1.  type 'a list = [] | (::) of 'a * 'a list
```

Ce type permet de définir des listes finies tout comme certaines listes infinies, ainsi que l'expose l'exemple suivant, qui est parfaitement licite :

```
2.  let list_finite = [ 1; 2; 3 ]        (* liste finie a 3 elements *)
3.  let rec list_ones = 1 :: list_ones   (* liste infinie toute a 1 *)
```

☐ 1 – Rendre compte, sous la forme d'un croquis constitué de maillons chaînés entre eux, de l'organisation en mémoire des variables `list_finite` et `list_ones`. Dire quelle région de la mémoire d'un programme est utilisée pour ce stockage.

Les termes de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ obéissent aux relations :

$$F_0 = 0, \quad F_1 = 1, \quad \text{et} \quad \forall n \in \mathbb{N},\ F_{n+2} = F_n + F_{n+1}.$$

Nous nous enhardissons à définir la liste des termes de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ en écrivant la déclaration récursive suivante :

```
4.  let list_fibo : int list =
5.    let rec list_fibo_with_internal_state (a:int) (b:int) : int list =
6.      a :: (list_fibo_with_internal_state b (a+b))
7.    in
8.    list_fibo_with_internal_state 0 1
```

Cependant, à l'évaluation de cette expression par un interpréteur (ou *toplevel* ou encore *REPL*), le système renvoie le message d'erreur suivant :

```
9.  Stack overflow during evaluation (looping recursion?).
```

☐ 2 – Traduire en français le mot *stack*. Expliquer le message d'erreur en présentant le déroulement des premières étapes de création de la variable `list_fibo` par l'interpéteur OCaml.

Nous recopions le code fautif dans un fichier `fibo.ml` et — *horresco referens* — nous nous apprêtons à appeler successivement depuis un terminal les commandes `ocamlopt fibo.ml -o fibo` pour la compilation de la source vers un exécutable puis `./fibo` pour l'exécution.

☐ 3 – Dire si l'une des deux étapes, compilation ou exécution, renvoie une erreur et, le cas échéant, préciser laquelle des étapes.

1.2. Définition des flots de données

Définition : Un *flot de données* est une suite, finie ou infinie, d'éléments de même type.

Indication OCaml : Afin de représenter les flots de données, nous fixons le type suivant :

```
10. type 'a stream = Nil | Cons of 'a * (unit -> 'a stream)
```

Le constructeur `Nil` qualifie le flot vide. Définir la suite infinie constante égale à 1 en tant que `int stream` demeure possible. Nous parvenons de surcroît à déclarer la suite de Fibonacci en tant que `int stream` sans encombre. En voici le code :

```
11. let rec (ones : int stream) = Cons (1, fun () -> ones)
12.
13. let fibo : int stream =
14.   let rec fibo_with_internal_state a b =
15.     Cons (a, fun () -> fibo_with_internal_state b (a+b))
16.   in
17.   fibo_with_internal_state 0 1
```

☐ 4 – Écrire une constante OCaml `integers : int stream` dont la valeur de retour est le flot des entiers naturels énumérés par ordre croissant.

☐ 5 – Écrire une fonction OCaml `range (a:int) (b:int) : int stream` dont la valeur de retour est le flot des entiers compris entre l'entier a inclus et l'entier b exclu énumérés par ordre croissant.

☐ 6 – Soit $(\sigma_n)_{n\in\mathbb{N}}$ une suite quelconque de chaînes de caractères. Dire s'il existe forcément un flot de donnée, de type `string stream`, qui encode la suite $(\sigma_n)_{n\in\mathbb{N}}$ et, le cas échéant, en préciser la construction en langage OCaml ou en démontrer l'inexistence.

1.3. Quelques manipulations de base des flots de données

☐ 7 – Écrire une fonction OCaml `hd (u : 'a stream) : 'a` dont la valeur de retour est le premier élément du flot de données u et qui lève une exception `Failure "hd"` si le flot u est vide.

☐ 8 – Écrire une fonction OCaml `tl (u : 'a stream) : 'a stream` dont la valeur de retour est le flot de données u privé de l'élément de tête et qui lève une exception `Failure "tl"` si le flot u est vide.

☐ 9 – Écrire une fonction `of_list (l : 'a list) : 'a stream` dont la valeur de retour est le flot des éléments de la liste ℓ.

☐ 10 – Écrire une fonction OCaml `iter (f : 'a -> unit) (t : int) (u : 'a stream) : unit` qui applique la fonction f aux t premières valeurs existantes du flot de données u si l'entier t est positif et ne fait rien sinon.

☐ 11 – Écrire une fonction OCaml `map (f : 'a -> 'b) (u : 'a stream) : 'b stream` dont la valeur de retour est le flot constitué de l'application de la fonction f aux éléments de u.

☐ 12 – Écrire une fonction `zip (w : 'a stream array) : 'a array stream` qui transforme un tableau de flots de données $w = [(u_n)_{n\in\mathbb{N}}, (v_n)_{n\in\mathbb{N}}, \dots]$ en un flot de données tabulaire $([u_n, v_n, \dots])_{n\in\mathbb{N}}$. Si l'un des flots de w est vide à partir d'un certain rang, la valeur de retour est aussi vide à partir de ce rang.
Par exemple, `zip [|ones; range 0 100 ; fibo|]` désigne le flot fini des tableaux d'entiers de longueur trois $([1, k, F_k])_{0\leqslant k<100}$.

☐ 13 – Écrire une fonction `intertwin (u : 'a stream) (v : 'a stream) : 'a stream` dont la valeur de retour est le flot obtenu en prenant alternativement des éléments du flot u et du flot v, en cessant de prendre des éléments d'un flot une fois que celui-ci est vide. Par exemple, `intertwin ones fibo` désigne le flot d'entiers 1, F_0, 1, F_1, 1, F_2, 1, F_3, etc., tandis que `intertwin (range 0 2) fibo` désigne le flot d'entiers 0, F_0, 1, F_1, F_2, F_3, F_4, etc.

Nous nous proposons d'écrire une fonction `product (u1 : 'a stream) (u2 : 'b stream) : ('a * 'b) stream` dont la valeur de retour est un flot qui énumère tous les couples tirés du flot u_1 et du flot u_2 dans un ordre non spécifié. Nous écrivons le code suivant :

```
18.  let rec product_wrong u1 u2 =
19.    match u1,u2 with
20.    | _, Nil | Nil, _ -> Nil
21.    | Cons(hd1, tl_begetter1), Cons(hd2, tl_begetter2) ->
22.        let part1 = product_wrong (tl_begetter1 ()) u2 in
23.        let part2 = map (fun y -> (hd1, y)) (tl_begetter2 ()) in
24.        Cons((hd1, hd2), fun () -> intertwin part1 part2)
```

☐ 14 – Montrer que l'appel `product_wrong u1 u2` ne se termine pas toujours mais qu'il suffit de déplacer quelques mots au sein du code pour le rendre correct.

1.4. Filtre d'un flot

☐ 15 – À titre préliminaire, rappeler brièvement la signification de l'entier 0 dans l'instruction `return 0` qui termine usuellement la fonction `main` en langage C ou dans l'instruction `exit 0` du langage OCaml.

☐ 16 – Énoncer le théorème de Turing relatif au problème de l'arrêt.

Pour répondre à la question 17, on pourra admettre sans justification l'existence d'une *machine universelle à chronomètre*, c'est-à-dire une fonction OCaml `run : string -> int -> int` qui se termine toujours et telle que :

— si la chaîne de caractères x est le code source d'une expression OCaml et si l'exécution du code x se termine sans erreur en moins de n étapes élémentaires de calcul, alors `run x n` a pour valeur de retour l'entier 0 ;

— sinon, `run x n` a pour valeur de retour l'entier 1.

☐ 17 – Décrire la construction ou bien réfuter l'existence d'une fonction OCaml `filter (f : 'a -> bool) (u : 'a stream) -> 'a stream` dont la valeur de retour est formé des éléments du flot u satisfaisant le prédicat f, l'ordre des éléments de u étant préservé. Par exemple, `let is_negative x = x < 0 in filter is_negative ones` vaut `Nil`.

2. Estimation du cardinal d'un flot de données

Définition : Nous appelons *cardinal* d'un flot de données le cardinal de l'ensemble des valeurs distinctes appartenant à ce flot. Ce cardinal peut être *fini* même si le flot est infini.

Dans cette section, nous cherchons à estimer le cardinal s d'un flot de chaînes de caractères $(u_n)_{n\in\mathbb{N}}$ de cardinal fini avec peu de mémoire et en supposant que $0 \leqslant s < 2^{62}$.

Seconde épreuve d'informatique MPI 2024

Définition : Nous appelons *premier bit activé d'un entier* $h \in [\![0, 2^{62}-1]\!]$, et notons $\varrho(h)$, le plus grand entier i tel que les $i-1$ bits les plus à droite dans la représentation binaire de h sont tous nuls. Par convention, nous fixons $\varrho(0) = 63$.
Par exemple, nous avons $\varrho(1) = \varrho\left(\overline{00001}_{(2)}\right) = 1$, $\varrho(6) = \varrho\left(\overline{00110}_{(2)}\right) = 2$, ou encore $\varrho(8) = \varrho\left(\overline{01000}_{(2)}\right) = 4$.

☐ 18 – Écrire une fonction OCaml `ffs (h:int) : int` dont la valeur de retour est le premier bit activé $\varrho(h)$ de l'entier h. En anglais, `ffs` désigne *find first set*.

Indication OCaml : Nous disposons d'une fonction de hachage $\mathcal{H}$ à valeur dans l'intervalle entier $[\![0, 2^{62}-1]\!]$ que nous réalisons à l'aide de `String.hash : string -> int`.

Soit s un entier naturel non nul. Nous tirons s chaînes de caractères aléatoirement et notons $(H_j)_{1 \leqslant j \leqslant s}$ leur haché par la fonction de hachage $\mathcal{H}$. Pour tout indice j compris entre 1 et s, nous supposons que chaque variable aléatoire H_j est distribuée selon une loi uniforme dans l'intervalle entier $[\![0, 2^{62}-1]\!]$. Nous posons

$$R = \max_{1 \leqslant j \leqslant s} \varrho(H_j).$$

Une analyse mathématique, que nous admettons, montre alors que les variables aléatoires $\varrho(H_j)$ sont tirées selon des lois géométriques et que la variable R est proche de $\log_2 s$, si bien que la quantité 2^R est une bonne approximation de s.

Soit $(u_n)_{n \in \mathbb{N}}$ un flot de chaînes de caractères de cardinal s, l'entier s étant inconnu. Nous proposons deux algorithmes qui estiment le cardinal s par l'observation des t premiers termes du flot, en supposant que les s valeurs distinctes apparaissent parmi les t premiers termes.

— Algorithme naïf : garnir à la volée une table de hachage avec les chaînes $(u_n)_{n \leqslant t}$, puis renvoyer le compte des éléments présents dans la table. En voici le code :

```
25. let cardinality_nf (t:int) (u: string stream) : float =
26.   let htb = Hashtbl.create 1 in
27.   iter (fun x-> Hashtbl.add htb x () ) t u;
28.   float_of_int (Hashtbl.length htb)
```

où l'on réutilise la fonction `iter` de la question 10 et des fonctions du module `Hashtbl` rappelées en annexe.

— Algorithme de Flajolet-Martin, inspiré par les observations ci-dessus. En voici le code :

```
29. let cardinality_fm (t:int) (u: string stream) : float =
30.   let r = ref 0 in
31.   let update x = r:= max !r (ffs (String.hash x)) in
32.   iter update t u;
33.   float_of_int (1 lsl !r) (* calcule 2^r en flottant *)
```

☐ 19 – Comparer la complexité en espace des algorithmes `cardinality_nf` et `cardinality_fm`.

Nous rappelons que la moyenne harmonique de m réels $z_1, \ldots, z_m$ strictement positifs se calcule par la formule

$$\mathsf{Harm}(z_1, \ldots, z_m) = m \cdot \left(\sum_{j=1}^{m} \frac{1}{z_j} \right)^{-1}.$$

☐ 20 – Écrire une fonction OCaml `harmonic_mean (z : float array) : float` dont la valeur de retour est la moyenne harmonique des termes du tableau z.

La suite de chaînes de caractères $(u_n)_{n\in\mathbb{N}}$ et l'entier m étant fixés, pour tout entier i compris entre 0 et $m-1$ et pour tout entier naturel t, nous notons U_t^i l'ensemble de chaînes de caractères

$$U_t^i = \{u_n \text{ où } n \leqslant t \text{ et } \mathcal{H}(u_n) \equiv i \mod m\}.$$

Afin d'obtenir une estimation de cardinal du flot $(u_n)_{n\in\mathbb{N}}$ plus robuste et de diminuer la variabilité associée à l'unicité du point de mesure, nous proposons une amélioration de la fonction `cardinality_fm` qui utilise un effet de moyenne.

— Algorithme HyperLogLog : Nous divisons les premiers termes du flot $(u_n)_{n\in\mathbb{N}}$ en m sous-flots disjoints $U_t^0, \ldots, U_t^{m-1}$. Nous estimons le cardinal de chaque sous-flot comme le fait l'algorithme de Flajolet-Martin et obtenons des valeurs $(2^{r_i})_{0\leqslant i\leqslant m-1}$, où

$$r_i = \max_{u\in U_t^i} \varrho\left(\mathcal{H}(u)/m\right),$$

qui, intuitivement, forment chacune une estimation grossière du quotient s/m. Nous calculons la moyenne harmonique $\mathsf{Harm}(2^{r_0}, \ldots, 2^{r_{m-1}})$, qui, intuitivement, constitue une estimation robuste du quotient s/m.

Une analyse mathématique poussée confirme notre intuition. Il s'avère que le cardinal s est très bien approché par la quantité

$$s \simeq \alpha_m \cdot m \cdot \mathsf{Harm}\left(2^{r_0}, \ldots, 2^{r_{m-1}}\right) \qquad (\clubsuit)$$

où α_m est une constante, valant ici $\alpha_m = \left(m \int_0^\infty \left(\log_2\left(\frac{2+u}{1+u}\right)\right)^m \mathrm{d}u\right)^{-1}$. On utilisera l'équation ($\clubsuit$) sans la démontrer durant l'épreuve.

Indication OCaml : Nous fixons les constantes globales pour le reste de la section 2.

```
34. let m = 16
35. let alpha16 = 0.6731 (* calcul numerique admis *)
```

L'algorithme HyperLogLog se présente alors sous la forme

```
1. let cardinality_hll (t:int) (u : string stream) : float =
2.   let r = Array.make m 0 in
3.   iter (hll_update r) t u;
4.   mean_estimate r
```

où les fonctions `hll_update` et `mean_estimate` seront écrites aux questions 21 et 22.

☐ 21 – Écrire une fonction OCaml `hll_update (r : int array) (x : string) : unit` ainsi spécifiée :

Précondition : Il existe un flot de chaînes de caractères $(u_n)_{n\in\mathbb{N}}$, des entiers m et t tels que $x = u_t$ et le tableau r contient $\left(\max_{u\in U_{t-1}^i} \varrho(\mathcal{H}(u)/m)\right)_{0\leqslant i\leqslant m-1}$.

Postcondition : Le tableau r contient $\left(\max_{u\in U_t^i} \varrho(\mathcal{H}(u)/m)\right)_{0\leqslant i\leqslant m-1}$.

☐ 22 – Écrire une fonction OCaml `mean_estimate (r : int array) : float` ainsi spécifiée
Précondition : Le tableau r contient les valeurs $(r_i)_{0\leqslant i\leqslant m-1}$ avec

$$r_i = \max_{u\in U_t^i} \varrho(\mathcal{H}(u)/m)$$

pour un certain flot de chaînes de caractères $(u_n)_{n\in\mathbb{N}}$ et un certain entier m.
Valeur de retour : Estimation du cardinal du flot $(u_n)_{n\in\mathbb{N}}$ d'après l'équation (♣).

Nous nous plaçons finalement dans la situation où deux flots de chaînes de caractères $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ nous parviennent simultanément. Nous chargeons deux fils d'exécution auxiliaires distincts de construire indéfiniment leurs tableaux respectifs de maxima de bit activés tandis que le fil d'exécution principal déclenche toute les cinq secondes l'affichage du cardinal. Voici une ébauche de code :

```
36.  let cardinality_threadedhll u v =
37.    let r_u = Array.make m 0 in
38.    let f1 () = iter (hll_update r_u) Int.max_int u in
39.    let _ = Thread.create f1 () in
40.
41.    let r_v = Array.make m 0 in
42.    let f2 () = iter (hll_update r_v) Int.max_int v in
43.    let _ = Thread.create f2 () in
44.
45.    while true do
46.     let r = Array.init m (fun i -> max r_u.(i) r_v.(i)) in
47.     print_float (mean_estimate r);
48.     Thread.delay 5. (* mise en pause de 5 secondes *)
49.    done
```

dans laquelle `Int.max_int` représente le plus grand entier représentable.

☐ 23 – Identifier les portions de code sujettes à des courses critiques. Indiquer un mécanisme qui permet de corriger l'ébauche de code.

3. Chiffrement par flot de booléens

Dans cette partie, nous nous appuyons sur des variables booléennes $(x_j)_{j\leqslant k}$ et sur les constantes $\top$ (tautologie) et $\bot$ (antilogie) pour former des formules, à l'aide des trois connecteurs $\neg$ (négation), $\wedge$ (conjonction) et $\vee$ (disjonction). Pour toutes formules de logique ϕ et ψ, nous notons $X(\phi,\psi)$ la nouvelle formule

$$X(\phi,\psi) = (\phi\vee\psi)\wedge(\neg\phi\vee\neg\psi).$$

Les règles d'inférence de la déduction naturelle sont rappelées dans l'annexe B.

☐ 24 – Soit ϕ une formule. Construire des arbres de preuve qui démontrent les séquents

$$X(\phi,\phi)\vdash\bot \qquad \text{et} \qquad \bot\vdash X(\phi,\phi)$$

à partir des règles d'inférence de la déduction naturelle.

☐ 25 – Soit ψ une formule. Construire des arbres de preuve qui démontrent les séquents

$$X(\bot, \psi) \vdash \psi \qquad \text{et} \qquad \psi \vdash X(\bot, \psi).$$

Nous admettons que, pour toutes formules de logique ϕ et ψ, il existe des arbres de preuve qui démontrent les séquents $X(X(\phi, \phi), \psi) \vdash \psi$ et $\psi \vdash X(X(\phi, \phi), \psi)$.

Définitions : Nous notons $\mathbb{B} = \{V, F\}$ l'ensemble des valeurs de vérité. Nous introduisons la fonction booléenne $\oplus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ telle que $\oplus(x, y)$ est l'évaluation de la formule $X(x, y)$.

☐ 26 – Justifier que les fonctions booléennes $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$ suivantes $(x, y) \mapsto \oplus(\oplus(x, x), y)$ et $(x, y) \mapsto y$ sont égales.

Nous admettons, plus généralement, que la fonction booléenne $\oplus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ est commutative et associative. Pour tout entier $m \geqslant 1$ et pour tous booléens $(b_i)_{1 \leqslant i \leqslant m}$, la somme $\bigoplus_{i=1}^{m} b_i$ est définie comme $\oplus\left(\cdots \oplus\left(\oplus(b_1, b_2), b_3\right), \ldots, b_m\right)$ mais peut être calculée dans n'importe quel ordre. La somme vide vaut, par convention, la valeur de vérité F.

Définition : Nous appelons *registre à décalage à rétroaction linéaire* (ou encore *LFSR* d'après l'anglais *linear feedback shift register*) de longueur m, de germe $(b_i)_{0 \leqslant i < m} \in \mathbb{B}^m$ et de connexion $C \subseteq [\![0, m-1]\!]$, le flot de booléens $(b_n)_{n \in \mathbb{N}}$ défini par la relation de récurrence

$$\forall n \in \mathbb{N}, \quad b_{n+m} = \bigoplus_{i \in C} b_{n+i}$$

Indication OCaml : Nous utilisons le type `bool array` pour représenter le germe et le type `int list` pour représenter la connexion d'un LFSR. La longueur s'obtient avec `Array.length`.

☐ 27 – Écrire une fonction OCaml `lfsr (g:bool array) (c:int list) : bool stream` dont la valeur de retour est le flot de données constitué par le LFSR de germe g et de connexion c.

Indication OCaml : Nous représentons les formules de logique à l'aide du type

```
type formula = | Var of int
               | Not of formula
               | And of formula * formula
               | Or  of formula * formula
```

Pour toute formule de logique ϕ définie à partir des variables booléennes $(x_j)_{1 \leqslant j \leqslant k}$ et pour tout k-upplet de valeurs de vérité $b = (b^1, \ldots, b^k) \in \mathbb{B}^k$, nous notons $\phi(b^1, \ldots, b^k)$ l'évaluation de la formule ϕ en b.

☐ 28 – Écrire une fonction OCaml `logical_map (phi : formula) (w : bool stream array) : bool stream` ainsi spécifiée :
Précondition : La formule ϕ ne dépend que des variables x_1, …, x_k. Le tableau w est de longueur k. Nous notons $w = \left[\left(b_n^{(1)}\right)_{n \in \mathbb{N}}, \ldots, \left(b_n^{(k)}\right)_{n \in \mathbb{N}}\right]$ son contenu.
Valeur de retour : Flot des évaluations booléennes $\left(\phi\left(b_n^{(1)}, \ldots, b_n^{(k)}\right)\right)_{n \in \mathbb{N}}$.

Dans la question 29, deux personnages, Alice et Bob, se sont accordés, lors de leur tout premier tête-à-tête et à l'abri de toute écoute par un tiers, au sujet de k germes et de k connexions de LFSR ainsi qu'au sujet d'une formule de logique $\psi_0(x_1, \ldots, x_k)$. Nous notons $\left(b_n^{(1)}\right)_{n\in\mathbb{N}}$, ..., $\left(b_n^{(k)}\right)_{n\in\mathbb{N}}$ les k flots de données booléens associés aux LFRS. À présent, Alice souhaite communiquer à Bob la suite de bits $c = (c_n)_{n\in\mathbb{N}}$ via un canal public. Elle transmet à Bob la suite des évaluations $\left(X(\psi_0(b_n^{(1)}, \ldots, b_n^{(1)}), c_n)\right)_{n\in\mathbb{N}}$.

☐ 29 – Dire si Bob peut toujours parvenir à reconstituer le flot $c = (c_n)_{n\in\mathbb{N}}$ et, le cas échéant, expliquer comment.

4. Énumération d'objets combinatoires

4.1. Flot des mots de Dyck

Soit Σ l'alphabet binaire $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Nous nous intéressons à la constante OCaml `dyck` définie par le code suivant

```
54.  let rec dyck =
55.    let f (x,y) = "a" ^ x ^ "b" ^ y in
56.    Cons("", fun () -> map f (product dyck dyck))
```

où la fonction `product` a été introduite à la question 14. Nous notons $\mathcal{D} \subseteq \Sigma^*$ le langage constitué des mots appartenant au flot de données `dyck`.

☐ 30 – Exhiber une grammaire non contextuelle $\mathcal{G}$ telle que le langage $L(\mathcal{G})$ engendré par la grammaire $\mathcal{G}$ coïncide avec le langage $\mathcal{D}$.

☐ 31 – Énoncer une propriété de la grammaire $\mathcal{G}$ de la question 30 qui permet d'établir que le flot de chaînes de caractères `dyck` n'énumère jamais deux fois le même mot. La démontrer.

4.2. Flot des permutations d'un tableau

Nous partageons une variante d'un algorithme dû à B. R. Heap sous la forme du pseudo-code suivant. Dans ce pseudo-code, les positions de tableau sont numérotées à partir de l'indice 0.

Algorithme 1 : Algorithme de B. R. Heap

Entrées : Entier k, tableau t de longueur $\geqslant k$.
Effet : Effectue des échanges en place dans le tableau t et des affichages de t.
Sortie : Aucune valeur de retour.

```
1 si k = 1 alors
2 |  Afficher le tableau t.
3 sinon
4 |  pour i = 0 à k − 1 (inclus) faire
5 |  |  Exécuter Heap((k − 1), t).
6 |  |  si k est pair alors
7 |  |  |  Échanger t[i] et t[k − 1]
8 |  |  sinon
9 |  |  |  Échanger t[0] et t[k − 1]
```

☐ 32 – Soient k un entier et t un tableau de longueur supérieure à k. Dénombrer le nombre d'appels $A(k)$ à la fonction « afficher » lorsque l'on exécute $\mathsf{Heap}(k,\, t)$.

☐ 33 – Soient k un entier et t un tableau de longueur supérieure à k. Démontrer, pour tout entier naturel non nul k, la proposition $\mathcal{P}(k)$ suivante :

> Après l'exécution de $\mathsf{Heap}(k,\, t)$, si k est un entier impair, alors le tableau t est inchangé ; en revanche, si k est un entier pair, alors les k premiers coefficients du tableau t ont subi une permutation circulaire vers la droite : autrement dit, le nouvel état du tableau est

$$\Big(t[k-1] \quad t[0] \quad t[1] \quad \cdots \quad t[k-3] \quad t[k-2] \quad t[k] \quad \cdots \quad t[n-1]\Big)$$

☐ 34 – Soient k un entier et t un tableau de longueur supérieure à k. Démontrer que l'exécution de $\mathsf{Heap}(k,\, t)$ affiche, pour toute permutation des k premiers alvéoles du tableau t, le tableau dans lequel les k premiers alvéoles ont été permutés et dans lequel les derniers alvéoles ont été laissés inchangés.

☐ 35 – Écrire une fonction OCaml `pivot (k:int) (i:int) (t:'a array) : unit` dont l'effet est d'appliquer au tableau t les opérations des lignes 6 à 9 de l'algorithme 1.

☐ 36 – Écrire une fonction OCaml `permstream_of_array (t:'a array) : 'a array stream` dont la valeur de retour est un flot de données fini constitué de toutes les permutations du tableau t énumérées dans l'ordre d'affichage de l'algorithme de Heap et qui utilise efficacement la mémoire.

A. Annexe : rappels de programmation

(D'après https://v2.ocaml.org/api/index.html)

Opérations générales : Le module `Stdlib`, chargé par défaut, offre les fonctions suivantes :

- `max : 'a -> 'a -> 'a`
  Return the greater of the two arguments. The result is unspecified if one of the arguments contains the float value nan.
- `float_of_int : int -> float`
  Convert an integer to floating-point.
- `(lsl) : int -> int -> int`
  `n lsl m` shifts n to the left by m bits.

Opérations sur les tableaux : Le module `Array` offre les fonctions suivantes :

- `make : int -> 'a -> 'a array`
  `make n x` returns a fresh array of length n, initialized with x.
- `init : int -> (int -> 'a) -> 'a array`
  `init n f` returns a fresh array of length n, with element number i initialized to the result of $f(i)$.
- `exists : ('a -> bool) -> 'a array -> bool`
  `exists f [|a1; ...; an|]` checks if at least one element of the array satisfies the predicate. That is, it returns `(f a1) || (f a2) || ... || (f an)`.

Opérations sur les tables de hachage : Le module `Hashtbl` offre les fonctions suivantes :

- `create : int -> ('a, 'b) t`
  `Hashtbl.create n` creates a new, empty hash table, with initial size n.
- `add : ('a, 'b) t -> 'a -> 'b -> unit`
  `Hashtbl.add t k d` adds a binding of key k to data d in table t.
- `length : ('a, 'b) t -> int`
  `Hashtbl.length tt` returns the number of bindings in t. It takes constant time. Multiple bindings are counted once each.

Opérations sur les fils d'exécution : Le module `Thread` offre les fonctions suivantes :

- `create : ('a -> 'b) -> 'a -> t`
  `Thread.create f x` creates a new thread of control, in which the function application $f(x)$ is executed concurrently with the other threads of the domain. The application of `Thread.create` returns the handle of the newly created thread.
- `delay : float -> unit`
  `delay d` suspends the execution of the calling thread for d seconds. The other program threads continue to run during this time.

Opérations sur les verrous : Le module `Mutex` offre les fonctions suivantes :

- `create : unit -> t`
  Return a new mutex.
- `lock : t -> unit`
  Lock the given mutex. Only one thread can have the mutex locked at any time. A thread that attempts to lock a mutex already locked by another thread will suspend until the other thread unlocks the mutex.
- `unlock : t -> unit`
  Unlock the given mutex. Other threads suspended trying to lock the mutex will restart. The mutex must have been previously locked by the thread that calls `Mutex.unlock`.

B. Annexe : règles de la déduction naturelle

Dans les tableaux suivants, la lettre Δ désigne un ensemble de formules de logique ; les lettres A, B et C désignent des formules de logique.

Axiome
$\dfrac{}{\Delta, A \vdash A}\ (\text{ax})$

	Introduction	Élimination
$\top$	$\dfrac{}{\Delta \vdash \top}\ (\top\text{i})$	
$\bot$		$\dfrac{\Delta \vdash \bot}{\Delta \vdash A}\ (\bot\text{e})$
$\wedge$	$\dfrac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \wedge B}\ (\wedge\text{i})$	$\dfrac{\Delta \vdash A \wedge B}{\Delta \vdash A}\ (\wedge\text{e})$ $\dfrac{\Delta \vdash A \wedge B}{\Delta \vdash B}\ (\wedge\text{e})$
$\vee$	$\dfrac{\Delta \vdash A}{\Delta \vdash A \vee B}\ (\vee\text{i})$ $\dfrac{\Delta \vdash B}{\Delta \vdash A \vee B}\ (\vee\text{i})$	$\dfrac{\Delta \vdash A \vee B \qquad \Delta, A \vdash C \qquad \Delta, B \vdash C}{\Delta \vdash C}\ (\vee\text{e})$
$\neg$	$\dfrac{\Delta, A \vdash \bot}{\Delta \vdash \neg A}\ (\neg\text{i})$	$\dfrac{\Delta \vdash A \qquad \Delta \vdash \neg A}{\Delta \vdash \bot}\ (\neg\text{e})$

Fin de l'épreuve

A2024 – INFO MP

**ÉCOLE DES PONTS PARISTECH,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARIS,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.**

**Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).**

CONCOURS 2024

Durée de l'épreuve : 3 heures

L'usage de la calculatrice et de tout dispositif électronique est interdit.

ÉPREUVE D'INFORMATIQUE MP

*Cette épreuve concerne uniquement les candidats de la filière MP.
Les candidats sont priés de mentionner de façon apparente
sur la première page de la copie :*

INFORMATIQUE - MP

L'énoncé de cette épreuve comporte 7 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Les sujets sont la propriété du GIP CCMP. Ils sont publiés sous les termes de la licence
Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 France.

Tout autre usage est soumis à une autorisation préalable du Concours commun Mines Ponts.

Épreuve d'informatique d'option MP 2024

Préliminaires

L'épreuve est formée d'un problème unique, constitué de 25 questions, qui porte sur l'algorithmique de quelques résultats mathématiques reliés à la notion de relation d'ordre. Le problème est divisé en trois sections qui peuvent être traitées séparément, à condition d'avoir lu les définitions introduites jusqu'à la question traitée.

Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractères différentes désigne la même entité mais du point de vue mathématique pour la police en italique (par exemple n) et du point de vue informatique pour celle en romain avec espacement fixe (par exemple `n`).

Des rappels des extraits du manuel de documentation de OCaml sont reproduits en annexe. Ces derniers portent sur le module `Array`.

Travail attendu

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat.

Il faudra coder des fonctions à l'aide du langage de programmation OCaml exclusivement, en reprenant l'en-tête de fonctions fourni par le sujet, sans s'obliger à recopier la déclaration des types. Il est permis d'utiliser la totalité du langage OCaml mais il est recommandé de s'en tenir aux fonctions les plus courantes afin de rester compréhensible. Quand l'énoncé demande de coder une fonction, sauf demande explicite, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté et de la concision des programmes : il est attendu que l'on choisisse des noms de variables intelligibles ou encore que l'on structure de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle.

1. Relation d'ordre et tri topologique

Nous fixons un ensemble fini V et appelons n son cardinal. Nous notons systématiquement v_0, v_1, …, v_{n-1} les éléments distincts de V.

Définition : Nous appelons *relation d'ordre sur* V toute relation binaire $\trianglelefteq$ qui est

1. *réflexive*, c'est-à-dire telle que, pour tout indice i compris entre 0 et $n-1$, nous avons la relation $v_i \trianglelefteq v_i$,
2. *antisymétrique*, c'est-à-dire telle que, pour tous indices i et j compris entre 0 et $n-1$, si les relations $v_i \trianglelefteq v_j$ et $v_j \trianglelefteq v_i$ sont vérifiées, alors nous avons l'égalité $v_i = v_j$
3. *transitive*, c'est-à-dire que, pour tous indices i, j et k compris entre 0 et $n-1$, si les relations $v_i \trianglelefteq v_j$ et $v_j \trianglelefteq v_k$ sont vérifiées, alors nous avons encore la relation $v_i \trianglelefteq v_k$.

Un ensemble fini muni d'une relation d'ordre s'appelle un *ensemble ordonné* et se note $(V, \trianglelefteq)$. Nous réservons le symbole $\leqslant$ pour désigner l'ordre usuel sur les entiers.

Indication OCaml : Nous représentons tout ordre sur un ensemble de cardinal n par une matrice carrée r de dimension $n \times n$, de type

```
1. type order = bool array array
```

où le coefficient `r.(i).(j)` vaut `true` si et seulement si la relation $v_i \trianglelefteq v_j$ est vérifiée.

☐ 1 – Écrire une fonction OCaml `check_reflexivity (r:order) : bool` qui vérifie si la relation r est une relation réflexive et qui relève du paradigme de *programmation impérative*.

☐ 2 – Écrire une fonction OCaml `check_reflexivity_bis (r:order) : bool` qui vérifie également si la relation r est une relation réflexive mais qui, par contraste avec la question 1, relève du paradigme de *programmation fonctionnelle*.

Dans le reste du sujet, le choix du paradigme de programmation est libre.

☐ 3 – Écrire une fonction OCaml `check_antisymmetry (r:order) : bool` qui vérifie si la relation r est une relation antisymétrique.

☐ 4 – Écrire une fonction OCaml `check_transitivity (r:order) : bool` qui vérifie si la relation r est une relation transitive.

☐ 5 – Écrire une fonction OCaml `is_order (r:order) : bool` qui vérifie si la relation r est une relation d'ordre en s'appuyant sur les questions 1, 3 et 4.

Définition : Nous appelons *graphe d'un ensemble ordonné* $(V, \trianglelefteq)$, ou simplement graphe de la relation d'ordre $\trianglelefteq$, le graphe orienté $G = (V, E)$ dont l'ensemble des arcs est

$$E = \{(v_i, v_j) \text{ où } (i, j) \in [\![0, n-1]\!]^2 \text{ et } v_i \trianglelefteq v_j\}.$$

Nous rappelons que le *degré sortant* d'un sommet v désigne le nombre d'arcs dont l'extrémité initiale vaut v.

☐ 6 – Écrire une fonction OCaml `outdegrees (r:order) : int array` dont la valeur de retour est le tableau d des degrés sortants des sommets du graphe de la relation d'ordre r. Autrement dit, pour tout indice i compris entre 0 et $n-1$, l'entier $d[i]$ est le degré sortant du sommet v_i.

☐ 7 – Écrire une fonction OCaml `argmin (d:int array) : int` dont la valeur de retour est un indice valide i tel que, pour tout indice j compris entre 0 et la longueur du tableau d'entiers d exclue, on a l'inégalité $d[i] \leqslant d[j]$.

Définition : Étant donné un ensemble ordonné $(V, \trianglelefteq)$ de cardinal n, nous appelons *tri topologique* de V toute permutation τ des entiers de l'intervalle $[\![0, n-1]\!]$ telle que, pour tous indices i et j compris entre 0 et $n-1$, si l'on a la relation $v_i \trianglelefteq v_j$, alors on a l'inégalité $\tau(i) \leqslant \tau(j)$. Pour tous indices i compris entre 0 et $n-1$, nous disons dans ce cas que le sommet v_i est le sommet de numéro $\tau(i)$.

☐ 8 – Montrer qu'il existe une constante entière δ_0 telle que, pour tout tri topologique τ, le sommet de numéro $n-1$, à savoir le sommet $v_{\tau^{-1}(n-1)}$, est de degré sortant δ_0 et le degré sortant des autres sommets est supérieur ou égal à δ_0.

☐ 9 – Écrire une fonction OCaml `topological_sort (r:order) : int array` dont la valeur de retour est un tri topologique de l'ensemble ordonné par r. Il est attendu que la fonction attribue les numéros par ordre décroissant et que, en cours d'exécution, elle s'appuie sur la question 8 pour identifier le prochain sommet à numéroter.

☐ 10 – Énoncer un ensemble d'invariants qui démontre que la fonction `topological_sort` de la question 9 est correcte. Il pourra s'agir d'*invariants de boucle* dans le cas d'un programme itératif ou de *postconditions* dans le cas d'un programme récursif.

☐ 11 – Exprimer la complexité en temps de la fonction `topological_sort r` en fonction du nombre d'éléments n ordonnés par la relation r. Dire comment on qualifie une telle complexité en fonction de l'espace en mémoire utilisé pour représenter l'argument d'entrée r.

2. Chaînes et antichaînes

Définition : Étant donné un ensemble ordonné $(V, \trianglelefteq)$, nous appelons *chaîne* toute partie C de V totalement ordonnée, c'est-à-dire, telle que pour tous éléments c et $c' \in C$, nous avons la relation $c \trianglelefteq c'$ ou la relation $c' \trianglelefteq c$.

Indication OCaml : Nous représentons une chaîne par le type `int list`.

Nous donnons le code suivant

```
2.  let rec is_chain (r:order) (c:int list) : bool =
3.     match c with
4.     | [] -> true
5.     | v::cc -> List.for_all (fun x -> x > v || x < v) cc
6.                && is_chain r cc
```

où nous avons recouru à la fonction `for_all` du module `List` ainsi décrite dans la documentation de OCaml :

> **val** `for_all : ('a -> bool) -> 'a list -> bool`
> `for_all f [a1; ...; an]` checks if all elements of the list satisfy the predicate `f`. That is, it returns `(f a1) && (f a2) && ... && (f an)` for a non-empty list and `true` if the list is empty.

☐ 12 – Confirmer ou réfuter la spécification suivante : la fonction `is_chain r c` codée comme ci-dessus teste si la liste de sommets c est une chaîne au sens de la relation d'ordre r.
Dans le premier cas, on fournira une démonstration ; dans le second cas, on modifiera la fonction `is_chain` afin qu'elle soit correcte.

☐ 13 – Déterminer la complexité en temps de la fonction `is_chain r c` codée comme ci-dessus en fonction de la longueur γ de la liste c. Justifier le calcul.

Nous nous proposons d'écrire une fonction `is_chain_bis r c` obéissant à la spécification de la question 12 et de complexité en temps $O(\gamma \log \gamma)$ où γ est la longueur de la liste c. À cette fin, nous nous préparons à introduire une structure de donnée mutable à capacité bornée permettant de stocker une collection d'éléments de $(V, \trianglelefteq)$ à travers l'interface suivante :

```
7.  type hp
8.  val init : order -> int -> hp
9.  val is_empty : hp -> bool
10. val push : hp -> int -> bool
11. val pop : hp -> bool
```

L'appel `init r gamma` doit permet d'initialiser une collection vide, informée de la relation d'ordre r, et de capacité γ. L'appel `is_empty q` doit indiquer si la collection q est vide. L'appel `push q i` doit insérer l'élément $v_i \in V$ à la collection q ; l'appel `pop q` doit retirer un élément de la collection q. Nous envisageons que les appels à `push` et de `pop` puissent échouer, ce qui est signalé par la valeur de retour de ces fonctions. Nous écrivons la fonction `is_chain_bis r c` sous la forme suivante :

```
12. let is_chain_bis (r:order) (c:int list) : bool =
13.    let q = init r (List.length c) in
14.    let rec insert c : bool =
15.       match c with
16.       | [] -> true
17.       | hd::tl -> (push q hd) && insert tl
18.    in
19.    let rec extract_all () : bool =
20.       if is_empty q then true
21.       else (pop q) && (extract_all ())
22.    in
23.    (insert c) && (extract_all ())
```

Cette fonction insère les éléments de la liste c à une collection initialement vide q, de type `hp`, puis les retire de cette collection q.

☐ 14 – Compléter la description du type `hp` en nommant une structure de donnée, dont on rappellera brièvement l'invariant de bonne constitution, de sorte que la fonction `is_chain_bis r c` obéisse à la spécification de la question 12 et s'exécute en temps $O(\gamma \log \gamma)$, où γ est la longueur de la liste c. Justifier la réponse.

Définition : Étant donné un ensemble ordonné $(V, \trianglelefteq)$, nous appelons *antichaîne* toute collection de sommets $A \subseteq V$ dont les éléments sont deux à deux incomparables, c'est-à-dire, telle que pour tous éléments distincts a et $a' \in A$, aucune des relations $a \trianglelefteq a'$ ou $a' \trianglelefteq a$ n'est vérifiée.

Indication OCaml : Nous représentons une antichaîne par le type `int list`.

☐ 15 – Modifier la fonction `is_chain` introduite à la question 12 afin d'écrire une fonction OCaml `is_antichain` obéissant à la spécification : la fonction `is_antichain r a` teste si la liste de sommets a est une antichaîne pour la relation d'ordre r. En déduire que la fonction `is_antichain` admet pour complexité en temps la même complexité que celle obtenue à la question 13.

☐ 16 – Confirmer ou réfuter l'existence d'une fonction `is_antichain_bis r a` obéissant à la même spécification que celle de la question 15 et de complexité en temps $O(\alpha \log_2 \alpha)$, où α est la longueur de la liste a.

3. Couverture par des chaînes

Définition : Nous disons qu'une famille finie $(C_\ell)_{1\leqslant \ell \leqslant \lambda}$ de chaînes de l'ensemble ordonné $(V, \trianglelefteq)$ est une *couverture* si la réunion $\bigcup_{1\leqslant \ell \leqslant \lambda} C_\ell$ est égale à l'ensemble V.

☐ 17 – Soit $(C_\ell)_{1\leqslant \ell \leqslant \lambda}$ une couverture de l'ensemble ordonné $(V, \trianglelefteq)$ et soit A une antichaîne de $(V, \trianglelefteq)$ de cardinal α. Démontrer l'inégalité

$$\alpha \leqslant \lambda.$$

Définition : Étant donné un ensemble ordonné $(V, \trianglelefteq)$, nous appelons *graphe des poursuites d'un ensemble ordonné* $(V, \trianglelefteq)$ le graphe biparti non orienté $P = (W_0 \sqcup W_1, F)$ où les ensembles de sommets W_0 et W_1 sont deux copies distinctes de V, dont on note les éléments

$$W_0 = \{w_i\,;\ i \in [\![0, n-1]\!]\} \qquad \text{et} \qquad W_1 = \{w_{n+i}\,;\ i \in [\![0, n-1]\!]\},$$

et dont l'ensemble des arêtes est

$$F = \{(w_i, w_{n+j}) \text{ où } (i,j) \in [\![0, n-1]\!]^2,\ i \neq j, \ \text{ et } v_i \trianglelefteq v_j\}.$$

Indication OCaml : Nous représentons un graphe biparti à $2n$ sommets par un tableau de longueur $2n$ contenant des listes de voisins

```
24. type bipartite = int list array
```

Chaque arrête est notée deux fois en mémoire.

☐ 18 – Écrire une fonction OCaml `bipartite_of_order (r:order) : bipartite` dont la valeur de retour est le graphe des poursuites tiré de la relation d'ordre r.

Définition : Nous appelons *couplage* tout ensemble d'arêtes dont les extrémités sont distinctes.

☐ 19 – Pour tout couplage M du graphe des poursuites $P = (W_0 \sqcup W_1, F)$, nous construisons la partition $(C_\ell)_{1\leqslant \ell \leqslant \lambda}$ de l'ensemble V telle que

1. pour tous indices i et j compris entre 0 et $n-1$, si le couple $(w_i, w_{n+j}) \in W_0 \times W_1$ est une arête de M, alors les deux éléments v_i et v_j appartiennent à une même part C_ℓ,
2. le cardinal λ est aussi grand que possible.

Montrer que pour tout indice ℓ compris entre 1 et λ, l'ensemble C_ℓ est une chaîne de $(V, \trianglelefteq)$.

Épreuve d'informatique d'option MP 2024

Indication OCaml : Nous représentons un couplage dans le graphe des poursuites $P = (W_0 \sqcup W_1, F)$ par un tableau m de longueur n, de type `int array`, tel que $m[j] = i$ si le couple $(w_i, w_{n+j}) \in W_0 \times W_1$ est une arête du couplage et $m[j] = -1$ si le sommet $w_{n+j} \in W_1$ n'est pas l'extrémité d'une arrête du couplage.

☐ 20 – Déduire de la question 19 une fonction OCaml `chains_of_matching (m:int array) : int list list` dont la valeur de retour est une liste de chaînes disjointes tirée du couplage m.

☐ 21 – Une couverture $(C_\ell)_{1 \leqslant \ell \leqslant \lambda}$ de l'ensemble ordonné $(V, \trianglelefteq)$ par des chaînes disjointes étant fixée, expliquer comment construire un couplage M tel que la construction de la question 19 conduit à la partition $(C_\ell)_{1 \leqslant \ell \leqslant \lambda}$.

☐ 22 – Montrer que la construction de la question 19 fournit une couverture de l'ensemble V par un nombre minimum de chaînes si et seulement si elle s'applique à un couplage M de cardinal maximum.

☐ 23 – Écrire une fonction OCaml `maximum_matching (p:bipartite) : int array` qui calcule un couplage de cardinal maximum dans le graphe biparti p.

☐ 24 – Écrire une fonction OCaml `minimum_cover (r:order) : int list list` qui calcule une couverture par une famille de chaînes disjointes de cardinal minimum.

☐ 25 – Déterminer la complexité en temps de la fonction `minimum_cover r` en fonction du nombre d'éléments n ordonnés par la relation r.

A. Annexe : aide à la programmation en OCaml

Opérations sur les tableaux : Le module `Array` offre les fonctions suivantes :

— `length : 'a array -> int`
Return the length (number of elements) of the given array.

— `make : int -> 'a -> 'a array`
`Array.make n x` returns a fresh array of length n, initialized with x. All the elements of this new array are initially physically equal to x (in the sense of the == predicate). Consequently, if x is mutable, it is shared among all elements of the array, and modifying x through one of the array entries will modify all other entries at the same time.

— `make_matrix : int -> int -> 'a -> 'a array array`
`Array.make_matrix dimx dimy e` returns a two-dimensional array (an array of arrays) with first dimension $dimx$ and second dimension $dimy$. All the elements of this new matrix are initially physically equal to e. The element (x, y) of a matrix m is accessed with the notation `m.(x).(y)`.

— `init : int -> (int -> 'a) -> 'a array`
`Array.init n f` returns a fresh array of length n, with element number i initialized to the result of $f(i)$. In other terms, `init n f` tabulates the results of f applied to the integers 0 to $n-1$.

— `copy : 'a array -> 'a array`
`Array.copy a` returns a copy of a, that is, a fresh array containing the same elements as a.

— `mem : 'a -> 'a array -> bool`
`mem a l` is true if and only if a is structurally equal to an element of l (i.e. there is an x in l such that `compare a x = 0`).

— `for_all : ('a -> bool) -> 'a array -> bool`
`Array.for_all f [|a1; ...; an|]` checks if all elements of the array satisfy the predicate f. That is, it returns `(f a1) && (f a2) && ... && (f an)`.

— `exists : ('a -> bool) -> 'a array -> bool`
`Array.exists f [|a1; ...; an|]` checks if at least one element of the array satisfies the predicate f. That is, it returns `(f a1) || (f a2) || ... || (f an)`.

— `map : ('a -> 'b) -> 'a array -> 'b array`
`Array.map f a` applies function f to all the elements of a, and builds an array with the results returned by f : `[| f a.(0); f a.(1); ...; f a.(length a - 1) |]`.

— `mapi : (int -> 'a -> 'b) -> 'a array -> 'b array`
Same as `Array.map`, but the function is applied to the index of the element as first argument, and the element itself as second argument.

— `iter : ('a -> unit) -> 'a array -> unit`
`Array.iter f a` applies function f in turn to all the elements of a. It is equivalent to `f a.(0); f a.(1); ...; f a.(length a - 1); ()`.

— `iteri : (int -> 'a -> unit) -> 'a array -> unit`
Same as `Array.iter`, but the function is applied to the index of the element as first argument, and the element itself as second argument.

D'après `https://v2.ocaml.org/api/Array.html`

Fin de l'épreuve

CONCOURS CENTRALE•SUPÉLEC

Informatique

MPI

2024

4 heures Calculatrice autorisée

Le jeu de go

I Introduction au jeu de go

I.A – Présentation du sujet

Ce sujet s'intéresse au jeu de go et à différentes facettes de la construction d'un programme jouant au jeu. La partie 1 présente des règles simplifiées du jeu de go qui seront utilisées tout au long de ce sujet, et qui seront particulièrement utiles aux parties 2 et 5. Plusieurs fonctions permettant de mener une partie de go seront à implémenter en langage **C** dans la partie 2. Les parties suivantes se penchent davantage sur la dimension stratégique du jeu. La partie 3 s'intéresse à l'estimation d'un coup pertinent à partir d'une base de données, en utilisant l'algorithme des **k plus proches voisins**, puis en mettant en place une **table de hachage** contenant des coups localement similaires, en langage **C**. Cette recherche de coup se poursuit en partie 4 à travers une **Recherche Arborescente de Monte-Carlo** utilisant des **fils d'exécutions concurrents**, en **OCaml**. Enfin, la partie 5 conclut ce sujet en montrant que décider si une position est gagnante ou non dans un jeu de go généralisé est un problème **NP-dur**.

I.B – Présentation et règles du jeu de go

Le jeu de *go* est un jeu de stratégie opposant deux adversaires, qui placent à tour de rôle des pierres sur un plateau quadrillé appelé *goban*. Ce plateau est modélisé par une grille de d lignes par d colonnes, avec d un entier appelé la *dimension* du goban. Dans ce sujet nous nous intéresserons principalement à des gobans standards de dimension $d = 19$. Chacune des d^2 intersections du goban est identifiée par sa *position*, qui est un couple $(ligne, colonne)$ représentant son emplacement sur le plateau.

Les deux joueurs sont nommés d'après la couleur des pierres qu'ils placent sur le goban : Noir, qui joue en premier et place des pierres noires, et Blanc qui place des pierres blanches. À chaque tour, un joueur place une pierre de sa couleur sur une intersection *libre* du goban, c'est-à-dire sans pierre. Un coup correspond donc simplement à l'ajout d'une pierre sur une intersection libre, les joueurs ne peuvent pas déplacer une pierre déjà posée.

Nous appellerons *configuration* du goban, ou simplement *goban*, l'état du plateau à un moment de la partie, c'est-à-dire la donnée des positions et couleurs de toutes les pierres placées sur le goban. La figure 1 illustre différentes configurations progressives au cours d'une partie possible de go.

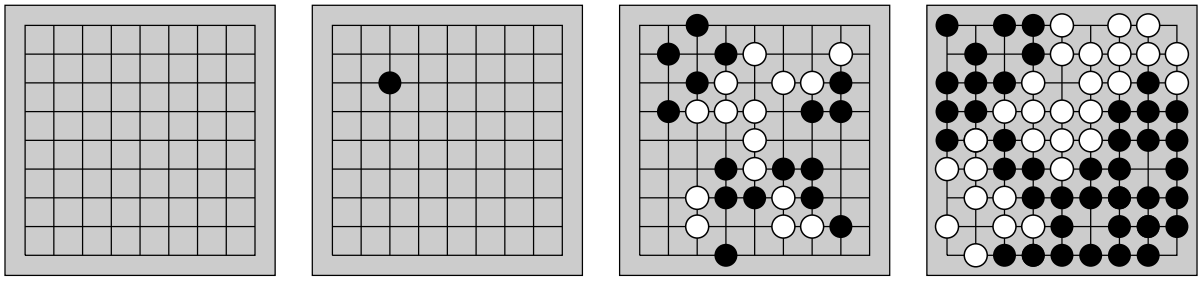


Figure 1 Un *goban* vide de dimension 9, un premier coup de Noir, une configuration du jeu à un moment intermédiaire, et une configuration à la *fin* de la partie dans laquelle chaque intersection est *contrôlée* par un des deux joueurs.

Chaque intersection possède plusieurs *intersections voisines* qui sont les intersections orthogonalement adjacentes à cette dernière (et non diagonalement adjacentes). Une intersection a donc au plus 4 intersections voisines, celles sur les extrémités du goban pouvant en avoir moins. Par extension, les intersections voisines d'une pierre sont celles de l'intersection où elle est placée.

Deux pierres d'une même couleur sont dites *connectées* si elles se trouvent sur des intersections voisines. Nous appellerons *groupe* de pierres une composante connexe pour cette relation de connexion. Chaque groupe possède un nombre de *libertés* qui est défini comme le nombre d'intersections libres qui sont voisines d'une pierre du groupe. Une intersection voisine de plusieurs pierres du groupe n'est comptée qu'une seule fois.

Un groupe de pierres auquel on vient de supprimer la dernière liberté est *capturé*, et les pierres de ce groupe sont alors immédiatement retirées du plateau. Ces notions de libertés et de capture sont illustrées dans la figure 2.

BY-NC-SA

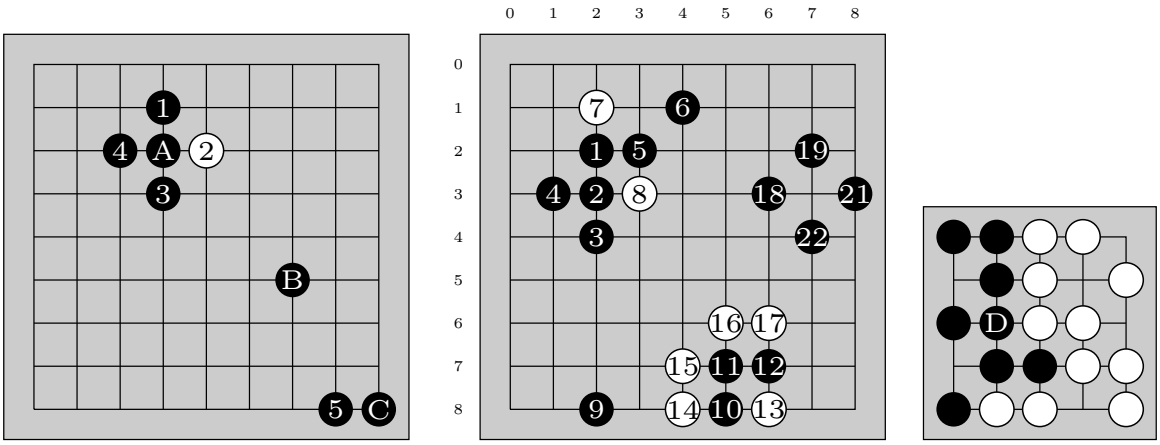


Figure 2 À gauche : la pierre A a une pierre sur chacune de ses 4 *intersections voisines*. La pierre B n'a pas de pierres sur ses 4 intersections voisines. La pierre C a une pierre sur une de ses deux intersections voisines. Au milieu : les pierres 1, 2, 3, 4 et 5 forment un *groupe* de pierres noires, la pierre 6 ne fait pas partie de ce groupe, ni les pierres blanches 7 et 8. Ce groupe a 7 *libertés*. La pierre 9 a 3 libertés. Le groupe de pierre 10, 11, 12 a une seule liberté, si Blanc joue en (7, 7) alors il *capture* immédiatement ce groupe, et les pierres 10, 11, 12 sont retirées du plateau. L'intersection (3, 7) est *contrôlée* par noir. À droite : un exemple de goban de dimension 5 en fin de partie.

Cette règle de capture permet d'aboutir à des configurations avec moins de pierres sur le goban. Afin de s'assurer que la partie termine, une règle supplémentaire interdit de jouer un coup qui fasse revenir à une configuration du goban déjà vue durant la partie.

Lorsqu'une pierre est placée, on commence par regarder si elle permet de capturer des pierres adverses, et dans ce cas on retire immédiatement ces pierres capturées du goban. On regarde ensuite si le groupe de la pierre placée possède des libertés. Si elle ne possède aucune liberté, le coup est interdit. Cependant, si elle a permis de capturer des pierres, son groupe a nécessairement au moins une liberté. Il est donc possible de jouer sur une position sans libertés, mais uniquement dans le cas où cela permet de capturer des pierres à l'adversaire.

Une intersection est dite *contrôlée* par un joueur si une de ses pierres est placée dessus, ou si l'intersection est vide mais que toutes les intersections voisines contiennent des pierres de sa couleur.

La partie s'arrête lorsque toutes les intersections sont contrôlées par un des deux joueurs. Une telle configuration du goban est alors dite *finale*. Le *score* de chaque joueur est défini comme le nombre d'intersections contrôlées par ce dernier. Le joueur avec le score le plus élevé remporte la partie.

I.C – Généralités sur le jeu de go

Q 1. Compter le nombre de libertés du groupe auquel appartient la pierre D dans le goban de la figure 2 Droite. Donner le score de Blanc, celui de Noir et le joueur gagnant de cette partie.

Q 2. Montrer qu'il y a toujours un gagnant dans une configuration finale sur un goban standard de dimension 19.

II Implémentation du jeu de go

L'objectif de cette première partie est d'implémenter diverses fonctions permettant de manipuler le goban pour mener une partie de go. Ces fonctions sont à écrire en langage **C**. Dans tout le sujet, il sera supposé que les entêtes suivantes ont été incluses :

```
#include <assert.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
```

Afin de représenter et manipuler un goban, nous utiliserons la structure de données suivante pour représenter une configuration.

```
1   struct goban_s {
2           int d;
3           int* m;
4   };
5   typedef struct goban_s goban;
```


BY-NC-SA

Le champ `d` est ici la dimension du goban, et `m` est un pointeur vers un tableau unidimensionnel d'entiers de taille $d \times d$ représentant les pierres placées. Une intersection libre est représentée par l'entier 0, une pierre noire par l'entier 1 et une pierre blanche par l'entier 2. Initialement, le tableau est donc rempli de 0.

Q 3. Écrire une fonction `goban initialisation(int d)` prenant en argument un entier `d` et qui renvoie un goban de dimension `d` avec un tableau `m` alloué dynamiquement, représentant un goban vide.

La position d'une intersection du goban sera représentée par une structure de couple d'entiers, contenant la ligne i et la colonne j de la position, avec $0 \leqslant i, j < d$.

```
1   struct couple_int {
2           int i;
3           int j;
4   };
5   typedef struct couple_int position;
```

Une position pourra être initialisée avec la fonction suivante :

```
1   position pos(int i, int j) {
2           position p = {.i = i, .j = j};
3           return p;
4   }
```

Un goban pourra être lu et modifié grâce aux deux fonctions ci-dessous, en supposant que la position correspond à une intersection valide du goban.

```
1   int lire(goban g, position p) {
2           return g.m[p.i * g.d + p.j];
3   }
4
5   void ecrire(goban g, position p, int couleur) {
6           g.m[p.i * g.d + p.j] = couleur;
7   }
```

Les questions suivantes vont permettre d'isoler un groupe de pierres. Pour cela, nous représentons les intersections voisines d'une intersection donnée grâce à la structure suivante :

```
1   struct voisins_s {
2           int nb;
3           position t[4];
4   };
5   typedef struct voisins_s voisins;
```

Le nombre de voisins de l'intersection est donné par le champ `nb`, et `t` est un tableau de positions de taille 4, dont les `nb` premières cases contiennent les positions des intersections voisines de l'intersection. La fonction incomplète suivante permet ce calcul des voisins :

```
1   voisins pos_voisins(goban g, position p) {
2           int i = p.i;
3           int j = p.j;
4           voisins v;
5           ...
6           if (...) {v.t[k] = pos(i - 1, j); k = k + 1;}
7           if (...) {v.t[k] = pos(i + 1, j); k = k + 1;}
8           if (...) {v.t[k] = pos(i, j - 1); k = k + 1;}
9           if (...) {v.t[k] = pos(i, j + 1); k = k + 1;}
10          v.nb = k;
11          return v;
12  }
```


BY-NC-SA

Q 4. Compléter cette fonction `pos_voisins` en donnant l'instruction à écrire en ligne 5 ainsi que les conditions à écrire pour chacune des lignes 6, 7, 8 et 9 pour qu'elle renvoie, dans une structure `voisins` les positions des intersections voisines de la position donnée en entrée.

Q 5. Écrire une fonction `goban groupe(goban g, position p)` qui renvoie un nouveau goban dans lequel seules les pierres du groupe de la pierre en position `p` sont présentes. Expliquer par une phrase, avant le code de votre fonction, l'approche choisie. Il pourra être utile de commencer par implémenter une fonction auxiliaire récursive, dont la spécification sera clairement explicitée. Donner la complexité obtenue.

Une première tentative pour calculer le nombre de libertés du groupe de la pierre en position `p` a abouti sur la fonction suivante :

```
1   int liberte(goban g, position p) {
2           goban g1 = groupe(g, p);
3           int cpt = 0;
4           int couleur = lire(g, p);
5           for (int i = 0; i < g.d; i = i + 1) {
6                   for (int j = 0; j < g.d; j = j + 1) {
7                           position p2 = pos(i, j);
8                           if (lire(g1, p2) == couleur) {
9                                   voisins v = pos_voisins(g1, p2);
10                                  for (int k = 0; k < v.nb; k = k + 1) {
11                                          if (lire(g, v.t[k]) == 0) {
12                                                  cpt = cpt + 1;
13                                          }
14                                  }
15                          }
16                  }
17          }
18          return cpt;
19  }
```

Q 6. Proposer un jeu de quatre tests permettant de tester cette fonction `liberte`. Deux tests seront passés par la fonction, et deux montreront qu'elle n'est pas correcte. Les gobans en entrées pourront être représentés par des schémas. On prendra soin de justifier l'intérêt des tests proposés et d'indiquer les valeurs attendues.

Q 7. Proposer des modifications à apporter au code de la fonction `liberte` pour qu'elle renvoie bien le nombre de libertés, et qu'elle libère les structures créées pendant son appel.

Q 8. Écrire une fonction `void retire_groupe(goban g, position p)` prenant en argument un goban, une position et qui retire toutes les pierres du groupe de la pierre à cette position. Une complexité linéaire en la taille du groupe à retirer est attendue.

Q 9. Écrire une fonction `void joue(goban g, position p, int couleur)` prenant en argument un goban, une position et une couleur et qui modifie le goban en ajoutant une pierre de la couleur donnée à la position donnée. On prendra en compte les possibles pierres à retirer et on utilisera des assertions pour vérifier que le coup est valable. On ne vérifiera cependant pas si la configuration du goban a déjà été rencontrée au cours de la partie. Donner la complexité de cette fonction.

III Prédiction et évaluation d'un prochain coup

Dans cette partie, nous envisageons de réaliser un programme qui renvoie ou évalue un prochain coup pertinent à jouer depuis une configuration donnée du goban, afin de simuler un adversaire. La dimension stratégique du jeu de go est réputée pour être complexe, c'est pourquoi une première approche simple peut être de s'inspirer de coups joués par des joueurs humains professionnels.

Pour cela nous nous baserons sur une base de données de 160 000 parties de maîtres, jouées sur des gobans standards de dimension 19. Nous avons commencé par décomposer ces parties en 29.4 millions de triplets (configuration du goban, joueur dont c'est le tour, coup suivant choisi), que nous appellerons *observations*, et qui peuvent être redondantes.

Q 10. À partir des informations sur la base de données, donner une estimation de la profondeur du jeu de go, c'est-à-dire du nombre de coups moyen d'une partie de go. En utilisant la description du jeu ainsi que la profondeur, donner une estimation de la largeur du jeu de go, c'est-à-dire le nombre moyen de coups valables lors d'une partie.
Est-il envisageable de mettre en place un algorithme *min-max* pour trouver une stratégie gagnante à ce jeu ? Justifier.

BY-NC-SA

III.A – K plus proches voisins

Afin de jouer en un temps raisonnable, dans cette sous-partie nous allons implémenter l'algorithme des *k plus proches voisins* pour prédire un prochain coup pertinent de manière supervisée. Pour cela, nous allons confronter la configuration d'entrée sur laquelle nous désirons trouver le prochain coup à toutes les configurations compatibles de la base de données, c'est-à-dire les configurations dont le coup suivant qui a été joué peut effectivement être réalisé. Le calcul d'une distance entre deux configurations du goban ne sera pas étudié ici, nous supposerons que nous disposons d'un tableau non trié dans lequel ces n distances ont été préalablement calculées. Dans cette sous-partie, pour simplifier les algorithmes et les programmes nous nous intéresserons seulement à déterminer les k plus petites valeurs de ce tableau de distances non trié, sans se préoccuper de garder en mémoire les configurations auxquelles correspondent ces distances.

Q 11. Proposer une fonction `double* k_plus_petits(double* distances, int n, int k)` qui prend en entrée un tableau de `n` distances et un entier `k` et renvoie un tableau de `k` distances correspondant aux `k` plus petites valeurs du tableau en entrée. Il est attendu ici une complexité en $O(nk)$ dans le pire cas. La fonction pourra, si besoin, modifier le tableau donné en entrée. Justifier la complexité obtenue et prouver la correction de cet algorithme à l'aide d'un invariant de boucle.

Q 12. Proposer une structure de données abstraite qui permette d'améliorer la complexité asymptotique de la fonction `k_plus_petits`. Rappeler en une phrase une façon d'implémenter cette structure de données ainsi que la complexité obtenue.

Dans le cadre d'un adversaire jouant en temps réel, il est préférable d'avoir une borne précise sur le nombre de comparaisons qui seront réalisées. Nous supposerons désormais que n est une puissance de 2 avec $n \geqslant 2$. Nous allons construire un algorithme permettant de trouver les indices des deux plus petits éléments d'un tableau de n distances. Il sera possible de supposer que les distances de ce tableau sont toutes différentes.

Nous appellerons comparatif un algorithme prenant en entrée un tableau de distances pour lequel la seule opération autorisée sur les distances du tableau est la comparaison entre deux distances du tableau.

Q 13. Montrer que tout algorithme comparatif renvoyant l'indice du plus petit élément d'un tableau devra réaliser la comparaison du plus petit élément et du second plus petit élément du tableau.

Q 14. Proposer un algorithme qui renvoie les indices des deux plus petits éléments d'un tableau de n distances en réalisant au plus $n + \log_2(n) - 2$ comparaisons. On pourra s'inspirer d'une approche de type diviser pour régner et commencer par comparer chaque élément d'indice $2i$ avec l'élément d'indice $2i + 1$, pour $0 \leqslant i < n/2$.

Il est ensuite possible d'adapter cette approche à la sélection des k plus petits éléments.

III.B – Hachage de Zobrist

Le but de cette sous-partie est de présenter une autre approche utilisant la base de données des parties de maîtres, cette fois pour apprendre une fonction d'évaluation des prochains coups qui s'intéresse uniquement à l'état local du goban autour d'une position envisagée. Nous chercherons, dans la base de données, si des configurations localement identiques ont mené à un coup à cette position. Cette recherche sera implémentée de manière efficace dans cette partie grâce à un prétraitement de la base de données utilisant des tables de hachage et une fonction de hachage adaptée, nommée *hachage de Zobrist*.

Un état local sera modélisé par différents *zooms* autour d'une position, qui considèrent la disposition du plateau autour d'une intersection en ignorant le reste du goban, appelée *motif*. Nous noterons $Z_i(g, (x_0, y_0))$ le motif obtenu à partir du zoom i sur la position (x_0, y_0) du goban g comme étant les couleurs des pierres aux positions (x, y) telles que $|x - x_0| + |y - y_0| \leqslant i$, et avec $(x - x_0, y - y_0) \neq (0, 0)$. Par exemple, $Z_1(g, p)$ correspond à la configuration des intersections voisines de l'intersection de position p. Nous dénoterons par $(x - x_0, y - y_0)$ la position du goban (x, y) dans ce motif $Z_i(g, (x_0, y_0))$. Pour avoir toujours le même nombre d'éléments dans les motifs, une couleur supplémentaire 3 sera assignée aux positions qui dépassent les bords du goban. Les intersections libres seront représentées par 0, celles avec une pierre noire par 1 et celles avec une pierre blanche par 2.

Le but d'un zoom sur une position est d'évaluer un coup potentiel à cette position. Étant donné un motif centré sur une intersection libre sur laquelle il est possible de jouer, nous allons chercher dans la base de données si des configurations possédant ce même motif ont mené à un coup à cette position. Nous considérerons toujours que le centre du motif est une intersection vide.

Q 15. Donner une borne supérieure sur le nombre de motifs distincts pour un zoom i. Justifier qu'il est raisonnable de supposer que tout motif de zoom 1 a été rencontré au moins une fois dans la base de données. On pourra faire cette supposition dans le reste du sujet.

Les fonctions de cette partie sont à écrire en langage **C**, les configurations seront manipulées avec les mêmes structures que celles de la partie 2, et les fonctions qui y sont définies peuvent être librement réutilisées.

Pour savoir combien de fois un motif a conduit à un coup en son centre, nous allons construire et remplir une table de hachage, pour chaque niveau de zoom. La fonction de hachage associée assignera à chaque motif un entier non signé de 64 bits, de type `uint64_t`. Cette fonction de hachage particulière se base sur une table d'entiers `uint64_t` fixés, ayant autant de colonnes que de positions dans le motif, et quatre lignes pour les quatre

BY-NC-SA

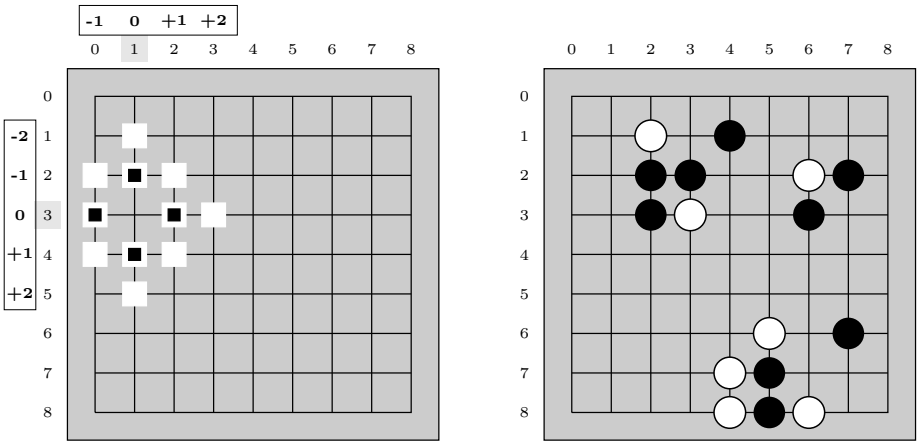


Figure 3 Gauche : l'étendue du zoom 1 (les 4 intersections avec un carré noir), et du zoom 2 (les 11 intersections avec des carrés blancs, et une intersection extérieure à la grille non représentée) sur la position $(3,1)$. La case $(2,1)$ est en position $(-1,0)$ dans les zooms sur la position $(3,1)$. Droite : un exemple de goban. Les motifs correspondant au zoom 1 en position $(2,4)$ et au zoom 1 en position $(3,7)$ sont identiques, ce motif est ici observé deux fois dans cette configuration. Si Noir choisit de jouer en $(2,4)$ on dira alors que ce motif a été choisi une fois.

couleurs possibles (libre, noir, blanc et hors du goban). Afin de manipuler à la main un nombre raisonnable de bits, sur la table donnée en exemple en figure 4, nous utilisons des entiers codés sur 8 bits et non sur 64 bits

La valeur de hachage est alors calculée comme le *ou exclusif* (XOR) bit à bit, noté $\oplus$, des entiers des cases de toutes les combinaisons position-couleur se trouvant dans le motif. La table de vérité du XOR pour un bit est rappelée en figure 4.

Formellement, en notant A la fonction qui associe à un couple de couleur et de position dans le motif l'entier fixé associé dans la table, nous définissons la fonction de hachage d'un motif h par :

$$h(Z_i(g,(x_0,y_0))) = \bigoplus_{\substack{(x_u,y_u)\in\mathbb{Z}^2\setminus\{(0,0)\} \\ \text{tels que } |x_u|+|y_u|\leqslant i}} A(g(x_0+x_u,y_0+y_u),(x_u,y_u))$$

a	b	a XOR b
0	0	0
1	0	1
0	1	1
1	1	0

Couleur et position	$(0,+1)$	$(+1,0)$	$(0,-1)$	$(-1,0)$
0 (vide)	00001111	11100111	00011000	01111011
1 (noir)	11110110	11111110	01101001	10011111
2 (blanc)	10100001	10001101	00110111	00110100
3 (au delà du bord)	10100000	01101010	10111011	11000010

Figure 4 Table de vérité du XOR, et une table d'entiers fixés sur 8 bits associés à des couples couleur et position dans le motif. Par exemple une pierre noire en position (+1,0) dans un motif est associée à 11111110.

Q 16. Dans le goban de la figure 3 Droite, et avec le tableau de valeurs sur 8 bits donné dans la figure 4, calculer la valeur de hachage du motif en position (7,6) pour le zoom 1. Expliquer comment il est possible de mettre à jour cette valeur en calculant seulement 2 XOR supplémentaires lorsque l'on joue une pierre blanche en (6,6).

Nous disposons désormais d'une fonction `uint64_t hachage(position p, goban g, int zoom)` qui calcule la valeur de hachage $h(Z_i(g,p))$ étant donnée une position `p`, un goban `g` et un zoom `zoom`.

De plus, en connaissant la valeur de hachage h_1 du motif $Z_i(g_1,p)$, nous disposons d'une fonction `uint64_t maj_hachage(position p, goban g1, goban g2, int zoom, uint64_t h1)` qui permet de calculer efficacement, en terme de nombre de XOR effectués, la valeur $h(Z_i(g_2,p))$ en mettant à jour $h(Z_i(g_1,p))$ à partir des différences entre les gobans `g1` et `g2`.

Nous nous intéressons maintenant uniquement aux choix des coups de Noir. Il nous faut dénombrer les motifs observés et les motifs choisis dans la base de données de parties. Pour un niveau de zoom fixé, nous stockerons ces valeurs dans une table de hachage, associant un motif à son nombre d'*observations* et de *choix*.

Pour un niveau de zoom fixé, dans une configuration du goban où c'est au tour de Noir de jouer, nous *observons* un motif par position libre de cette configuration. Noir va alors jouer dans une de ces positions libres, et on dit que le motif associé à la position libre où Noir joue est *choisi* (voir la figure 3).

BY-NC-SA

Pour chaque niveau de zoom, les valeurs stockées dans sa table de hachage associée seront représentées par la structure suivante :

```
1   struct observation_s {
2           int n; //nombre d'observations du motif au total
3           int v; //nombre de fois ou ce motif a ete choisi
4   };
5   typedef struct observation_s observation;
```

Le champ `n` correspond au nombre d'observations du motif dans des gobans où c'était à Noir de jouer. Le champ `v` correspond au nombre de fois où le motif était observé au moment où c'était à Noir de jouer et où le coup de Noir a été de placer une pierre au centre de ce motif.

Nous définissons alors une structure de table de hachage composée d'un tableau d'observations, et d'un entier `nt` correspondant à sa taille.

```
1   struct table_de_hachage_s {
2           int nt;
3           observation* t;
4   };
5   typedef struct table_de_hachage_s t_hachage;
```

La table de hachage est initialisée avec $n = 0$ et $v = 0$, pour chaque case. Pour accéder à la valeur de la table associée à une clé, nous prendrons la clé modulo `nt`. Ce sujet ne s'intéresse pas aux collisions, et nous pourrons supposer qu'il n'y en a aucune.

Pour remplir la table de hachage, nous allons simuler la partie de maître configuration après configuration, en utilisant la fonction `maj_hachage` pour mettre à jour les valeurs de hachages. Au départ, nous partons donc de la configuration initiale du goban, et nous allons garder en mémoire les valeurs de hachage pour chaque position du goban, et les faire évoluer au fur et à mesure de la partie.

Q 17. Écrire une fonction `uint64_t* init_t_hash(int d, int zoom)` qui prend en entrée un entier `d` correspondant à la dimension d'un goban, et un entier `zoom` correspondant au niveau de zoom d'intérêt, et qui renvoie un tableau d'entiers non signés sur 64 bits alloué dynamiquement, de taille $d \times d$ où chaque case correspond à la valeur de hachage initiale sur cette position dans un goban vide.

On considère un goban `gN` où Noir doit jouer, le goban suivant `gB` après le coup de Noir, et le goban après le coup de Blanc, quand Noir doit jouer à nouveau `gNs`. La fonction `maj_coup`, donnée ci-dessous, met à jour dans toutes les positions possibles du goban, si nécessaire, le nombre d'observations des motifs en prenant en compte le goban `gN`, avec la fonction `maj_n`, le nombre de choix des motifs avec la fonction `maj_v`, en s'intéressant au coup choisi par Noir entre `gN` et `gB`. Elle prend notamment en entrée `tableau_hache` qui contient les valeurs de hachage des positions du goban `gN`. La fonction `maj_t_hache`, donnée ci-dessous, met à jour les valeurs de hachage associées aux positions avec la fonction `maj_h`, le tableau en entrée `tableau_hache` contient les valeurs de hachage des positions du goban `gN` et doit être à jour sur le goban `gNs` à l'issue de l'exécution de la fonction.

```
1   void maj_coup(t_hachage table, uint64_t* tableau_hache, goban gN, goban gB, int zoom) {
2           for (int i = 0; i < gN.d; i = i + 1) {
3                   for (int j = 0; j < gN.d; j = j + 1) {
4                           position p = pos(i, j);
5                           maj_n(table, tableau_hache, gN, p);
6                           maj_v(table, tableau_hache, gN, gB, p);
7                   }
8           }
9   }
10
11  void maj_t_hache(uint64_t* tableau_hache, goban gN, goban gNs, int zoom) {
12          for (int i = 0; i < gN.d; i = i + 1) {
13                  for (int j = 0; j < gN.d; j = j + 1) {
14                          position p = pos(i, j);
15                          maj_h(tableau_hache, gN, gNs, zoom, p);
16                  }
17          }
18  }
```

Q 18. Écrire les fonctions

BY-NC-SA

`void maj_n(t_hachage table, uint64_t* tableau_hache, goban gN, position p)`,

`void maj_v(t_hachage table, uint64_t* tableau_hache, goban gN, goban gB, position p)`,

`void maj_h(uint64_t* tableau_hache, goban gN, goban gNs, int zoom, position p)`,

pour que les fonctions `maj_coup` et `maj_t_hache` mettent correctement à jour la table de hachage et le tableau des valeurs de hachage.

Une partie de go sera représentée par une structure de liste simplement chaînée, chaque coup étant représenté par le goban actuel, le joueur qui doit jouer, et un pointeur vers le reste de la partie. Nous supposerons que les parties commencent par le goban vide associé au joueur d'indice 1 (Noir).

```
1   struct partie_s {
2           goban g;
3           int joueur;
4           struct partie_s* suivant;
5   };
6   typedef struct partie_s partie;
```

Q 19. En s'intéressant uniquement aux coups de Noir, écrire une fonction `t_hachage lit_partie(partie* lp, int zoom, t_hachage table)` qui prend en entrée une partie et met à jour pour le zoom donné et dans toutes les positions libres du goban le nombre d'occurrences des motifs, et le nombre de choix des motifs choisis, dans la table de hachage donnée en entrée (qui a déjà été initialisée et mise à jour sur d'autres parties). On pourra supposer que la partie contient au moins 3 coups.

Q 20. Étant donné un tableau des valeurs de hachage initialisé avec le goban initial, montrer qu'au plus $4c$ XOR sont nécessaires pour calculer toutes les valeurs de hachage d'un motif à cette position au cours d'une partie de c coups.

Nous allons représenter toutes les tables de hachages pour les différents niveaux de zoom allant de 1 à $n_z - 1$, inclus, dans une table de tables de hachage, représentée par la structure suivante :

```
1   struct table_table_de_hachage {
2           int nz;
3           t_hachage* t;
4   };
5   typedef struct table_table_de_hachage tt_hachage;
```

Nous construisons cette table de tables de hachages `tt` telle que `tt[i]` contienne la table de hachage associée au zoom `i`. La première case de ce tableau, `tt[0]`, ne sera ainsi pas utilisé.

Pour évaluer la pertinence d'un coup dans une configuration, nous allons chercher le plus grand zoom pour lequel le motif a été observé au moins une fois dans la base de données. La qualité estimée de ce coup est alors le nombre de fois où ce motif a été choisi sur le nombre de fois où il a été vu.

Q 21. Écrire une fonction booléenne `bool appartient(position p, goban g, t_hachage table, int zoom)` qui renvoie vrai si le motif en position `p` et avec le zoom `zoom` sur le goban `g` est présent dans la table de hachage, c'est-à-dire a été observé au moins une fois dans la base de données, et faux sinon.

Q 22. Écrire une fonction `float evalue_zoom(position p, goban g, t_hachage table, int zoom)` qui renvoie la valeur associée au motif, c'est-à-dire le rapport entre le nombre d'observations et le nombre de choix.

Q 23. Écrire une fonction `float evaluation(position p, goban g, tt_hachage tt)` qui renvoie l'évaluation du coup en position `p` sur le goban `g`. Une fonction de complexité logarithmique en le nombre de zooms n_z est attendue. Donner une preuve de la terminaison et de la correction partielle de ce programme, et justifier sa complexité.

IV Recherche arborescente de Monte-Carlo

Dans cette partie nous nous intéressons à la construction d'un programme en OCaml jouant au go, en utilisant l'approche de Recherche Arborescente de Monte-Carlo. C'est notamment sur cette méthode que s'est basé AlphaGo pour vaincre le champion du monde de go en 2017. Nous supposerons dans cette section que les parties de go finissent toujours dans une configuration finale, c'est-à-dire complètement contrôlée, sans égalité et que le goban est de dimension standard $d = 19$.

Les fonctions de cette partie sont à écrire en langage OCaml. Une annexe rappelant des éléments de langage **OCaml**, modules Thread et Mutex, module List, type enregistrement et fonction de conversion de type, est proposée en fin de partie.

Commençons par décrire quatre types pour représenter les joueurs, les gobans et les positions :

BY-NC-SA

```
type joueur = Noir | Blanc
type intersection = Libre | Pierre_noire | Pierre_blanche
type goban = intersection array array
type position = int * int
```

Nous supposerons que les fonctions suivantes ont été déjà implémentées et peuvent être librement utilisées.

— `liste_coup_prior : goban -> joueur -> (position * float) list` : prend un goban et un joueur et renvoie une liste, pour toutes les positions jouables dans le goban pour le joueur, de couples `(position, prior)` où `prior` est un flottant entre 0 et 1 correspondant à une estimation de la probabilité de victoire de Noir lorsque le joueur joue en position `position`.

— `strategie_defaut : goban -> joueur -> position` : prend un goban et un joueur et renvoie une position où jouer, en suivant une stratégie par défaut définie *a priori*, et qui peut contenir une part d'aléatoire. Cette fonction renvoie l'exception `Pas_de_coup_valide` lorsque le plateau est complètement contrôlé et qu'il n'y a plus de coup valide.

— `gagnant : goban -> joueur` : prend un goban complètement contrôlé et renvoie le joueur gagnant.

— `joue : goban -> position -> joueur -> goban` : prend un goban, une position et un joueur et renvoie le goban obtenu après que le joueur a joué en cette position. Le goban en entrée n'est pas modifié.

Q 24. Écrire une fonction de signature `js : joueur -> joueur` qui prend un joueur et renvoie le joueur suivant.

IV.A – Simulation de partie pour évaluer une configuration

Pour trouver un prochain coup pertinent, il est important de pouvoir évaluer si une configuration du goban est prometteuse ou non, ce qui est une tâche complexe. L'évaluation de Monte-Carlo permet d'estimer cette évaluation de goban, en jouant une partie depuis la configuration actuelle jusqu'au bout, et en choisissant les coups successifs grâce à une stratégie définie *a priori*. Celle-ci peut-être complètement aléatoire, ou basée sur des données d'apprentissage par exemple.

Q 25. Écrire une fonction de signature `sim_defaut : goban -> joueur -> joueur` qui prend un goban et un joueur, qui utilise la fonction `strategie_defaut` pour choisir des coups jusqu'à atteindre une configuration finale, et qui renvoie alors le joueur gagnant.

IV.B – Valeur d'action

Dans la recherche arborescente de Monte-Carlo, nous explorons l'arbre de toutes les parties possibles depuis le goban actuel, mais sans stocker explicitement cet arbre tout entier. De multiples explorations sont réalisées partant chacune de la racine de l'arbre et descendant jusqu'à une configuration finale, sans représentation explicite de l'arbre, en suivant des stratégies définies *a priori*. L'idée principale de la recherche arborescente de Monte-Carlo est d'ajouter à cette approche sans mémoire la construction d'un arbre partiel de jeu, dont les nœuds vont correspondre aux configurations souvent explorées, et pour lesquelles il sera possible d'enregistrer différentes informations :

— `n` : le nombre de visites du nœud, entier mutable

— `v` : le nombre de visites du nœud ayant mené à une victoire de Noir, entier mutable

— `prior` : la probabilité *a priori* de victoire de Noir donnée par la fonction `liste_coups_prior`, flottant

— `pos` : la position du coup associé au nœud, position

— `t` : un verrou, de type `t.Mutex`, qui sera utilisé pour mettre en place des parcours et mises à jour concurrentes de l'arbre.

Afin de stocker ces données, les nœuds de l'arbre seront étiquetés par le type enregistrement `etiquette`.

```
type etiquette =
        {mutable n:int; mutable v:int; prior:float; pos:position; t:Mutex.t}
```

Enfin, nous définissons le type suivant pour l'arbre partiel :

```
type arbre = N of etiquette * arbre list
```

La première étape de la recherche arborescente de Monte-Carlo est une descente qui consiste à suivre un chemin dans l'arbre partiel qui semble prometteur jusqu'à atteindre une feuille. Pour un nœud a de l'arbre, nous notons n_a, v_a, et prior_a les nombres de visites, de victoires et la prior, c'est-à-dire l'estimation à priori de victoire de Noir, du nœud a. Dans un nœud de l'arbre partiel n'étant pas une feuille, pour savoir quel enfant choisir nous définissions la quantité V_a^N appelée *valeur d'action* du nœud a pour Noir :

BY-NC-SA

$$V_a^N = \begin{cases} \dfrac{v_a}{n_a} + \dfrac{\text{prior}_a}{1+n_a} & \text{si } n_a > 0 \\ \text{prior}_a & \text{sinon} \end{cases}$$

Cette valeur est un compromis entre la prior apprise par apprentissage et le taux de victoires obtenu lors des descentes dans l'arbre, l'importance de la prior décroissant avec l'augmentation du nombre de descentes dans l'arbre passant par le nœud. Une valeur d'action analogue peut être définie pour Blanc. En partant de la racine de l'arbre partiel, la feuille à explorer est atteinte en descendant le long d'une branche, en choisissant itérativement comme nœud suivant le fils de valeur d'action maximale pour le joueur qui doit jouer.

Q 26. Écrire une fonction de signature `etiq : arbre -> etiquette` qui prend un arbre et renvoie l'étiquette de cette arbre.

Q 27. Écrire une fonction de signature `valeurActionNoir : arbre -> float` qui prend un arbre et renvoie la valeur d'action pour Noir de cet arbre basée sur son étiquette. Les champs mutables de l'étiquette étant modifiables par d'autres fils d'exécution concurrents, une attention particulière sera donc accordée à l'utilisation du verrou du nœud lors des accès à ces champs.

Q 28. Donner la valeur à maximiser au tour de Blanc en suivant le même principe que pour la valeur d'action de Noir. Nous supposerons par la suite que la fonction `valeurActionBlanc : arbre -> float` a été également implémentée.

Q 29. Écrire une fonction de signature `popargmax : 'a list -> ('a -> 'b) -> 'a * 'a list` qui prend en entrée une liste d'éléments de type `'a`, `l`, et une fonction `f` qui associe à un type `'a` un type `'b` tel que les éléments de type `'b` soient comparables avec les opérateurs polymorphes de comparaison de OCaml (comme `>`), et qui renvoie le premier élément de `l` maximum pour `f` ainsi qu'une liste des autres éléments de `l`. Informellement, nous sortons l'argument maximum de `l` pour `f` pour renvoyer cet argument maximum ainsi que la liste privée de ce dernier.

IV.C – Descente et remontée dans l'arbre

Pour affiner les évaluations des configurations stockées dans l'arbre partiel, la recherche arborescente de Monte-Carlo procède en trois étapes clés :

— une descente dans l'arbre partiel en choisissant à chaque nœud le fils de valeur d'action maximale pour le joueur associé, afin de sélectionner une feuille qui semble prometteuse ;
— une simulation de partie depuis cette feuille comme implémentée en partie IV.A, donnant un joueur vainqueur ;
— une remontée qui met à jour le nombre d'observations et de victoires pour Noir dans chaque nœud rencontré, le long de la branche reliant la feuille explorée à la racine de l'arbre.

Ces étapes pourront être exécutées en parallèles par plusieurs fils d'exécution. Il est donc nécessaire de faire attention à ce que deux fils ne rentrent pas en conflit d'écriture et de lecture pour un champ mutable d'une étiquette de l'arbre, ce qui justifie l'intérêt des verrous associés à chaque nœud.

De plus, pour décourager deux fils d'exécution de suivre la même branche lors de la descente, lorsque l'enfant d'un nœud est sélectionné par un fil d'exécution, ce fil va modifier l'étiquette de cet enfant pour lui ajouter des *défaites virtuelles*, ce qui baissera sa valeur d'action et donc son attractivité du point de vue des autres fils. Nous ajouterons 5 défaites virtuelles à un nœud s'il est sélectionné, en prenant soin d'enlever ces défaites virtuelles lors de la remontée.

Lors de la descente, nous empêcherons ainsi un fil d'exécution de commencer à traiter un nœud, (c'est-à-dire de chercher l'enfant de ce nœud de valeur d'action maximale) si un autre fil d'exécution est déjà en train de traiter ce nœud. Nous attendrons que le fil d'exécution en cours ait ajouté des défaites virtuelles à la branche qu'il a choisi de suivre avant de commencer le traitement.

Q 30. Écrire la fonction `descente_remontee_arbre : goban -> joueur -> arbre -> joueur` qui prend un goban, le joueur dont c'est le tour et un arbre de jeu, et qui réalise une descente, une simulation et une remontée de l'arbre. Cette fonction renverra le joueur gagnant, et lors de la remontée mettra à jour les champs mutables des nœuds rencontrés. Une attention particulière sera portée sur le fait que cette fonction puisse être exécutée par plusieurs fils d'exécution concurrents.

IV.D – Expansion de l'arbre

L'idée de la recherche arborescente de Monte-Carlo est d'alterner entre des étapes de descente, simulation et remontée de l'arbre pour améliorer les évaluations des nœuds, et des étapes d'expansion, où nous ajoutons de nouveaux nœuds à l'arbre partiel de jeu.

Q 31. Écrire une fonction de signature `expansion_feuille : goban -> joueur -> arbre list` qui prend un goban et un joueur, et qui renvoie la liste des enfants du nœud correspondant à ce goban et ce joueur. Il est attendu d'utiliser la fonction `liste_coup_prior` et d'initialiser les étiquettes des enfants avec 0 observation, 0 victoire et un nouveau verrou `Mutex.t`.

Une étape d'expansion consiste à étendre une feuille, qui sera sélectionnée en partant de la racine et en choisissant à chaque nœud le fils ayant été le plus visité.

BY-NC-SA

Q 32. Écrire une fonction de signature `expansion : goban -> joueur -> arbre -> arbre` qui réalise une étape complète d'expansion, c'est-à-dire part de l'arbre et descend jusqu'à une feuille en choisissant à chaque étape le fils le plus visité, étend la feuille rencontrée en ajoutant tous ses fils, puis qui renvoie l'arbre partiel de jeu étendu.

IV.E – Algorithme complet

Pour pouvoir commencer à réaliser l'étape d'expansion, il est nécessaire d'attendre que tous les fils d'exécution réalisant des descentes et remontées aient terminé.

Q 33. Écrire une fonction de signature `attendre : Thread.t list -> unit` permettant d'attendre que tous les fils d'exécutions de la liste donnée en argument aient terminé leur exécution.

Q 34. Écrire une fonction de signature `etape_complete : int -> goban -> joueur -> arbre -> arbre` qui réalise une étape complète de l'algorithme, telle que `etape_complete nt g j a` lance `nt` fils d'exécution qui réalisent chacun une descente, simulation puis remontée de l'arbre `a` où la racine correspond au goban `g` et au joueur `j`, et une fois que tous les fils ont terminé, réalise une étape d'expansion avant de renvoyer l'arbre obtenu après cette expansion.

Q 35. Montrer que cette fonction termine. Il est notamment attendu de montrer qu'il n'y a pas d'interblocages dans les exécutions de la fonction `descente_remontee_arbre` en se référant au code proposé en question 30.

Pour estimer le coup à jouer, il reste alors à réaliser plusieurs étapes complètes, par exemple un nombre `ns` fixé à l'avance. Puis, à la fin de ces étapes, l'algorithme renvoie la position du coup correspondant au nœud le plus visité parmi les fils de la racine.

Le sous-arbre associé à ce coup sera ensuite utilisé comme base pour la suite de la partie, ce qui permet de ne pas repartir de zéro à chaque coup.

Q 36. Écrire une fonction de signature `recherche_arborescente : int -> int -> goban -> joueur -> arbre -> arbre * position` telle que `recherche_arborescente ns nt g j a` effectue `ns` étapes de simulations sur l'arbre `a` en partant du goban `g` avec le joueur `j`, chaque étape de simulations utilisant `nt` fils d'exécutions, avant de renvoyer le coup choisi, ainsi que l'arbre à conserver pour le coup suivant.

Éléments du langage OCaml

Module Thread

- `Thread.t` : Le type des fils d'exécution.
- `Thread.create : ('a -> 'b) -> 'a -> Thread.t` : `Thread.create funct arg` crée un nouveau fil d'exécution dans lequel l'application de `funct arg` est exécuté en même temps que les autres fils d'exécutions. L'application de `Thread.create` renvoie le fil d'exécution crée. Le nouveau fil termine quand l'application `funct arg` termine, soit normalement soit en remontant une exception `Thread.Exit` ou en remontant d'autres exceptions non rattrapées. Dans ce dernier cas, l'exception non rattrapée est affichée sur la sortie d'erreur standard, mais pas propagée au fil d'exécution parent. De la même manière le résultat de l'application `funct arg` est perdu et n'est pas directement accessible au fil d'exécution parent.
- `Thread.join : t -> unit` : `join th` supsend l'exécution du fil qui appelle cette fonction jusqu'à ce que le fil d'exécution `th` ait terminé.

Module Mutex

- `Mutex.t` : Le type des mutex.
- `Mutex.create : unit -> Mutex.t` : Renvoie un nouveau mutex.
- `Mutex.lock : Mutex.t -> unit` : Verrouille le mutex donné. Un seul fil d'exécution peut verrouiller le mutex à un moment donné. Un fil d'exécution tentant de verrouiller un mutex déjà verrouillé attendra jusqu'à ce que l'autre fil d'exécution déverouille le mutex.
- `Mutex.unlock : Mutex.t -> unit` : Déverouille le mutex donné. Les autres fils d'exécution en pause tentant de verouiller le mutex reprennent. Le mutex doit avoir été verrouillé par le fil d'exécution qui appelle `Mutex.unlock`.

Module List

- `List.iter : ('a -> unit) -> 'a list -> unit` : `iter f [a1; ...; an]` applique la fonction `f` tour à tour à `[a1; ...; an]`. Cet application est équivalente à `f a1; f a2; ...; f an`.
- `List.map : ('a -> 'b) -> 'a list -> 'b list` : `map f [a1; ...; an]` applique la fonction `f` à `a1, ..., an`, et construit la liste `[f a1; ...; f an]` avec les résultats renvoyés par `f`.

BY-NC-SA

Type enregistrement
— `type exemple = {mutable a : int; b int}` : permet de définir un type enregistrement.
— `let c = {a = 1; b = 3}` permet de définir une variable du type `exemple`.
— `c.a` permet d'accéder au champ `a` de `c`.
— `c.a<-2` permet de modifier le champ mutable `a` du type enregistrement `c`.

Fonction de conversion de type
— `float_of_int : int -> float` : convertit un `int` en `float`.
— `int_of_float : float -> int` : convertit un `float` en `int`.

V Complexité du problème de la stratégie gagnante au go

Un problème classique qui survient lorsque l'on s'intéresse à des jeux est de se demander quel joueur possède une stratégie gagnante à partir d'une configuration donnée. Ce problème de décision sera étudié ici sous le nom de go généralisé.

Go généralisé
Entrée : Une configuration d'une partie de go géneralisé, c'est-à-dire un goban de dimension d, avec N un entier arbitraire, sur lequel sont placées des pierres, et la donnée du joueur qui doit réaliser le prochain coup.
Sortie : Oui s'il existe une stratégie gagnante pour Blanc à partir de cette configuration, non sinon.

Il est possible de montrer que ce problème est NP-dur à partir d'une chaîne de réductions depuis le problème SAT, dont la dernière étape sera montrée dans cette partie.

Q 37. Rappeler la définition de la classe NP, et expliquer ce que signifie être NP-dur et NP-complet. Donner un exemple de problème NP-complet classique autre que SAT et 3SAT, sans prouver son appartenance à cette classe de complexité.

Dans cette partie nous considérons une variante du problème 3SAT, nommée *Rectilinear Planar 3SAT*, qui est également NP-dur. Tout comme pour 3SAT, les instances de ce problème de décision sont des formules sous forme normale conjonctive avec exactement trois littéraux par clauses, mais désormais chaque clause ne possède que des littéraux positifs, ou que des littéraux négatifs. De plus, la formule en question doit pouvoir se représenter graphiquement de manière planaire rectiligne comme illustré en figure 5, avec toutes les variables alignées horizontalement et les clauses reliées verticalement à leurs variables associées sans aucune intersection. Une telle représentation graphique est également donnée en entrée du problème avec la formule, et n'a donc pas besoin d'être trouvée.

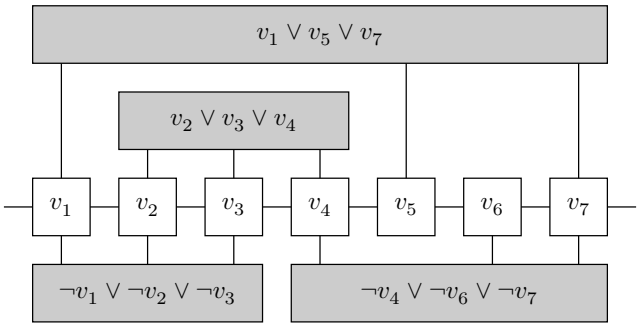


Figure 5 La formule $\varphi = (v_1 \vee v_5 \vee v_7) \wedge (v_2 \vee v_3 \vee v_4) \wedge (\neg v_1 \vee \neg v_2 \vee \neg v_3) \wedge (\neg v_4 \vee \neg v_6 \vee \neg v_7)$ représentée graphiquement sous forme planaire rectiligne. Il est à noter que sur une telle représentation graphique, certaines clauses peuvent en enjamber d'autres. Ici, la clause $(v_1 \vee v_5 \vee v_7)$ enjambe la clause $(v_2 \vee v_3 \vee v_4)$.

Soit φ et sa représentation graphique planaire rectiligne une instance de Rectilinear Planar 3SAT. Nous allons construire une configuration du jeu de go généralisé telle que Blanc possède une stratégie gagnante si et seulement si φ est satisfiable. Pour cela, nous supposons qu'une grande quantité de pierres blanches se trouvent presque encerclées par des pierres noires, de sorte que l'issue de la partie soit uniquement déterminée par le fait que Blanc parvienne à relier ces pierres blanches à une structure permettant de sécuriser son groupe (figure 6), (d)), (auquel cas il sécurise un nombre de points suffisamment grand pour lui garantir la victoire), ou pas (auquel cas Noir encercle et capture toutes ces pierres blanches, et gagne). En s'affrontant pour cette capture décisive, les deux joueurs vont placer des pierres dans des *gadgets* reliés entre eux et à la réserve de pierres blanches par des tuyaux (figure 6, (b) et (c)).

Q 38. Expliquer pourquoi si Blanc parvient à relier sa réserve à un gadget (d) de la figure 6, alors Noir ne peut plus capturer les pierres blanches de la réserve.

Ce gadget à atteindre pour sécuriser le groupe de pierres sera nommé des *yeux*. En tentant à tout prix de relier sa réserve à des yeux, Blanc va progressivement relier sa réserve à des gadgets, qui sont des dispositions

BY-NC-SA

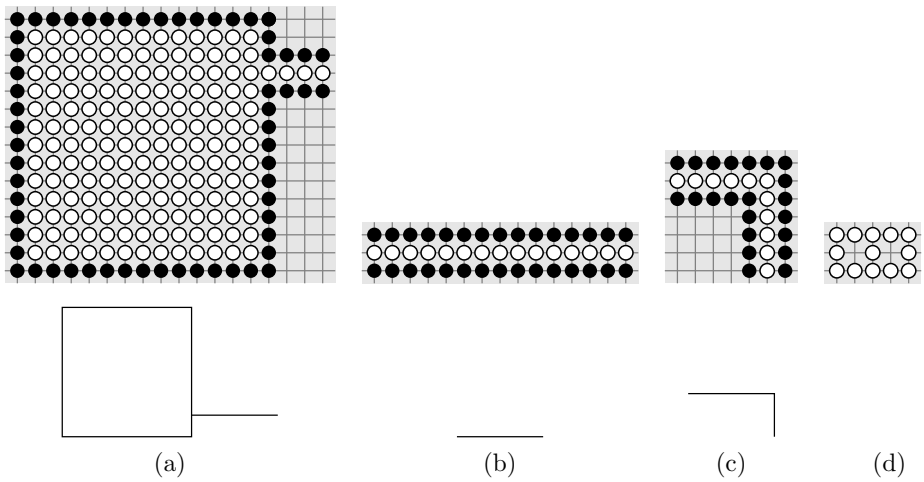


Figure 6 Un exemple de réserve de pierres blanches presque capturée par le joueur noir (a), ainsi que deux sections de tuyaux (b) et (c). Des représentations simplifiées sont données en bas. En (d) une configuration de 13 pierres blanches que Blanc va chercher à relier à sa réserve de pierres blanches pour les sécuriser.

locales de pierres qui vont permettre de simuler un comportement souhaité. Ces gadgets sont construits de sorte que le nombre de libertés du groupe de pierres blanches qui risquent d'être capturées soit toujours de 1, ou exceptionnellement de 2 dans le gadget (figure 7 (a)) ce qui permet à Blanc de faire un choix ponctuellement. Ce premier gadget (figure 7 (a)) permet de modéliser un choix pour Blanc, qui décidera s'il veut relier le tuyau du haut (supposé relié à la réserve de pierres) au tuyau de gauche ou au tuyau de droite. Nous pourrons supposer que c'est à Blanc de poser la première pierre dans chaque gadget.

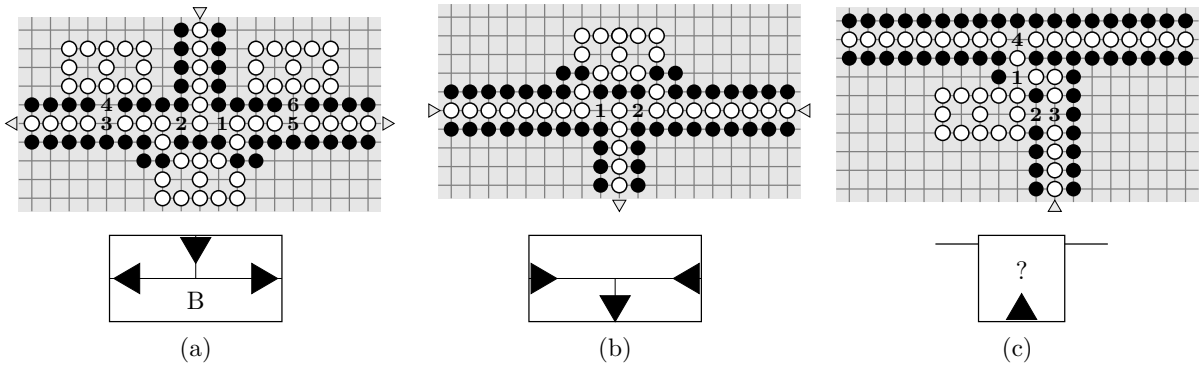


Figure 7 Trois gadgets qui seront utilisés pour la réduction, ainsi que leurs représentations simplifiées associées.

Q 39. Expliciter précisément les deux suites de coups possibles pour le gadget (a), et justifier à l'aide d'un arbre que si Blanc ou Noir s'écarte de l'une de ces deux stratégies, alors l'autre joueur peut arriver à ses fins en au plus deux coups.

Q 40. Décrire des modifications à apporter au gadget (a) pour obtenir un gadget représentant un branchement mais dans lequel c'est Noir qui décide si le tuyau de gauche ou de droite sera relié au tuyau du haut. On rappelle qu'on se place toujours dans le cas où Blanc joue en premier dans les gadgets.

Nous utiliserons, pour ce nouveau gadget, la même représentation simplifiée que le gadget (a) en remplaçant le B par un N.

Q 41. Que permet de réaliser le gadget (b) ? Justifier.

Q 42. Justifier que le gadget (c) permet, en arrivant par le bas, de tester si le tuyau transversal du haut a déjà été relié à la réserve de pierres blanches par le passé ou pas. On pourra notamment justifier que la partie se décidera sur ce gadget précis, et l'issue en sera uniquement déterminée par le fait que le tuyau transversal du haut ait été préalablement relié à la réserve ou non.

Nous nous intéressons désormais à l'assemblage de ces gadgets pour former un circuit, qui pourra être représenté en utilisant les notations simplifiées. Dans un premier temps, nous allons simuler la valuation de chaque variable par le fait d'avoir relié une section à la réserve ou non.

Q 43. Vérifier que pour chaque variable v_i, il existe une stratégie pour Noir qui permette qu'au plus un des deux tuyaux v_i ou $\neg v_i$ soit relié à la réserve de pierres blanches.

Q 44. Vérifier qu'il existe, pour toute valuation des variables, une stratégie permettant à Blanc de relier à la réserve de pierres blanches les sections associées à cette valuation.

BY-NC-SA

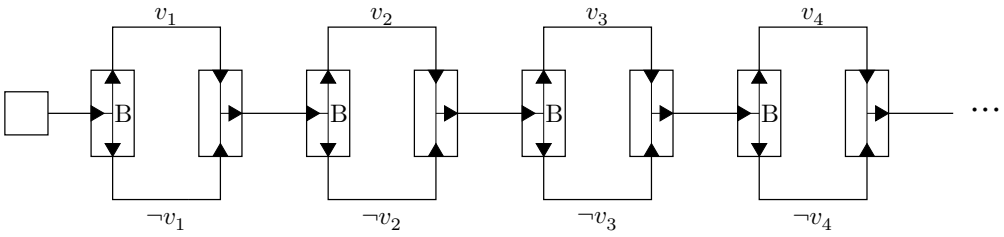


Figure 8 Un circuit représentant le choix d'une valuation, modélisée par le fait de choisir de relier le tuyau du haut ou le tuyau du bas à la réserve pour chaque variable.

Ainsi, Blanc a choisi de relier certaines sections à sa réserve de pierres blanches et la partie est encore en cours. Il reste désormais à déterminer si la valuation choisie par Blanc est un modèle de φ ou non. Pour cela, l'idée est de laisser Noir choisir la clause C qu'il souhaite dans φ. Ensuite, Blanc choisit d'accéder au littéral de son choix au sein de cette clause, afin de témoigner qu'elle est bien satisfaite. Le gadget suivant (figure 9) est une première tentative pour modéliser ce choix de Blanc, mais sa construction bloque le passage et empêche Noir d'accéder à d'éventuelles autres clauses enjambées, qui pourraient être représentées à l'intérieur dans la représentation planaire rectiligne de φ.

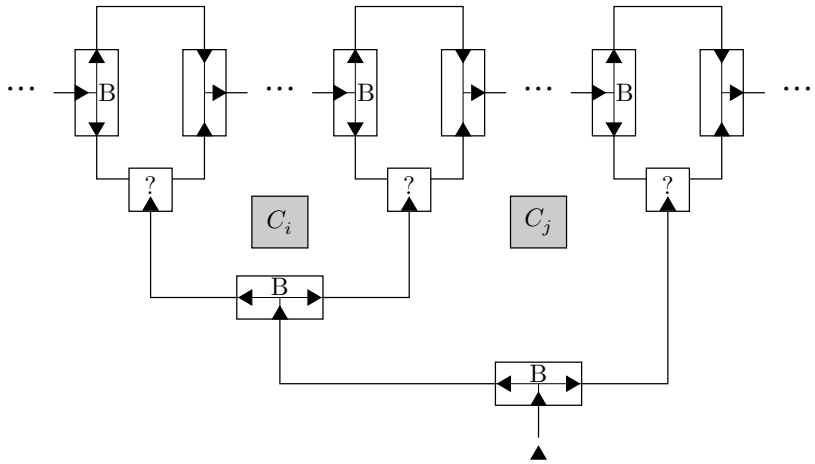


Figure 9 Un gadget permettant à Blanc de vérifier que le littéral de son choix est bien satisfait dans une clause, mais qui bloque le passage à d'autres clauses éventuelles comme C_i ou C_j qui peuvent être enjambées.

Q 45. Modifier le circuit de vérification de clause proposé, afin que Noir puisse en plus décider d'accéder à une clause intérieure s'il le souhaite, sans en modifier le comportement.

Q 46. En utilisant les questions précédentes, conclure en proposant une réduction complète de Rectilinear Planar 3SAT au jeu de go généralisé.

• • • FIN • • •

BY-NC-SA

Optimisation de réseau

Option informatique

CONCOURS CENTRALE•SUPÉLEC

MP

2024

4 heures Calculatrice autorisée

Créer un réseau routier entre différentes villes, en s'autorisant des intersections intermédiaires, dont la longueur totale est minimale est un problème très difficile à résoudre. Ce problème est une généralisation de deux problèmes pourtant faciles à résoudre : la recherche d'arbre couvrant de poids minimal, correspondant au cas où il n'y a pas d'intersections intermédiaires, et la recherche de plus court chemin, correspondant au cas où on ne souhaite relier que deux villes.

En 1836, un an seulement après l'apparition d'une ligne de train en Allemagne, Carl Friedrich Gauß s'interrogeait dans une lettre à l'astronome Christian Schumacher sur la meilleure manière de relier les villes de Hambourg, Brême, Hanovre et Brunswick par un réseau ferré.

Figure 1 Les quatre villes allemandes et une manière théoriquement optimale de les relier.

De tels arbres trouvent des applications dans la construction de réseaux (électroniques, routiers, ...). Ils peuvent également apparaître dans des phénomènes naturels : des films de savon reliant des tiges entre deux plaques peuvent former un arbre avec des points intermédiaires.

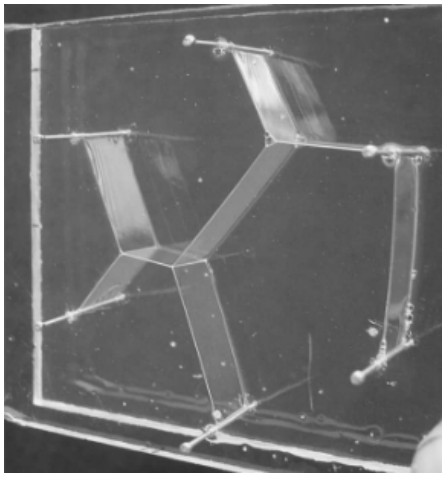


Figure 2 Films de savon reliant des tiges métalliques entre deux plaques de plexiglas. Il y a trois sommets intermédiaires.[1]

La partie I de ce problème étudie des généralités sur les arbres couvrants et une fonction de tri. La partie II présente l'algorithme de Kruskal inversé pour trouver un arbre couvrant de poids minimal dans un graphe

[1] Crédits : Dutta, P., Khastgir, S. P. & Roy, A. (2010). Steiner trees and spanning trees in six-pin soap films. American Journal of Physics, 78(2), 215-221.

BY-NC-SA

connexe. La partie III montre la complexité du problème d'optimisation de réseau en se ramenant au problème de satisfiabilité. Enfin, la partie IV présente comment approcher une solution au problème d'optimisation en utilisant des arbres couvrants.

Consignes aux candidates et candidats

On identifiera une même grandeur écrite dans deux polices de caractères différentes, en italique du point de vue mathématique (par exemple n) et en Computer Modern à chasse fixe du point de vue informatique (par exemple `n`).

Les candidates et candidats devront répondre aux questions de programmation en utilisant le langage OCaml. S'ils écrivent une fonction non demandée par l'énoncé, ils devront préciser son rôle, ainsi que sa signature (son type). Les candidates et candidats sont également invités, lorsque c'est pertinent, à décrire le fonctionnement global de leurs programmes. On autorisera toutes les fonctions des modules `Array` et `List`, ainsi que les fonctions de la bibliothèque standard (celles qui s'écrivent sans nom de module, comme `max`, `incr` ainsi que les opérateurs comme `@`). Sauf précision de l'énoncé, l'utilisation d'autres modules sera interdite.

Lorsqu'une question de programmation demande l'écriture d'une fonction, la signature de la fonction demandée est indiquée à la fin de la question. Les candidates et candidats pourront écrire une fonction dont la signature est **compatible** avec celle demandée. Par exemple, si l'énoncé demande l'écriture d'une fonction `int list -> int` qui renvoie le premier élément d'une liste d'entiers, écrire une fonction `'a list -> 'a` qui renvoie le premier élément d'une liste d'éléments quelconques sera considéré comme correct.

Une annexe disponible en fin de sujet rappelle différentes fonctions en OCaml.

Définitions et notations

Pour un ensemble fini E, on note $|E|$ le cardinal de E. On note $\mathcal{P}_2(E)$ l'ensemble des paires d'éléments de E, c'est-à-dire les sous-ensembles de E de cardinal 2.

On définit un **graphe non orienté** comme un couple $G = (S, A)$ tel que S est un ensemble fini d'éléments appelés **sommets** et A un ensemble de paires d'éléments distincts de S appelées **arêtes**, c'est-à-dire $A \subset \mathcal{P}_2(S)$. S'il n'y a pas d'ambiguïté, une paire $\{s, t\}$ sera notée st. Si st est une arête du graphe, on dit que s et t sont **adjacents**. Le graphe $G_1 = (\{s_0, s_1, s_2, s_3, s_4, s_5\}, \{s_0s_2, s_0s_3, s_0s_4, s_1s_2, s_1s_3, s_2s_3, s_2s_5, s_3s_4\})$ est représenté sur la figure 3.

On appelle **sous-graphe** de $G = (S, A)$ un graphe $H = (X, B)$ tel que $X \subset S$ et $B \subset A \cap \mathcal{P}_2(X)$. Si $X \subset S$, on appelle **sous-graphe induit par X** le graphe $G[X] = (X, A \cap \mathcal{P}_2(X))$, c'est-à-dire le graphe ayant X comme ensemble de sommets et contenant toutes les arêtes de G dont les deux extrémités sont dans X.

Pour $(s, t) \in V^2$, on appelle **chaîne** de s à t une suite de sommets distincts $(s_0, s_1, ..., s_k)$ telle que $s_0 = s$, $s_k = t$ et pour $i \in [\![1, k]\!]$, $s_{i-1}s_i \in A$. On appelle **cycle** de G une suite de sommets $(s_0, s_1, ..., s_k)$ telle que $k \geqslant 3$, $(s_0, ..., s_{k-1})$ est une chaîne, $s_0 = s_k$ et $s_0s_{k-1} \in A$.

Un graphe est dit **connexe** s'il existe toujours une chaîne entre deux sommets distincts. Si $G = (S, A)$, on appelle **composante connexe** de G un sous-ensemble X de S tel que $G[X]$ est connexe et pour tout $s \in S \setminus X$, $G[X \cup \{s\}]$ n'est pas connexe.

Un graphe est dit un **arbre** (à différencier des arbres enracinés) s'il est connexe et ne contient pas de cycle. On dit qu'un sous-graphe $T = (X, B)$ de $G = (S, A)$ qu'il est un **arbre couvrant** si $S = X$ et T est un arbre.

On appelle **graphe pondéré** un triplet (S, A, f) tel que (S, A) est un graphe et $f : A \to \mathbb{R}_+$ est une fonction qui à chaque arête associe un **poids** qui est un réel positif. La figure 3 contient une représentation graphique d'un graphe pondéré G_2.

Dans un graphe pondéré $G = (S, A, f)$, le **poids** d'un sous-graphe $H = (X, B, f)$ de G est la somme des poids des arêtes qui le composent, c'est-à-dire $f(H) = \sum_{a \in B} f(a)$.

Figure 3 Les graphes G_1 et G_2

BY-NC-SA

I Généralités

Q 1. Dessiner sans justifier un arbre couvrant de G_2 de poids minimal.

Q 2. Soit $G = (S, A)$ un graphe non orienté. Montrer les deux propriétés suivantes :

— Si G est connexe, alors $|A| \geqslant |S| - 1$.

— Si G est sans cycle, alors $|A| \leqslant |S| - 1$.

Q 3. En déduire l'équivalence entre les trois propriétés suivantes, pour $G = (S, A)$ un graphe non orienté :

— G est un arbre.

— G est connexe et $|A| = |S| - 1$.

— G est sans cycle et $|A| = |S| - 1$.

Pour les deux questions suivantes, on suppose que $G = (S, A, f)$ est un graphe pondéré tel que f est injective (toutes les arêtes ont un poids différent).

Q 4. Montrer que G possède un unique arbre couvrant de poids minimal.

Q 5. Soit $T^* = (S, B^*)$ l'unique arbre couvrant de poids minimal de G. On suppose que $a \in A$ est une arête dont le poids est maximal dans un cycle de G. Montrer que $a \notin B^*$.

Les trois questions qui suivent ont pour objectif d'implémenter un algorithme de tri efficace.

Q 6. Écrire une fonction `separation` telle que si `lst` est une liste, alors `separation lst` renvoie un couple `(lst1, lst2)` tel que chaque élément de `lst` apparaît soit dans `lst1`, soit dans `lst2`, et les tailles de `lst1` et `lst2` diffèrent d'au plus 1.

```
separation : 'a list -> 'a list * 'a list
```

Q 7. Écrire une fonction `fusion` telle que si `lst1` et `lst2` sont deux listes triées par ordre croissant, alors `fusion lst1 lst2` renvoie une liste triée par ordre croissant contenant tous les éléments de `lst1` et `lst2`.

```
fusion : 'a list -> 'a list -> 'a list
```

Q 8. En déduire une fonction `tri_fusion` qui prend en argument une liste et renvoie une liste triée par ordre croissant contenant les mêmes éléments. Quelle est la complexité temporelle de cette fonction ? (On ne demande pas de justifier.)

```
tri_fusion : 'a list -> 'a list
```

II Algorithme de Kruskal inversé

Q 9. Rappeler le principe de l'algorithme de Kruskal et sa complexité temporelle en fonction de $|S|$ et $|A|$ lorsqu'il est appliqué à un graphe pondéré connexe $G = (S, A, f)$.

L'algorithme de Kruskal inversé est une variante de l'algorithme de Kruskal qui permet de trouver un arbre couvrant de poids minimal dans un graphe pondéré connexe. En voici une description en pseudo-code.

Entrées : Graphe pondéré $G = (S, A, f)$ non orienté connexe
$B \leftarrow$ copie de A
Trier B par ordre décroissant de poids
pour $a \in B$ par ordre décroissant de poids **faire**
  si $(S, B \setminus \{a\})$ est connexe **alors**
    $B \leftarrow B \setminus \{a\}$
  fin si
fin pour
renvoyer (S, B)

Algorithme 1 Algorithme de Kruskal

Q 10. Appliquer l'algorithme de Kruskal inversé au graphe G_3 de la figure figure 4. On ne demande pas la description de l'exécution détaillée, mais simplement une représentation graphique de l'arbre obtenu.

BY-NC-SA

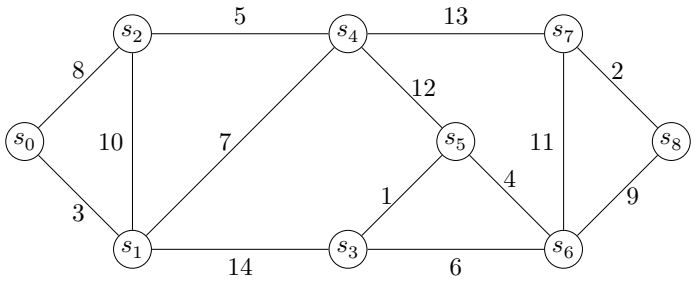


Figure 4 Le graphe pondéré G_3

On implémente un graphe pondéré par tableau de listes d'adjacences en OCaml, selon le type suivant :

```
type graphe = (int * float) list array
```

Si $G = (S, A, f)$ est implémenté par un objet `g` de type `graphe`, alors :

— l'ensemble des sommets est $S = [\![0, n-1]\!]$, où $n = |S|$;

— `g` est un tableau de taille n tel que pour tout $s \in S$, `g.(s)` est une liste contenant des couples (t, p) tels que $st \in A$ et $f(st) = p$.

Les listes d'adjacence ne sont pas nécessairement supposée triées par ordre croissant des sommets ou des poids. Par exemple, le graphe G_2 de la figure 3 peut être créé par la commande :

```
let g2 = [|[(2, 7.); (3, 6.); (1, 10.); (4, 7.)]; [(2, 5.); (0, 10.)];
          [(0, 7.); (1, 5.); (3, 9.)]; [(0, 6.); (2, 9.)]; [(0, 7.)]|]
```

On rappelle que les opérations de comparaisons (`max`, `min`, `<`, `>`, …) peuvent s'effectuer sur des uplets selon l'ordre lexicographique.

Q 11. Écrire une fonction `tri_aretes` telle que si `g` est la représentation d'un graphe pondéré $G = (S, A, f)$, alors `tri_aretes g` renvoie la liste des triplets $(f(st), s, t)$, triée par ordre **décroissant** des $f(st)$, tels que $st \in A$ et $s < t$. Cette liste ne devra pas contenir de doublons.

```
tri_aretes : graphe -> (float * int * int) list
```

Q 12. Écrire une fonction `connectes` telle que si `g` est la représentation d'un graphe pondéré $G = (S, A, f)$, $(s, t) \in S^2$ et `poids_max` est un flottant, alors `connectes g s t poids_max` renvoie un booléen qui vaut `true` si et seulement s'il existe un chemin de s à t dans G ne passant que par des arêtes dont le poids est strictement inférieur à `poids_max`. La fonction devra avoir une complexité linéaire en $|S| + |A|$ et on demande de justifier brièvement.

```
connectes : graphe -> int -> int -> float -> bool
```

Q 13. En déduire une fonction `kruskal_inverse` qui prend en argument un graphe $G = (S, A, f)$ et renvoie le graphe obtenu selon le principe de l'algorithme de Kruskal inversé. On supposera que le graphe G donné en argument est connexe et que la fonction de pondération f est injective.

```
kruskal_inverse : graphe -> graphe
```

Q 14. Montrer que si G est connexe, le graphe renvoyé par l'algorithme de Kruskal inversé appliqué à G est un arbre couvrant de G.

Q 15. Montrer que l'arbre couvrant renvoyé par l'algorithme de Kruskal inversé appliqué à un graphe pondéré connexe G est de poids minimal. On pourra commencer par traiter le cas où la fonction de pondération est injective.

Q 16. Déterminer la complexité temporelle de la fonction `kruskal_inverse`. Commenter.

III Difficulté du calcul de l'arbre optimal

Dans cette partie, on présente le problème du calcul de l'arbre optimal, appelé arbre de Steiner, et on veut montrer que c'est un problème difficile à résoudre.

BY-NC-SA

Soit $G = (S, A)$ un graphe non orienté connexe. On appelle **arbre de Steiner de sommets terminaux X** un sous-graphe de G de la forme $T = (Y, B)$ tel que T est un arbre et $X \subset Y$. Les sommets de $Y \setminus X$ sont appelés les **sommets de Steiner**. Le problème de l'arbre de Steiner consiste, étant donné un graphe $G = (S, A)$, un ensemble $X \subset S$ et un entier k, à déterminer s'il existe un arbre de Steiner de sommets terminaux X ayant moins de k sommets de Steiner.

Q 17. Représenter graphiquement un arbre de Steiner du graphe G_1 de la figure 3, ayant $X = \{s_1, s_4, s_5\}$ comme sommets terminaux et ayant moins de 2 sommets de Steiner. Justifier qu'il n'en existe pas ayant moins de 1 sommet de Steiner.

Pour montrer la difficulté de ce problème, on se ramène au problème de satisfaisabilité d'une formule booléenne.

On rappelle qu'un **littéral** est une variable ou la négation d'une variable. Une **clause** est une disjonction ($\vee$) de littéraux. Une **forme normale conjonctive** (FNC) est une conjonction ($\wedge$) de clauses. Une formule φ est dite en 3-FNC si elle est en forme normale conjonctive telle que chaque clause contient au plus trois littéraux.

On admet qu'étant donnée une formule propositionnelle φ en 3-FNC, il est difficile de savoir si φ possède un modèle, c'est-à-dire une valuation qui la satisfait.

III.A – De la satisfaisabilité au stable

Soit $G = (S, A)$ un graphe non orienté. On dit qu'une partie $X \subset S$ est un **stable** si c'est un ensemble de sommets deux à deux non adjacents, c'est-à-dire si pour $(s, t) \in X^2$, $st \notin A$.

Q 18. Déterminer, en justifiant, un stable de cardinal maximal dans le graphe G_1 représenté figure 3.

Soit $\mathcal{V} = \{v_1, ..., v_n\}$ un ensemble de variables et φ une formule booléenne en 3-FNC, de la forme $\varphi = \bigwedge_{j=1}^{m} C_j$, où $C_j = \bigvee_{k=1}^{m_j} \ell_{jk}$, les ℓ_{jk} étant des littéraux (de la forme v_i ou $\neg v_i$) et $m_j \in [\![1, 3]\!]$.

On définit le graphe $G_\varphi = (S_\varphi, A_\varphi)$ par :

— S_φ contient un sommet par littéral apparaissant dans une clause, c'est-à-dire $S_\varphi = \bigcup_{j=1}^{m} S_j$ où :

$$S_j = \left\{v_i^{(j)} \mid v_i \text{ apparait dans } C_j\right\} \cup \left\{\overline{v_i}^{(j)} \mid \neg v_i \text{ apparait dans } C_j\right\}$$

— A_φ contient les arêtes entre chaque sommet associé à une même clause, et entre les sommets associés à une variable et sa négation :

$$A_\varphi = \left\{st \mid (s, t) \in S_j^2, s \neq t\right\} \cup \left\{v_i^{(j)}\overline{v_i}^{(j')} \mid i \in [\![1, n]\!], j \neq j'\right\}$$

La figure 5 représente le graphe G_{φ_0} pour $\varphi_0 = (v_1 \vee \neg v_2 \vee \neg v_3) \wedge (\neg v_1 \vee v_2) \wedge (v_1 \vee v_2 \vee v_3)$.

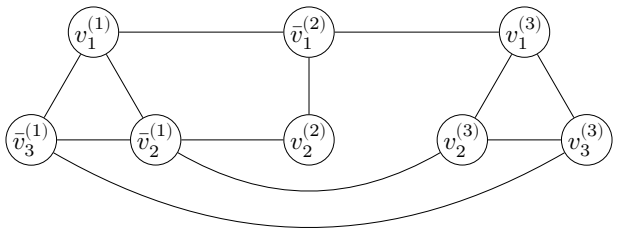


Figure 5 Le graphe G_{φ_0} pour $\varphi_0 = (v_1 \vee \neg v_2 \vee \neg v_3) \wedge (\neg v_1 \vee v_2) \wedge (v_1 \vee v_2 \vee v_3)$.

Q 19. Tracer le graphe G_{φ_1} pour $\varphi_1 = (\neg v_0 \vee v_2) \wedge (v_0 \vee \neg v_1 \vee \neg v_2) \wedge (v_1 \vee \neg v_2) \wedge (\neg v_0 \vee \neg v_1 \vee v_2)$.

Q 20. Montrer que s'il existe un stable X de taille m dans G_φ, alors φ est satisfaisable. On expliquera comment construire un modèle de φ en fonction de X.

Q 21. De même, montrer que si φ est satisfaisable, alors il existe un stable de taille m dans G_φ.

III.B – Du stable à l'arbre de Steiner

Si $G = (S, A)$ est un graphe non orienté, on pose $G' = (S_G, A_G)$ un graphe tel que :

— S_G contient S et un nouveau sommet par arête de G, c'est-à-dire :

$$S_G = S \cup \{s_a \mid a \in A\}$$

BY-NC-SA

— A_G contient toutes les arêtes entre deux sommet de S, et des arêtes entre chaque sommet s_a vers chacune des extrémités de a, c'est-à-dire :

$$A_G = \{st \mid (s,t) \in S^2\} \cup \bigcup_{st \in A} \{ss_{st}, ts_{st}\}$$

Par exemple, la figure 6 est une représentation de G'_1 où G_1 est le graphe de la figure 3.

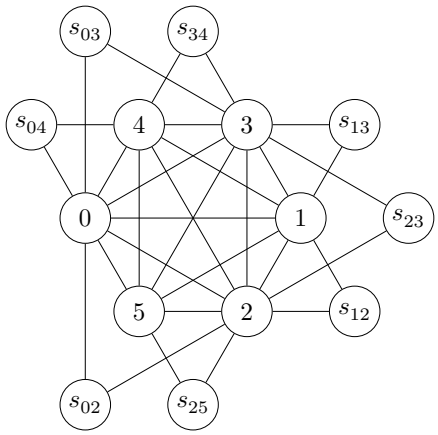


Figure 6 Une représentation de G'_1. Pour simplifier, on a renommé chaque sommet s_i par i.

Q 22. Montrer que s'il existe un stable de G de cardinal k, alors le graphe $G' = (S_G, A_G)$ admet un arbre de Steiner ayant $X = \{s_a \mid a \in A\}$ comme sommets terminaux et $|S| - k$ sommets de Steiner.

Q 23. Réciproquement, montrer que si le graphe $G' = (S_G, A_G)$ admet un arbre de Steiner ayant $X = \{s_a \mid a \in A\}$ comme sommets terminaux et k sommets de Steiner, alors G admet un stable de cardinal $|S| - k$.

Q 24. En déduire que si on sait trouver un arbre de Steiner ayant un nombre minimal de sommets de Steiner en temps polynomial, on peut déterminer la satisfaisabilité d'une formule propositionnelle en 3-FNC en temps polynomial.

IV Approximation du problème

Étant donné un graphe pondéré $G = (S, A, f)$ et un sous-ensemble $X \subset S$, le **problème de l'arbre de Steiner pondéré** consiste à déterminer un sous-graphe $T = (Y, B, f)$ tel que $X \subset Y \subset S$, T est un arbre et $f(T)$ est minimal.

On peut voir que le problème de l'arbre de Steiner pondéré est plus difficile que le cas non pondéré, qui n'est qu'un cas particulier où tous les poids sont égaux à 1. Malgré sa difficulté, il est possible de calculer en temps polynomial un arbre de Steiner pondéré dont le poids est moins du double d'un arbre optimal.

IV.A – Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall est un algorithme de programmation dynamique permettant de calculer toutes les distances pondérées entre deux sommets d'un graphe pondéré.

On rappelle que la représentation de G par **matrice d'adjacence** est une matrice M de dimensions $n \times n$, où $n = |S|$, telle que pour $(s,t) \in S^2$, $M_{st} = f(st)$ s'il existe $st \in A$, et $M_{st} = +\infty$ sinon.

En OCaml, `infinity` est un flottant qui se comporte comme $+\infty$: plus grand que tous les autres flottants et absorbant par addition ou par multiplication par un flottant strictement positif.

Q 25. Écrire une fonction `matrice_adjacence` qui prend en argument un graphe et renvoie sa matrice d'adjacence.

```
matrice_adjacence : graphe -> float array array
```

Soit $G = (S, A, f)$ un graphe pondéré. Si $c = (s_0, s_1, ..., s_k)$ est une chaîne de s_0 à s_k, le poids de c est $f(c) = \sum_{i=1}^{k} f(s_{i-1}s_i)$. Pour $(s,t) \in S^2$, on appelle **distance de s à t**, notée $d_G(s,t)$ (ou $d(s,t)$ s'il n'y a pas d'ambiguïté sur le graphe), le poids minimal d'une chaîne de s à t. S'il n'existe pas de telle chaîne, on pose $d(s,t) = +\infty$ par convention. On remarque qu'avec une telle définition, pour $s \in S$, $d(s,s) = 0$.

BY-NC-SA

L'algorithme de Floyd-Warshall permet de calculer les distances des sommets deux à deux en considérant une suite de matrices $M^{(k)}$, pour $k \in [\![0, n]\!]$ définies de la manière suivante : pour $(s,t) \in S^2$, $M_{st}^{(k)}$ est le poids minimal d'une chaîne de s à t dont les sommets intermédiaires (autres que les extrémités) sont strictement inférieurs à k.

Q 26. Que vaut $M^{(0)}$?

Q 27. Montrer que pour $k \in [\![0, n-1]\!]$ et $(s,t) \in S^2$, $M_{st}^{(k+1)} = \min(M_{st}^{(k)}, M_{sk}^{(k)} + M_{kt}^{(k)})$.

Q 28. Écrire une fonction `floyd_warshall` qui prend en argument une matrice d'adjacence $n \times n$ d'un graphe $G = (S, A, f)$ et renvoie un couple `(dist, pred)` de matrices $n \times n$ telles que pour $(s,t) \in S^2$:

— `dist.(s).(t)` est un flottant correspondant à $d(s,t)$;

— `pred.(s).(t)` est le prédécesseur de t dans un chemin de poids minimal de s à t si un tel chemin existe, et -1 sinon.

```
floyd_warshall : float array array -> float array array * int array array
```

Q 29. Déterminer la complexité temporelle de la fonction `floyd_warshall`.

Q 30. Écrire une fonction `chaine_min` qui prend en argument une matrice `pred` telle que décrite à la Q 28 et deux sommets s et t et renvoie une chaîne de poids minimal de s à t sous la forme d'une liste de ses sommets. La fonction renverra une liste vide s'il n'existe pas de tel chemin.

```
chaine_min : int array array -> int -> int -> int list
```

IV.B – Solution approchée

En considérant $G = (S, A, f)$ un graphe pondéré non orienté connexe et $X \subset S$, la solution approchée pour le problème de l'arbre de Steiner est donnée par l'algorithme suivant :

— poser $H_1 = (X, \mathcal{P}_2(X), g)$: H_1 est un graphe complet entre les sommets de X, dont les pondérations sont données par les distances dans G (c'est-à-dire $g(st) = d_G(s,t)$) ;

— calculer $T_1 = (X, B_1, g)$ un arbre couvrant de poids minimal de H_1 ;

— pour chaque arête $st \in B_1 \setminus A$, la remplacer dans T_1 par une chaîne de poids minimal dans G, en rajoutant les sommets nécessaires. Le sous-graphe de G obtenu sera noté $H_2 = (Y, A_2, f)$.

— calculer et renvoyer $T = (Y, B, f)$ un arbre couvrant de poids minimal de H_2.

Q 31. Montrer que l'arbre T est un arbre de Steiner de G ayant X pour sommets terminaux.

Q 32. Montrer que si T possède des sommets de Steiner de degré 1, leur suppression permet d'obtenir un arbre de Steiner ayant X pour sommets terminaux de poids inférieur.

On pose T^* un arbre de Steiner de poids minimal parmi ceux ayant X comme sommets terminaux.

Q 33. Montrer que $f(T) \leqslant g(T_1)$, puis en déduire que $f(T) \leqslant 2f(T^*)$.

On représente une partie $X \subset S$ comme un tableau de booléens `tab_X` de taille $n = |S|$ tel que pour $s \in S$, $s \in X$ si et seulement si `tab_X.(s)` vaut `true`.

Pour créer un graphe induit par une partie X en OCaml, il est nécessaire de renuméroter des sommets, via une bijection $num : X \to [\![0, |X|-1]\!]$.

Q 34. Écrire une fonction `renumeroter` qui prend en argument une partie $X \subset S$ et renvoie un tableau `num` d'entiers de taille n tel que pour $s \in S$, `num.(s)` vaut -1 si $s \notin X$ et `num.(s)` $= num(s)$ sinon, où num est une bijection de $X \to [\![0, |X|-1]\!]$ choisie arbitrairement.

```
renumeroter : bool array -> int array
```

Q 35. En déduire une fonction `steiner_approche` qui, étant donné un graphe $G = (S, A, f)$ et une partie $X \subset S$ calcule un arbre de Steiner ayant X pour sommets terminaux selon l'algorithme présenté précédemment. On ne demande pas de supprimer les sommets de Steiner de degré 1.

```
steiner_approche : graphe -> bool array -> graphe
```

Q 36. Déterminer la complexité temporelle de la fonction `steiner_approche` en fonction de $|S|$, $|A|$ et $|X|$.

BY-NC-SA

Annexe : rappels de programmation

Cette annexe rappelle le fonctionnement de différentes fonctions des modules `List` et `Array` :

— `List.iter (f: 'a -> unit) (lst: 'a list) : unit` fait un appel à la fonction `f` pour chaque élément de la liste `lst`. Cette fonction s'exécute en temps linéaire en la taille de la liste, si la fonction `f` se calcule en temps constant ;

— `List.rev (lst: 'a list) : 'a list` renvoie une liste contenant les mêmes éléments que `lst`, mais dans l'ordre inverse. Cette fonction s'exécute en temps linéaire en la taille de la liste ;

— `Array.length (tab: 'a array) : int` renvoie la longueur du tableau `tab`. Cette fonction s'exécute en temps constant ;

— `Array.make (n: int) (x: 'a) : 'a array` renvoie un tableau de longueur n dont toutes les cases contiennent la valeur `x`. Cette fonction s'exécute en temps linéaire en n ;

— `Array.make_matrix (m: int) (n: int) (x: 'a) : 'a array array` renvoie une matrice de dimension $m \times n$ dont toutes les cases contiennent la valeur `x`. Cette fonction s'exécute en temps linéaire en $m \times n$.

• • • FIN • • •

BY-NC-SA

SESSION 2024 **MPI5IN**

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 - Algorithme Single Pass

Cette partie comporte des questions nécessitant un code en **langage C**.

On cherche à classer $n = 5$ documents textuels $doc_i, i \in [\![1, n]\!]$ dans lesquels les termes T_1, T_2 et T_3 apparaissent un certain nombre de fois, ces occurrences étant décrites dans le tableau suivant :

	doc_1	doc_2	doc_3	doc_4	doc_5
T_1	1	2	0	1	1
T_2	3	1	1	3	0
T_3	3	0	0	1	1

Chaque document doc_i est donc représenté par un ensemble de $p = 3$ valeurs. On notera $d_i, i \in [\![1, n]\!]$ un vecteur de $\mathbb{N}^3$, de composantes $d_{ij}, j \in [\![1, p]\!]$, d_{ij} indiquant le nombre d'occurrences du terme T_j dans le document doc_i.
On cherche à voir si des textes traitent des mêmes thématiques, en faisant l'hypothèse que des textes sont sémantiquement proches si des termes communs apparaissent.

Q1. Recopier et remplir le tableau suivant en appliquant l'algorithme des k-moyennes avec $k = 2$ et d_2 et d_5 comme centres de classe initiaux. On utilisera la distance $\delta(d_i, d_j) = \sum_{\ell=1}^{p} |d_{i\ell} - d_{j\ell}|$. On notera de plus c_i les centres de classe et $\mathcal{A}_i$ les classes correspondantes.

Itération	c_1	c_2	$\mathcal{A}_1$	$\mathcal{A}_2$
1	d_2	d_5		
2				
3				

On souhaite maintenant traiter ce problème par une méthode dite "single pass" décrite dans l'**algorithme 1**.

Algorithme 1 - Algorithme Single Pass

Entrées : $(d_1 \cdots d_n)$ les vecteurs des documents, θ un seuil appartient à $\mathbb{R}$.
Sorties : $(\mathcal{A}_1 \cdots \mathcal{A}_j)$ les classes de centres $(c_1 \cdots c_j)$
début
- $c_1 = d_1$; `// Initialisation`
- $\mathcal{A}_1 = \{d_1\}$;
- $j = 1$;
- **pour** *i de 2 à n* **faire**
  - **pour** *k de 1 à j* **faire**
    - Étape **(i)** Calculer $\delta(d_i, c_k)$;
  - **si** Étape **(ii)** $\delta(d_i, c_k) > \theta \ \forall c_k$ **alors**
    - $j = j + 1$; `// Création d'une nouvelle classe`
    - $\mathcal{A}_j = \{d_i\}$;
    - $c_j = d_i$;
  - **sinon**
    - `// Indice du centre de classe le plus proche de` d_i `au sens de` δ
    - Étape **(iii)** $\ell = \arg\min_{1 \le k < i} (\delta(d_i, c_k))$;
    - $\mathcal{A}_\ell = \mathcal{A}_\ell \cup \{d_i\}$; `// Affectation de` d_i `à la classe` ℓ
    - $c_\ell = \frac{1}{|A_\ell|} \sum_{d_i \in \mathcal{A}_\ell} d_i$; `// Recalcul de` c_ℓ

Q2. Appliquer l'**algorithme 1** avec $\theta = 5.0$. Détailler les résultats des étapes de l'algorithme.

On propose d'implémenter cet algorithme en langage C. À cet effet, on définit un type structuré `vecteur` permettant d'encoder les centres de classe et les textes.

```
struct vecteur_s {
    double *v;          // pointeur vers les coordonnées
    int taille;         // taille du vecteur
    int num_classe;     // classe du vecteur
};
typedef struct vecteur_s vecteur;
```

Puisque le nombre de centres de classe varie au cours de l'algorithme, on utilise une liste chaînée de vecteurs pour représenter l'ensemble des centres de classe.

```
struct noeud_s {
    vecteur *c;
    struct noeud_s *suivant;
};
typedef struct noeud_s noeud;
```

Q3. Écrire une fonction de prototype `void ajoutVecteur(vecteur *vec, noeud **tete)` permettant d'ajouter un vecteur `vec` en tête de la liste des centres de classe pointée par `tete`. On prendra soin de vérifier que l'allocation mémoire s'est bien passée.

On suppose dans la suite disposer :
- de la variable `vecteur *documents[nb_documents]` qui contient l'ensemble des documents, où pour tout $i \in [\![0, \texttt{nb_documents} - 1]\!]$, `documents[i]` est égal à d_{i+1}.
- d'une fonction `void recalculCentre (vecteur *documents, noeud **tete, int l)` qui effectue le recalcul du centre c_ℓ et met à jour le nœud correspondant dans la liste chaînée des centres de classe.

Q4. Écrire une fonction de prototype `double delta(vecteur *di,vecteur *c)` qui calcule la distance entre le document `di` et le centre de classe `c` (étape **(i)** de l'**algorithme** 1). On suppose que d_i et c ont la même taille.

Q5. Écrire une fonction de prototype `bool distmax(double dists[],int j,double theta)` qui réalise l'étape **(ii)** de l'**algorithme** 1. Le tableau `dists` contient les distances de d_i à tous les c_k : pour tout $k \in [\![0, j-1]\!]$ l'élément `dists[k]` du tableau `dists` contient la distance de d_i à c_{k+1}. La fonction renvoie `true` si $\delta(d_i, c_k) > \theta \ \forall c_k$ et `false` sinon.

Q6. Écrire une fonction de prototype `int distmin(double dists[],int j)` qui réalise l'étape **(iii)** de l'**algorithme** 1. Le tableau `dists` contient les distances de d_i à tous les c_k comme dans la question précédente. La fonction renvoie l'entier l décrit dans l'algorithme.

Q7. À l'aide des questions précédentes et de la fonction `recalculCentre`, proposer une implémentation de l'**algorithme** 1 sous la forme d'une fonction de prototype
`noeud *algorithme1(vecteur *documents, int nb_documents, double theta)`.
Évaluer la complexité de l'**algorithme** 1.

Partie II - Langages réguliers

Soit $\Sigma = \{a, b, c\}$ un alphabet. Σ^* est l'ensemble des mots de longueur finie sur l'alphabet Σ. Pour $u \in \Sigma^*$, $|u| \in \mathbb{N}$ désigne la longueur du mot u. On note $u_i, i \in [\![1, |u|]\!]$ la i-ème lettre de u. Pour $k \in [\![1, |u|]\!]$, on note $u[1, k]$ le préfixe de u de longueur k, c'est-à-dire le mot $u_1 ... u_k$.
On définit sur Σ^* la relation symétrique $\mathcal{R}$ de la manière suivante : $u \mathcal{R} v$ s'il existe $w, z \in \Sigma^*$ tels que $u = wabz$ et $v = wbaz$. On note $\mathcal{R}^*$ la fermeture réflexive transitive de $\mathcal{R}$: $u \mathcal{R}^* v$ si $u = v$ ou s'il existe des mots $x_0 \cdots x_n$ tels que :

(i). $x_0 = u$,

(ii). pour $i \in [\![0, n-1]\!], i < min(|u|, |v|),\ x_i \mathcal{R} x_{i+1}$,

(iii). $x_n = v$.

Q8. Montrer que $\mathcal{R}^*$ est une relation d'équivalence.

Soit $u \in \Sigma^*$. La classe de u modulo $\mathcal{R}^*$ est l'ensemble des mots v vérifiant $v \mathcal{R}^* u$. Tous les mots de cette classe ont la même longueur. Ainsi, par exemple, {abcabb, abcbab, abcbba, bacabb, bacbab, bacbba} est une classe modulo $\mathcal{R}^*$.

Q9. Montrer que les classes modulo $\mathcal{R}^*$ sont des langages réguliers.

Q10. Donner les classes de mots de longueur 3.

On définit la relation $\leq$ sur Σ telle que $a < b < c$ par : pour $u, v \in \Sigma^*$, $u \leq v$ si l'une des deux conditions suivantes est réalisée :

(i). $u = v[1, |u|]$,

(ii). il existe $i > 1$ tel que $u[1, i-1] = v[1, i-1]$ et $u_i < v_i$.

Q11. Montrer que la relation $\leq$ est réflexive et transitive. En déduire que $\leq$ est un ordre total sur Σ^*.

Soit $\mathcal{U}$ une classe modulo $\mathcal{R}^*$. Le *représentant* de $\mathcal{U}$ est le plus petit élément de cette classe pour l'ordre $\leq$. On note $\mathcal{L}$ l'ensemble des représentants des classes modulo $\mathcal{R}^*$.

Q12. Donner le représentant de la classe contenant le mot bacbab.

Q13. Donner une expression régulière du langage régulier $\mathcal{L}$.

Q14. Proposer un automate fini déterministe complet reconnaissant $\mathcal{L}$.

Partie III - Correspondance de Burge

La correspondance de Burge permet d'exprimer une bijection entre des graphes simples et des tableaux de Young semi-standards. Ces objets combinatoires ont de nombreuses applications, notamment dans l'étude de groupes symétriques et la géométrique algébrique. En théorie des graphes, ils permettent également de trouver, s'ils existent, les graphes simples dont les sommets sont contraints à avoir des degrés donnés.

Cette partie comporte des questions nécessitant un **code OCaml**. Pour ces questions, **les réponses ne feront pas appel aux fonctionnalités impératives du langage** (références, champs mutables, exceptions).

Notations
Dans toute la suite on notera :
- $[l] = [\![1, l]\!]$ l'ensemble des entiers naturels de 1 à l,
- $\left|E\right|$ le cardinal de l'ensemble E,
- $G = ([l], A)$ un *graphe simple*, c'est-à-dire un graphe non orienté à l sommets et m arêtes, sans boucles ni arêtes multiples,
- $d_i = \left|\{j \in [l], (i, j) \in A\}\right|$ le degré du sommet $i \in [l]$ dans le graphe $G = ([l], A)$.

De plus, les sommets de G seront ordonnés par degrés décroissants, de sorte que $d_1 \geq d_2 \geq \cdots \geq d_l \geq 0$.

Définition 1 (Suite de degrés d'un graphe)
Soit $G = ([l], A)$ un graphe simple. Si $d_i, i \in [l]$ est le degré du sommet i, avec $d_1 \geq d_2 \cdots \geq d_l \geq 0$, la *suite de degrés* de G est le l-uplet $(d_1, d_2 \cdots d_l)$.

III.1 - Partitions d'un entier

Définition 2 (Partition)
Une *partition* d'un entier $n \in \mathbb{N}^*$, notée $\lambda \vdash n$, est une suite décroissante $\lambda = (\lambda_1, \lambda_2, \cdots \lambda_l)$ de nombres entiers strictement positifs de somme n.

Par exemple, $n = 4$ a cinq partitions : (4), (3,1), (2,2), (2,1,1) et (1,1,1,1).

Une partition $\lambda = (\lambda_1 \cdots \lambda_l) \vdash n$ peut être vue comme une suite de degrés sous certaines conditions.

Q15. Montrer que si $\sum_{i=1}^{l} \lambda_i$ est impaire, alors la partition $\lambda \vdash n$ ne peut pas générer de graphe simple ayant la suite de degrés $(\lambda_1, \lambda_2 \cdots \lambda_l)$.

Dans la suite, on considérera que la somme des termes de la suite d'entiers $(\lambda_1, \lambda_2 \cdots \lambda_l)$ est paire. Même avec cette contrainte, cette suite peut ne pas pouvoir générer de graphe simple.

Définition 3 (Suite graphique)
Une suite d'entiers $d_1 \geq d_2 \cdots \geq d_l \geq 0$ est dite *graphique* s'il existe un graphe simple dont la suite des degrés est $(d_1, d_2 \cdots d_l)$.

Notons qu'une suite graphique vérifie par définition $d_1 \geq d_2 \cdots \geq d_l$.

Pour déterminer si une suite d'entiers donnée est graphique, on peut utiliser l'algorithme d'Havel-Hakimi, basé sur le théorème suivant :

Théorème 1 (Havel-Hakimi)

(i). Pour tout $(d_1, ..., d_l) \in \mathbb{N}^l$, si $(d_1, \cdots, d_l)$ est graphique, alors $d_{d_1+1} > 0$ et toute permutation décroissante de $(d_2 - 1, d_3 - 1, \cdots, d_{d_1+1} - 1, d_{d_1+2}, \cdots, d_l)$ est graphique.

(ii). Réciproquement, pour tout $(d_2, \cdots, d_l) \in \mathbb{N}^{l-1}$, si $(d_2, \cdots, d_l)$ est graphique, alors pour $d_1 \in [\![d_2 + 1, l - 1]\!]$, la suite $(d_1, d_2 + 1, \cdots, d_{d_1+1} + 1, d_{d_1+2}, \cdots, d_l)$ est graphique.

Q16. Montrer que si $(d_2 \cdots d_l)$ est graphique, alors il existe un graphe simple $G = (S, A)$ à l sommets, de sommet 1 de degré $d_1 \in [\![d_2 + 1, l - 1]\!]$ tel que $(d_1, d_2 + 1 \cdots d_{d_1+1} + 1, d_{d_1+2}, d_{d_1+3}, \cdots d_l)$ soit la suite des degrés des sommets de G.

Q17. On suppose que $(d_1, d_2 \cdots, d_l)$ est graphique. Montrer qu'il existe $G = (S, A)$, $S = [l]$, le degré du sommet i étant d_i pour tout $i \in [l]$, tel que : $\forall j \in [\![2, d_1 + 1]\!]$, $(1, j) \in A$. Pour cela, on pourra raisonner par l'absurde et supposer que le sommet 1 est adjacent à un maximum de sommets dans $(2 \cdots d_1 + 1)$, mais pas à tous.

Q18. En déduire que si $d_{d_1+1} > d_{d_1+2}$, alors $(d_2 - 1, d_3 - 1 \cdots d_{d_1+1} - 1, d_{d_1+2}, d_{d_1+3} \cdots d_l)$ est graphique.

Q19. Déterminer si les suites (4,3,3,3,3) et (6,4,4,2,2,1,1) sont graphiques.

Q20. Écrire une fonction de signature `compare_entiers : int -> int -> int` qui compare deux entiers et telle que l'appel à `compare_entiers m n` renvoie 1 si `m< n`, -1 si `m> n` et 0 sinon.

Q21. Écrire une fonction de signature `decr_n : n -> int list -> int list option` telle que pour tout $n \geq 0$ l'appel `decr_n n l` retourne `Some l'`, avec `l'` la liste `l` dont les n premiers éléments sont décrémentés, s'ils étaient tous supérieurs à 1 et `None` sinon.

Q22. Écrire une fonction récursive de signature `havel_hakimi : int list -> bool` qui retourne `true` si la liste passée en entrée, triée par ordre décroissant, est graphique et `false` sinon. À chaque étape utilisant le **théorème 1**, il sera nécessaire de réordonner par ordre décroissant la liste `list` construite, ce qui pourra être fait à l'aide de l'appel à `List.sort compare_entiers list`.

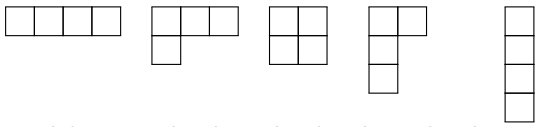
Q23. Donner deux graphes simples à $l = 5$ sommets ayant une même suite de degrés $(3, 2, 2, 2, 1)$.

Ainsi, il n'existe pas de correspondance univoque entre suite de degrés et graphe simple.

III.2 - Diagramme et tableau de Young

Définition 4 (Diagramme de Young d'une partition)
Le diagramme de Young de forme λ, noté $Y(\lambda)$, d'une partition $\lambda = (\lambda_1, \lambda_2, \cdots \lambda_l) \vdash n$ est un tableau de cases constitué de l lignes alignées à gauche, chaque ligne $i \in [l]$ ayant λ_i cases.
Par convention, la ligne associée à λ_1 est la première ligne du tableau.

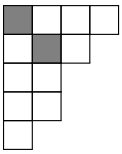
Par exemple, les diagrammes de Young des partitions de l'entier 4 sont



(4) (3,1) (2,2) (2,1,1) (1,1,1,1)

Définition 5 (Diagonale d'un diagramme de Young)
Soit $Y(\lambda)$ un diagramme de Young. La diagonale de $Y(\lambda)$ est définie par les r cases $(i, i), i \in [r]$.

Dans le tableau suivant, $r = 2$ et les cases de la diagonale sont grisées.



Définition 6 (Partition conjuguée)
Soit $Y(\lambda)$ un diagramme de Young associé à $\lambda \vdash n$. La partition conjuguée de λ, notée $\lambda^* = (\lambda_1^*, \cdots, \lambda_k^*)$, est la partition obtenue en énumérant le nombre de cases de $Y(\lambda)$ par colonne, en partant de la colonne de gauche.

On remarque immédiatement que pour tout j, $\lambda^*_j = \left|\{i, \lambda_i \geq j\}\right|$.

Q24. Soit $\lambda = (5, 4, 1, 1, 1, 1) \vdash 13$. Donner λ^*.

Définition 7 (Représentation de Frobenius d'un diagramme de Young)
Soient $(\lambda_1, \lambda_2, \cdots \lambda_l) \vdash n$ une partition, $Y(\lambda)$ son diagramme de Young et r la taille de sa diagonale. Pour $i \in [r]$, soient $\alpha_i = \lambda_i - i$ le nombre de cases à droite de la case (i, i) dans la i-ème ligne de $Y(\lambda)$ et $\beta_i = \lambda^*_i - i$ le nombre de cases en-dessous de la case (i, i) dans la i-ème colonne de $Y(\lambda)$. Alors $\alpha_1 > \alpha_2 \geq \cdots \alpha_r \geq 0$, $\beta_1 > \beta_2 \geq \cdots \beta_r \geq 0$ et la notation de Frobenius de λ est donnée par $\lambda = \begin{pmatrix} \alpha_1 & \alpha_2 & \cdots \alpha_r \\ \beta_1 & \beta_2 & \cdots \beta_r \end{pmatrix}$.

Q25. Soit $(4, 3, 2, 2, 1) \vdash 12$. Donner la représentation de Frobenius de $Y(\lambda)$.

Définition 8 (Tableau de Young)
Soit $\lambda \vdash n$ une partition de $n > 0$. Un λ-tableau de Young est un tableau obtenu en remplissant les cases d'un diagramme Y(λ) par des entiers dans $[n]$.

Par exemple, pour $\lambda = (2, 1) \vdash 3$, les tableaux suivants sont des λ-tableaux

$$\begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 \\ \cline{1-1}\end{array} \quad \begin{array}{|c|c|}\hline 2 & 1 \\ \hline 3 \\ \cline{1-1}\end{array} \quad \begin{array}{|c|c|}\hline 1 & 3 \\ \hline 2 \\ \cline{1-1}\end{array} \quad \begin{array}{|c|c|}\hline 3 & 1 \\ \hline 2 \\ \cline{1-1}\end{array} \quad \begin{array}{|c|c|}\hline 2 & 3 \\ \hline 1 \\ \cline{1-1}\end{array} \quad \begin{array}{|c|c|}\hline 3 & 2 \\ \hline 1 \\ \cline{1-1}\end{array} \quad \begin{array}{|c|c|}\hline 1 & 1 \\ \hline 2 \\ \cline{1-1}\end{array} \quad \begin{array}{|c|c|}\hline 1 & 3 \\ \hline 1 \\ \cline{1-1}\end{array}$$

Dans la suite, on notera $T(i, j)$ l'entier en position (i, j) dans le tableau T.

Définition 9 (Tableau de Young (semi)standard)
Un tableau de Young est dit *semi standard* si les éléments de chaque ligne (respectivement colonne) forment une suite croissante de gauche à droite (respectivement strictement croissante de haut en bas). Il est dit *standard* s'il est semi-standard et si les entiers de 1 à n n'apparaissent qu'une et une seule fois.

Dans les exemples précédents, les premier, troisième et septième tableaux sont semi standards. Les premier et troisième tableaux sont standards.

III.3 - Insertion de Schensted

Pour construire un tableau de Young semi-standard à partir d'une suite d'entiers $d_1, \cdots d_l$, on utilise une méthode d'insertion proposée par Schensted.
On cherche à insérer dans un tableau de Young semi standard T un entier k, de sorte à ce que le tableau créé (contenant une case de plus) soit toujours un tableau de Young semi standard. On note cette opération d'insertion $T \leftarrow k$.
L'entier k est inséré dans T en le comparant aux valeurs de la première ligne et en déplaçant le premier entier plus grand que k en lisant la ligne de gauche à droite. S'il n'y a pas de plus grand élément, k est ajouté à la fin de la ligne. Si un élément est déplacé, il est inséré dans la deuxième ligne et les lignes suivantes du tableau sont traitées de la même manière. L'**algorithme 2** décrit l'insertion de k dans la i-ème ligne de T.

Soit T_0 le tableau de Young semi standard $\begin{array}{|c|c|c|c|c|}\hline 1 & 1 & 2 & 2 & 4 \\ \hline 2 & 3 & 3 \\ \cline{1-3} 3 \\ \cline{1-1} 4 \\ \cline{1-1}\end{array}$

Q26. Insérer la valeur 5 dans T_0 et donner le tableau de Young semi standard résultant. Même question pour l'insertion de la valeur 1 dans T_0.

Q27. Énoncer une précondition de l'**algorithme 2** permettant de prouver sa correction, c'est-à-dire qu'il retourne bien un tableau de Young semi standard contenant k.

Pour coder les tableaux de Young, on choisit d'utiliser un type OCaml `type tableau = int list list` et on encode la structure par liste de lignes.

Algorithme 2 - Algorithme d'insertion d'un entier k dans la ligne i d'un tableau de Young semi standard

Insertion(T,k,i)
Entrées : un tableau de Young semi standard T, un entier k à insérer, i la ligne traitée
Sorties : un tableau de Young semi standard contenant k
début
  si *(la ligne i est vide) OU (k plus grand que l'élément le plus à droite de la ligne i)* **alors**
    Ajouter k en bout de la ligne i de T
  sinon
    Soit j le plus petit indice tel que $k < T(i, j)$
    $p = T(i, j)$
    $T(i, j) = k$
    Insertion(T,p,$i + 1$)

Q28. Écrire une fonction récursive de signature `insereligne :int -> int list -> int*int list` qui, à partir d'un entier k et d'une ligne i d'un tableau de Young semi standard, retourne un couple (m, r) où
- $m = 0$ et r est la ligne i où k a été ajouté en queue, si k est plus grand que tous les éléments de la ligne i,
- m, qui est dans la ligne i et r est la ligne i où m a été remplacé par k sinon.

Q29. En déduire une fonction récursive de signature `insertion : int -> tableau -> tableau` réalisant l'**algorithme 2**.

Q30. Écrire une fonction récursive de signature `construit_tableau :int list -> tableau` qui construit un tableau semi standard à partir d'une suite d'entiers.

L'**algorithme 2** permet de définir une position (s, t), coordonnées de la case où k a été ajouté à T.

III.4 - Correspondance de Burge

Définition 10 (Tableau de Burge)
Soit $G = ([l], A)$ un graphe simple, $|A| = m$. Le *tableau de Burge* associé à G est défini par

$$\mathcal{B}_G = \begin{pmatrix} u_1 & u_2 & \cdots u_m \\ v_1 & v_2 & \cdots v_m \end{pmatrix}$$

où pour tout $i \in [m]$ (u_i, v_i) est une arête de G, avec $u_i > v_i$, le tableau étant formé de sorte que pour $i \in [m-1]$, $u_i \leq u_{i+1}$ et si $u_i = u_{i+1}$ alors $v_i > v_{i+1}$.

Soit le graphe G de la **figure 1**.

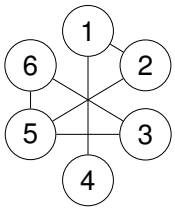


Figure 1 - Graphe exemple

Q31. Donner le tableau $\mathcal{B}_G$ correspondant au graphe de la **figure 1**.

On utilise alors $\mathcal{B}_G$ pour associer à G un diagramme de Young $Y(\lambda)$ représenté par une forme de Frobenius particulière, notée $\mathcal{F}_Y = \begin{pmatrix} \alpha_1 & \alpha_2 & \cdots \alpha_r \\ \alpha_1 + 1 & \alpha_2 + 1 & \cdots \alpha_r + 1 \end{pmatrix}$, où r est le nombre de cases de la diagonale principale du diagramme de Young $Y(\lambda)$ associé.

Q32. Montrer que, dans ce cas, pour tout $i \in [r]$ $\lambda_i^* = \lambda_i + 1$.

Ainsi, $Y(\lambda)$ est divisé en deux parties symétriques. La partie inférieure est constituée de toutes les cases qui se trouvent strictement sous la diagonale et la partie supérieure est constituée du reste. Chaque position dans la partie supérieure (inférieure) du diagramme de Young correspond à une position unique, appelée *position opposée*, dans la partie inférieure (supérieure) de $Y(\lambda)$. Par exemple, dans le diagramme de Young suivant, ayant comme représentation de Frobenius $\mathcal{F}_Y$, les positions opposées sont codées par le même symbole.

$$\mathcal{F}_Y = \begin{pmatrix} 3 & 1 \\ 4 & 2 \end{pmatrix}$$

$\diamond$	$\bullet$	$\times$	$\oplus$
$\diamond$	∇	$\heartsuit$	
$\bullet$	∇		
$\times$	$\heartsuit$		
$\oplus$			

Q33. Soit (s, t) une case. Donner la position opposée en fonction de s et t.

L'**algorithme 3** utilise alors la représentation $\mathcal{B}_G$ d'un graphe G pour construire un tableau de Young semi standard dont le diagramme correspondant $Y(\lambda)$ admet une représentation de Frobenius du type $\mathcal{F}_Y$. Dans cet algorithme, $T \leftarrow v_i$ insère la valeur v_i dans le tableau de Young semi standard T.

Algorithme 3 - Algorithme de Burge

Entrées : $\mathcal{B}_G$ de taille $m \times 2$
Sorties : tableau de Young semi standard T dont la représentation de Frobenius du diagramme correspondant est du type $\mathcal{F}_Y$

```
début
    T = tableau vide ;                                  // Initialisation
    pour i de 1 à m faire
        (s, t) = T ← v_i
        Placer u_i dans la position opposée à (s, t)
```

Cet algorithme produit, à partir du graphe de la **figure 1**, le tableau :

1	1	2	3
2	3	5	
4	5		
5	6		
6			

Q34. Donner les tableaux de Young semi standard intermédiaires produits à chaque itération.

Q35. À quoi correspond le nombre d'apparitions de chaque entier contenu dans les cases du tableau de Young résultat de l'**algorithme 3** ?

Notons pour terminer qu'il est à l'inverse possible de produire un tableau bidimensionnel $\mathcal{B}_G$ (et donc un graphe G) à partir d'un tableau de Young semi standard $Y(\lambda)$ de type $\mathcal{F}_Y$.

FIN

IMPRIMERIE NATIONALE – 24 1124 – D'après documents fournis

SESSION 2024 **MP7IN**

ÉPREUVE MUTUALISÉE AVEC E3A-POLYTECH

ÉPREUVE SPÉCIFIQUE - FILIÈRE MP

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 indépendantes.

Partie I - Coloration de graphes

L'objectif de cette partie est de proposer une implémentation en **Python** d'une solution au problème de coloration d'un graphe.

I.1 - Définitions et propriétés

Soit $G = (S, A)$ un graphe fini non orienté avec S son ensemble de sommets et A son ensemble d'arêtes. On suppose que le graphe est simple c'est-à-dire qu'il ne comporte pas de boucles et que chaque paire de sommets est reliée par au plus une arête. On note n, le cardinal de l'ensemble S. Les sommets sont numérotés de 0 à $n-1$. Étant donné un entier naturel k, une k-coloration des sommets de G est une application $c : S \to \{0, 1, ..., k-1\}$ telle que pour chaque arête $\{x, y\}$ d'extrémités x et y, $c(x) \neq c(y)$. Si $c(x) = i$, on considèrera que la couleur i est affectée au sommet x. Si G admet une k-coloration, il est k-coloriable. On définit le nombre chromatique $\chi(G)$ d'un graphe G fini par $\chi(G) = \min\{k \in \mathbb{N}, \mathrm{G\ est\ } k\text{-coloriable}\}$.
Une clique est un sous-ensemble de sommets du graphe, adjacents 2 à 2. On dit qu'un graphe est complet si il est une clique. On notera K_p le graphe complet à p sommets.
On pose $\omega(G) = \max\left\{p \in \mathbb{N} | K_p \text{ est une clique de G}\right\}$, avec $\mathbb{N}$ l'ensemble des entiers naturels.

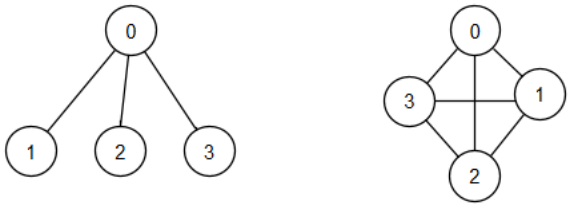


Figure 1 - de gauche à doite, le graphe G_1 et le graphe K_4

Q1. Le graphe G_1 de la **figure 1** ci-dessus est-il 2-coloriable ? Justifier votre réponse.

Q2. Pour un entier naturel $n \geq 1$, déterminer le nombre chromatique du graphe K_n.

Q3. Montrer que pour tout graphe G à n sommets, on a $\omega(G) \leq \chi(G) \leq n$.

I.2 - Algorithmique et programmation en Python (Informatique Commune)

La coloration d'un graphe G avec $\chi(G)$ couleurs est un problème complexe. Dans cette sous-partie, nous présentons une heuristique permettant de construire une coloration d'un graphe donné.
Dans la suite, on implémente un graphe par **un dictionnaire (type `dict` en Python), contenant les listes d'adjacence des sommets**. Les clés du dictionnaire sont les numéros des sommets et la valeur correspondant à la clé `i` du dictionnaire est la liste d'adjacence du sommet numéro `i`.
Notons qu'il serait ici possible d'implémenter le graphe par des listes d'adjacence dans un tableau, dans la mesure où les sommets sont numérotés de 0 à $n-1$.

Q4. Définir en Python le dictionnaire `d1` correspondant au graphe G_1.

Q5. Écrire une fonction Python `degres_sommets(d)` qui prend en paramètre un dictionnaire `d` représentant un graphe et renvoie une liste de couples `(di,i)`, `i` étant le numéro d'un sommet parcourant l'ensemble $\{0, ..., n-1\}$ et `di` le degré du sommet `i`.

Q6. On suppose qu'on dispose d'une fonction Python `tri(l)` qui trie une liste de couples `l` dans l'ordre décroissant par rapport à la première composante du couple. En déduire une fonction `tri_degres(d)` qui prend en paramètre un dictionnaire `d` représentant un graphe et renvoie une liste contenant les numéros des sommets, triés par degrés décroissants.

Q7. Écrire une fonction Python `test(d,c)` qui prend en paramètre un dictionnaire `d` représentant un graphe G, un dictionnaire `c` dont les clés représentent les sommets du graphe G et les valeurs, leurs couleurs. La fonction renvoie `True` si `c` est une 2-coloration pour G.

On considère ci-dessous, l'algorithme de coloriage de Welsh-Powel.

Algorithme 1 : Welsh-Powel (coloration de graphe)

Entrée : un graphe `G` à `n` sommets
Sortie : liste d'entiers contenant en position `i` la couleur du sommet numéro `i`

```
Début
    Ordonner les sommets selon les degrés décroissants dans une liste li ;
    colorie : dictionnaire vide qui à terme, associera à chaque clé i, la couleur du sommet i ;
    Tant qu' il reste des sommets à colorier faire
        Chercher dans li le premier sommet non colorié et le colorier avec la plus petite couleur
          c non utilisée ;
        Colorier avec cette même couleur, en respectant leur ordre dans li, tous les sommets
          non coloriés et non adjacents à des sommets de couleur c ;
    Fin
    Retourner colorie
Fin
```

Q8. Que contient `colorie` si on déroule l'algorithme de coloriage ci-dessus avec le graphe G_1 en entrée ?

Q9. Écrire une fonction Python `adjacent(d,dc,s,c)` qui prend en paramètre un graphe représenté par un dictionnaire `d`, un dictionnaire `dc` contenant la couleur des sommets coloriés, le numéro d'un sommet `s`, une couleur `c` et renvoie `True` si le sommet `s` est adjacent à l'un des sommets de couleur `c`, `False` sinon.

Q10. Proposer une implémentation en Python de l'algorithme de Welsh-Powel.

Application
Le tableau ci-dessous représente les liens d'amitiés entre huit étudiants : Alice (A), Béatrice (B), Carl (C), David (D), Eloïs (E), Fanny (F), Gary(G) et Hedge (H).

Prénom	A	B	C	D	E	F	G	H
Ami·e avec	B,C,G	A,C,E,F	A, B	E,F	B,D,F	B,D,E,H	A,H	F,G

On souhaite créer des groupes de travail. Dans le contexte de l'application, un groupe contient au moins 2 étudiants tel que chaque étudiant soit dans un groupe différent de celui de ses amis.
Q11. Modéliser la situation par un graphe et en déduire une solution.

Partie II - Satisfiabilité d'une formule propositionnelle

Le langage utilisé dans cette partie est OCaml.
Une formule propositionnelle est construite à l'aide de constantes propositionnelles, de variables propositionnelles et de connecteurs logiques. Les connecteurs logiques seront notés $\neg$ (négation), $\wedge$ (conjonction), $\vee$ (disjonction). Dans cette partie, on étudie le problème de satisfiabilité d'une formule et son application à la détermination d'une conséquence logique entre 2 formules propositionnelles.
Le problème CNF-SAT est défini de la façon suivante. Étant donné une formule sous forme normale conjonctive, admet-elle un modèle, c'est-à-dire une valuation des variables, qui rende la formule vraie ? On souhaite écrire un programme qui teste si une valuation donnée rend une telle formule vraie.
Dans cette partie, on considère que si une formule contient n variables propositionnelles, elles seront désignées par $x_0, x_1, ..., x_{n-1}$.
On définit le type OCaml suivant :

```
type clause=Var of int|Non of clause |Ou of clause*clause
```

L'argument du constructeur `Var` correspond au numéro de la variable concernée.
Une formule sous forme normale conjonctive ayant m clauses sera implémentée par une liste de m clauses. Les tableaux seront implémentés par le module `Array` dont les éléments suivants pourront être utilisés :

- `type 'a array`, notations `[| |]`
- création d'un tableau : `make : int -> 'a -> 'a array`
- accès à l'élément d'indice `i` du tableau `t` : `t.(i)`
- modification de l'élément placé à l'indice `i` du tableau `t` : `t.(i) <- v`
- taille du tableau : `length : 'a array -> int`

Q12. Donner le code OCaml correspondant à la clause $c = (x_0 \vee x_1) \vee \neg x_2$.

Q13. Donner le code OCaml permettant de définir la formule : $f = (x_0 \vee x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$.

Q14. Écrire une fonction de signature `evalue_clause : clause -> bool array -> bool` qui prend en paramètre une clause et une valuation représentée par un tableau contenant à l'indice i, la valeur de vérité de la variable x_i et renvoie la valeur de vérité de la clause.

Q15. Écrire une fonction de signature `evalue_FNC : clause list -> bool array -> bool` qui prend en paramètre une clause et une valuation représentée par un tableau contenant à l'indice i, la valeur de vérité de la variable x_i et évalue une formule donnée sous forme normale conjonctive.

Q16. Quel résultat obtient-on avec la formule F et le tableau de valuations `[|false;true;true|]` ? Justifier.

Q17. On souhaite énumérer toutes les valuations possibles pour un nombre de variables fixé. Étant donné une valuation, on considérera que si la valeur `true` correspond à 1 et la valeur `false` correspond à 0, la valuation suivante correspond à l'ajout de 1 au nombre binaire associé. Ainsi, la valuation suivante de `[|false;true;false|]` est `[|false;true;true|]`. On considère que la valuation suivante de `[|true;true;true|]` n'existe pas.
Écrire une fonction de signature `suivant : bool array -> bool` qui prend en paramètre un tableau de booléens, lui attribue la valuation "suivante" si possible et renvoie `true` ; sinon renvoie `false`.

Q18. En déduire une fonction de signature `satisfiable : clause list -> int -> bool` qui prend en paramètre une formule en forme normale conjonctive, son nombre de variables et renvoie `true` si il existe une valuation qui rend la formule vraie, `false` sinon.

Q19. Quelle est la complexité en temps de cette fonction par rapport aux paramètres d'entrée ?

Q20. Proposer une stratégie de retour sur trace pour résoudre le problème de satisfiabilité d'une formule.

Conséquence logique entre 2 formules

Définition

Une formule ϕ est une conséquence logique d'un ensemble fini de n formules $\Gamma = \{F_1, ..., F_n\}$, n étant un entier naturel supérieur ou égal à 1, si tout modèle de ϕ est un modèle de Γ. On note $\Gamma \vDash \phi$. On admettra que toute formule admet une formule équivalente sous forme normale conjonctive.

Q21. Déduire de la fonction précédente, un algorithme en pseudo-code permettant de déterminer si une formule F est une conséquence logique d'un ensemble de formules $\Gamma : F_1, ..., F_n$.

Q22. Afin de déterminer si $\Gamma \vDash \phi$, on peut prouver le séquent $\Gamma \vdash \phi$. Justifier cette méthode, puis construire un arbre de preuve qui démontre le séquent $\Gamma : P \to Q, Q \to R, P \vdash P$ où P, Q, R désignent des variables propositionnelles représentant des formules logiques, à partir des règles d'inférence de la déduction naturelle ; les règles et notations utilisées seront clairement mentionnées.

Partie III - Automates et reconnaissance de motifs

Dans cette partie, le langage utilisé est OCaml.

On s'intéresse à la reconnaissance de motifs dans un texte en utilisant une méthode naïve, puis des automates.

III.1 - Autour de l'algorithme naïf de recherche de caractères

Définitions

Un alphabet A est un ensemble fini non vide d'éléments appelés lettres. Un mot défini sur A est une suite finie d'éléments de A. Le mot vide sera noté ϵ. L'ensemble des mots sur l'alphabet A est noté A^*. La longueur d'un mot x, notée $|x|$ est le nombre de lettres de ce mot. Pour i allant de 0 à $|x|-1$, x_i désignera la lettre en position i . Ainsi, aab est un mot de longueur 3, sur tout alphabet contenant les lettres a et b.
Un mot x est un facteur d'un mot y si il existe 2 mots u et v tels que $y = uxv$. Dans ce cas, on dit qu'il y a une occurrence de x dans y. Quand $u = \epsilon$, x est un préfixe de y et quand $v = \epsilon$, x est un suffixe de y.
Soit x un mot non vide. Un entier $0 < p \leq |x|$ est une période de x si $x_i = x_{i+p}$ pour $0 \leq i \leq |x| - p - 1$. On définit la période de x comme étant la plus petite des périodes.
Un bord d'un mot non vide x est un facteur de x, différent de x et qui est à la fois un préfixe et un suffixe de la chaîne de caractères. Par exemple, ϵ, a,aa,$aabaa$ sont les bords du mot $aabaabaa$. On note $Bord(x)$ le plus long bord de x. On constate que lorsqu'il est défini, le bord d'un bord d'un mot x est aussi un bord de x. On le note $Bord^2(x)$. On définit ainsi récursivement $Bord^n(x)$ avec n un entier naturel.

En OCaml, les fonctions suivantes pourront être utilisées sur le type `string` :
- `String.length` : longueur de la chaîne de caractères
- `s.[i]` : accès à la lettre d'indice i de la chaîne s
- `^` : opérateur concaténation

Q23. Justifier le fait que tout mot non vide possède au moins une période.

Q24. Écrire une fonction de signature `occurrence : string -> string -> bool` qui prend en paramètre 2 chaînes de caractères `x` , `y` et renvoie `true` si il existe une occurrence de `x` dans `y`, `false` sinon.

Q25. Quelle est la complexité en temps de la fonction `occurrence` en fonction de la longueur des chaînes de caractères en entrée ?

Q26. Déterminer la période du mot $x = aabaabaa$ défini sur l'alphabet $A = \{a, b\}$.

Q27. Écrire une fonction de signature `periode : string -> int` qui renvoie la période d'une chaîne de caractères.

Q28. Soit x une chaîne de caractères non vide et p un entier naturel tel que $0 < p \leq |x|$. Montrer que p est une période de x si et seulement si il existe 3 chaînes de caractères u, v, w telles que $x = uw = wv$ et $|u| = |v| = p$.

Q29. Soit x une chaîne de caractères non vide et n le plus grand entier tel que $Bord^n(x)$ est bien défini. On a donc $Bord^n(x) = \epsilon$. Montrer par récurrence sur la longueur des mots que $Bord(x), Bord^2(x), ..., Bord^n(x)$ est la suite des bords de x dans l'ordre décroissant de leur longueur et que $|x| - |Bord(x)|, |x| - Bord^2(x)|, ..., |x| - |Bord^n(x)|$ est la suite des périodes de x dans l'ordre croissant.

Q30. Expliquer brièvement comment la connaissance des bords peut permettre d'améliorer la complexité en temps dans le pire cas, de l'algorithme naïf de recherche d'une occurrence d'un mot.

III.2 - Localisation des occurrences d'un motif à l'aide d'un automate

Un automate déterministe M sur l'alphabet A est composé d'un ensemble d'états finis Q, d'un état initial q_0, d'un ensemble d'états terminaux $T \subseteq Q$ et d'une fonction de transitions $\delta : Q \times A \to Q$. On le désigne par un quintuplet (Q, A, q_0, T, δ). On dit qu'il est complet si δ est définie pour chaque élément de $Q \times A$.
Certains automates peuvent être utilisés comme machine de recherche pour le traitement séquentiel de textes. Étant donné un alphabet A, un motif X représente un langage non vide ne contenant pas le mot vide. Le problème de la recherche de motifs est celui de la localisation d'occurrences de mots du langage dans d'autres mots. On suppose qu'on dispose d'un automate déterministe M complet qui reconnaît le langage A^*X, autrement dit, l'automate M reconnaît les mots qui ont comme suffixe un mot de X.
L'algorithme suivant permet de localiser les mots de X reconnus dans un texte t, considéré comme une chaîne de caractères.
Dans la suite, l'automate M de la **figure 2** ci-après sera cité en exemple. Il est défini sur l'alphabet $A = \{a, b, c\}$

Algorithme 2 : reconnaissance de mots dans un texte

```
1  Cherche(M,t);
2  Début
3      e← initial[M];
4      lst← liste vide;
5      Pour chaque lettre λ de t prise dans l'ordre croissant de leurs indices faire
6          e ← δ(e,λ);
7          Si e est un état terminal alors
8              ajouter à lst l'indice de λ dans t;
9          Fin
10     Fin
11     Retourner lst;
12 Fin
```

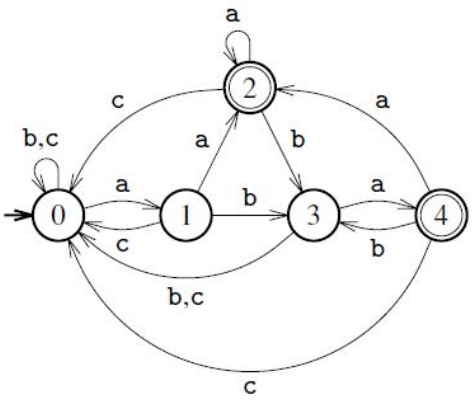


Figure 2 - Automate M

On définit en OCaml un type automate par :

```
type auto={etats: int list; alphabet : char list; initial: int;
transition: int -> char ->int; final : int list};;
```

Q31. Créer une variable `automate` qui représente l'automate M de la **figure 2**.

Q32. L' automate M est-il émondé ? Justifier.

Q33. Présenter les premières étapes de l'algorithme d'élimination des états appliqué à l'automate M. On éliminera l'état 0.

Q34. Que représentent les indices mémorisés dans la liste `lst` de l'algorithme `cherche` ? Que contient la liste `l` de l'algorithme `cherche` avec l'automate ci-dessus et $t = cabaac$?

Q35. Écrire une fonction de signature `cherche : auto -> string -> int list` qui implémente l'algorithme `cherche`.

Q36. Montrer la correction de l'algorithme `cherche`.

FIN

IMPRIMERIE NATIONALE – 24 1094 – D'après documents fournis

Épreuve disciplinaire

Préliminaires

Notes de programmation Python

Il est demandé d'accompagner les programmes Python d'explications de conception rédigées.

Vous disposez, pour répondre aux questions de ce sujet, des fonctions de listes suivantes :

— création d'une liste `li` de taille n dont tous les éléments ont pour valeur v : `li = [v]*n`.
— renvoi de la taille d'une liste `li` : `len(li)`.
— insertion (avec décalage à droite) à un indice i dans une liste `li` : `li.insert(i,v)`.
— ajout de la valeur v en fin de liste : `li.append(v)`.
— position du premier élément de la liste égal à v (en comptant à partir de 0) : `li.index(v)`.
— énumération des éléments d'une liste, dans l'ordre : `for x in li`.
— construction de listes par *compréhension* : `[x**2 for x in li]` est une nouvelle liste construite en mettant au carré chaque élément de la liste `li`.
— les fonctions permettant de sélectionner des tranches (*slices*) : `li[start:end]` sélectionne la sous-liste d'indices de `start` à `end-1`.

Il est autorisé d'utiliser l'évaluation gauche droite paresseuse des expressions booléennes, par exemple, si l'expression `expr_bool1` s'évalue à `False`, alors l'expression `expr_bool1 and f(42)` est directement évaluée à False (sans appel de la fonction `f`).

Enfin, on utilise la possibilité en Python des versions supérieures à 3.10 de décrire les types des paramètres et de retour de fonction, par exemple : `foo(a:int, b:int)->bool` est une fonction nommée `foo`, à deux paramètres entiers, retournant un Booléen.

Notes de complexité

La complexité d'un algorithme est le nombre d'opérations élémentaires réalisées par celui-ci et calculée en fonction des données d'entrées, et selon le contexte (à rappeler), exprimée dans le meilleur cas, le pire cas, ou en moyenne. Cette complexité sera dans tout ce sujet exprimée à l'aide de l'opérateur O, dont on rappelle la définition ici : on dit que $f = O(g)$ s'il existe une constante K et une constante M telle que pour tout $n > M$, $f(n) \leqslant Kg(n)$.

On rappelle que les *listes* Python sont en fait des tableaux redimensionnables. En conséquence, le calcul de la taille, l'accès à un élément et l'ajout en fin de liste sont considérés comme des opérations élémentaires, avec une complexité $O(1)$. L'insertion et le parcours sont des opérations linéaires en la taille de la liste.

Vocabulaire d'arbres

Dans ce sujet :

— les arbres sont des graphes acycliques connectés, enracinés et étiquetés ;
— un nœud sans enfant est dénommé feuille ;
— on appelle la longueur d'un chemin le nombre de nœuds de ce chemin, extrémités inclues ;
— on appelle arité d'un nœud le nombre d'enfants de ce nœud ; par extension on appelle arité d'un arbre le maximum de l'arité de ses nœuds. Un arbre binaire est d'arité 2.
— on appelle distance entre deux nœuds A et B la valeur $l - 1$, où l désigne la longueur du chemin (A, B) ;
— on appelle niveau d'un nœud d'un arbre la distance de ce nœud à la racine de l'arbre (la racine d'un arbre est de niveau 0) ;
— la hauteur d'un arbre est la longueur du plus grand chemin de la racine à une feuille (en nombre de nœuds) ;
— un arbre est équilibré si tous les chemins de la racine aux feuilles ont la même longueur ;

Rappel mathématique

$$\text{Pour } x \neq 1 \ : \ \sum_{i=0}^{i=n-1} x^i = \frac{1-x^n}{1-x}$$

1 Arbres Binaires de Recherche

Un arbre binaire de recherche est un *arbre binaire* (i.e. chaque nœud possède 0, 1 ou 2 enfants), dans lequel les nœuds sont étiquetés par des *entiers naturels* appelés *clefs*. De plus, si l'on considère un nœud de clef k, alors chaque nœud de son sous-arbre gauche (resp. droit) a une clef inférieure ou égale à k (resp. supérieure ou égale). Un exemple de tel arbre est donné à la Figure 1.

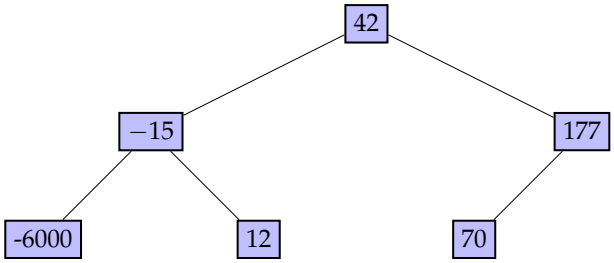


FIGURE 1 – Un arbre binaire de recherche

Nous proposons la définition suivante pour les arbres binaires en Python : tout d'abord, nous définissons un type `Noeud` sous forme d'une classe (un nœud a forcément deux enfants, qui sont soit des nœuds, soit `None`) :

```
class Noeud:
    """
    Noeud d'un arbre
    """
    def __init__(self, clef:int, gauche= None, droite= None):
        """ Initialisation, noeud non vide"""
        self.gauche = gauche
        self.droite = droite
        self.clef = clef
```

Par la suite, un arbre binaire est soit l'arbre vide (`None`) soit enraciné sur un nœud (`Noeud`).
L'arbre de la Figure 1 peut donc être construit de la façon suivante :

```
mon_arbre = Noeud(42, Noeud(-15, Noeud(-6000), Noeud(12)), Noeud(177, Noeud(70), None));
```

À ce stade, rien ne permet de s'assurer que l'arbre construit est un arbre binaire de recherche.

Question #1

Écrire une fonction récursive en Python de recherche d'une clef dans un arbre binaire de recherche (contenant au moins un nœud). Cette fonction aura la signature suivante :
`recherche(root : Noeud, k : int)->bool`

Question #2

Donner la complexité algorithmique de votre fonction en fonction du nombre de nœuds de l'arbre binaire de recherche. On distinguera des cas selon la forme des arbres en entrée.

Question #3

En utilisant un parcours récursif, écrire une fonction `collecte` qui collecte et ajoute dans une liste Python les valeurs des clefs des nœuds d'un arbre binaire de recherche par ordre croissant, et renvoie cette liste. On pourra utiliser la concaténation de listes et l'ajout en fin de liste (avec `append`) sans se préoccuper de la complexité.

Question #4

Nous proposons l'algorithme suivant pour déterminer si un arbre binaire est un arbre binaire de recherche :

```
1     def est_ABR_1(self):
2     """
3     renvoie True ssi l'arbre est un ABR
4     """
5     temp = True
6     val = self.clef
7     if self.gauche is not None:
8         temp = self.gauche.clef < val and self.gauche.est_ABR_1()
9     if self.droite is not None:
10        temp = temp and self.droite.clef > val and self.droite.est_ABR_1()
11     return temp
```

En exhibant un arbre (non feuille) bien choisi, montrer que cet algorithme est incorrect.

Nous désirons maintenant proposer un algorithme correct : celui-ci est basé sur l'invariant structurel suivant : lorsque l'on observe un sous-arbre (d'un arbre binaire de recherche) enraciné en un nœud de clef k :

— Dans le sous-arbre de gauche, le maximum des clefs est inférieur ou égal à k.

— Dans le sous-arbre de droite, le minimum des clefs est supérieur ou égal à k.

Pour simplifier, il est supposé que l'on connaît une borne inférieure (`inf=INF`) et une borne supérieure (`sup=SUP`) des valeurs des clefs de l'arbre. L'algorithme consiste alors à propager un intervalle (initialement égal à `(inf,sup)`) de la racine vers les feuilles à chaque appel récursif du parcours.

Question #5

Recopier sur votre copie l'arbre de la Figure 1 en indiquant à côté de chaque nœud les intervalles propagés par l'algorithme décrit ci-dessus.

Question #6

Sur votre contre-exemple de la question 4, comment cet algorithme va-t-il décider que ce n'est pas un arbre binaire de recherche ?

Question #7

Implémenter la fonction `est_ABR_2_aux(self, inf:int, sup:int)->bool` qui propage les intervalles de haut en bas et décide si l'arbre binaire courant est un arbre binaire de recherche.

2 B-trees

Pour une première définition, nous nous reportons à Wikipédia :

> « En informatique, un arbre B (appelé aussi B-arbre par analogie au terme anglais « B-tree ») est une structure de données en arbre équilibré. Les arbres B sont principalement mis en œuvre dans les mécanismes de gestion de bases de données et de systèmes de fichiers. Ils stockent les données sous une forme triée et permettent une exécution des opérations d'insertion et de suppression en temps toujours logarithmique.
>
> Le principe est de permettre aux nœuds parents de posséder plus de deux nœuds enfants : c'est une généralisation de la notion d'arbre binaire de recherche. Ce principe minimise la taille de l'arbre et réduit le nombre d'opérations d'équilibrage. De plus un B-tree grandit à partir de la racine, contrairement à un arbre binaire de recherche qui croît à partir des feuilles. »

Dans cette partie nous allons tout d'abord implémenter ces B-trees et en caractériser certaines opérations.

2.1 Définition, algorithmes élémentaires

Contrairement à la section précédente, les B-trees ne seront pas binaires, mais d'arité quelconque. De plus, les nœuds ne contiennent pas une clef unique, mais un ensemble de clefs.

Définition
Un B-tree d'ordre t (cf. Figure 2) est un arbre qui satisfait les propriétés suivantes :

- (S1) Tous les chemins de la racine à une feuille ont la même longueur (le même nombre de nœuds), appelée hauteur de l'arbre et dénotée par h.
- (S2) La racine est une feuille ou bien a au moins 2 enfants.
- (S3) Chaque nœud qui n'est ni racine ni feuille possède au moins $t+1$ enfants.
- (S4) Chaque nœud a au plus $2t+1$ enfants.
- (C1) Les nœuds contiennent des clefs : la racine contient de 1 à $2t$ clefs ; et les autres nœuds, entre t et $2t$ clefs.
- (C2) Dans chaque nœud, les éléments sont stockés par ordre croissant.
- (C3) Chaque nœud qui contient ℓ clefs (et qui n'est pas une feuille) a $\ell+1$ enfants.
- (C4) Considérons un nœud P contenant ℓ clefs $x_0 \dots x_{\ell-1}$. Soient $P_0, \dots P_\ell$ ses enfants, et $K(P_i)_{0 \leqslant i \leqslant \ell}$ l'ensemble des clefs du sous-arbre dont la racine est P_i. Alors :
  - $\forall y \in K(P_0), y \leqslant x_0$;
  - $\forall 1 \leqslant i \leqslant \ell-1, \forall y \in K(P_i), x_{i-1} \leqslant y \leqslant x_i$;
  - $\forall y \in K(P_\ell), x_{\ell-1} \leqslant y$;

Les propriétés sont de deux ordres : quatre propriétés **structurelles**, préfixées par « S », et des propriétés de contenu/de clefs, préfixées par « C ». Remarquons que la propriété C4 est une généralisation de la propriété fondamentale des arbres binaires de recherche.

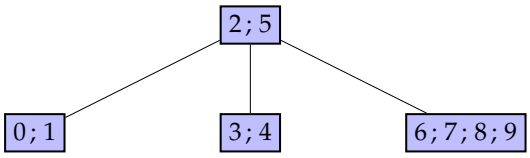
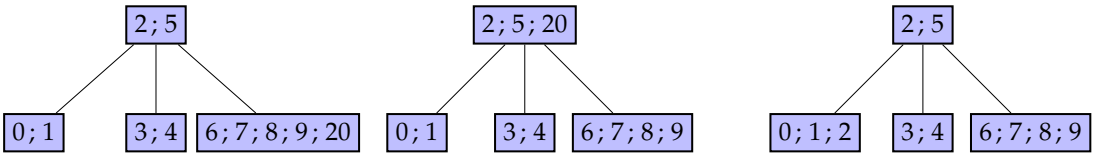


FIGURE 2 – Un exemple de B-tree d'ordre 2. Les nœuds contiennent un ensemble de clefs, triées par ordre croissant.

Question #8

Les arbres de la Figure 3 sont-ils des B-trees d'ordre 2 ?



(a) Arbre 1 (b) Arbre 2 (c) Arbre 3

FIGURE 3 – Quelques arbres.

L'implémentation du type B-tree en Python est fournie ci-dessous :

```
class BTreeNode:
  def __init__(self, t:int, f=False):
    self.feuille = f # True si le noeud est une feuille
    self.clefs = []    # Les valeurs des clefs du noeud
    self.enfants = []  # les enfants du noeud (qui sont des BTreeNode)
    self.ordre = t        # l'ordre du noeud
    self.n = 0        # nombre de clefs du noeud
```

```
class BTree:
  def __init__(self, t):
    self.root = BTreeNode(t,True)
    self.ordre = t  # l'ordre du btree
```

Ainsi l'appel `BTree(2)` renvoie une instance de la classe BTree d'ordre 2 représentant une feuille sans clef ni enfant. Remarquons en particulier que :

— Rien ne garantit les invariants de la définition d'un B-tree. Il faut donc bien faire attention à les maintenir.
— La numérotation des enfants et celle des clefs commencent à 0.

Par commodité, toutes les fonctions écrites seront des méthodes de la classe BTree.

Question #9

Écrire une fonction récursive d'impression d'un B-tree, `def print_node(self, niveau=0)`, qui affiche chacun des nœuds du `BTreeNode` courant, à partir du nœud x . Chaque nœud sera affiché sur une nouvelle ligne, avec le nombre de clefs qu'il contient, ainsi que les valeurs de ces clefs. Ainsi, sur l'exemple de la Figure 2, l'appel `B.root.print_node()` doit donner l'impression suivante :

```
Niveau  0   2:2 5
Niveau  1   2:0 1
Niveau  1   2:3 4
Niveau  1   4:6 7 8 9
```

2.2 Recherche dans un B-tree

La recherche d'une clef dans un B-tree va suivre le même motif que pour un arbre binaire de recherche : en effet, elle va combiner deux algorithmes : la recherche dans une liste triée (`keys`) et la recherche parmi les enfants (`child`) d'un nœud donné.

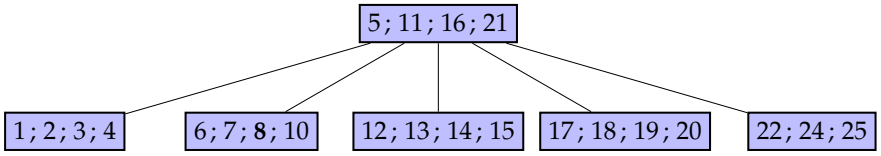


FIGURE 4 – Exemple de B-tree (d'ordre 2) pour la recherche

Question #10

Dans le B-tree de la figure 4, expliquer les étapes de recherche de la clef dont la valeur est 8. Utiliser les caractéristiques du B-tree pour ne pas réaliser un parcours entier de l'arbre.

Question #11

En déduire, pour un nœud fixé, une condition d'arrêt de la recherche dans ce nœud ainsi que le numéro du sous-arbre dans lequel rechercher récursivement.

Question #12

Implémenter en Python `recherche_clef(self : BTreeNode, k : int)-> bool` cherchant la clef k dans l'arbre courant à partir du nœud courant.

Étudions maintenant la complexité algorithmique de la recherche. Considérons des B-trees non vides, d'ordre $t > 1$, de hauteur $h \geqslant 1$. Dans un premier temps, nous allons compter le nombre de nœuds minimum N_{min} et maximum N_{max} d'un tel B-tree. Il est recommandé de procéder en comptant les nœuds par niveau.

Question #13

Montrer que $N_{min} = 1 + \frac{2}{t}\left((t+1)^{h-1} - 1\right)$ pour $h \geqslant 2$.

Question #14

Montrer que $N_{max} = \frac{1}{2t}\left((2t+1)^h - 1\right)$ pour $h \geqslant 1$.

Question #15

En déduire un encadrement du nombre de clefs contenues dans un B-tree d'ordre t et de hauteur h.

Question #16

Expliquer rapidement comment optimiser le coût de la recherche dans un nœud. Quelle est la complexité de cet algorithme ?

Question #17

Conclure sur la complexité algorithmique de la recherche dans un B-tree en fonction du nombre de nœuds n.

2.3 Insertion dans un B-tree

L'algorithme d'insertion d'une clef dans un B-tree réalise des étapes similaires à celui de l'insertion dans un arbre binaire de recherche : l'arbre est parcouru récursivement à partir de la racine, jusqu'à trouver une feuille dans laquelle insérer l'élément. L'insertion se réalise donc **dans une feuille**. Les B-trees ont néanmoins une contrainte supplémentaire : un nombre minimum et un nombre maximum de clefs pour les nœuds. Avant d'insérer une clef dans un nœud, il faudra s'assurer qu'un nœud « n'est pas complet » (au sens où le nombre maximum de clefs par nœud n'est pas atteint).

Question #18

Expliquer comment insérer la clef 23 dans le B-tree de la Figure 4.

Question #19

Implémenter une fonction `insertion_position(self : BTreeNode, k : int) -> int` qui, appelée sur un BTreeNode, calcule l'indice auquel il faudrait insérer la clé k dans la liste `self.clefs` (en respectant le critère C4 et en supposant que la liste `self.clefs` ne soit pas déjà trop pleine).

Question #20

En utilisant la fonction précédente, écrire la fonction `insertion_non_complet(self : BTreeNode, k:int)` qui réalise l'insertion d'une clef k dans un `BTreeNode`, dans le cas où celui-ci n'est pas complet.

Question #21

Donner la condition en Python qui permet de déterminer qu'un nœud x est complet.

Dans le cas où l'endroit d'insertion est complet, le nœud va être « scindé en deux » (*split*), afin de récupérer deux nœuds de tailles plus petites. La Figure 5 explique le processus. On considère un B-tree d'ordre $t = 2$ et un sous-arbre enraciné en x, dont l'un des enfants y, doit être scindé, car il contient trop de clefs (ici 5 clefs). Le nœud y est donc scindé en deux nœuds y et z de taille égale à 2 et les sous-arbres associés sont « distribués » en conséquence. L'élément « en trop », ici 8, est alors **éjecté**, pour être inséré par la suite dans l'ensemble des clefs du nœud parent, x (qui lui même peut être complet, comme nous le verrons par la suite).

Question #22

Écrire la méthode `def split(self : BTreeNode)-> (int, BTreeNode)` d'un `BTreeNode` dont le nombre de clefs est supposé (temporairement) égal à $2t+1$. La méthode actualise le nœud y et renvoie la valeur « éjectée » (8 dans la Figure 5), ainsi que le nouveau nœud créé (z dans la Figure 5). *Cette procédure, non récursive, ne réalise pas l'insertion dans le nœud parent*. Il est fortement recommandé d'utiliser les *slices* (« tranches ») Python.

La procédure d'insertion récursive peut réaliser de multiples scindages, si la valeur éjectée d'un nœud doit être insérée dans un nœud déjà complet. Le résultat final de la procédure d'insertion est donc possiblement un B-tree très différent du B-tree initial. Ainsi, l'insertion de la clé 9 dans le B-tree de la Figure 4 a pour résultat le B-tree de la Figure 6.

Question #23

Que se passe-t-il lors de l'insertion si la racine est complète ?

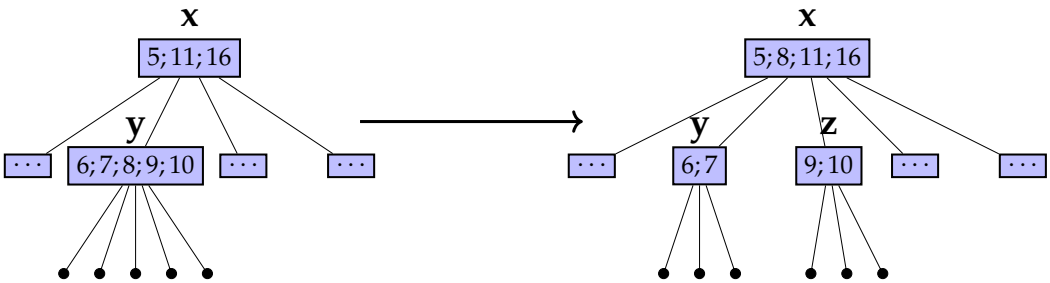


FIGURE 5 – Illustration (partielle) de l'insertion de 8 dans le nœud **y** avec « split » : lors de l'insertion, le nœud **y** possède temporairement 5 clefs, ce qui n'est pas permis pour un B-tree d'ordre 2. La procédure de découpage ("split") va donc créer le nouveau nœud **z** et éjecter la valeur 8. Cette valeur sera retournée au parent à la fin de l'appel récursif d'insertion.

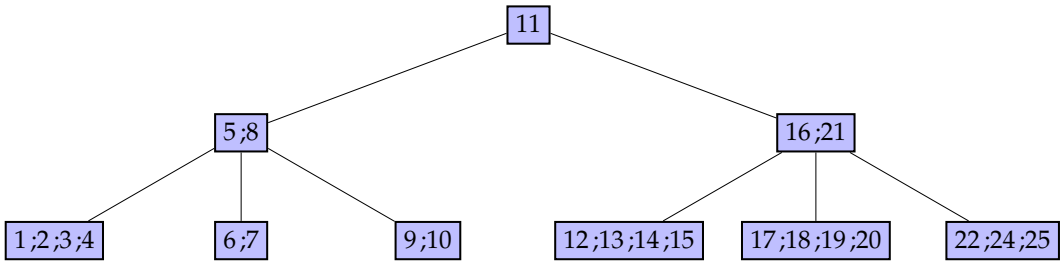


FIGURE 6 – B-tree résultant de l'insertion de la clef 9 dans le B-tree de la Figure 4

La méthode d'insertion au niveau `BTree` est fournie ci-dessous.

```
def insert(self, k): # Version bottom-up de l'insertion dans un Btree
    val, newnode = self.root.insert(k)
    if newnode is not None:
        self.root = self.root.make_new_root(val, newnode)
```

Question #24

Implémenter la méthode `make_new_root` qui traite le cas de la racine complète.

Question #25

Terminer l'implémentation de l'insertion en implémentant la méthode `insert` de la classe `BTreeNode`.

3 Application : Indexation dans les Bases de Données

Dans cette partie, nous allons étudier une application des B-trees, le stockage *persistant* des tables dans un système de gestion de bases de données (SGBD). En effet, les données des tables sont stockées dans des fichiers, pour pouvoir traiter leur potentiel grand volume et les sauvegarder dans le temps. Ces fichiers sont supposés stockés sur un disque dur local.

Une grande partie de la littérature des bases de données est consacrée à l'étude et à la conception de structures de données performantes pour le stockage et le requêtage efficace de ces données. Le but principal est de limiter la quantité de données lues sur le support persistant.

Dans cette partie, nous considérons à titre d'exemple, une base de données de taille moyenne, contenant des informations sur l'oeuvre de Shakespeare (https://www.opensourceshakespeare.org).

3.1 Schéma relationnel et requêtes

La Figure 7 donne un aperçu de deux des relations de la base de données précitée, avec des exemples d'*enregistrements*.

Characters (Personnages)

CharID	CharName	Abbrev	WorkID	SpeechCount
'1citizen'	'First Citizen'	'FC'	'romeojuliet'	3
'1musician-oth'	'First Musician'	'FM'	'othello'	5
'1musician-rj'	'First Musician'	'FM'	'romeojuliet'	10
'2gentleman-oth'	'Second Gentleman'	'SG'	'othello'	5
'2musician-rj'	'Second Musician'	'SM'	'romeojuliet'	3

Works (Œuvres)

WorkID	Title ,	LongTitle	Date
'othello'	'Othello'	'The Tragedy of Othello, Moor of Venice'	1604
'hamlet'	'Hamlet'	'The Tragedy of Hamlet, Prince of Denmark'	1600
'romeojuliet'	'Romeo and Juliet'	'The Tragedy of Romeo and Juliet'	1594

FIGURE 7 – Extrait de la Base de Données Open Shakespeare (simplifiée en deux tables).

Question #26

Dans la relation `Characters`, que signifie le fait que `CharID` est une clef primaire ? `WorkID` est-il une clef étrangère pour la relation `Characters` ?

On souhaite effectuer plusieurs recherches dans la base de données. Les recherches sont décrites comme suit, et sont désignées par une étiquette R1, R2, R3, R4.

R1 : Rechercher toutes les informations décrites dans la table 'Characters', concernant le personnage désigné par l'identifiant '2musician-rj'.

R2 : Rechercher le nombre de mots prononcés (attribut `SpeechCount`) par le personnage désigné par l'identifiant '2musician-rj'.

R3 : Lister tous les titres longs des oeuvres de Shakespeare écrites entre 1589 et 1600.

R4 : Lister l'ensemble des titres longs des oeuvres où apparaissent des musiciennes ou des musiciens (dont l'attribut `CharName` contient *Musician*).

Dans la suite, les recherches seront désignées par leur étiquette.

Question #27

Exprimer chacune des recherches (R1, R2, R3 et R4) sous forme d'une requête SQL distincte ; chaque requête est indépendante des autres.

3.2 Stockage des enregistrements

Nous considérons par la suite que le type 'char' est stocké sur 8 bits (1 octet), et le type 'mediumint' est stocké sur 3 octets. Chaque enregistrement est stocké comme la suite des informations des attributs de cet enregistrement.

Le stockage des données sur disque dur se fait par *blocs* de 4096 octets. Chaque bloc comprend une entête de 200 octets. Les enregistrements de la relation `Characters` sont stockés dans un fichier, dont la taille est un multiple de 4096 octets (*blocs*).

La relation `Characters` comporte 1264 enregistrements et la table correspondante est créée en SQL par la commande suivante :

```
CREATE TABLE IF NOT EXISTS `Characters` (
  `CharID` varchar(50) NOT NULL DEFAULT '',
  `CharName` varchar(75) NOT NULL DEFAULT '',
  `Abbrev` varchar(10) NOT NULL DEFAULT '',
  `WorkID` varchar(50) NOT NULL DEFAULT '',
  `SpeechCount` mediumint unsigned NOT NULL DEFAULT 0,
  PRIMARY KEY (`CharID`)
) DEFAULT CHARSET=latin1;
```

Question #28

Dans un enregistrement, les champs décrivant des chaînes de caractères sont munis d'un premier octet donnant le nombre effectif de caractères de la chaine, sans sentinelle de fin de chaîne. Donner la taille, en octets, de l'enregistrement `('1musician-rj', 'First Musician', 'FM', 'romeojuliet', 10)`.

Question #29

Donner la taille minimale $SIZE_{min}$ et maximale $SIZE_{max}$ d'un enregistrement de la relation `Characters`.

Question #30

Calculer le nombre maximal d'enregistrements de taille $SIZE_{max}$ par bloc. En déduire la taille maximale (en octets) du fichier stockant la relation `Characters`.

3.3 Recherche dans une table

Nous nous intéressons maintenant à la complexité d'une opération de sélection simple sur la table `Characters`, qui serait par exemple réalisée lors de l'exécution de la commande SQL associée à la recherche `R1`.

Nous supposons que la base de données est implantée sur un disque dur magnétique et que le temps moyen d'accès à un bloc sur disque est 15 ms ; une fois le bloc chargé en mémoire, la recherche à l'intérieur du bloc est réalisée séquentiellement, et le temps d'accès à l'enregistrement au sein d'un bloc est jugé négligeable.

Question #31

En l'absence de structure supplémentaire, l'accès aux blocs du fichier contenant la table `Characters` est séquentiel. Donner une estimation du temps maximal nécessaire à la recherche d'un enregistrement particulier dans la table `Characters`.

En général, l'accès à la base de données est réalisée au travers d'un SGBD (Système de Gestion de Base de Données). Le SGBD réalise les opérations de création / modification / sauvegarde / requêtage sur des éléments de la base de données. En particulier, il maintient un *système d'indexation* qui permet d'améliorer les temps d'accès aux enregistrements des tables de la base.

L'indexation consiste à maintenir, dans une structure dédiée, l'information minimale permettant de localiser chaque enregistrement de façon efficace. Chaque enregistrement est décrit par un couple : (identifiant unique de l'enregistrement, numéro de bloc dans lequel l'enregistrement est créé). Cette structure est maintenue cohérente par le SGBD lors de la création ou de la destruction d'un enregistrement dans une table.

Cette structure d'indexation peut être réalisée au travers de *tables d'index*. Une table d'index associée à une relation stocke les couples (identifiant unique, numéro de bloc) de la table décrivant la relation ; les couples sont rangés par identifiant croissant. On suppose qu'un numéro de bloc est stocké sur 8 octets.

Question #32

On considère la relation `Characters`. Indiquer le champ à utiliser pour représenter l'identifiant unique de l'enregistrement dans la table d'index, et la relation d'ordre adaptée. Déterminer une évaluation de la taille maximale de la table d'index associée à la table `Characters`, (en nombre d'octets et en nombre de blocs).

Question #33

Donner les avantages et inconvénients d'utiliser un index sous forme de table lors des opérations d'insertion, de suppression ou de recherche d'un enregistrement d'une table représentant une relation de la base de données.

Une autre implémentation de la structure d'indexation est basée sur les B-trees vus précédemment, avec la variante suivante :

— Seules les feuilles contiennent des couples (clef/ numéro de bloc) : les clefs seront les identifiants

— Les nœuds qui ne sont pas des feuilles ne contiennent que des clefs (et un lien vers leurs enfants). Ces clefs sont des recopies de certaines clefs apparaissant dans les feuilles ; elles servent de balises lors des parcours du B-tree, afin d'orienter la recherche vers le sous-arbre adéquat.

Question #34

Montrer qu'un B-tree d'ordre 3 et de hauteur 4 suffit à permettre la construction d'un index pour la clé `charID` pour la table `Characters`. En considérant qu'une adresse (de bloc) est stockée sur 8 octets, et en ignorant la place prise par le chaînage parent-enfant, donner une évaluation de la place en mémoire prise par ce B-tree. Expliquer l'intérêt de la structure d'index sous forme de B-tree par rapport à la structure en table.

Question #35

Quelles sont les étapes à effectuer pour répondre à la requête associée à la recherche `R2` ?

On souhaite réaliser un parcours efficace des index et tables lors de la recherche `R3`. Pour ce faire, on cherche à construire un index relatif à la table `Works`, permettant un parcours séquentiel des enregistrements de cette table, selon un index combinant différents attributs de la relation `Works`.

Question #36

Quel couple d'attributs de la relation `Works` faut-il intégrer dans l'index et quelle est la relation d'ordre associée à ces couples pour traiter efficacement la requête `R3` ?

Question #37

Quelle modification simple apporteriez-vous au B-tree correspondant à l'indexation des enregistrements de la relation `'Works'` afin de parcourir séquentiellement les feuilles du B-tree, en respectant l'ordre associé ?

On suppose que des structures d'index en B-tree ont été construites, pour les deux tables `'Characters'` et `'Works'`, en intégrant la modification de la question précédente.

Question #38

Quelles sont les étapes à effectuer pour réaliser la requête `R3` ?

Question #39

Quel index pourrait être proposé pour réaliser efficacement les parcours d'index et de table associés à la requête R4 ?

Question #40

Quels sont les critères à privilégier pour associer un attribut de table à un index ? Pourquoi faut-il éviter de maintenir trop d'index différents pour une table donnée ?

On suppose maintenant que l'on travaille sur une base de données bien plus grande. Elle comprend une table `T` de 4,2 millions d'enregistrements, stockés sur 300 000 blocs.

Question #41

Déterminer dans ce cas, le temps maximal pour la recherche d'un enregistrement particulier dans `T` lorsqu'il n'y a pas de mécanisme d'indexation.

On cherche à construire une structure d'index en B-tree pour optimiser les accès à `T`. Différents arbres sont possibles :

1. Un B-tree d'ordre 3 et de taille 8 et
2. Un B-tree d'ordre 6 et de hauteur 6 permettent de stocker l'index de T.

Question #42

On suppose que chacun des deux arbres tient en mémoire RAM. Parmi ces deux arbres, quelle est la structure la plus adaptée et pourquoi ?

Partie A

Dans le thème "La photographie numérique" du programme de SNT, l'une des capacités attendues est de "traiter par programme une image pour la transformer en agissant sur les trois composantes de ses pixels.".

On donne ci-dessous un extrait du programme de SNT (thème photographie numérique) :

Contenus	Capacités attendues
Photosites, pixels, résolution (du capteur, de l'image), profondeur de couleur	Distinguer les photosites du capteur et les pixels de l'image en comparant les résolutions du capteur et de l'image selon les réglages de l'appareil.
Métadonnées EXIF	Retrouver les métadonnées d'une photographie.
Traitement d'image	Traiter par programme une image pour la transformer en agissant sur les trois composantes de ses pixels.
Rôle des algorithmes dans les appareils photo numériques	Expliciter des algorithmes associés à la prise de vue. Identifier les étapes de la construction de l'image finale.

1. En vous aidant de la documentation de la bibliothèque Python Pillow donnée dans l'annexe 1, écrivez un programme Python permettant de transformer une image couleur au format JPEG (nom : "`photo.jpg`" ; définition de l'image en pixels : 800 x 600) en image en niveaux de gris de même format puis de l'afficher.
2. Proposez une activité destinée à vos élèves de seconde leur permettant d'écrire (ou de compléter) un programme Python ayant pour but de modifier "une image pour la transformer en agissant sur les trois composantes de ses pixels."
3. Proposez une correction, à destination des élèves, de l'activité que vous aurez rédigée à la question 2.

Partie B

L'étude des "types et valeurs de base" est au programme de la spécialité NSI en classe de première :

Contenus	Capacités attendues	Commentaires
Écriture d'un entier positif dans une base b ≥ 2	Passer de la représentation d'une base dans une autre.	Les bases 2, 10 et 16 sont privilégiées.
Représentation binaire	Évaluer le nombre de bits	Il s'agit de décrire les tailles

d'un entier relatif	nécessaires à l'écriture en base 2 d'un entier, de la somme ou du produit de deux nombres entiers. Utiliser le complément à 2.	courantes des entiers (8, 16, 32 ou 64 bits). Il est possible d'évoquer la représentation des entiers de taille arbitraire de Python.
Représentation approximative des nombres réels : notion de nombre flottant	Calculer sur quelques exemples la représentation de nombres réels : 0.1, 0.25 ou 1/3.	0.2 + 0.1 n'est pas égal à 0.3. Il faut éviter de tester l'égalité de deux flottants. Aucune connaissance précise de la norme IEEE-754 n'est exigible.
Valeurs booléennes : 0, 1. Opérateurs booléens : and, or, not. Expressions booléennes	Dresser la table d'une expression booléenne.	Le ou exclusif (xor) est évoqué. Quelques applications directes comme l'addition binaire sont présentées. L'attention des élèves est attirée sur le caractère séquentiel de certains opérateurs booléens.
Représentation d'un texte en machine. Exemples des encodages ASCII, ISO-8859-1, Unicode	Identifier l'intérêt des différents systèmes d'encodage. Convertir un fichier texte dans différents formats d'encodage.	Aucune connaissance précise des normes d'encodage n'est exigible.

4. Vous trouverez dans l'annexe 2 un exercice donné à des élèves de première NSI dans le cadre d'une évaluation. Proposez une correction de cet exercice à destination de vos élèves.
5. Expliquez le principe de la représentation en machine des nombres à virgule flottante, par exemple en vous appuyant sur la norme IEEE-754.
6. Expliquez la phrase suivante du programme : "0.2 + 0.1 n'est pas égal à 0.3".

Partie C

L'annexe 3 est un extrait de la note de service qui précise les modalités de l'épreuve pratique de l'enseignement de spécialité NSI au baccalauréat.

7. Voici l'énoncé de l'exercice 1 d'un sujet tiré de la banque 2022 :

   Écrire une fonction maxi qui prend en paramètre une liste tab de nombres entiers et qui renvoie un couple donnant le plus grand élément de cette liste ainsi que l'indice de la première apparition de ce maximum dans la liste.

   Exemple : >>> maxi([1,5,6,9,1,2,3,7,9,8])
   (9,3)

   Proposez un corrigé de cet exercice.

8. Voici l'énoncé de l'exercice 2 d'un sujet tiré de la banque 2022 :

   La fonction recherche prend en paramètres deux chaines de caractères gene et seq_adn et renvoie True si on retrouve gene dans seq_adn et False sinon. Compléter le code Python ci-dessous pour qu'il implémente la fonction recherche.

```
def recherche(gene, seq_adn):
    n = len(seq_adn)
    g = len(gene)
    i = ...
    trouve = False
    while i < ... and trouve == ... :
        j = 0
        while j < g and gene[j] == seq_adn[i+j]:
            ...
        if j == g:
            trouve = True
        ...
    return trouve
```

   Proposez un corrigé de cet exercice.

9. Dans le cadre de la préparation de vos élèves de première à cette épreuve pratique, vous élaborez le contenu d'une séance de Travaux Pratiques qui s'appuiera sur la résolution d'un sujet. Rédigez un sujet type « épreuve pratique », pour vos élèves de première NSI en respectant les préconisations de la note de service dont l'extrait est en annexe 3.

   Le premier exercice devra porter sur la partie suivante du programme de première de l'enseignement de spécialité NSI Rédigez un exercice de type "Exercice I". Cet exercice devra porter sur la recherche dichotomique.

   On donne ci-dessous un extrait du programme de première NSI :

Contenus	Capacités attendues	Commentaires
Recherche dichotomique dans un tableau trié	Montrer la terminaison de la recherche dichotomique à l'aide d'un variant de boucle.	Des assertions peuvent être utilisées. La preuve de la correction peut être présentée par le professeur.

10. Proposez une correction pour l'exercice proposé à la question 9

11. Rédigez un exercice de type "Exercice II". Cet exercice devra porter sur la partie "dictionnaire" du programme de première.

    On donne ci-dessous un extrait du programme de première NSI :

Contenus	Capacités attendues	Commentaires
	Construire une entrée de	Il est possible de présenter les

Dictionnaires par clés et valeurs	dictionnaire. Itérer sur les éléments d'un dictionnaire.	données EXIF d'une image sous la forme d'un enregistrement. En Python, les p-uplets nommés sont implémentés par des dictionnaires. Utiliser les méthodes keys(), values () et items ().

12. Proposez une correction pour l'exercice rédigé à la question 11.

Partie D

Le tri par insertion et le tri par sélection sont au programme de la spécialité NSI en classe de première.

Contenus	Capacités attendues	Commentaires
Tris par insertion, par sélection de l'enseignement de spécialité NSI en classe de terminale :	Écrire un algorithme de tri. Décrire un invariant de boucle qui prouve la correction des tris par insertion, par sélection.	La terminaison de ces algorithmes est à justifier. On montre que leur coût est quadratique dans le pire cas.

13. Expliquez, comme vous le feriez devant une classe de première, le principe du tri par insertion.
14. Expliquez, comme vous le feriez devant une classe de première, le principe du tri par sélection.
15. Montrez que la complexité en temps dans le pire des cas de l'algorithme du tri par insertion est quadratique.
16. Démontrez la correction de l'algorithme du tri par insertion à l'aide d'un invariant de boucle.
17. Écrivez une fonction Python *tri_selection* qui prend en paramètre une liste d'entiers *L*. Cette fonction devra trier la liste passée en paramètre en utilisant l'algorithme du tri par sélection.

L'étude de la méthode "diviser pour régner" est au programme de l'enseignement de spécialité NSI en classe de terminale :

Contenus	Capacités attendues	Commentaires
Méthode « diviser pour régner ».	Écrire un algorithme utilisant la méthode « diviser pour régner ».	La rotation d'une image bitmap d'un quart de tour avec un coût en mémoire constant est un bon exemple. L'exemple du tri fusion permet également d'exploiter la récursivité et d'exhiber un algorithme de coût en $n.\log_2(n)$ dans les pires des cas.

Il est précisé dans la rubrique "commentaires", que "l'exemple du tri fusion permet également d'exploiter la récursivité et d'exhiber un algorithme de coût en $n.\log_2(n)$ dans les pires des cas.".

18. Expliquez, comme vous le feriez devant une classe de terminale, le principe du tri-fusion.
19. Expliquez, comme vous le feriez devant une classe de terminale, le coût en $n.\log_2(n)$ dans le pire des cas du tri-fusion.
20. À la suite d'une activité sur le tri-fusion, une élève vient vous dire qu'elle a regardé une vidéo consacrée au tri-rapide (quicksort) et qu'elle a l'impression que cet algorithme ressemble au tri-fusion étudié en classe, comment lui expliquez-vous les points communs et les différences entre ces deux algorithmes ?

Partie E

L'étude de la récursivité est au programme de terminale de la spécialité NSI.

Contenus	Capacités attendues	Commentaires
Récursivité	Écrire un programme récursif. Analyser le fonctionnement d'un programme récursif.	Des exemples relevant de domaines variés sont à privilégier

Après avoir proposé à vos élèves un premier exemple de fonction récursive avec la fonction permettant de calculer la factorielle d'un entier, vous décidez de faire travailler vos élèves sur la suite de Fibonacci. On rappelle que cette suite est définie par :

$F_0 = 0$
$F_1 = 1$
Pour tout $n \geq 2$, $F_n = F_{n-1} + F_{n-2}$

21. Écrivez une fonction Python récursive *fib* qui prend en paramètre un entier *n* et qui renvoie le terme F_n de la suite de Fibonacci.
22. Un élève a réussi à écrire la fonction *fib* et s'interroge sur le fait que l'exécution de *fib(120)* "fasse planter l'ordinateur", que pouvez-vous lui répondre ?

Plus tard dans l'année, vous décidez de reprendre cet exemple de la suite de Fibonacci pour illustrer le début de votre cours sur la programmation dynamique.

On donne ci-dessous un extrait du programme de l'enseignement de spécialité NSI en Terminale :

Contenus	Capacités attendues	Commentaires
Programmation dynamique.	Utiliser la programmation dynamique pour écrire un algorithme.	Les exemples de l'alignement de séquences ou du rendu de monnaie peuvent être présentés. La discussion sur le coût en mémoire peut être développée.

23. Expliquez, en quelques lignes, le principe de la programmation dynamique.

24. Utilisez la programmation dynamique afin d'améliorer l'efficacité de la fonction écrite à la question 21. Vous écrirez deux programmes Python : un premier programme qui utilisera l'approche ascendante (bottom-up) et un deuxième programme qui utilisera l'approche descendante (top-down).

Après ce premier exemple d'utilisation de la programmation dynamique, vous décidez de faire travailler vos élèves sur le problème du rendu de monnaie. Comme ce problème a déjà été traité en première dans le cadre du cours sur les algorithmes gloutons, vous décidez de donner un exercice à vos élèves de terminale afin de voir ce qu'ils ont retenu de la résolution du problème du rendu de monnaie par une méthode gloutonne.

On donne ci-dessous un extrait du programme de l'enseignement de spécialité NSI en première :

Contenus	Capacités attendues	Commentaires
Algorithmes gloutons	Résoudre un problème grâce à un algorithme glouton.	Exemples : problèmes du sac à dos ou du rendu de monnaie. Les algorithmes gloutons constituent une méthode algorithmique parmi d'autres qui seront vues en terminale.

25. Expliquez, en quelques lignes, le principe d'un algorithme glouton.

26. Voici une fonction Python *rendu* proposée par un élève de terminale permettant de résoudre le problème du rendu de monnaie en utilisant une méthode gloutonne. Cette fonction prend en paramètre la somme à rendre (en euro) et renvoie le nombre minimum de pièces qu'il est possible de rendre. Quel est, selon vous, le problème qui risque de se poser lors de l'exécution de ce programme ? Proposez une modification afin de supprimer ce problème.

```
def rendu(s):
    index = 0
    tab_piece = [2, 1, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01]
    somme_rendu = 0
    nbre_piece = 0
    while somme_rendu != s :
        somme_a_rendre = s - somme_rendu
        if somme_a_rendre < tab_piece[index]:
            index = index + 1
        else :
            somme_rendu = somme_rendu + tab_piece[index]
            nbre_piece = nbre_piece + 1
    return nbre_piece
```

27. Pour les questions suivantes, on supprime la pièce de 1 centime du tableau *tab_piece* (valeur 0.01). Donnez une valeur de *s* (paramètre de la fonction *rendu*) pour laquelle la méthode gloutonne ne permet pas de trouver une solution alors que cette solution existe.

28. Suite aux constatations faites à la question 27, vous décidez d'abandonner la méthode gloutonne pour résoudre le problème du rendu de monnaie et d'écrire à la place une fonction récursive *rendu_rec* qui prend en paramètre la somme à rendre *s*

(en centimes) et qui renvoie le nombre minimum de pièces qu'il est possible de rendre (cette fonction doit trouver la solution, si cette solution existe). Écrivez cette fonction en Python.

29. En utilisant la programmation dynamique, améliorer l'efficacité du programme écrit à la question 28.

Partie F

Vous trouverez en annexe 4 l'énoncé d'un exercice sur les arbres binaires donné à des élèves de terminale NSI lors d'une évaluation.

On donne ci-dessous des extraits du programme de l'enseignement de spécialité NSI de terminale :

Contenus	Capacités attendues	Commentaires
Arbres : structures hiérarchiques. Arbres binaires : nœuds, racines, feuilles, sous-arbres gauches, sous-arbres droits.	Arbres : structures hiérarchiques. Arbres binaires : nœuds, racines, feuilles, sous-arbres gauches, sous-arbres droits.	On fait le lien avec la rubrique « algorithmique ».

Contenus	Capacités attendues	Commentaires
Algorithmes sur les arbres binaires et sur les arbres binaires de recherche.	Calculer la taille et la hauteur d'un arbre. Parcourir un arbre de différentes façons (ordres infixe, préfixe ou suffixe ; ordre en largeur d'abord). Rechercher une clé dans un arbre de recherche, insérer une clé.	Une structure de données récursive adaptée est utilisée. L'exemple des arbres permet d'illustrer la programmation par classe. La recherche dans un arbre de recherche équilibré est de coût logarithmique.

30. Proposez un corrigé de cet exercice à destination de vos élèves.
31. Proposez un barème sur 10 points pour cet exercice en expliquant vos choix.
32. On donne dans l'annexe 5 un extrait d'une copie d'élève. Corrigez cet extrait de copie, la copie corrigée sera rendue à l'élève.
33. Globalement, cet exercice est difficile. Proposez, en les justifiant, au moins deux modifications permettant de le rendre plus accessible.

ANNEXES

Annexe 1

Documentation succincte de la bibliothèque Pillow.

La bibliothèque Pillow propose la classe Image. La méthode open de la classe Image permet d'obtenir une instance de la classe Image. Cette méthode open prend en paramètre l'url d'un "fichier image".

Exemple :

```
from PIL import Image
img = Image.open("chat.jpg")
```

L'attribut size de la classe Image permet d'obtenir la dimension ou taille de l'image (en pixel) sous la forme d'un tuple.

Exemple :

```
from PIL import Image
img = Image.open("chat.jpg")
res = img.size
# le contenu de la variable res est le tuple (800,600) pour une image de
# taille 800 pixels par 600 pixels
```

La méthode `getpixel` de la classe Image renvoie un tuple correspondant aux composantes "rouge, vert, bleu" du pixel de coordonnées (x,y). Cette méthode prend en paramètre un tuple contenant les coordonnées (x,y) du pixel considéré.

On rappelle que le pixel de coordonnées (0,0) correspond au pixel en haut à gauche de l'image.

Exemple :

```
from PIL import Image
img = Image.open("chat.jpg")
r, v, b = img.getpixel((100,150))
# si le pixel de coordonnées (100,150) est rouge
# on obtiendra r = 255, v = 0, b = 0
```

La méthode `putpixel` de la classe Image permet de modifier la couleur du pixel de coordonnées (x,y). Cette méthode prend en paramètres 2 tuples : un premier tuple correspondant aux coordonnées du pixel que l'on souhaite modifier, un deuxième tuple correspondant à la nouvelle couleur du pixel de coordonnées (x,y).

Exemple :

```
from PIL import Image
img = Image.open("chat.jpg")
img.putpixel((100,150), (0,255,0))
# le pixel de coordonnées (100,150) est désormais vert
```

La méthode `show` de la classe image permet d'afficher l'image qui vient d'être modifiée (par exemple à l'aide de la méthode putpixel).

Exemple :

```
from PIL import Image
img = Image.open("chat.jpg")
img.putpixel((100,150), (0,255,0))
img.show()
```

Annexe 2

Exercice proposé à des élèves de première de spécialité NSI lors d'une évaluation :

Exercice 2

1. Convertissez 27_{10} en base 2.
2. Convertissez 1000101_2 en base 10.
3. Convertissez 44_{10} en base 16.
4. Convertissez $A1_{16}$ en base 10.
5. Convertissez 100110_2 en base 16.
6. Convertissez $F3_{16}$ en base 2.
7. En utilisant le complément à 2 sur 8 bits, convertissez -9_{10} en base 2.
8. En utilisant le complément à 2 sur 8 bits, convertissez 10010110_2 en base 10.
9. Convertissez $1{,}0100_2$ en base 10.
10. Convertissez $3{,}125_{10}$ en base 2.

Annexe 3

NOR : MENE2001797N
Extrait de la note de service n° 2020-030 du 11-2-2020
MENJS - DGESCO A2-1

[…]

2. Partie pratique

Durée : 1 heure

Modalités

La partie pratique consiste en la résolution de deux exercices sur ordinateur, chacun étant noté sur 4 points.

Le candidat est évalué sur la base d'un dialogue avec un professeur-examinateur. Un examinateur évalue au maximum quatre élèves. L'examinateur ne peut pas évaluer un élève qu'il a eu en classe durant l'année en cours.

L'évaluation de cette partie se déroule au cours du deuxième trimestre pendant la période de l'épreuve écrite de spécialité.

- Premier exercice

Le premier exercice consiste à programmer un algorithme figurant explicitement au programme, ne présentant pas de difficulté particulière, dont on fournit une spécification. Il s'agit donc de restituer un algorithme rencontré et travaillé à plusieurs reprises en cours de formation. Le sujet peut proposer un jeu de test avec les réponses attendues pour permettre au candidat de vérifier son travail.

- Deuxième exercice

Pour le second exercice, un programme est fourni au candidat. Cet exercice ne demande pas l'écriture complète d'un programme, mais permet de valider des compétences de programmation suivant des modalités variées : le candidat doit, par exemple, compléter un programme « à trous » afin de répondre à une spécification donnée, ou encore compléter un programme pour le documenter, ou encore compléter un programme en ajoutant des assertions, etc.

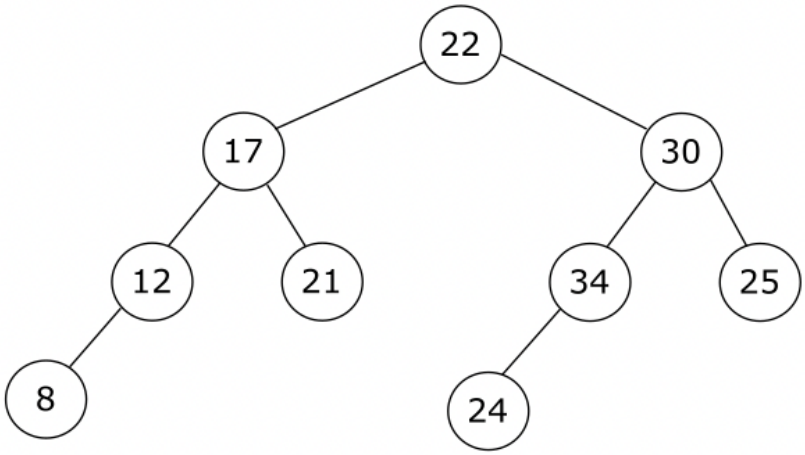
[…]

Annexe 4

Extrait du sujet d'une évaluation de terminale

Exercice III

Cet exercice porte sur les arbres binaires et les arbres binaires de recherche.
Dans cet exercice, on considérera que la racine d'un arbre binaire a une profondeur de 1.



Arbre 1

PARTIE A

1. Quelle est la taille de l'arbre 1 ?
2. Quelle est la hauteur de l'arbre 1 ?
3. L'arbre 1 est-il un arbre binaire de recherche ? Justifiez votre réponse.
4. Parcourir l'arbre 1 dans l'ordre suffixe

PARTIE B

Dans la suite de cet exercice, on utilisera la classe ArbreBinaire donnée ci-dessous.

```
class ArbreBinaire:

    def __init__(self, valeur):
        self.valeur = valeur
        self.enfant_gauche = None
        self.enfant_droit = None

    def insert_gauche(self, valeur):
        if self.enfant_gauche == None:
            self.enfant_gauche = ArbreBinaire(valeur)
        else:
            new_node = ArbreBinaire(valeur)
            new_node.enfant_gauche = self.enfant_gauche
            self.enfant_gauche = new_node
```

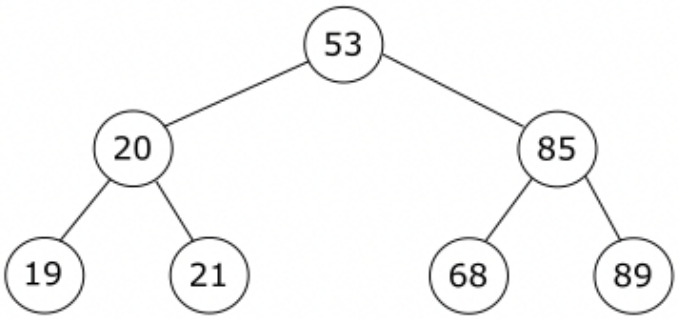
```
def insert_droit(self, valeur):
    if self.enfant_droit == None:
        self.enfant_droit = ArbreBinaire(valeur)
    else:
        new_node = ArbreBinaire(valeur)
        new_node.enfant_droit = self.enfant_droit
        self.enfant_droit = new_node

def get_valeur(self):
    return self.valeur

def get_gauche(self):
    return self.enfant_gauche

def get_droit(self):
    return self.enfant_droit
```

5. En utilisant la classe `ArbreBinaire`, implémenter en Python l'arbre binaire donné ci-dessous (arbre 2)



arbre 2

6. Écrire la fonction *taille* qui prend en paramètre une instance T de la classe `ArbreBinaire` et qui renvoie la taille de l'arbre T.
7. Écrire la fonction *hauteur* qui prend en paramètre une instance T de la classe `ArbreBinaire` et qui renvoie la hauteur de l'arbre T.
8. Écrire la fonction *parcours_infixe* qui prend en paramètre une instance T de la classe `ArbreBinaire` et qui permet d'afficher dans la console le parcours de l'arbre binaire T dans l'ordre infixe
9. Écrire la fonction *arbre_recherche* qui prend en paramètres une instance T de la classe `ArbreBinaire` (cette instance représentant un arbre binaire de recherche dont les valeurs des nœuds sont des entiers) et un entier k. Cette fonction renvoie True si k appartient à T et False dans le cas contraire.

Annexe 5 **À RENDRE AVEC LA COPIE**

Extrait d'une copie d'élève

Exercice III

PARTIE A

1. 9

2. 3

3. Oui, cet arbre binaire est bien un arbre binaire de recherche, car tous les nœuds situés à gauche de la racine sont plus petits que 22 et tous les nœuds situés à droite de la racine sont plus grands que 22.

4. 22 - 17 - 12 - 8 - 21 - 30 - 34 - 24 - 25

PARTIE B

5.
```
arbre = ArbreBinaire(53)
arbre.insert_gauche(20)
arbre.insert_droit(85)
n_20 = arbre.get_gauche()
n_85 = arbre.get_droit()
n_20.insert_gauche(19)
n_20.insert_droit(21)
n_85.insert_gauche(68)
n_85.insert_droit(89)
```

6.
```
def taille(T):
    if T==None:
        return 0
    else :
        return 1 + taille(T-1)
```

7.

```
def hauteur(T):
    if T == None:
        return 0
    else :
        return 1 + hauteur(T.get_gauche()) + hauteur(T.get_droit())
```

8.

```
def parcours_infixe(T):
    if T == None:
        return 0
    else :
        print(T)
        parcours_infixe(T.get_gauche())
        parcours_infixe(T.get_droit())
```

9.

```
def arbre_recherche(T,k):
    while T != None:
        if T.get_valeur() == k:
            return True
        elif T.get_valeur() < k:
            T = T.get_gauche()
        else :
            T = T.get_droit()
    return False
```

MINISTÈRE
DE L'ÉDUCATION
NATIONALE
ET DE LA JEUNESSE
Liberté
Égalité
Fraternité

EAE INF 1

SESSION 2024

AGREGATION
CONCOURS EXTERNE

Section : INFORMATIQUE

COMPOSITION EN INFORMATIQUE

Durée : 5 heures

L'usage de tout ouvrage de référence, de tout dictionnaire et de tout matériel électronique (y compris la calculatrice) est rigoureusement interdit.

Il appartient au candidat de vérifier qu'il a reçu un sujet complet et correspondant à l'épreuve à laquelle il se présente.

Si vous repérez ce qui vous semble être une erreur d'énoncé, vous devez le signaler très lisiblement sur votre copie, en proposer la correction et poursuivre l'épreuve en conséquence. De même, si cela vous conduit à formuler une ou plusieurs hypothèses, vous devez la (ou les) mentionner explicitement.

NB : Conformément au principe d'anonymat, votre copie ne doit comporter aucun signe distinctif, tel que nom, signature, origine, etc. Si le travail qui vous est demandé consiste notamment en la rédaction d'un projet ou d'une note, vous devrez impérativement vous abstenir de la signer ou de l'identifier.
Le fait de rendre une copie blanche est éliminatoire.

Tournez la page S.V.P.

A

INFORMATION AUX CANDIDATS

Vous trouverez ci-après les codes nécessaires vous permettant de compléter les rubriques figurant en en-tête de votre copie.

Ces codes doivent être reportés sur chacune des copies que vous remettrez.

Concours	Section/option	Epreuve	Matière
EAE	6200A	101	9422

IMPRIMERIE NATIONALE – 24 0113 – D'après documents fournis

EAE INF 1

Dépendances. Ce sujet contient trois parties indépendantes qui doivent être traitées toutes les trois. On veillera à bien indiquer sur la copie les changements de partie.

Attendus. Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction.

Partie I. Robot de dépôt de solutions chimiques

Cette partie doit être traitée dans le langage C avec des appels de type POSIX. Un rappel de certaines fonctions est disponible en annexe. Aucun `#include` n'est demandé. Ce sujet s'intéresse principalement à la synchronisation de processus légers (appelés dans la suite threads) à l'aide de mutex et de sémaphores. Les trois sections sont indépendantes.

On s'intéresse à la partie contrôle d'un ou plusieurs robots d'injections de produits chimiques dans des éprouvettes. On considère avoir un bras injecteur contrôlable par un ou plusieurs threads sur un ordinateur connecté à ce bras. Dans la suite, on supposera avoir une bibliothèque de fonctions dont on donnera la spécification mais dont on supposera ne pas pouvoir modifier ni consulter le code.

On s'intéresse à résoudre les problèmes découlant du cas où plusieurs threads d'un programme interagissent avec le robot au travers de la bibliothèque de contrôle en même temps. On considère exister la structure `struct formule`.

Dans l'ensemble du sujet, on suppose avoir un seul processeur avec un seul cœur.

1 Une rangée d'éprouvettes

On suppose dans cette partie être dans le cas où le programme composé de plusieurs threads contrôle un seul bras pouvant se déplacer au dessus d'une rangée unique de `N>0` éprouvettes.

On suppose avoir trois fonctions appelée, `int libre()`, `void aller(int i)` et `void injecter(struct formule *f)` qui respectivement permettent d'obtenir l'indice d'une éprouvette vide, au bras d'aller à l'éprouvette `i` entre `0` inclus et `N` exclus, et enfin d'injecter une solution dont la formule est passée en argument dans l'éprouvette actuellement sous le bras que l'on considère vide. Dans cette section on considère avoir assez d'éprouvettes par rapport au nombre d'expériences prévues.

Le comportement d'une fonction utilisant le bras sera alors par exemple (sous l'hypothèse qu'il y a toujours au moins une éprouvette vide disponible) :

```
1 int déposer(struct formule *f) {
2    int i = libre();
3    aller(i);
4    injecter(f);
5    return i;
6 }
```

Question 1.1. On se place dans le cas où deux threads tentent « en même temps » d'utiliser la fonction `déposer` décrite ci-dessus. Explicitez un exemple montrant qu'il est possible que les deux threads déposent leur solution dans la même éprouvette.

Question 1.2. On tente de résoudre le problème avec des mutex

```
1  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
2
3  int déposer(struct formule *f) {
4    pthread_mutex_lock(&mutex);
5    int i = libre();
6    pthread_mutex_unlock(&mutex);
7
8    aller(i);
9    injecter(f);
10   return i;
11 }
```

Quel est le problème de cette solution ?

Question 1.3. On tente de corriger le problème de la solution précédente comme suit :

```
1  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
2
3  int déposer(struct formule *f) {
4    pthread_mutex_lock(&mutex);
5    int i = libre();
6    aller(i);
7    injecter(f);
8    return i;
9    pthread_mutex_unlock(&mutex);
10 }
```

Cette version fonctionne-t-elle ? Expliquez pourquoi et corrigez-la si elle ne fonctionne pas.

Question 1.4. On suppose, pour cette question **uniquement**, que la formule est à encoder pour le modèle spécifique de bras : la fonction `injecter` n'est pas capable de directement traiter la formule de type `struct formule`. La fonction `encoder` permet de traduire la formule pour le bras utilisé. Le code de base deviendrait:

```
1  int déposer(struct formule *f) {
2     int i = libre();
3     aller(i);
4     struct codage code = encoder(f, "bras␣modèle␣1");
5     injecter(&code);
6
7     return i;
8  }
```

On considère aussi que la fonction (sans effets de bords) `encoder` est très lente, beaucoup plus lente que le temps de déplacement du bras. Modifiez le code précédent pour qu'il soit possible par plusieurs threads de l'appeler en parallèle sans problème de synchronisation et que ces appels soient le plus efficace possible.

1.1 Ajout du vidage des éprouvettes

On considère maintenant avoir la fonction `struct resultat *analyser()` qui permet de vider et d'analyser l'éprouvette qui est sous le bras. Le code d'un thread complet serait alors (en considérant que le mutex `mutex` a été créé en tant que variable globale) :

```
1 ...
2 struct formule f = creer_formule(..);
3 int pos = déposer(&f);
4 ... // attente avec calcul du temps de déplacement du bras
5 pthread_mutex_lock(&mutex);
6 aller(pos)
7 struct resultat *res = analyser()
8 pthread_mutex_unlock(&mutex);
9 ...
```

Question 1.5. Dans cette question, on considère toujours avoir au moins une éprouvette vide lors de l'appel à la fonction `libre`. Est-il possible de garantir que l'attente est d'une heure (à la seconde près) entre l'appel à `injecter` (dans la fonction `déposer`) et l'appel à la fonction `analyser` dans le cas de deux threads ? On considère que le temps de déplacement du bras est déterministe (calculable en fonction de sa position actuelle et de sa destination) et est de l'ordre de la minute. On justifiera la réponse.

Question 1.6. Même question si on est dans le cas d'un seul thread toujours à la précision de la seconde ? Avec une précision d'un cycle de processeur ? On justifiera les deux réponses.

Question 1.7. On suppose avoir maintenant plus d'expériences à faire que d'éprouvettes. On propose la modification suivante de la fonction `déposer`. On considère toujours avoir un mutex `mutex` global initialisé. On considère aussi qu'après que la fonction `analyser` termine de vider une éprouvette, la fonction `libre` est capable de renvoyer sa position. Lorsqu'il n'y a pas d'éprouvette vide, la fonction `libre` renvoie la valeur `-1`. On transforme la fonction `déposer` :

```
1  int déposer(struct formule *f) {
2    int i = -1;
3    while (i == -1) {
4      pthread_mutex_lock(&mutex);
5      i = libre();
6
7      if (i != -1) {
8        aller(i);
9        injecter(f);
10     }
```

```
11      pthread_mutex_unlock(&mutex);
12    }
13    return i;
14  }
```

Est-ce que cela fonctionne ? Justifier votre réponse

Question 1.8. En utilisant la fonction `déposer` de la question précédente, explicitez quel impact négatif a cette approche sur les autres threads et/ou sur les autres processus ?

Question 1.9. On suppose maintenant avoir accès aux sémaphores, et il n'y a plus accès qu'aux fonctions `aller`/`injecter`/`analyser` (i.e. plus à la fonction `libre`). Proposer une implémentation en C des fonctions `déposer` et `récupérer` en précisant les arguments et en justifiant ceux éventuellement ajoutés. Le but est d'autoriser l'utilisation suivante:

```
1 ???
2 struct formule f = creer_formule(...)
3 int pos = déposer(&f, ???)
4 /// attendre en fonction de la formule
5 struct resultat *res = récupérer(pos, ???)
6 ???
```

Précisez les variables partagées utilisées et l'initialisation du sémaphore dans la fonction `main` si besoin. On s'attachera à ce que l'implantation proposée résolve le problème soulevé dans la question précédente. On considérera avoir accès à une fonction `int trouver(int *tab, int n)` qui trouve et renvoie la valeur de l'indice dans le tableau `tab` de `n` entiers qui a pour valeur `0`. Si aucune valeur n'est `0` alors la fonction renvoie `-1`.

Question 1.10. L'implémentation proposée de la synchronisation est dans le cadre des threads, expliquer les types de modifications qu'il aurait fallu faire pour réaliser cette synchronisation dans un cadre multi-processus.

1.2 Passage à la 2D

Pour augmenter le nombre d'expériences possibles en même temps, on ajoute plusieurs rangées d'éprouvettes (toutes de la même taille), tout en gardant un seul bras. On ajoute un moteur permettant de changer de rangée d'éprouvettes. On a donc maintenant deux fonctions indépendantes `allerX` et `allerY`. On a respectivement maintenant `NY` rangées de `NX` éprouvettes. La fonction `libreXY` modifie ses arguments pour renvoyer les coordonnées d'un emplacement vide au moment de son appel. Dans cette partie, on suppose à nouveau avoir assez d'emplacement pour l'ensemble des expériences.

```
1 void déposer(struct formule *f) {
2   int x,y;
3   libreXY(&x, &y);
4   allerX(x);
5   allerY(y);
6   injecter(f);
7 }
```

Question 1.11. Combien de mutex au minimum sont nécessaires pour permettre l'utilisation de la fonction `déposer` par plusieurs threads du même programme en garantissant uniquement le bon fonctionnement du système (i.e. on ne s'intéressera pas à la problématique de performance dans cette question). Modifier le code précédent pour ajouter cette ou ces mutex.

Question 1.12. Les deux moteurs sont indépendants, plutôt que d'attendre que le déplacement sur X soit fini avant de faire le déplacement sur Y, il est possible de faire les deux en même temps. Proposez une implémentation de la fonction déposer réalisant cette amélioration.

1.3 Analyse non destructive dans le cas d'une seule rangée

On change légèrement l'expérience tout en gardant le même matériel. Les expérimentateurs mettent des produits dans chacune des éprouvettes et modifient légèrement le matériel pour que l'analyse ne vide plus les éprouvettes. Les expérimentateurs veulent étudier l'évolution du produit dans le temps et vont donc appeler la fonction `analyser` régulièrement. On revient dans le cas où l'on a une seule rangée et on considère que les éprouvettes sont déjà remplies au lancement du programme.

Une version séquentielle d'un thread de suivi d'une des expériences serait donc :

```
1 ...
2 while(true) {
3   aller(i)
4   struct resultat *res = analyser()
5   traiter(res, thread_data);
6   free(res);
7 }
```

La fonction `traiter` utilise uniquement des données internes à un thread (la variable `thread_data`) et donc il n'y a aucun problème à en exécuter plusieurs en même temps.

On se rend compte que le moteur est assez lent, mais est 10 fois plus rapide pour aller en valeurs croissantes que décroissantes. Par exemple, en étant en position 1, aller(2) est dix fois plus rapide que aller(0).

Question 1.13. Expliciter le déplacement de la tête dans le cas où les évènements suivants se produisent :

- L'analyse de 5 est en cours et a commencé à T0 (qui finira à T0+6)
- Un autre thread lance à T0+1 l'analyse de 3 (qui prendra 1)
- Un autre thread lance à T0+2 l'analyse de 7 (qui prendra 4)
- Un autre thread lance à T0+8 l'analyse de 9 (qui prendra 1)

Toutes les durées sont en Unité de Temps (UT). On donnera l'ordre des actions ainsi que les UT correspondantes par rapport à T0 (donc T0+3 sera donné comme 3). On considère que déplacer le bras d'une position en valeur croissante prend une UT. On considère aussi que l'on a déjà réglé le problème de synchronisation entre les threads.

Question 1.14. Écrire une fonction `analyser_en(i)` qui garantit que si plusieurs analyses sont en attente, celle qui sera débloquée sera soit celle dont la position est la plus proche supérieurement si elle existe, soit celle dont la position est la plus petite sinon.

Annexe

```
pthread_create // Créer un nouveau thread
int pthread_create(pthread_t * thread,
                   pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void * arg);

pthread_join // Attendre la fin d'un autre thread
int pthread_join(pthread_t th, void **thread_return);


pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock
                  , pthread_mutex_destroy // Opérations sur les mutex
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

sem_init // Initialiser un sémaphore non nommé
int sem_init(sem_t *sem, int pshared, unsigned int value);

sem_destroy // Détruire un sémaphore non nommé
int sem_destroy(sem_t *sem);

sem_post // Déverrouiller un sémaphore
int sem_post(sem_t *sem);

sem_wait // Verrouiller un sémaphore
int sem_wait(sem_t *sem);
```

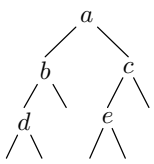
Partie II. Tableaux flexibles

Cette partie doit être traitée dans le langage OCaml.

Dans ce problème, on représente un tableau par un arbre binaire. Si le tableau est vide, il est représenté par l'arbre vide. Sinon, l'élément d'indice 0 du tableau est stocké à la racine de l'arbre, le sous-arbre gauche contient une représentation des éléments d'indices de la forme $2i+1$ et le sous-arbre droit contient une représentation des éléments d'indices de la forme $2i+2$. Ainsi, le tableau de taille 5

0	1	2	3	4
a	b	c	d	e

est représenté par l'arbre binaire



avec l'élément a d'indice 0 à la racine, le sous-tableau $[b, d]$ des éléments d'indices impairs récursivement organisé pour former le sous-arbre gauche et le sous-tableau $[c, e]$ des éléments d'indices pairs récursivement organisé pour former le sous-arbre droit.

Question 2.1. Dessiner l'arbre binaire correspondant à un tableau de taille 12.

Mise en œuvre en OCaml. On suppose que les éléments sont d'un type `elt` donné qui n'est pas précisé. Un tableau flexible est représenté par une valeur du type OCaml suivant :

```
type tree = E | N of tree * elt * tree
```

La valeur `E` représente l'arbre binaire vide. Une valeur $\mathtt{N}(\ell, x, r)$ représente un arbre binaire non vide, avec la valeur x à la racine, un sous-arbre gauche ℓ et un sous-arbre droit r. La taille d'un arbre t, notée $|t|$, est son nombre de nœuds. On a donc :

$$\begin{aligned} |\mathtt{E}| &= 0 \\ |\mathtt{N}(\ell, x, r)| &= 1 + |\ell| + |r| \end{aligned}$$

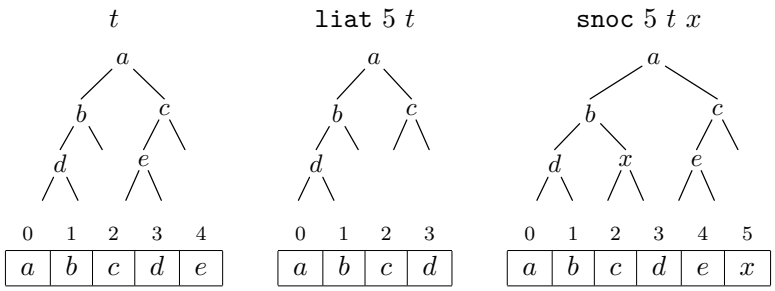
Question 2.2. Écrire une fonction `get: int -> tree -> elt` qui reçoit en arguments un entier i et un tableau flexible t, avec $0 \leq i < |t|$, et qui renvoie l'élément d'indice i du tableau représenté par t. La complexité doit être proportionnelle à la hauteur de t. On ne demande pas de la justifier.

Invariant structurel. On remarque que dans un tableau flexible, le sous-arbre gauche a toujours au moins autant de nœuds que le sous-arbre droit, car il y a au moins autant d'indices de la forme $2i+1$ que d'indices de la forme $2i+2$. Plus précisément, *pour tout sous-arbre* $\mathtt{N}(\ell, x, r)$ d'un tableau flexible, on a

$$|r| \leq |\ell| \leq |r| + 1 \tag{1}$$

Question 2.3. Montrer que la hauteur d'un tableau flexible de taille n est en $\mathcal{O}(\log n)$.

Agrandir/rétrécir du côté droit. Notre objectif est de réaliser quatre opérations pour respectivement agrandir ou rétrécir le tableau d'une case sur l'une ou l'autre de ses extrémités. Commençons par le côté droit, c'est-à-dire celui des indices les plus grands. L'opération `liat` n t supprime le dernier élément d'un (arbre représentant le) tableau t de taille n. L'opération `snoc` n t x ajoute un élément x à la fin d'un (arbre représentant le) tableau t de taille n. Si on reprend l'exemple du tableau donné en introduction, ces deux opérations donnent respectivement les arbres suivants :



En dessous de chaque arbre, on a dessiné le tableau qu'il représente. Voici par exemple le code de la fonction `liat` :

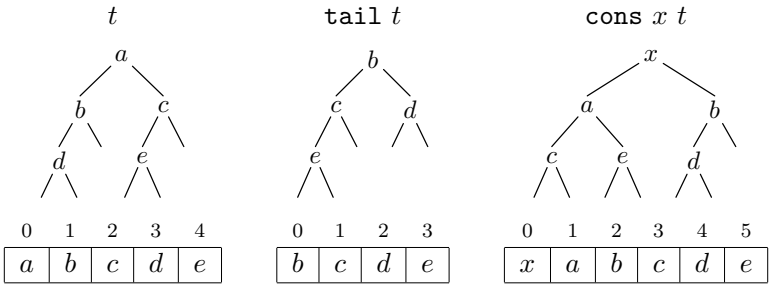
```
let rec liat n = function
  | N(E, _, E) -> E
  | N(l, x, r) -> if n mod 2 = 0 then N(liat (n / 2) l, x, r)
                                 else N(l, x, liat (n / 2) r)
  | E -> assert false
```

Question 2.4. Pour un tableau flexible t de taille n, donner la complexité de `liat` n t en fonction de n.

Question 2.5. Montrer la correction totale de la fonction `liat`, c'est-à-dire que, sous l'hypothèse que t est un tableau flexible de taille $n \geq 1$, alors `liat` n t termine et renvoie bien un tableau flexible de taille $n-1$ avec les éléments attendus.

Question 2.6. Donner un code pour la fonction `snoc: int -> tree -> elt -> tree`. Pour un tableau flexible t de taille n, et un élément x, l'appel `snoc` n t x renvoie un tableau flexible de taille $n+1$ avec les mêmes éléments que t aux indices $0 \leq i < n$ et l'élément x à l'indice n. La complexité doit être $\mathcal{O}(n)$. On ne demande pas de la justifier.

Agrandir/rétrécir du côté gauche. Agrandir ou rétrécir le tableau du côté gauche, c'est-à-dire du côté des indices les plus petits, est plus délicat à réaliser, car on décale alors les indices des éléments. L'opération `tail` t supprime le premier élément du tableau t. L'opération `cons` x t ajoute un élément x au début du tableau t. Si on reprend toujours le même exemple, alors ces deux opérations donnent respectivement les arbres suivants :



On prendra le temps de bien comprendre ce qui se passe ici. En particulier, on observera comment les éléments passent d'un côté à l'autre de l'arbre. En effet, les éléments situés à des indices pairs (resp. impairs) avant se retrouvent à des indices impairs (resp. pairs) après.

Question 2.7. Donner un code pour la fonction `tail: tree -> tree`. Pour un tableau flexible t de taille $n \geq 1$, l'appel `tail` t renvoie un tableau flexible de taille $n-1$ où le premier élément de t a été supprimé. La complexité doit être $\mathcal{O}(\log n)$ et on demande de la justifier.

Question 2.8. Donner un code pour la fonction `cons: elt -> tree -> tree`. Pour un tableau flexible t de taille n, l'appel `cons` x t renvoie un tableau flexible de taille $n+1$ où un premier élément x a été ajouté à gauche des éléments de t. La complexité doit être $\mathcal{O}(\log n)$. On ne demande pas de la justifier.

Construction à partir d'un vrai tableau. Si on se donne un (vrai) tableau `a`, de type `elt array` et de taille n, on peut chercher à construire un tableau flexible contenant les mêmes éléments que `a`. Si on le fait naïvement en appelant n fois la fonction `snoc` (ou n fois la fonction `cons`) alors la complexité sera $\mathcal{O}(n \log n)$ au total. Mais on peut faire mieux.

Question 2.9. Écrire une fonction `of_array: elt array -> tree` qui reçoit en argument un tableau `a` de taille n et construit un tableau flexible ayant les mêmes éléments que `a`, en temps $\mathcal{O}(n)$. Indication : on pourra introduire une fonction auxiliaire qui, pour deux entiers $m \geq 1$ et $k \geq 0$ donnés, construit le tableau flexible des éléments de `a` d'indices $mi + k$.

Tableau flexible constant. Pour cette dernière question, on cherche à construire un tableau flexible d'une certaine taille n dont tous les éléments sont identiques à un élément donné. Comme pour la question précédente, on pourrait le construire à partir des fonctions `cons` ou `snoc`, avec une complexité totale $\mathcal{O}(n \log n)$. Mais là encore, on peut faire mieux, et même beaucoup mieux !

Question 2.10. Écrire une fonction `make: int -> elt -> tree` qui reçoit en arguments un entier $n \geq 0$ et un élément x, et renvoie un tableau flexible de taille n dont tous les éléments sont égaux à x. La complexité en temps doit être $\mathcal{O}(\log n)$. On demande de la justifier. Donner par ailleurs la complexité en espace.

Partie III. Gestion d'un concours de recrutement

Cette partie considère une version très simplifiée de la gestion d'un concours de recrutement comme celui de l'agrégation. La première section considère des bases de données pour la gestion de l'inscription. La deuxième section modifie ces bases de données avec les notes des candidats (sans considérer la correction des copies). La troisième section vise la gestion des épreuves orales.

1 Bases de données pour l'inscription

Tous les candidats et candidates doivent s'inscrire sur un site web. Ils doivent fournir un certain nombre d'informations dont leur identité et les concours où ils sont candidats. Ce formulaire web permet alors de remplir la base de données dont nous donnons une version très simplifiée en figure 1. Les noms des tables sont sur fond grisé, les clés primaires sont en gras et les clés étrangères en italique.

La table Académie liste les académies de métropole et d'outre-mer avec d'une part une clé primaire **aid** qui est un code interne et d'autre part le nom usuel de l'académie dans le champs « nom ». La table Candidat reprend les informations personnelles du candidat. Le formulaire web force le genre à être 'H' ou 'F'.

La table Aménagement liste les aménagements possibles lors des épreuves avec d'une part une clé primaire **amid** et une description textuelle comme 'Tiers-temps' ou 'Salle isolée'. Selon le ou les handicaps reconnus (qui n'apparaissent pas dans ces tables), un candidat peut avoir droit à un ou plusieurs aménagements. La table ListeAmén liste les aménagements de chaque candidat. Chaque candidat peut avoir plusieurs aménagements et chaque aménagement peut s'appliquer à plusieurs candidats. C'est donc la paire **{ *cid*, *amid* }** qui est la clé primaire de ListeAmén.

Pour l'instant, on ne considère que deux concours : le capes NSI et l'agrégation externe d'informatique. Pour chaque concours, nous avons une table avec les numéros des candidats qui souhaitent y concourir ainsi que l'académie où il passe ce concours (on pourrait envisager de passer les deux concours dans deux académies distinctes, mais chaque concours correspond à une unique académie).

Pour l'agrégation, le candidat a un choix sur la troisième épreuve, représenté par le champs *ep* qui vaut 'A' pour « Étude de cas informatique » et 'B' pour « Fondements de l'informatique ».

Les champs *aid* de AgrégExtInfo et CapesNSI sont des clés étrangères provenant de Académie. Les champs *cid* de ces deux tables sont des clés étrangères provenant de Candidat ainsi que la clé primaire de ces tables. Le champs *cid* de ListeAmén est également une clé étrangère provenant de Candidat. Le champs *amid* de ListeAmén est une clé étrangère provenant de Aménagement.

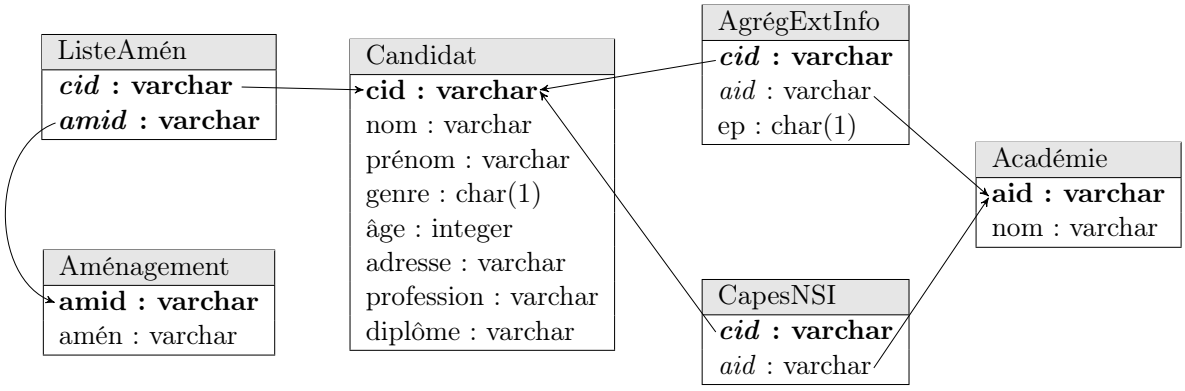


FIGURE 1 – Bases de données pour l'inscription

Question 3.1. Justifier brièvement l'intérêt d'avoir *cid* comme clé primaire de AgrégExtInfo.

Question 3.2. Écrire une requête SQL qui répond à : combien de candidats passent l'agrégation externe d'informatique dans chaque académie ? Le but est de savoir combien de sujets imprimer, on affichera en première colonne le nom de l'académie et en deuxième le nombre de candidats.

Question 3.3. Écrire à la fois une requête SQL et une formule en algèbre relationnelle qui répondent à : pour les candidats qui passent les deux concours dans deux académies différentes, donner leur nom de famille et le nom de ces deux académies.

Question 3.4. Écrire une requête SQL qui répond à : pour les candidats à l'agrégation et pas au CAPES, donner le nombre de candidats par profession.

Question 3.5. Écrire une requête SQL qui répond à : donner la proportion de femmes qui choisissent l'épreuve spécifique « Étude de cas informatique » sur le nombre de candidats ayant choisi cette épreuve spécifique.

Question 3.6. Écrire à la fois une requête SQL et une formule en algèbre relationnelle qui répondent à : donner les noms de famille des candidats ayant l'aménagement 'Salle isolée' mais pas l'aménagement 'Tiers-temps'.

Question 3.7. Écrire à la fois une requête SQL et une formule en algèbre relationnelle qui répondent à : donner les noms des académies où il n'y a pas de candidat ayant un aménagement 'Tiers-temps' pour l'agrégation.

Question 3.8. Écrire une requête SQL qui répond à : donner les noms des académies telles que cette académie a le nombre maximal d'aménagements 'Salle isolée' pour l'agrégation.

2 Bases de données pour les notes des épreuves écrites

On suppose que la correction se fait dans un autre outil (scan des copies, double correction de chaque copie, anonymat...) et que les notes sont ensuite basculées dans notre base de données initiale. On modifie alors AgrégExtInfo en

AgrégExtInfo
***cid* : varchar**
aid : varchar
ep : char(1)
note-ep1 : real
note-ep2 : real
note-ep3 : real

où note-epX est la note obtenue à l'épreuve X.

Question 3.9. Écrire une requête SQL qui répond à : faire afficher les noms des 42 candidats à l'agrégation ayant les meilleurs notes finales, triés par note finale décroissante. On rappelle que les trois épreuves ont le même coefficient, la note finale est donc la somme des notes des 3 épreuves écrites.

Question 3.10. Écrire une requête SQL qui répond à : calculer la moyenne de la note finale des candidats ayant au moins deux aménagements.

Question 3.11. Il est bien sûr simpliste de ne considérer que deux concours : il y a plus d'une centaine de concours, dont les CAPES, CAPET, agrégations externes, internes... Proposer une modélisation en tables qui permet d'avoir un nombre de concours non borné : vous indiquerez les tables et leur champs, ainsi que les clés primaires et étrangères.

Question 3.12. Écrire une requête SQL qui répond à : quel est le maximum du nombre de concours passé par une seule personne (dans votre modélisation) ?

3 Gestion des épreuves orales

Cette partie doit être traitée dans le langage Python.

Une fois les candidats admissibles, il faut organiser les épreuves orales et convoquer chaque admissible pour passer les trois épreuves orales. Une possibilité est d'avoir pour base un fichier `oral.csv` de ce type :

```
21/06, Leçon, 13h15, 13h30, 17h30, 18h30, A, 2845, Recoque, Alice
22/06, TP,    8h,    8h15,  13h15,  14h15,  C, 7743, Turing,  Alan
...
```

Ainsi, Alice Recoque a pour numéro de candidate 2845 et elle passe l'épreuve de leçon le 21 juin. Elle est convoqué à 13h15 pour une préparation de 13h30 à 17h30 et une interrogation de 17h30 à 18h30 avec le jury A (chaque jury est représenté par une lettre). On rappelle que chaque candidat a 3 épreuves, leçon, modélisation et TP avec 4h ou 5h de préparation.

Le but est de vérifier que ce fichier csv ne comporte aucune erreur ou typo, car il est utilisé pour convoquer les candidats, organiser l'accueil des candidats et le travail du jury.

Pour un traitement en Python de ce fichier, on peut commencer par l'importer :

```
import csv
t_oral = []
with open('oral.csv', newline='') as csvfile:
    spamreader = csv.reader(csvfile, delimiter=',')
    for row in spamreader:
        t_oral.append(row)
```

qui donne alors le tableau de tableaux `t_oral` (temporaire) suivant :

```
[
  ['21/06', 'Leçon', '13h15', '13h30', '17h30', '18h30', 'A',
        '2845', 'Recoque', 'Alice']
  ['22/06', 'TP', '8h', '8h15', '13h15', '14h15', 'C', '7743',
        'Turing', 'Alan']
... ]
```

FIGURE 2 – Extrait de la valeur de la variable `t_oral` après appel au code Python précédent.

Le but est de programmer plusieurs fonctions de vérification. Chacune devra renvoyer un booléen indiquant si le fichier est correct ou non. Si non, elle devra écrire un message d'erreur explicatif.

Question 3.13. Écrire une fonction Python `verif_nblc(N,t_oral)` qui, à partir du nombre de candidat `N` et de `t_oral`, vérifie que le nombre de lignes de `t_oral` est correct, et pour chaque ligne, que son nombre de colonnes est correct.

Pour la suite des vérifications, une première étape consiste à « nettoyer » `t_oral` de façon à obtenir le tableau de tableaux `oral` suivant :

```
       0      1       2     3     4      5     6     7        8          9
 [
    [ 21,  'Leçon',  795,  810,  1050,  1110,  'A',  2845,  'Recoque',  'Alice'  ]
    [ 22,   'TP',    480,  495,   795,   855,  'C',  7743,  'Turing',   'Alan'   ]
    ...
 ]
```

FIGURE 3 – Extrait de la valeur de la variable `oral` après « nettoyage » de `t_oral`. On a indiqué à la première ligne en italique la numérotation des colonnes, qui n'apparaît pas dans `oral`.

Sachant que les oraux se déroulent en juin, la première colonne ne contient que le numéro (entier) du jour. Les colonnes d'horaires sont le nombres de minutes depuis 0h00 : ainsi '13h15' correspond à $13 \times 60 + 15 = 795$ minutes. Le numéro de candidat est transformé en un entier. Les colonnes restantes sont inchangées.

Question 3.14. Écrire une fonction Python `verif_temps(oral)` qui vérifie la cohérence des horaires, c'est-à-dire pour chaque ligne :

- il y a 15 minutes entre la convocation et le début de préparation,
- il y a 4h de préparation en modélisation et leçon et 5h en TP,
- il y a 1h d'interrogation.

Question 3.15. Écrire une fonction Python `verif_nbj(oral, nb_jury)` qui vérifie que pour chaque horaire de chaque jour, il y a moins de `nb_jury` candidats convoqués.

Question 3.16. Écrire une fonction Python `verif_ep(oral)` qui vérifie que chaque candidat a 3 épreuves qui sont 'TP', 'Leçon' et 'Modélisation' sur 3 jours différents.

Pour aider l'humain à lire le tableau, le fichier csv en entrée a en fait une première ligne explicative :

Jour,	**Ép**,	**Hconvoc**,	**Hdéb**,	**Hpass**,	**Hfin**,	**Jury**,	**cid**,	**Nom**,	**Prénom**
21/06,	Leçon,	13h15,	13h30,	17h30,	18h30,	A,	2845,	Recoque,	Alice
22/06,	TP,	8h,	8h15,	13h15,	14h15,	C,	7743,	Turing,	Alan
...									

On peut alors, en utilisant plutôt `csv.DictReader`, obtenir le tableau de dictionnaires `t_dict_oral` :

```
[
  {'Jour': '21/06', 'Ép': 'Leçon', 'Hconvoc': '13h15',
    'Hdéb': '13h30', 'Hpass': '17h30', 'Hfin': '18h30',
    'Jury': 'A', 'cid': '2845', 'Nom': 'Recoque',
    'Prénom': 'Alice'},
  {'Jour': '22/06', 'Ép': 'TP', 'Hconvoc': '8h',
    'Hdéb': '8h15', 'Hpass': '13h15', 'Hfin': '14h30',
    'Jury': 'C', 'cid': '7743', 'Nom': 'Turing',
    'Prénom': 'Alan'},
  ...
]
```

FIGURE 4 – Extrait de la valeur de la variable `t_dict_oral`.

On peut alors le « nettoyer » de façon similaire au passage de `t_oral` en figure 2 à `oral` en figure 3. On obtient le tableau de dictionnaires `dict_oral` suivant.

```
[
  {'Jour': 21, 'Ép': 'Leçon', 'Hconvoc': 795, 'Hdéb':
    810, 'Hpass': 1050, 'Hfin': 1110,  'Jury': 'A',
    'cid': 2845, 'Nom': 'Recoque', 'Prénom': 'Alice'},

  {'Jour': 22, 'Ép': 'TP', 'Hconvoc': 480, 'Hdéb':
    495, 'Hpass': 795, 'Hfin': 855,  'Jury': 'C',
    'cid': 7743, 'Nom': 'Turing', 'Prénom': 'Alan'},
  ...
]
```

FIGURE 5 – Extrait de la valeur de la variable `dict_oral` après « nettoyage » de `t_dict_oral`.

Question 3.17. Écrire une fonction Python `dict_verif_nomno(dict_oral)`, qui vérifie que, dans la variable `dict_oral`, chaque numéro de candidat correspond bien à un seul nom de famille et prénom.

Question 3.18. Écrire une fonction Python `dict_verif_ep(dict_oral)` qui vérifie que chaque candidat a 3 épreuves qui sont 'TP', 'Leçon' et 'Modélisation' sur 3 jours différents.

Question 3.19. Donner trois avantages ou inconvénients de l'utilisation des dictionnaires par rapport à l'approche précédente.

Question 3.20. Donner trois avantages ou inconvénients de l'utilisation d'un fichier csv par rapport à une base de données pour la problématique de cette section.

Question 3.21. Écrire une fonction Python `clean_dict(t_dict_oral)` de « nettoyage » qui, à partir de la variable `t_dict_oral` de la figure 4, renvoie une valeur qui correspond à la figure 5. Elle doit également vérifier que la date et les horaires sont dans des intervalles possibles (entre 1 et 31 pour la date et entre 0h et 24h pour les horaires).

* *
*

MINISTÈRE
DE L'ÉDUCATION
NATIONALE
ET DE LA JEUNESSE
Liberté
Égalité
Fraternité

EAE INF 2

SESSION 2024

AGREGATION
CONCOURS EXTERNE

Section : INFORMATIQUE

ÉTUDE D'UN PROBLÈME INFORMATIQUE

Durée : 6 heures

L'usage de tout ouvrage de référence, de tout dictionnaire et de tout matériel électronique (y compris la calculatrice) est rigoureusement interdit.

Il appartient au candidat de vérifier qu'il a reçu un sujet complet et correspondant à l'épreuve à laquelle il se présente.

Si vous repérez ce qui vous semble être une erreur d'énoncé, vous devez le signaler très lisiblement sur votre copie, en proposer la correction et poursuivre l'épreuve en conséquence. De même, si cela vous conduit à formuler une ou plusieurs hypothèses, vous devez la (ou les) mentionner explicitement.

NB : Conformément au principe d'anonymat, votre copie ne doit comporter aucun signe distinctif, tel que nom, signature, origine, etc. Si le travail qui vous est demandé consiste notamment en la rédaction d'un projet ou d'une note, vous devrez impérativement vous abstenir de la signer ou de l'identifier.
Le fait de rendre une copie blanche est éliminatoire.

Tournez la page S.V.P.

A

INFORMATION AUX CANDIDATS

Vous trouverez ci-après les codes nécessaires vous permettant de compléter les rubriques figurant en en-tête de votre copie.

Ces codes doivent être reportés sur chacune des copies que vous remettrez.

Concours	Section/option	Epreuve	Matière
EAE	6200A	102	9423

EAE INF 2

RECTIFICATIF

La formule en bas de la page 5 devrait être écrite à l'horizontale et non à la verticale.

Au lieu de :

1/4
1/4
E0= [1/4]
1/4

Lire :

E0 = [1/4 1/4 1/4 1/4]

EAE INF 2

Les modèles de Markov cachés

Dans ce sujet, nous allons aborder l'implémentation des modèles de Markov cachés. Il s'agit de modèles probabilistes qui permettent d'engendrer des mots, et plus spécifiquement des séquences d'ADN (acide désoxyribonucléique). Ils permettent de représenter des familles de séquences d'ADN proches, comme un gène codant pour une même protéine dans différents organismes. Ils sont constitués d'états dans lesquels sont émis des caractères selon une distribution probabiliste propre à chaque état, et les transitions entre les états sont également régies par une loi de probabilité. Lorsque l'on observe une séquence produite par le modèle, on ne connaît pas les états qui ont été visités pour la produire, d'où le terme *caché* dans le nom du modèle. La première partie sert à se familiariser avec cette notion de modèle, la deuxième partie s'intéresse à leur représentation informatique dans le paradigme de la programmation objet. La troisième partie s'intéresse aux algorithmes permettant de faire fonctionner ces modèles, ainsi qu'à leur implémentation. Enfin, la quatrième partie permet de raffiner le modèle pour leur donner un pouvoir de représentation plus réaliste.

Préliminaires

Programmation. Les questions de programmation doivent être traitées en langage Python. L'usage ou l'importation de tout module sera interdit. Lorsque le candidat écrira une fonction, il pourra faire appel à des fonctions définies dans les questions précédentes, même si elles n'ont pas été traitées. Il pourra également définir des fonctions ou des classes auxiliaires, mais devra préciser leurs rôles ainsi que les types et significations de leurs arguments. Les candidats sont encouragés à expliquer les choix d'implémentation de leurs fonctions lorsque ceux-ci ne découlent pas directement des spécifications de l'énoncé. Si les paramètres d'une fonction à coder sont supposés vérifier certaines hypothèses, il ne sera pas utile dans l'écriture de cette fonction de tester si les hypothèses sont bien satisfaites.

Notations. On identifiera une même grandeur écrite dans deux polices de caractères différentes, en italique du point de vue mathématique (par exemple n) et en Computer Modern à chasse fixe du point de vue informatique (par exemple `n`).

Probabilités. Il est fait mention de probabilités dans la suite. Une probabilité est notée $Prob$, et notée $Prob_H$ lorsqu'il s'agit d'une probabilité dans le modèle H.

Complexité. Sans précision supplémentaire, lorsqu'une question demande la complexité d'une fonction, il s'agira de la complexité temporelle dans le pire des cas. On considérera que toutes les opérations arithmétiques sur les entiers s'effectuent en temps constant. La complexité sera exprimée sous la forme $O(f(n, m))$ où n et m sont les tailles des arguments de la fonction, et f une expression simple. Les calculs de complexité seront justifiés succinctement. Lorsqu'une question de programmation précise qu'une complexité est attendue, sauf demande explicite, il n'est alors pas nécessaire de justifier que la fonction écrite vérifie cette contrainte.

Dépendances. Ce sujet contient plusieurs parties. Chaque partie utilise des définitions et des résultats des parties précédentes. Sauf mention explicite du contraire, les questions restent néanmoins indépendantes, au sens où toute question peut être traitée en admettant les résultats énoncés dans les questions précédentes.

Attendus. Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction.

Partie I. Définitions et propriétés élémentaires

En bioinformatique, les modèles de Markov cachés sont utilisés pour représenter une famille de séquences d'ADN (acide désoxyribonucléique). Ce sont des modèles probabilistes qui se composent de plusieurs éléments. Un modèle de Markov caché H est un quintuplet (Σ, S, T, E, P), tel que :

— l'ensemble Σ est l'alphabet sur lequel les séquences sont construites. On pose m la taille de cet alphabet. Pour les séquences génomiques, $\Sigma = \{\text{A,C,G,T}\}$ et $m = 4$;

— l'ensemble S est un ensemble fini non vide dont les éléments sont appelés *états* du modèle. On note n la taille de S. Dans toute la suite on supposera que $S = \{1, 2, \ldots, n\}$;

— la matrice T est une matrice réelle de dimension $n \times n$, qui est la matrice des probabilités de transition $(\forall (i,j) \in \{1, \ldots n\}^2, T(i,j) \in [0,1])$;

— la matrice E est une matrice réelle de dimension $n \times m$, qui est la matrice des probabilités d'émission $(\forall (i,c) \in \{1, \ldots n\} \times \Sigma, E(i,c) \in [0,1])$;

— le vecteur P est un vecteur réel de taille n, qui est le vecteur des probabilités de départ $(\forall i \in \{1, \ldots n\}, P(i) \in [0,1])$.

Les contraintes classiques sur un modèle de Markov caché sont les suivantes :

— La somme des probabilités des états initiaux est égale à 1 :

$$\sum_{i=1}^{n} P(i) = 1 \tag{1}$$

— La somme des probabilités des transitions partant d'un état est égale à 1 :

$$\forall i \in \{1, \ldots, n\}, \sum_{j=1}^{n} T(i,j) = 1 \tag{2}$$

— La somme des probabilités d'émission dans un état est égale à 1 :

$$\forall i \in \{1, \ldots, n\}, \sum_{c \in \Sigma} E(i,c) = 1 \tag{3}$$

La représentation graphique d'un modèle de Markov caché est constituée d'un graphe dont les sommets, représentant les états, contiennent les probabilités d'émission, et les arcs représentent les probabilités de transition non nulles, qui les étiquettent. Un exemple de représentation sous

forme de graphe d'un modèle de Markov caché est donné en figure 1. Il correspond au modèle H_1 suivant : $\Sigma = \{\text{A, C, G, T}\}$, $n = 4$, et

$$P = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad T = \begin{bmatrix} 0.5 & 0 & 0.5 & 0 \\ 0 & 0.1 & 0.8 & 0.1 \\ 0 & 0.2 & 0.5 & 0.3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad E = \begin{array}{c} \begin{array}{cccc} \text{A} & \text{C} & \text{G} & \text{T} \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 \\ 0.5 & 0 & 0.5 & 0 \\ 0.2 & 0.1 & 0.4 & 0.3 \end{bmatrix} \end{array}.$$

Les probabilités de départ ne sont pas représentées dans le graphe de la figure 1.

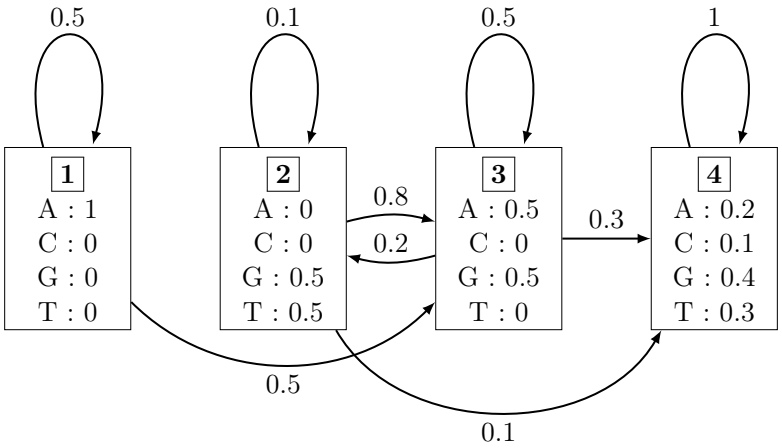


FIGURE 1 – Modèle de Markov caché H_1 représenté sous forme de graphe.

On considèrera dans toute la suite uniquement des modèles de Markov dont tous les états sont accessibles depuis un état de probabilité de départ non nulle.

1 Quelques définitions

1.1 Les marches

Une *marche* $m = m_1 m_2 \dots m_k$ dans le modèle de Markov caché H est une suite d'états de H, telle que :

— le premier état de la marche a une probabilité non nulle dans le vecteur des probabilités de départ : $P(m_1) \neq 0$;

— chaque succession d'état dans la marche a une probabilité non nulle dans la matrice des probabilités de transition : $\forall i \in \{1, \dots k-1\}$, $T(m_i, m_{i+1}) \neq 0$.

Par exemple 11332 est une marche possible dans le modèle H_1 défini ci-dessus, alors que la marche 2112 ne l'est pas.

La probabilité $Prob_H(m)$ d'une marche m dans le modèle H est définie comme le produit des probabilités :

— de départ dans le premier état de m ;

— de transition d'un état de m au suivant.

Ainsi :

$$Prob_H(m) = P(m_1) \times \prod_{i=1}^{k-1} T(m_i, m_{i+1)}.$$

Par exemple, la probabilité de la marche 11324 dans le modèle H_1 ci-dessus est :

$$Prob_{H_1}(11324) = P(1) \times T(1,1) \times T(1,3) \times T(3,2) \times T(2,4) = 1 \times 0.5 \times 0.5 \times 0.2 \times 0.1 = 5 \times 10^{-3}.$$

On pourra remarquer qu'une marche m est possible dans le modèle H si et seulement si $Prob_H(m) \neq 0$.

Question 1 : Pour chaque marche suivante, indiquer si elle est possible dans le modèle H_1 et donner sa probabilité dans le modèle H_1 :

a) 13444

b) 234

c) 132323232

d) 1234

e) 13432

1.2 Les séquences

Une séquence $w = w_1 \dots w_k$ de caractères de Σ peut être *produite* dans le modèle de Markov caché H s'il existe une marche $m = m_1 \dots m_k$ de même longueur que la séquence, possible dans H, et telle que chaque caractère a une probabilité non nulle d'émission dans l'état de même indice : $\forall i \in \{1, \dots, k\}, E(m_i, w_i) \neq 0$.

Par exemple, la séquence AAGAT peut être produite dans le modèle H_1, notamment grâce à la marche 11332, mais la séquence TACCG ne peut pas être produite par le modèle H_1, car aucune marche ne commence dans un état qui permet d'émettre T avec une probabilité non nulle.

Question 2 : Pour chacune des séquences suivantes, indiquer si elle peut être produite dans le modèle H_1, et le cas échéant, indiquer une marche qui permet de la produire :

a) AATATA

b) CATGTG

c) AACGAA

d) AAGGGG

Les séquences sont des événements associés aux modèles de Markov cachés. Si l'on connaît une marche m du modèle H qui permet de produire la séquence w, on peut calculer la probabilité de produire la séquence w sachant m et H, en considérant les probabilités d'émission de caractères de w dans les états de m :

$$Prob_{m,H}(w) = \prod_{i=1}^{k} E(m_i, w_i).$$

Par exemple, la probabilité de produire la séquence AAGAT dans le modèle H_1 sachant que l'on a suivi la marche 11332 est :

$$Prob_{11332,H_1}(\text{AAGAT}) = E(1,\text{A}) \times E(1,\text{A}) \times E(3,\text{G}) \times E(3,\text{A}) \times E(2,\text{T}) = 1 \times 1 \times 0.5 \times 0.5 \times 0.5 = 0.125.$$

On va par la suite plutôt s'intéresser à la probabilité d'événements du type « la marche m a produit la séquence w, sachant le modèle H », notée $Prob_H(m, w)$, qui est calculée de la façon suivante :

$$Prob_H(m, w) = Prob_H(m) \times Prob_{m,H}(w) = P(m_1) \times \prod_{i=1}^{k-1} T(m_i, m_{i+1}) \times \prod_{i=1}^{k} E(m_i, w_i).$$

Question 3 : Pour les séquences suivantes, donner **une** marche ayant permis de la produire, avec la probabilité que cette marche produise la séquence dans le modèle H_1 :

a) AGCGAT

b) AGAAA

Question 4 : Montrer que dans H_1, il existe une marche qui peut produire plusieurs séquences. À quelle(s) condition(s) sur le modèle aurait-on une unique séquence par marche possible (sans pour autant que toutes les marches soient supposées possibles) ?

Question 5 : Montrer que dans H_1, il existe une séquence qui peut être produite par plusieurs marches.

Question 6 : Donnez toutes les marches possibles produisant la séquence AAG dans le modèle H_1. Laquelle maximise la probabilité de produire cette séquence dans le modèle H_1 ?

Question 7 : Pour une séquence w, indiquer à quelle probabilité correspond la somme pour toutes les marches possibles m des probabilités que m produise la séquence w dans un modèle de Markov H.

2 Quelques propriétés

On s'intéresse au modèle de Markov caché très particulier décrit ci-dessous, que l'on appelle le modèle nul : $H_0 = (\Sigma_0, S_0, P_0, E_0, T_0)$ avec $\Sigma_0 = \{\text{A, C, G, T}\}$, $S_0 = \{1\}$, $P_0 = [1]$,

$E_0 = \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}$, et $T_0 = [1]$.

Question 8 : Donner la représentation sous forme de graphe de ce modèle et vérifier qu'il satisfait bien aux contraintes de base des modèles de Markov cachés, données par les équations (1), (2), (3).

Question 9 : Montrer que H_0 peut produire des séquences non vides de longueur arbitraire.

Question 10 : Montrer que H_0 peut produire toutes les séquences non vides sur l'alphabet Σ_0, avec une marche que l'on précisera.

Question 11 : On considère Σ_0^k l'ensemble des séquences de taille $k \geq 1$ fixée sur l'alphabet Σ_0.

a) Donner le nombre de séquences possibles dans Σ_0^k.

b) Donner la probabilité d'une séquence de Σ_0^k dans le modèle H_0.

c) Calculer la somme des probabilités des séquences de Σ_0^k dans le modèle H_0.

On s'intéresse maintenant à nouveau au modèle H_1 décrit en figure 1.

Question 12 : Donner toutes les séquences produites par H_1 de taille 1, ainsi que leurs probabilités et la somme de ces probabilités.

Question 13 : Donner toutes les séquences produites par H_1 de taille 2, ainsi que leurs probabilités et la somme de ces probabilités.

Question 14 : Justifier le fait qu'il soit possible de produire des séquences non vides de taille arbitraire dans H_1.

On revient enfin au cas le plus général d'un modèle de Markov caché H quelconque.

Question 15 : Donner une condition suffisante sur les propriétés du graphe qui le représente pour qu'un modèle de Markov caché puisse produire des séquences arbitrairement longues. Montrer que tout modèle de Markov caché satisfait cette propriété.

Question 16 : Démontrer que, dans un modèle de Markov caché H quelconque, on a la propriété suivante : « Pour tout $k \geq 1$, la somme des probabilités des séquences de taille k produites par le modèle H est égale à 1. ». On pourra procéder par récurrence sur $k \geq 1$.

Partie II. Représentation des modèles de Markov cachés

On s'intéresse dans cette partie à la représentation des modèles de Markov cachés dans le langage Python. On implémentera ceux-ci dans le paradigme objet. La classe implémentant un modèle de Markov caché s'appelle la classe `HMM` (de *Hidden Markov Model*).

3 La classe `HMM`

Question 17 : En s'appuyant sur la description des modèles de Markov cachés, décrire la structure d'un objet `HMM` représentant un modèle de Markov caché pour des séquences ADN (donc avec le même alphabet). On demande ici une description textuelle des attributs et méthodes, pas une implémentation en Python.

Question 18 : Donner une implémentation de la classe `HMM`, avec un constructeur prenant en paramètre les différents éléments du modèle, et une méthode pour afficher les paramètres du modèle. Pour implémenter les matrices en Python, qui n'a pas de type intégré pour cette structure, il est conseillé d'utiliser des listes imbriquées. On demande explicitement ici de donner le code des méthodes en Python.

Question 19 : Modifier le constructeur de cette classe pour qu'en l'absence de paramètre dans l'appel, il initialise le modèle avec le modèle nul vu dans la section 2.

Question 20 : Donner l'instruction Python permettant d'instancier le modèle H_1 vu en figure 1.

Question 21 : En Python, pour comparer l'égalité entre deux objets instances d'une même classe, on peut utiliser l'opérateur d'égalité `==`. Quand on utilise cet opérateur, on fait en réalité appel à une méthode `__eq__` propre à la classe, qui prend comme paramètres `self` et un deuxième paramètre `other`. Lorsque l'opérateur d'égalité `__eq__` n'est pas surchargé, donc si on ne le définit pas spécifiquement dans la classe, il est par défaut identique à l'opérateur d'identité qui renvoie vrai si les deux paramètres désignent la même instance d'une classe. Ainsi, si on initialise deux `HMM`, `H1` et `H2` avec les mêmes valeurs initiales, l'expression `H1 == H2` est évaluée à `False`. Ce n'est pas un comportement adéquat puisque l'on va considérer que deux modèles de Markov cachés sont égaux s'ils ont les mêmes valeurs pour les attributs correspondants aux éléments du modèle. Implémenter l'opérateur `__eq__` qui correspond à cette notion d'égalité.

Question 22 : Implémenter une méthode `estValide` qui indique si le modèle respecte les contraintes de base des modèles de Markov cachés, données par les équations (1), (2), (3). On pourra utiliser l'opérateur `sum` qui prend une liste en paramètre et renvoie la somme des éléments de la liste. Exemple : `sum([1,2,3])` renvoie la valeur 6.

4 Adaptation à une famille de séquences

Dans la section précédente, nous avons vu une façon très générique d'implémenter les modèles de Markov cachés. Mais pour s'adapter à des familles de séquences proches bien définies (par exemple, une famille de séquences représentant un même gène dans plusieurs espèces), il est indispensable d'initialiser le modèle avec des informations suffisamment pertinentes vis-à-vis de cette famille de séquences. On s'appuie pour cela sur un alignement de q séquences, défini comme un tableau de dimension $q \times p$, où q est le nombre de séquences alignées, et p la longueur de l'alignement. Dans la ligne i de ce tableau, apparaissent dans l'ordre les caractères de la $i$ème séquence, entre lesquels peuvent s'intercaler des caractères tiret - comme dans la figure 2. On va s'intéresser ici à des alignements optimaux. Nous ne détaillons pas ici comment les obtenir, mais de cette optimalité découlent quelques propriétés que nous supposerons satisfaites dans la suite :

— la longueur de l'alignement p est inférieure à la somme des longueurs des séquences ;

— la longueur de l'alignement p est supérieure ou égale à la plus petite des longueurs des séquences ;

— chaque colonne contient au moins un caractère de l'alphabet ;

— il peut y avoir des tirets dans n'importe quelle colonne.

À partir de cet alignement, on calcule les fréquences d'apparition des différents caractères de l'alphabet $\Sigma = \{A,C,G,T\}$ dans les colonnes, comme sur la figure 2. Les tirets - dans l'alignement

indiquent que la séquence s'aligne avant et après mais qu'il n'y a pas de caractère dans la séquence à mettre en correspondance dans la colonne où figure le tiret. Les successions de tirets s'appellent des *gaps*. Ainsi, dans la figure 2, la première séquence GATTCA est sans *gap*, la deuxième séquence GCTA présente deux *gaps* de taille 1, la troisième séquence GATTT présente un *gap* de taille 1, et la quatrième séquence GTCG présente un *gap* de taille 2.

colonne n°	1	2	3	4	5	6
séq. 1	G	A	T	T	C	A
séq. 2	G	-	C	T	-	A
séq. 3	G	A	T	T	-	T
séq. 4	G	-	-	T	C	G

A	0	1	0	0	0	$\frac{1}{2}$
C	0	0	$\frac{1}{3}$	0	1	0
G	1	0	0	0	0	$\frac{1}{4}$
T	0	0	$\frac{2}{3}$	1	0	$\frac{1}{4}$

FIGURE 2 – En haut, l'alignement des séquences GATTCA, GCTA, GATTT et GTCG. En bas, les fréquences d'apparition des caractères dans l'alignement. Le modèle de Markov caché généré à partir de cet alignement est noté H_2.

Le modèle de Markov caché correspondant à un alignement est alors défini comme suit :

— il est composé d'autant d'états que la longueur p de l'alignement,

— le départ se fait de façon sûre (*i.e.* avec probabilité égale à 1) dans le premier état,

— les probabilités d'émission dans l'état i correspondent aux fréquences des caractères apparaissant dans la colonne i de l'alignement,

— les probabilités de transition sont calculées de la façon suivante : une transition passe forcément d'un état où l'on émet vers un état où l'on émet également, elle ne peut pas passer par un *gap*. Ainsi, considérant un état i, pour chaque ligne ℓ où un caractère différent de `'-'` apparaît (donc pour lequel on émet dans l'état i), on va regarder dans quelle colonne $j_{i,\ell}$ apparaît le prochain caractère émis (donc différent de `'-'`). S'il y a c_i caractères de l'alphabet Σ apparaissant dans la colonne i, la probabilité de transition de l'état i vers l'état j est le nombre de lignes ℓ telles que $j_{i,\ell} = j$, divisé par c_i, et le dernier état transitionne de façon sûre (*i.e.* avec une probabilité égale à 1) vers lui-même.

Ainsi, dans la figure 2 par exemple, pour calculer les probabilités de transition de l'état $i = 1$ vers les autres états, on va considérer le nombre de caractères différents de - dans la colonne 1, c'est-à-dire $c_1 = 4$.

À la ligne 1, le caractère émis après celui de la colonne 1 ('G') est celui de la colonne 2 ('A') donc $j_{1,1} = 2$. À la ligne 2, le caractère émis après celui de la colonne 1 ('G') est celui de la colonne 3 ('C') donc $j_{1,2} = 3$. À la ligne 3, le caractère émis après celui de la colonne 1 ('G') est celui de la colonne 2 ('A') donc $j_{1,3} = 2$. À la ligne 4, le caractère émis après celui de la colonne 1 ('G') est celui de la colonne 4 ('T') donc $j_{1,4} = 4$.

La probabilité de transition de l'état $i = 1$ vers l'état $j = 2$ correspond à la fréquence des lignes ℓ telles que $j_{1,\ell} = 2$, ce qui donne une probabilité égale à $\frac{2}{4} = 0.5$. La probabilité de transition de l'état $i = 1$ vers l'état $j = 3$ correspond à la fréquence des lignes ℓ telles que $j_{1,\ell} = 3$, ce qui donne une probabilité de $\frac{1}{4} = 0.25$. La probabilité de transition de l'état $i = 1$ vers l'état $j = 4$ correspond à la fréquence des lignes ℓ telles que $j_{1,\ell} = 4$, ce qui donne une probabilité de $\frac{1}{4} = 0.25$. Toutes les autres probabilités de transition de l'état $i = 1$ vers les autres états sont nulles.

Question 23 : Représenter sous forme de graphe le modèle de Markov caché H_2 correspondant à l'alignement de la figure 2. Préciser à part le vecteur des probabilités de départ.

Question 24 : Un alignement est composé de séquences de même longueur (en comptant les *gaps*). On supposera que ces séquences ne comportent que les caractères `'A'`, `'C'`, `'G'`, `'T'` et `'-'` en majuscules. Choisir en justifiant une structure de données pour représenter un alignement.

Question 25 : Implémenter un constructeur `fromAlignement` de la classe `HMM` qui prend en paramètre un alignement et initialise les paramètres du modèle.

Question 26 : Montrer qu'un modèle de Markov caché initialisé avec un alignement satisfait nécessairement les contraintes de base des modèles de Markov cachés, données par les équations (1), (2) et (3).

5 Sur la représentation du modèle

Nous avons admis que les paramètres probabilistes du modèle de Markov caché sont représentés sous forme de matrices (implémentées par des listes de listes en Python). On s'intéresse à la matrice de probabilités de transition, dans le cas où le modèle est construit à partir d'un alignement. Il est admis qu'un alignement de bonne qualité se caractérise par des *gaps* de longueur faible.

Question 27 : Majorer le nombre d'entrées non nulles dans la matrice de probabilités de transition lorsque le modèle est construit à partir d'un alignement de longueur p et dont les *gaps* sont de longueur maximale g.

Question 28 : La densité d'une matrice est calculée comme étant le nombre d'éléments non nuls de la matrice divisé par le nombre total d'éléments dans la matrice. Donner un majorant de la densité de la matrice des probabilités de transition, en fonction de p.

Question 29 : La matrice des probabilités de transition peut être vue comme la matrice d'adjacence du graphe représentant le modèle de Markov. Donner une autre représentation possible de ce graphe, plus adaptée à la densité du graphe (correspondant à la densité de sa matrice d'adjacence). Appliquer cette représentation à la matrice de probabilités de transition du modèle H_2.

Question 30 : Donner de façon textuelle (on ne demande pas d'implémenter) les grandes modifications à réaliser pour implémenter la structure de données choisie. On indiquera notamment quelles sont les méthodes impactées et les changements éventuels de la complexité de l'accès aux éléments des matrices.

Partie III. Algorithmes

6 L'algorithme de décodage

Le problème du décodage consiste à fournir, étant donnés une séquence w et un modèle de Markov caché H, une marche de probabilité maximale dans H qui produit w.

Question 31 : Montrer que, pour une séquence donnée $w_1 \dots w_k$, si $m_1 \dots m_k$ est une marche de probabilité maximale qui produit w, alors pour tout $k' \leq k$, $m_1, \dots m_{k'}$ est, parmi tous les préfixes de taille k' de marches produisant w se terminant en l'état m'_k, une marche qui produit $w_1 \dots w_{k'}$ avec une probabilité maximale.

Question 32 : Soit $w = w_1 \dots w_k$ une séquence de caractères de Σ. On définit le tableau suivant δ, de taille $k \times n$, avec $\delta[i, j]$ qui contient la probabilité maximale qu'une marche se terminant en l'état j produise le début de séquence $w_1 \dots w_i$.

a) Que contient la première ligne de ce tableau ?

b) En supposant les $i - 1$ premières lignes de ce tableau remplies, comment remplit-on la $i-$ième ligne ? Au besoin, faire un schéma pour appuyer l'explication.

c) Dans quelle case du tableau se trouve la probabilité maximale qu'une marche produise w ?

L'algorithme de Viterbi calcule une marche la plus probable pour une séquence $w = w_1 \dots w_k$ donnée, dans un modèle de Markov caché $H = (\Sigma, S, T, E, P)$. Il utilise le principe de la programmation dynamique, et comporte une phase de remplissage du tableau δ permettant de trouver la probabilité maximale qu'une marche produise w, puis une phase de retour arrière (*backtracking*) permettant d'identifier une marche qui atteint cette probabilité. La sortie de l'algorithme est cette marche la plus probable. On donne cet algorithme en figure 3.

Question 33 : Donner le résultat de cet algorithme sur la séquence AGG et le modèle H_1 de la figure 1. On indiquera également les tableaux δ et b remplis.

Question 34 : Donner une preuve de la correction de cet algorithme.

Question 35 : Donner la complexité de cet algorithme en temps dans le pire des cas.

7 Les algorithmes d'évaluation

L'algorithme de Viterbi permet d'identifier une marche qui produit une séquence de façon la plus probable dans un modèle de Markov caché. Une autre question, plus simple, consiste à calculer la probabilité d'émission d'une séquence donnée dans un modèle de Markov caché. On utilise également le principe de la programmation dynamique pour réaliser ce calcul, en remplissant un tableau α de taille $k \times n$. Ce tableau contient les probabilités $\alpha(i, j)$ de produire le préfixe $w_1 \dots w_i$ dans le modèle H en arrivant dans l'état j.

```
1  VITERBI(H , w)
2  Début
3      Soit k la taille de w et n la taille de S
4      Soient δ[1..k, 1..n] et b[1..k, 1..n] deux nouveaux tableaux bidimensionnels
5      Soit z un tableau d'entiers de taille k
6      Pour j = 1 à n
7          δ[1, j] = P(j) × E(j, w_1)
8          b[1, j] = 0
9      Fin Pour
10     Pour i = 2 à k
11         Pour j = 1 à n
12             δ[i, j] = E(j, w_i) × max_{ℓ∈S} {δ[i − 1, ℓ] × T(ℓ, j)}
13             b[i, j] = arg max_{ℓ∈S} {δ[i − 1, ℓ] × T(ℓ, j)}
14         Fin Pour
15     Fin Pour
16     z[k] = arg max_{ℓ∈S} {δ[k, ℓ]}
17     Pour i = k à 2
18         z[i − 1] = b[i, z[i]]
19     Fin Pour
20     Renvoyer z
21 Fin
```

FIGURE 3 – Algorithme de Viterbi pour le décodage des séquences dans les modèles de Markov cachés. La notation $\arg\max_{\ell \in S}\{f(l)\}$ signifie « une valeur de ℓ dans S qui maximise la fonction $f(l)$ ».

Question 36 : En vous appuyant sur la question 6, donner la probabilité que le modèle H_1 émette la séquence AAG.

Question 37 : Proposer des modifications de l'algorithme de Viterbi pour calculer le tableau α. Indiquer notamment s'il faut enlever ou rajouter des structures de données ou des phases. Justifier.

Question 38 : Comment trouver la probabilité de la séquence w à partir du tableau α renvoyé ?

L'algorithme modifié s'appelle l'algorithme *Forward*. Il existe un autre algorithme permettant de calculer la probabilité d'une séquence produite par un modèle de Markov caché en remplissant un tableau avec des probabilités partielles. Il s'agit de l'algorithme *Backward*, donné en figure 4.

Question 39 : Préciser les différences entre l'algorithme *Forward* et l'algorithme *Backward*. Notamment, que contient le tableau β ?

Question 40 : Indiquer comment trouver la probabilité de la séquence w produite par le modèle H à partir de la sortie de l'algorithme *Backward*.

Question 41 : Donner la complexité en temps et en espace de l'algorithme *Forward* et de l'algorithme *Backward*.

Question 42 : Donner l'implémentation d'une méthode `backward` correspondant à l'algorithme ci-dessus.

```
BACKWARD(H , w)
Début
    Soit k la taille de w et n la taille de S
    Soit β[1..k, 1..n] un nouveau tableau bidimensionnel
    Pour j = 1 à n
        β[k, j] = E(j, w_k)
    Fin Pour
    Pour i = k − 1 à 1
        Pour j = 1 à n
            β[i, j] = 0
            Pour ℓ = 1 à n
                β(i, j) = β(i, j) + E(j, w_i) × β(i + 1, ℓ) × T(j, ℓ)
            Fin Pour
        Fin Pour
    Fin Pour
    Pour j = 1 à n
        β(1, j) = P(j) × β(1, j)
    Fin Pour
    Renvoyer β
Fin
```

FIGURE 4 – Algorithme Backward permettant l'évaluation des séquences dans un modèle de Markov caché.

Partie IV. Pour aller plus loin

Le modèle de Markov caché généré à partir d'un alignement permet de bien capturer la structure globale des séquences qui composent l'alignement, mais ne permet pas nécessairement de représenter des séquences proches de façon satisfaisante. Cette partie a pour objectif d'améliorer la génération du modèle de Markov caché à partir d'un alignement.

8 Les *pseudo-counts*

Un modèle de Markov caché construit à partir d'un alignement est souvent trop restreint dans son expressivité, et des séquences proches de celles de la famille ne peuvent pas être produites, à cause de sa structure. Ainsi, dans le modèle H_2 produit à partir de l'alignement de la figure 2, la séquence AATTCA, très proche de la première séquence, ne pourrait être produite car la probabilité d'émission de A en l'état 1 est nulle, du fait de n'avoir observé que des G dans cette colonne. Pour pallier ce problème, on introduit de légères perturbations dans les probabilités d'émission, les *pseudo-counts*, définis de la façon suivante, avec n le nombre d'états du modèle et pour tout $i \in \{1, \ldots, n\}$, en notant n_i le nombre de caractères c tels que $E(i, c) = 0$. Alors, pour tout $c \in \Sigma$:

— si $E(i, c) = 0$, remplacer $E(i, c)$ par $\frac{1}{n \times n_i}$,

— sinon, remplacer $E(i, c)$ par $\frac{n-1}{n} E(i, c)$.

Question 43 : Démontrer que les contraintes classiques sur les modèles de Markov restent respectées.

Question 44 : Que devient le modèle H_2 lorsque l'on introduit les *pseudo-counts* ? On demande uniquement la matrice de probabilités d'émission modifiée.

Question 45 : Modifier l'implémentation du constructeur `fromAlignement` de la classe `HMM`, écrit à la question 25, pour intégrer les *pseudo-counts*.

9 Les zones mal alignées

Un autre problème peut se poser avec les alignements de séquences pour initialiser les modèles de Markov cachés : certaines colonnes successives contiennent des *gaps*, ce qui indique un endroit où l'alignement est de mauvaise qualité localement à ces colonnes, que l'on va qualifier de « mal alignées ». Lorsque l'on connaît un intervalle $[i, i+t]$ de colonnes mal alignées (avec $t > 0$), on peut remplacer les t états correspondant par un seul état, appelé état d'insertion, dans lequel les probabilités de transition et d'émission sont calculées globalement. Ainsi, dans l'exemple de la figure 5, l'état d'insertion couvre les trois colonnes 4, 5 et 6, dans lesquelles la fréquence d'apparition des A est $\frac{1}{5}$, la fréquence d'apparition des C est $\frac{2}{5}$, la fréquence d'apparition des G est $\frac{1}{5}$ et la fréquence d'apparition des T est $\frac{1}{5}$. Pour ce qui est des transitions, on compte dans cet état :

— une transition de la colonne 4 à la colonne 5 (interne)

— une transition de la colonne 5 à la colonne 6 (interne)

— une transition de la colonne 6 à la colonne 7 (externe)

— deux transitions de la colonne 4 à la colonne 7 (externes),

soit des probabilités de transition de l'état d'insertion à lui-même égale à $\frac{2}{5}$ et de l'état d'insertion à l'état suivant égale à $\frac{3}{5}$.

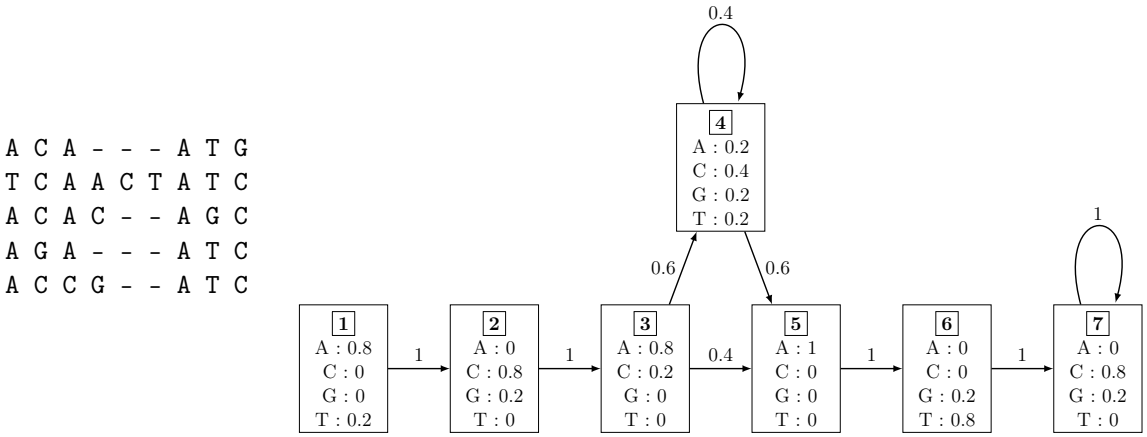


FIGURE 5 – À gauche, un alignement de séquences présentant une zone mal alignée au niveau de l'intervalle de colonnes $[4, 6]$. À droite, le modèle de Markov caché (sans *pseudo-counts*) généré à partir de cet alignement, et de l'intervalle des colonnes mal alignées.

Tournez la page S.V.P.

Remarque : une colonne isolée avec des tirets, comme la colonne 5 de l'alignement de la figure 2, n'a pas besoin d'être transformée en état d'insertion (elle est déjà un état d'insertion).

Question 46 : Si l'on part du principe qu'une colonne mal alignée est une colonne qui contient au moins un tiret, appliquer ce principe à l'alignement de la figure 2 et donner le graphe correspondant au modèle de Markov ainsi généré.

Question 47 : Donner un algorithme qui prend en entrée un alignement de séquences, et renvoie en sortie une liste d'intervalles de colonnes, dans laquelle les colonnes mal alignées ont été fusionnées. Par exemple, cet algorithme doit renvoyer la liste :

```
[(0, 0), (1, 2), (3, 3), (4, 4), (5, 5)]
```

à partir de l'alignement de la figure 2 et la liste :

```
[(0, 0), (1, 1), (2, 2), (3, 5), (6, 6), (7, 7), (8, 8)]
```

à partir de l'alignement de la figure 5. Quelle est la complexité en temps dans le pire des cas de cet algorithme ?

Question 48 : Donner l'implémentation de cet algorithme sous forme d'une ou plusieurs fonction(s).

Question 49 : Modifier le constructeur de la classe `HMM` qui prend en entrée un alignement, et une liste d'intervalles de colonnes comme construite à la question précédente. Ce constructeur doit construire le modèle en tenant compte de ces colonnes mal alignées.

* *
*

MINISTÈRE
DE L'ÉDUCATION
NATIONALE
ET DE LA JEUNESSE
Liberté
Égalité
Fraternité

EAE INF 3

SESSION 2024

AGREGATION
CONCOURS EXTERNE

Section : INFORMATIQUE

ÉPREUVE SPÉCIFIQUE SELON L'OPTION CHOISIE :
- **ÉTUDE DE CAS INFORMATIQUE**
- **FONDEMENTS DE L'INFORMATIQUE**

Durée : 6 heures

L'usage de tout ouvrage de référence, de tout dictionnaire et de tout matériel électronique (y compris la calculatrice) est rigoureusement interdit.

Il appartient au candidat de vérifier qu'il a reçu un sujet complet et correspondant à l'épreuve à laquelle il se présente.

Si vous repérez ce qui vous semble être une erreur d'énoncé, vous devez le signaler très lisiblement sur votre copie, en proposer la correction et poursuivre l'épreuve en conséquence. De même, si cela vous conduit à formuler une ou plusieurs hypothèses, vous devez la (ou les) mentionner explicitement.

NB : Conformément au principe d'anonymat, votre copie ne doit comporter aucun signe distinctif, tel que nom, signature, origine, etc. Si le travail qui vous est demandé consiste notamment en la rédaction d'un projet ou d'une note, vous devrez impérativement vous abstenir de la signer ou de l'identifier.
Le fait de rendre une copie blanche est éliminatoire.

Tournez la page S.V.P.

A

INFORMATION AUX CANDIDATS

Vous trouverez ci-après les codes nécessaires vous permettant de compléter les rubriques figurant en en-tête de votre copie.

Ces codes doivent être reportés sur chacune des copies que vous remettrez.

► **Etude de cas informatique** :

Concours	Section/option	Epreuve	Matière
EAE	6200A	103	9424

► **Fondement de l'informatique** :

Concours	Section/option	Epreuve	Matière
EAE	6200A	103	9425

Épreuve spécifique

Étude de cas informatique 3

Fondements de l'informatique 15

IMPRIMERIE NATIONALE – 24 0115 – D'après documents fournis

EAE INF 3

Étude de cas informatique

Préliminaires

Cette épreuve a pour objet l'étude du cas des compétitions d'ultimate. Chaque partie comprend la définition d'un certain nombre de problématiques à résoudre et présente des objectifs concrets, ainsi que la réflexion sur les moyens de les atteindre. Il est conseillé de bien lire l'ensemble du sujet avant de commencer à répondre à l'une des parties.

Attendus. Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction. Notamment, tout code doit être lisible, intelligible et documenté.

Dépendances. Ce sujet contient cinq parties. Les différentes parties et un grand nombre de leurs questions sont largement indépendantes. Il est possible d'aborder les différentes parties dans l'ordre qui vous conviendra le mieux mais en indiquant clairement à quelle question il est répondu.

Description du système étudié

L'ultimate. L'ultimate (ou *flying disc* en anglais) est un sport collectif inscrit au mouvement olympique. L'origine de ce sport est attribuée à des étudiants de l'Université de Yale, en 1940. Le jeu consiste à se transmettre un disque volant (aussi connu sous le nom de *frisbee*) [1] entre les joueurs d'une même équipe pour progresser sur le terrain de jeu. Lorsqu'un joueur réceptionne le disque, il ne peut plus se déplacer, mais peut établir un pied pivot (comme au handball, par exemple). Il a alors 10 secondes pour effectuer une passe à un(e) membre de son équipe.

Le but du jeu est donc d'amener le disque dans la zone de but adverse par une succession de passes entre les joueurs d'une même équipe. Si une passe est échouée, atterrit hors des limites du terrain ou est interceptée par l'équipe adverse, alors cette dernière récupère la possession du disque (l'attaque) et doit à son tour progresser par passes successives pour rejoindre son extrémité de terrain. Un point est marqué quand un joueur réceptionne le disque transmis par un membre de son équipe dans la zone d'en-but. Après chaque point marqué, les équipes se tiennent sur leur ligne d'en-but. L'équipe qui a marqué le dernier point lance le disque. L'autre équipe prend possession du disque là où il atterrit et devient alors l'équipe attaquante.

Les matchs. Un match d'ultimate se remporte, soit en 15 points, soit au temps (100 min). Dans le second cas, à la fin du temps réglementaire, le score à atteindre est alors le score le

1. « Frisbee » est une marque déposée de l'entreprise Wham-O-Holding basée à Hong Kong.

plus haut +1 point—ce qu'on appelle le « cap à 1 », dans la limite des 15 points. Par exemple, si le score est de 13 – 10 à la fin du temps réglementaire, alors la première équipe à atteindre 14 points (13+1) gagne le match. L'ultimate se pratique dans sa version la plus courante sur terrain en herbe à l'extérieur (7 contre 7), mais peut aussi se pratiquer en intérieur (5 contre 5), ou sur la plage (5 contre 5).

L'ultimate se démarque des autres sports collectifs par le fait qu'il s'agit d'un sport auto-arbitré, quel que soit le niveau auquel il est pratiqué. Chaque joueuse/-eur ou équipe peut donc appeler une faute, ce qui suspend le jeu (mais pas le décompte du temps) et engage une phase d'échange entre les joueurs pour décider si le jeu doit se poursuivre, ou si les joueurs doivent revenir à la situation de jeu précédant la faute. Par exemple, les contacts étant interdits, un joueur peut appeler une faute s'il s'estime gêné dans ses déplacements ou empêché d'effectuer une passe.

Les compétitions. En compétition, on retrouve plusieurs catégories : *open*, femmes, mixte (4 hommes/3 femmes ou 4 femmes/3 hommes). Il n'existe donc pas de catégorie réservée exclusivement aux hommes, la catégorie *open* étant aussi ouverte aux femmes. Les compétitions d'ultimate peuvent prendre plusieurs formes.

La formule championnat regroupe un ensemble d'équipes qui doivent toutes se rencontrer. Pour chaque journée du championnat, une équipe rencontre une et une seule autre équipe adverse. Un championnat peut être organisé en 2 phases : une phase aller et une retour. Le nombre de phases est décidé en amont d'un championnat et détermine le nombre de journées à jouer en fonction du nombre d'équipes inscrites.

La formule tournoi consiste en un mini-championnat qui est organisé sur un jour—ou un week-end—entre un nombre restreint d'équipes qui se rencontrent en une seule phase. En fonction du nombre d'équipes inscrites à un tournoi, il est possible de répartir ces dernières de manière équilibrée (plus ou moins une équipe selon le nombre d'équipes inscrites au tournoi) dans un ensemble de poules afin de limiter le nombre total de matchs joués.

La formule division consiste quant à elle en une succession de tournois. Les équipes concurrentes peuvent alors être réparties sur un ensemble de divisions. Par exemple, 30 équipes peuvent être réparties en 6 divisions de 5 équipes. À chaque journée, le premier de chaque division est promu dans la division supérieure (s'il en existe une), tandis que le dernier du groupe descend dans la division inférieure (si elle existe). Le nombre de journées est déterminé à l'avance.

Les classements. À la fin de toute compétition, il est possible d'établir un classement qui correspond à un ordre total des équipes engagées. Les règles de classement d'une compétition sont déterminées avant que celle-ci ne débute. Ces règles déterminent notamment le nombre de points attribués par victoire, ainsi que la gestion des situations d'égalité entre plusieurs équipes (e.g., en favorisant l'équipe qui a marqué le plus de points).

Toutes les formules peuvent éventuellement se conclure par une session de *playoff*. Durant cette session, les 2^N meilleures équipes de la phase préliminaire sont alors convoquées pour se

rencontrer dans une série de matchs à éliminations directes (e.g., quart de finale, demi-finale, finale pour une session de *playoff* à 8 équipes). La sélection des couples d'équipes pour les $N/2$ premiers matchs d'une session *playoff* peuvent être déterminés aléatoirement entre les équipes sélectionnées, ou en tenant compte du classement (e.g., la 1e équipe contre la 8e, la 2e contre la 7e, etc.). L'ordre dans lequel les couples sont établis fixe alors la suite des matchs jusqu'à la finale. L'équipe qui gagne alors la finale est alors proclamée gagnante de la compétition.

En dehors des compétitions officielles, il existe également des tournois ouverts, appelés *hat*. Les joueuses/-eurs s'y inscrivent de manière individuelle et la composition des équipes est tirée au sort, en respectant des conditions de parités et en essayant d'équilibrer les équipes. Outre, le genre, les joueuses/-eurs indiquent leur niveau de jeu et leur poste de prédilection (passeur, receveur, sans préférence) afin de constituer des équipes d'un niveau équivalent. Une fois les équipes constituées, ces tournois ouverts suivent le même règlement qu'une compétition de formule tournoi.

Dans la suite de ce sujet, nous nous intéressons à la modélisation d'un système information en ligne, commme une application web par exemple, qui permettrait de gérer des compétitions organisées par les fédérations d'ultimate, depuis l'inscription des joueurs à une compétition, jusqu'à la gestion des résultats.

Partie I. Application mobile

Dans un premier temps, nous souhaitons concevoir une application web permettant *i)* aux spectateurs de suivre l'évolution du score d'un match depuis leur téléphone, et *ii)* aux joueurs de mettre à jour ce score en direct. Nous proposons notamment de nous intéresser à l'interface graphique permettant de consulter le score des matchs d'une compétition.

Question 1. Le domaine sur lequel est déployé l'application est `https://app.ultimate.fr/`. Expliquez à quoi fait référence le caractère `s` du protocole `https://`. Quel est le port réseau standard associé à ce protocole ?

Question 2. Quel est l'intérêt d'avoir recours au protocole HTTPS pour une application web ?

Question 3. Sur quelles couches du modèle OSI (*Open Systems Interconnection*) repose le protocole HTTPS ?

Question 4. Écrire une requête HTTP minimale permettant d'afficher la page principale de l'application web étudiée, et expliquer le rôle de chacun de ses champs.

Question 5. Chaque compétition doit pouvoir disposer de sa propre adresse pour consulter facilement les scores et autres informations relatives à son organisation (composition des équipes, planning des matchs, classement temporaire, etc.). Proposez une structuration d'URL pour l'application web qui permette de consulter le score d'un match `m` organisé dans le cadre de la compétition `c`.

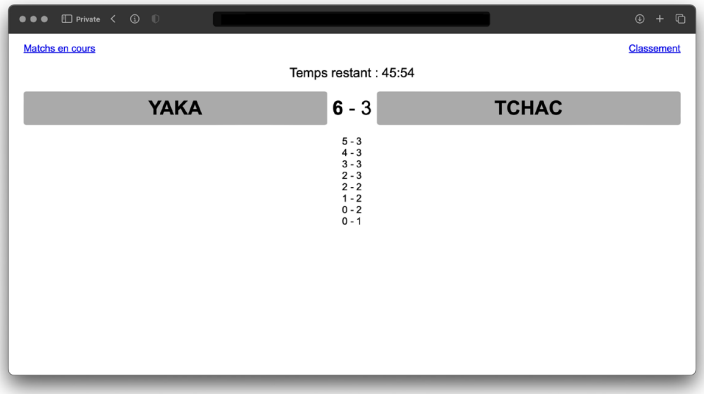


FIGURE 1 – IHM de la page de suivi d'un match accessible pour un spectateur. Le contenu de la page est centré et les polices sont volontairement agrandies pour faciliter la lecture des informations importantes. De haut en bas, les liens hypertextes permettent d'accéder à la liste des matchs en cours et le(s) classement(s) de la compétition. Le temps restant indique le nombre de minutes et secondes à jouer avant la fin du temps officiel. En cas d'application du "cap à 1", le temps restant compte à partir de zéro jusqu'à ce que le vainqueur du match soit désigné. L'historique des scores, indiqué dans la partie inférieure de l'écran, liste la séquence des précédents scores du match.

Question 6. Est-il possible de faire en sorte que la même application web côté serveur puisse servir des requêtes en provenance de clients de types «navigateur web» ou «application mobile dédiée» ? Si non, pourquoi ? Si oui, comment peut-elle le faire ?

Question 7. Le code compilé d'une telle application mobile de suivi des matchs peut-il s'exécuter sur n'importe quel équipement ? Pourquoi ? Est-ce également le cas du code de l'application web ? Pourquoi ?

Question 8. Du point de vue du réseau, discutez les avantages et inconvénients d'utiliser une application mobile dédiée ou une application web pour fournir un service de suivi de scores.

Partie II. Gestion d'un match d'ultimate

Nous proposons ensuite de nous intéresser plus spécifiquement à l'interface humain machine (IHM) permettant de suivre en direct le score des matchs d'une compétition. Dans cette partie, nous considérons plus précisément le cas d'une interface HTML qui est accessible depuis n'importe quel navigateur web comme, par exemple, celle illustrée dans la figure 1.

Question 9. Donnez le code HTML/CSS statique qui correspond à l'esquisse d'IHM décrite dans la figure 1. Les informatiques relatives au nom des équipes, des scores et de l'historique peuvent être indiqués dans le texte de la réponse.

Question 10. Enrichissez le code HTML/CSS précédent pour proposer une solution supportant la mise-à-jour dynamique, i.e. sans rechargement de la page, de l'affichage du score d'un match en cours, ainsi que l'historique de l'évolution du score pour ce match.

Question 11. Donnez le code JavaScript qui permet à l'application cliente de recevoir les informations relatives à l'évolution du score du match consulté. Le code proposé doit permettre d'être informé qu'un point vient d'être marqué par l'une ou l'autre des deux équipes sur le terrain. Le nombre de points de l'équipe qui vient de marquer doit s'afficher en gras.

Dans un second temps, nous proposons d'enrichir cette page HTML pour ajouter des boutons de contrôle du score qui ne soient accessibles qu'aux joueurs. Ces boutons correspondent aux parties grisées dans la figure 1. En cliquant sur le nom d'une équipe des deux équipes, un joueur autorisé incrémente alors immédiatement le score de cette équipe.

Question 12. Pourquoi l'utilisation de HTTPS ne suffit pas à limiter l'accès aux boutons de contrôle du score ? Décrivez précisément une solution, sans en fournir le code, pour restreindre l'accès à cette page à un ensemble de joueurs autorisés.

Question 13. Donnez le code JavaScript de la fonction `marquerPoint(equipe)`, invoquée en cliquant sur le bouton associé à l'équipe `equipe` pour incrémenter le score d'une unité en sa faveur.

Question 14. Expliquez précisément comment il est possible pour l'application web de s'assurer qu'un point qui vient d'être marqué n'est pas comptabilisé plusieurs fois par plusieurs joueurs qui seraient connectés simultanément sur cette page.

Question 15. Proposez une solution côté client, en JavaScript, pour que des accès concurrents des joueurs ne conduisent pas à incrémenter plusieurs fois le score d'une même équipe pour enregistrer un point qu'elle viendrait de marquer.

Question 16. Est-il également nécessaire de modifier le code de l'application côté serveur pour éviter cette situation ? Pourquoi ?

On considère la définition Python d'un match d'ultimate entre 2 équipes sous la forme de classes Python permettant de consulter et manipuler le score comme suit :

```
class Equipe:
    def __init__(self,name):
        self._name = name

    def __str__(self):
        return self._name

    def __repr__(self):
        return f'Equipe({self._name})'
```

```
class Match:
    def __init__(self, locaux, visiteurs):
        if locaux == visiteurs:
            raise Exception("Les équipes doivent être distinctes.")
        self._score = {locaux:0, visiteurs:0}
        self._dernier = None

    def equipes(self):
        return self._score.keys()

    def score(self):
        return self._score.copy()

    def gagnant(self):
        return max(self._score, key=self._score.get)

    def perdant(self):
        return min(self._score, key=self._score.get)

    def marquerPoint(self, equipe):
        self._score[equipe] += 1
        self._dernier = equipe

    def annulerPoint(self):
        if self._dernier is None:
            raise Exception("Aucun point à annuler.")
        self._score[self._dernier] -= 1
        self._dernier = None

    def __str__(self):
        return self._score
```

À chaque match est donc associé 2 équipes distinctes (les locaux et les visiteurs) et un score initial fixé à 0 - 0.

Question 17. Dans la suite du sujet, nous souhaitons utiliser la classe `Equipe` dans différentes structures de données, comme des dictionnaires, en nous assurant que le nom de l'équipe est bien unique, i.e. on s'attend à ce que l'expression `Equipe("TCHAC") == Equipe("Tchac")` soit évaluée à vrai. Complétez le code de la classe Python `Equipe` décrite ci-dessus afin de pouvoir l'utiliser comme clé d'identification d'une équipe au sein d'une compétition.

Question 18. Donnez le code Python d'une classe `GestionMatch` qui permet de gérer un match d'ultimate. La solution proposée doit notamment permettre d'implémenter la règle du "cap à 1" en tenant compte de tous les états dans lesquels un match peut être (*planifié*, *en cours*, *cap*, *terminé*). Le code attendu doit expliciter ces états et l'ensemble des méthodes et attributs requis pour suivre l'évolution du score d'un match.

Pour information, les scores sont mémorisés dans une base de données SQL via une table `HistoriqueScores` dont la définition est donnée comme suit :

```
CREATE TABLE Equipes (
    EquipeID INT PRIMARY KEY AUTO_INCREMENT,
    Nom VARCHAR(255) NOT NULL UNIQUE
);

CREATE TABLE Matchs (
    MatchID INT PRIMARY KEY AUTO_INCREMENT,
    EquipeAID INT,
    EquipeBID INT,
    FOREIGN KEY (EquipeAID) REFERENCES Equipes(EquipeID)
    FOREIGN KEY (EquipeBID) REFERENCES Equipes(EquipeID)
);

CREATE TABLE HistoriqueScores (
    HistoriqueID INT PRIMARY KEY AUTO_INCREMENT,
    MatchID INT,
    ScoreEquipeA INT,
    ScoreEquipeB INT,
    Estampille TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (MatchID) REFERENCES Matchs(MatchID)
);
```

Question 19. Donnez la requête SQL permettant de récupérer le score courant du match qui oppose les équipes YAKA et TCHAC.

Question 20. Proposez une solution pour annuler le dernier point marqué si une équipe appelle une faute qui remet en cause ce point. On considère ici qu'il n'est pas utile de conserver une trace de la faute qui a conduit à l'annulation du point. Si le code client nécessite d'être modifié, décrivez les changements à apporter au code HTML, CSS, et JavaScript pour mettre en place cette fonctionnalité du côté client. Si le code serveur nécessite d'être modifié, décrivez les changements à intégrer au code Python et à la base de données.

Partie III. Gestion de tournois d'ultimate

Nous proposons désormais d'étudier l'organisation de tournois d'ultimate qui se dérouleraient en deux temps : une étape préliminaire de poules, suivie d'une étape finale de *playoff* permettant de déterminer l'équipe gagnante d'un tournoi. Le nombre de poules d'un tournoi est déterminé en fonction du nombre de créneaux horaires disponibles, du nombre de terrains et du nombre de phases (par défaut, une seule phase est organisée). Lors de l'étape préliminaire, par souci d'équité, il est souhaitable qu'une équipe ne joue pas deux matchs dans deux plages consécutives d'une même journée, leurs laissant ainsi un temps suffisant de récupération.

Question 21. En partant du principe qu'un tournoi dispose de `t` terrains et `c` créneaux horaires, donnez le code Python de l'algorithme `liste_combinaisons(t,c)` qui liste les différentes combinaisons de nombres d'équipes (`n`) et de poules (`p`) acceptables en cherchant à maximiser le nombre d'équipes qu'il est possible d'inscrire à un tournoi en fonction du nombre de poules retenues par les organisateurs. En sachant qu'il est inutile qu'une équipe ne s'inscrive pour jouer moins de 3 matchs, ignorez toute combinaison qui ne respecte pas cette contrainte. Par exemple,

l'appel à la fonction `liste_combinaisons(4, 10)` pour calculer les combinaisons possibles avec 4 terrains et 10 créneaux devra retourner `{3: 12, 2: 10, 1: 6}`—i.e., un maximum de 12 équipes peuvent s'inscrire si les organisateurs privilégient 3 poules, 10 équipes pour 2 poules ou 6 équipes pour une poule unique.

Question 22. Donnez le code Python d'une fonction `liste_matchs(n,p)` qui génère la liste des matchs à jouer en fonction de la liste des `n` équipes inscrites et le nombre de poules souhaitées `p`. Les équipes sont réparties aléatoirement et de manière équilibrée dans les poules (plus ou moins une équipe selon le nombre total d'équipes). Le code produit doit obligatoirement faire apparaître la notion de `Poule` d'un tournoi sous la forme d'une classe dont le constructeur prend en paramètre la liste des équipes inscrites dans une poule donnée. La liste des matchs est générée à la construction de la poule. La classe `Poule` doit également fournir le code d'une méthode `classement()` permettant de retourner le classement courant d'une poule sous la forme d'une liste ordonnée d'équipes. Les équipes sont classées en priorité par nombre de matchs gagnés, puis par différence entre le nombre total de points marqués et le nombre total de points encaissés.

On s'intéresse maintenant à la planification d'un tournoi, c'est-à-dire l'association d'un match à un créneau horaire et un terrain, exactement. Pour ce faire, on souhaite disposer d'une classe `Planning` qui est initialisée avec une liste de matchs, un nombre de terrains et un nombre de créneaux disponibles pour le tournoi. La liste de matchs est triée par terrain, puis par créneau—*i.e.*, les n premiers éléments de la liste correspondent aux matchs du 1er créneau sur les n terrains disponibles, et ainsi de suite. La classe `Planning` permet ensuite de consulter un même planning par terrain ou par créneau. La méthode `par_terrain()` retourne notamment un ensemble de terrains qui contient une liste de matchs ordonnés par le créneau (un créneau non-occupé par un match est associé à la valeur `None`). De manière similaire, la méthode `par_creneau()` retourne une liste de créneaux qui contient un ensemble de terrains associés à un match (un terrain non-occupé par un match est associé à la valeur `None`).

Question 23. Donnez le code Python de la classe `Planning` qui permet d'initialiser la structure de données servant à gérer un planning de tournoi à partir des paramètres d'initialisation. La classe doit fournir le code des fonctions `par_terrain()` et `par_creneau()` afin de consulter le planning sous différentes vues.

Question 24. Donnez le code Python de 4 tests unitaires pour la classe `Planning` permettant de s'assurer qu'un planning de tournoi généré respecte les contraintes d'organisation décrites dans le sujet.

Question 25. Donnez le code Python d'une fonction `creer_planning(matchs,t,c)` qui construit le planning d'un tournoi en tenant compte du nombre `t` de terrains disponibles et le nombre `c` de plages horaires. Un planning alloue à chaque match un créneau horaire et d'un terrain. N'hésitez pas à préciser quelle(s) optimisation(s) vous avez mise(s) en œuvre dans le code de la fonction `creer_planning(matchs,t,c)` pour améliorer le temps de calcul d'un planning correct.

Question 26. Donnez le code Python d'une fonction `creer_playoff(nb_tours, poules)` qui sélectionne les premières équipes parmi la liste de `poules` d'un tournoi pour générer une liste initiale de matchs de *playoff*, à élimination directe, qui se dérouleront en `nb_tours` tours jusqu'à la finale.

Question 27. Donnez le code Python d'une fonction `tour_suivant(matchs)` qui analyse la liste des `matchs` joués pour préparer le tour suivant de la phase de `playoff` sous la forme d'une nouvelle liste de matchs en faisant se rencontrer les gagnants du tour courant uniquement.

Question 28. Donnez le code Python d'une fonction `joue_playoff(tours, poules)` qui utilise les fonctions des 2 questions précédentes pour orchestrer le déroulement de l'étape de *playoff* et retourner le podium du tournoi sous la forme d'une liste ordonnée de 3 équipes, en sachant que la troisième place est attribuée au terme d'une petite finale qui oppose les équipes qui ont perdu leur demi-finale.

Partie IV. Gestion d'une compétition en divisions

On s'intéresse ici à la formule division qui offre une forme de compétition hybride, entre championnats et tournois. À la différence des poules d'un tournoi, la composition des divisions n'a rien d'aléatoire puisqu'elle s'appuie sur le classement des équipes à l'issue de la précédente saison (qui n'était pas nécessairement organisée sous une formule division) pour déterminer comment les équipes doivent être regroupées. À noter que toute nouvelle équipe doit intégrer la dernière division.

Question 29. Donnez le code Python d'une classe `CompetitionDivision` dont le constructeur prend pour paramètres *i)* un `classement` d'équipes (de la meilleure à la moins bonne), *ii)* la `taille` maximale des divisions en terme de nombre d'équipes et *iii)* le nombre de `journées` prévues pour cette compétition. Vous proposerez la structure de données adéquate pour gérer une division, telle que décrite dans l'énoncé. Outre la définition du constructeur, la classe doit également fournir une fonction `classement()` qui retourne le classement courant d'une compétition comme une liste ordonnée d'équipes.

Question 30. Donnez le code Python d'une méthode `prochaine_journee()` au sein de la classe `CompetitionDivision` qui, tant que le nombre de journées planifiées n'a pas été atteint, récupère le classement de chaque division, pour créer les divisions de la prochaine journée de compétition en respectant la règle décrite dans l'énoncé. La solution proposée doit conserver l'historique des précédentes journées et la fonction `classement_journee(journee)`, dont vous fournirez le code Python, doit notamment permettre de récupérer le classement de la journée `journee`. Il est attendu que cette méthode et cette fonction déclenchent une exception `ValueError` si elles sont invoquées dans des conditions incorrectes (que vous expliciterez).

Question 31. Donnez le code Python d'une fonction `jouer_division(anciennes,nouvelles,taille,j)` qui crée une compétition de type division, à partir de la `taille` maximale des divisions en terme de nombre d'équipes, des équipes ordonnées `anciennes` qui étaient classées la saison précédentes et des équipes ordonnées `nouvelles` qui sont inscrites à partir de cette saison, puis organise `j` journées de compétition. La fonction retourne le podium de la compétition sous la forme d'une liste de 3 équipes ordonnées.

Question 32. Les notions de divisions et de poules sont-elles différentes ? Est-il possible de factoriser le code des classes associées ?

Partie V. Gestion d'un tournoi *hat*

Cette partie couvre enfin le cas particulier des tournois *hat* et de la gestion des joueuses/-eurs qui y participent en s'inscrivant individuellement. Pour s'inscrire à un tournoi *hat*, les joueuses/-eurs saisissent notamment leurs noms/prénoms, adresse, ville, niveau de jeu (évalué entre 1 et 10, 10 correspond à un niveau «excellent») et postes de prédilection (passeur, receveur, sans préférence) dans un formulaire en ligne, avant de s'acquitter des frais d'inscription. On considère que ces informations sont accessibles via une classe Python `Joueur`.

Question 33. Proposez une définition de la classe Python `Joueur` et une extension de la classe Python `Equipe`, nommée `EquipeHat`, pour faciliter le calcul d'indicateurs comme le niveau de jeu, le ratio des genres et la répartition des postes via des fonctions dont vous expliciterez la signature et le type de retour.

Question 34. Donnez le code de 4 tests unitaires qui permettent de s'assurer qu'une fonction Python `creer_equipes(joueurs,n)` constitue bien les `n` équipes d'un tournoi en équilibrant le niveau de jeu cumulé des joueurs, la répartition des postes de prédilection et les genres au sein des équipes.

Question 35. Donnez le code Python de la fonction `creer_equipes(joueurs,n)` qui retourne une liste de `n` instances de la classe `EquipeHat` qui se répartissent la liste des `joueurs` de manière équitable.

En amont de leur participation à des tournois *hat*, les participant(e)s peuvent communiquer pour organiser leurs trajets respectifs vers le lieu du tournoi. En particulier, il est généralement possible d'avoir recours à du co-voiturage pour transporter un maximum de joueuses/-eurs dans un minimum de véhicules. Pour ce faire, tout(e) joueuse/-eur inscrit(e) peut se déclarer *i)* en recherche d'un véhicule ou *ii)* disposant d'un véhicule, auquel cas elle/il indique le nombre de sièges disponibles. On considère qu'on dispose d'une matrice des plus courtes distances entre les `N+1` villes en lien avec le tournoi—i.e., la ville organisatrice et la liste des villes des joueuses/-eurs inscrits pour un co-voiturage. Les villes de départs sont celles où un(e) joueuse/-eur se déclare comme disposant d'un véhicule, tandis que la ville d'arrivée est celle où est organisée le tournoi.

On s'intéresse donc à établir un arbre couvrant de poids minimal dont la racine est la ville organisatrice et dont les feuilles sont les villes dans lesquelles des véhicules sont disponibles. On souhaite donc, à partir de la matrice de distance entre toutes les villes, du nombre de joueuses/-eurs à récupérer par ville, et de la liste des villes de départ, construire une liste ordonnée de villes à visiter pour chaque ville de départ.

Question 36. Quelle est la pré-condition à respecter pour pouvoir calculer un tel arbre couvrant de poids minimal ?

Par exemple, on considère un tournoi *hat*, organisé à Rennes, pour lequel les joueuses/-eurs inscrit(e)s habitent Paris, Lille, Lyon, Nantes, Bordeaux, Auxerre, Amiens.

Question 37. En partant du principe que les villes de départ sont Lille, Lyon, Nantes—avec des capacités respectives de 4, 3 et 6 places—et que les villes de Paris, Nantes, Bordeaux, Auxerre et Amiens recensent 2, 1, 3, 3 et 2 personnes à véhiculer, donnez la liste des co-voiturages à envisager pour permettre à tous les joueuses/-eurs de rejoindre le tournoi organisé à Rennes. Pour illustrer votre proposition, vous pouvez préciser le contenu de la matrice des distances avec des valeurs fictives qui vous serviront à justifier que votre solution minimise la distance totale parcourue.

Question 38. Expliquez pourquoi plusieurs co-voiturages sont possibles.

Pour construire la liste des co-voiturages possibles, on introduit la notion de véhicule dont la définition Python est donnée sous la forme d'une classe `Vehicule` :

```
class Vehicule:
    def __init__(self,conducteur,capacite,depart,distances):
        self._conducteur = conducteur
        self._capacite = capacite
        self._position = depart
        self._distances = distances
        self._distance = 0
        self.passagers = []

    def places_disponibles(self):
        return self._capacite - len(self.passagers)

    def distance_parcourue(self):
        return self._distance

    def ajouter_passager(self,passager):
        if self.places_disponibles() == 0:
            raise ValueError("Le véhicule est complet")
        self.passagers.append(passager)

    def visiter_ville(self,ville):
        if ville in self._distances:
            self._distance += self._distances[self._position][ville]
            self._position = ville
```

Question 39. Donnez le code Python de la fonction `lister_covoiturages(destination,distances,passagers,vehicules)` qui calcule une liste des co-voiturages à prévoir pour chaque conducteur déclaré. Le paramètre `vehicules` prend la forme d'un ensemble de tuples `(ville,listeVehicules)` qui indique, pour chaque ville de départ, la liste des véhicules disponibles. Le paramètre `passagers` prend la forme d'un ensemble de tuples `(ville,listeJoueurs)` qui indique, pour chaque ville à visiter, la liste des joueurs à récupérer. La fonction retourne une liste de `Vehicule`, mis à jour avec les joueurs qui doivent être récupérés, auxquels sont associés la liste ordonnée des villes à traverser. La fonction doit s'assurer que la capacité de chaque véhicule est respectée et que la distance parcourue par l'ensemble des véhicules est minimale.

Question 40. Donnez le diagramme de classes, avec la liste des attributs et la signature des méthodes (sans le code), qui permet de gérer les différentes formules de compétitions couvertes par le sujet (championnat, tournoi, division, *playoff* et *hat*), tout en faisant apparaître les autres concepts abordés dans le sujets. Vous prendrez soin d'indiquer les classes et méthodes abstraites, ainsi que les différentes relations entre les concepts représentés.

Question 41. Après avoir rappelé ce que sont les principes SOLID, vous illustrerez quel(s) principe(s) peuvent s'appliquer à la modélisation des compétitions que vous avez proposé.

Question 42. Dans quelle mesure la modélisation de ce système logiciel est dédié à la pratique de l'ultimate ? Précisez quelles sont les classes qui sont spécifiques à la gestion des règles de ce sport et quel mécanisme de conception pourrait être appliqué pour permettre de généraliser cette modélisation à la gestion de compétitions pour d'autres sports collectifs.

EAE INF 3

Fondements de l'informatique

Préliminaires généraux : Ce sujet porte sur le thème de la logique et s'intéresse plus particulièrement à la possibilité d'automatiser la recherche de preuves de formules logiques. Les travaux de Gentzen sur les systèmes formels de déduction se sont révélés fondamentaux en théorie de la démonstration. Nous nous intéressons ici à leur côté pratique pour l'automatisation du raisonnement. En particulier, l'aspect syntaxique des règles de déduction est un point positif quant à l'objectif d'un usage de la machine pour mener à bien les raisonnements.

Ce sujet se décompose en trois parties. La partie I justifie l'usage de systèmes déductifs formels via la démonstration du théorème de complétude de la déduction naturelle dans le cadre de la logique propositionnelle classique. La partie II porte sur l'implémentation d'un algorithme de recherche de preuve, toujours en logique propositionnelle classique, mais avec un système déductif plus adapté à la mécanisation que la déduction naturelle : le calcul des séquents. Enfin, la partie III porte sur le passage à la logique du premier ordre et l'usage de la méthode dite des tableaux pour la recherche de preuve. Les trois parties sont indépendantes, mais il est nécessaire de connaître le système du calcul des séquents, introduit en partie II, pour traiter la partie III.

Les figures et types OCaml de cet énoncé sont reproduits dans une annexe en fin de sujet afin de faciliter la composition.

Attendus : Les questions de programmation doivent être traitées en langage OCaml. On pourra utiliser toutes les fonctions du module `List`, ainsi que les fonctions de la bibliothèque standard (celles qui s'écrivent sans nom de module, comme `max`, `incr` ainsi que les opérateurs comme `@`). L'utilisation d'autres modules est interdite.

Dans l'écriture d'une fonction, on pourra faire appel à des fonctions définies dans les questions précédentes, même si ces dernières n'ont pas été traitées. On pourra également définir des fonctions auxiliaires, mais il faudra alors préciser leur rôle ainsi que le type et la signification de leurs arguments. Si les paramètres d'une fonction à coder sont supposés vérifier certaines hypothèses, il ne sera pas utile dans l'écriture de cette fonction de tester si les hypothèses sont bien satisfaites. Lorsque les choix d'implémentation ne découlent pas directement des spécifications de l'énoncé, il est conseillé de les expliquer.

On identifiera une même grandeur écrite dans deux polices de caractères différentes, en italique du point de vue mathématique (par exemple n) et en police à chasse fixe du point de vue informatique (par exemple `n`).

Sans précision supplémentaire, lorsqu'une question demande la complexité d'une fonction, il s'agira de la complexité temporelle dans le pire des cas. La complexité sera exprimée sous la forme $\mathcal{O}(f(n))$ où n est la taille de l'argument de l'algorithme, et f une expression la plus simple possible. Les calculs de complexité seront justifiés succinctement.

Il est attendu des candidates et des candidats des réponses construites. Leur évaluation portera aussi sur la précision, le soin et la clarté de la rédaction.

Partie I. Théorème de complétude

Dans cette partie, nous nous intéressons aux formules du calcul propositionnel. Nous supposons fixé un ensemble $\mathcal{P}$ dénombrable de variables propositionnelles. Nous considérons l'ensemble $\mathcal{F}_0$ des formules défini inductivement à partir des variables propositionnelles et à l'aide des connecteurs $\neg$ (négation), $\vee$ (disjonction), $\wedge$ (conjonction) et $\rightarrow$ (implication). Nous incluons une formule antilogique $\perp$ dans cette définition inductive, qui est rappelée en Figure 1a. Nous rappelons également les règles de la déduction naturelle pour le calcul propositionnel en logique classique en Figure 1b.

$$\frac{p \in \mathcal{P}}{p \in \mathcal{F}_0} \qquad \frac{}{\perp \in \mathcal{F}_0} \qquad \frac{A \in \mathcal{F}_0}{\neg A \in \mathcal{F}_0}$$

$$\frac{A \in \mathcal{F}_0 \qquad B \in \mathcal{F}_0}{A \wedge B \in \mathcal{F}_0} \qquad \frac{A \in \mathcal{F}_0 \qquad B \in \mathcal{F}_0}{A \vee B \in \mathcal{F}_0} \qquad \frac{A \in \mathcal{F}_0 \qquad B \in \mathcal{F}_0}{A \rightarrow B \in \mathcal{F}_0}$$

(a) Définition inductive des formules du calcul propositionnel

$$\frac{}{\Gamma, A \vdash A}\,ax \qquad \frac{\Gamma, \neg A \vdash \perp}{\Gamma \vdash A}\,RAA \qquad \frac{\Gamma, A \vdash \perp}{\Gamma \vdash \neg A}\,\neg_i \qquad \frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \perp}\,\neg_e$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,\wedge_i \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,\wedge_{e,g} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,\wedge_{e,d}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,\vee_{i,g} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,\vee_{i,d} \qquad \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\,\vee_e$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\,\rightarrow_i \qquad \frac{\Gamma \vdash A \rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\,\rightarrow_e$$

(b) Règles d'inférence de la déduction naturelle classique

FIGURE 1 – Formules et règles de la déduction naturelle pour le calcul propositionnel classique

Pour toute famille finie de formules $(A_i)_{i \in [\![0,n]\!]}$ et tout entier $k \in [\![0, n]\!]$, nous nous permettrons d'écrire $\bigwedge_{i=0}^{k} A_i$ pour désigner la formule définie par :

$$\bigwedge_{i=0}^{k} A_i = \begin{cases} A_0 \text{ si } k = 0 \\ \left(\bigwedge_{i=0}^{k-1} A_i\right) \wedge A_k \text{ sinon} \end{cases}.$$

Une définition similaire s'appliquera également au connecteur $\vee$, et si Γ est un ensemble fini de formules, nous nous permettrons d'écrire $\bigvee_{A \in \Gamma} A$ pour désigner $\bigvee_{i=0}^{n} A_i$, où $A_0, \ldots, A_n$ est une numérotation arbitraire des éléments de Γ. Par convention, nous définirons $\bigvee_{A \in \emptyset} A = \perp$.

Nous rappelons qu'une valuation est une fonction $v : \mathcal{P} \to \{V, F\}$ donnant une valeur de vérité à chaque variable propositionnelle, et nous noterons $[\![A]\!]_v$ l'évaluation de la formule A selon la valuation v. Nous noterons $\Gamma \vDash A$ le fait que la formule A est une conséquence sémantique de l'ensemble de formules Γ, i.e. pour toute valuation v, si $[\![B]\!]_v = V$ pour toute formule $B \in \Gamma$, alors $[\![A]\!]_v$. Ainsi, $\vDash A$ signifie que A est une tautologie. Enfin, l'équivalence sémantique entre deux formules A et B sera notée $A \equiv B$.

L'objectif de cette partie est de démontrer la complétude du système déductif présenté en Figure 1b, c'est-à-dire le théorème suivant.

Théorème 1 (Complétude)**.** *Soit $A \in \mathcal{F}_0$ une formule du calcul propositionnel.*

Si $\vDash A$, alors le séquent $\vdash A$ est dérivable.

I.1 Restriction de l'ensemble des formules

Nous allons commencer par démontrer certaines équivalences classiques, ce qui nous permettra de restreindre l'ensemble des formules possibles pour simplifier les démonstrations.

Question 1. Soit une formule $A \in \mathcal{F}_0$. On cherche une dérivation du séquent $\vdash \neg\neg A \leftrightarrow A$, où le connecteur $\leftrightarrow$ est défini comme suit : $A \leftrightarrow B = (A \to B) \wedge (B \to A)$.
En exploitant la règle appropriée pour se ramener aux dérivations des séquents $\vdash \neg\neg A \to A$ et $\vdash A \to \neg\neg A$, proposer une telle dérivation.
Indication : la règle RAA servira exactement une fois, dans la dérivation du séquent $\vdash \neg\neg A \to A$.

Question 2. Nous rappelons les lois de De Morgan : pour toutes formules $A, B \in \mathcal{F}_0$, on a

2.a) $\neg(A \vee B) \equiv \neg A \wedge \neg B$;

2.b) $\neg(A \wedge B) \equiv \neg A \vee \neg B$.

Donner une dérivation du séquent $\vdash \neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$, où $A, B \in \mathcal{F}_0$.

Question 3. Étant donné deux formules $A, B \in \mathcal{F}_0$, proposer une formule équivalente à $A \to B$ n'utilisant pas l'opérateur d'implication. À la manière des questions précédentes, donner un séquent représentant cette équivalence et le démontrer.

Les deux questions qui suivent donnent des outils permettant d'exploiter les équivalences précédentes dans le contexte de démonstrations.

Question 4. Soit deux formules $A, B \in \mathcal{F}_0$ telles que $\vdash A \leftrightarrow B$ est dérivable. Montrer que $\vdash A$ est dérivable si et seulement si $\vdash B$ l'est également.

Question 5. Démontrer le théorème d'affaiblissement, i.e. : si un séquent $\Gamma \vdash A$ est dérivable, alors pour tout ensemble de formules Δ le séquent $\Gamma, \Delta \vdash A$ est dérivable.

Nous admettons alors que pour toute formule $A \in \mathcal{F}_0$ il existe une formule $[A]_1 \in \mathcal{F}_1$ qui lui est équivalente sémantiquement et telle que le séquent $\vdash A \leftrightarrow [A]_1$ est dérivable, où l'ensemble de formules $\mathcal{F}_1$ est défini inductivement comme en Figure 2.

$$\frac{p \in \mathcal{P}}{p \in \mathcal{F}_1} \qquad \frac{}{\bot \in \mathcal{F}_1} \qquad \frac{A \in \mathcal{F}_1}{\neg A \in \mathcal{F}_1} \qquad \frac{A \in \mathcal{F}_1 \qquad B \in \mathcal{F}_1}{A \wedge B \in \mathcal{F}_1}$$

FIGURE 2 – Définition inductive de $\mathcal{F}_1$

Question 6. Montrer que l'on peut déduire le théorème de complétude du théorème de complétude restreint aux formules de $\mathcal{F}_1$.

Nous allons donc nous focaliser sur la démonstration du théorème de complétude restreint aux formules de $\mathcal{F}_1$.

I.2 Démonstration du théorème de complétude

L'ensemble $\mathcal{P}$ des variables propositionnelles étant dénombrable, nous nous donnons une énumération de ces variables : $\mathcal{P} = (p_n)_{n\in\mathbb{N}}$. Nous appelons alors valuation initiale de rang $n \in \mathbb{N}$ une valuation partielle donnant une valeur de vérité uniquement aux variables p_i pour $i \in [\![0, n]\!]$. Nous disons qu'une valuation initiale est compatible avec une formule si elle permet de l'évaluer, i.e. si elle donne une valeur de vérité à toutes les variables de la formule.

Question 7. On considère la formule $A = (p_0 \to (p_0 \to p_2)) \to p_2$. A est-elle une tautologie ? Si oui, le justifier. Sinon, proposer une valuation initiale compatible avec A, qui réfute A, et de rang minimal.

Question 8. Montrer qu'une formule $A \in \mathcal{F}_1$ est une tautologie si et seulement s'il existe un $n \in \mathbb{N}$ tel que toute valuation initiale v de rang n est compatible avec A et telle que $[\![A]\!]_v = V$.

Nous voulons exploiter les valuations initiales afin de montrer que toute tautologie est démontrable. Soit v une valuation initiale de rang $n \in \mathbb{N}$. Nous définissons une formule représentant v par $h_v = \bigwedge_{i=0}^{n} l(v, i)$, où pour tout $i \in [\![0, n]\!]$ le littéral $l(v, i)$ est défini par :

$$l(v, i) = \begin{cases} p_i \text{ si } v(p_i) = V \\ \neg p_i \text{ sinon} \end{cases} .$$

Question 9. Soit $A_0, \ldots, A_n \in \mathcal{F}_1$ et $i \in [\![0, n]\!]$. Montrer que $\bigwedge_{j=0}^{n} A_j \vdash A_i$ est dérivable.

Question 10. Soit $A \in \mathcal{F}_1$ et v une valuation initiale compatible avec A, dont on note n le rang. Montrer que

— si $[\![A]\!]_v = V$, alors le séquent $h_v \vdash A$ est dérivable ;

— si $[\![A]\!]_v = F$, alors le séquent $h_v \vdash \neg A$ est dérivable.

Nous voulons maintenant exploiter le résultat de la question 10 en démontrant par récurrence la propriété P_n : si pour toute valuation initiale v de rang n le séquent $h_v \vdash A$ est dérivable, alors $\vdash A$ est dérivable.

Question 11. Démontrer P_0.

Question 12. Soit $n \in \mathbb{N}$ tel que P_n est vraie. Démontrer P_{n+1}.

Question 13. Démontrer le théorème de complétude pour les formules de $\mathcal{F}_1$.

Partie II. Mécanisation de la recherche de preuve

Le théorème de complétude étant démontré, nous voulons maintenant confier la recherche de preuve à un ordinateur. Cependant, le système déductif présenté en Figure 1b présente plusieurs freins à cet usage :

— certaines règles, comme $\vee_e$ et $\rightarrow_e$, nécessitent de trouver une formule qui n'apparaît pas nécessairement dans le contexte initial. Trouver une telle formule de manière automatique ne paraît pas aisé.

— il est possible de « se tromper » en construisant un arbre de preuve, par exemple si l'on applique la règle $\vee_{i,g}$ alors que c'était le membre droit de la disjonction qui était démontrable.

Nous allons commencer par formaliser cette notion d'erreur possible dans l'application des règles. Un séquent $\Gamma \vdash A$ est dit valide si et seulement si $\Gamma \vDash A$. Une règle d'inférence est dite inversible si, dès lors que sa conclusion est un séquent valide, ses prémisses le sont aussi. Une règle non inversible est donc une règle pour laquelle il est possible lors de la recherche de preuve d'obtenir des prémisses non démontrables (car non valides) à partir d'une conclusion démontrable (car valide).

Question 14. Donner, en justifiant à l'aide d'exemples, les règles non inversibles de la déduction naturelle. On ne justifiera pas que les autres règles sont inversibles.

Pour résoudre ces problèmes techniques de la déduction naturelle, Gentzen a introduit un système déductif appelé calcul des séquents et qui s'appuie sur la notion de séquent symétrique. Un séquent symétrique est un séquent dont les deux membres sont des ensembles de formules. Nous écrirons donc un séquent symétrique sous la forme $\Gamma \vdash \Delta$ où Γ et Δ sont tous deux des ensembles de formules, contrairement aux séquents asymétriques, de la forme $\Gamma \vdash A$ avec A une formule unique. Le séquent symétrique $\Gamma \vdash \Delta$ doit être interprété de la manière suivante : de la conjonction des formules de Γ, on peut déduire la disjonction des formules de Δ. Les règles du calcul des séquents classique sont présentées en Figure 3.

$$\frac{}{\Gamma, A \vdash A, \Delta}\,ax \qquad \frac{}{\Gamma, \bot \vdash \Delta}\,\bot_g$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\,\neg_g \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\,\neg_d$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,\wedge_g \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\,\wedge_d$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\,\vee_g \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}\,\vee_d$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}\,\to_g \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}\,\to_d$$

FIGURE 3 – Règles d'inférence du calcul des séquents classique

Bien que le système déductif de la Figure 3 ne contienne aucune règle telle que la règle RAA de la déduction naturelle pour le raisonnement par l'absurde, c'est bien un système pour la logique classique, comme le démontre la question suivante.

Question 15. Donner une dérivation du tiers exclu : $\vdash A \vee \neg A$.

Nous admettons que le calcul des séquents est équivalent à la déduction naturelle, c'est-à-dire que :

— un séquent $\Gamma \vdash A$ est dérivable en déduction naturelle si et seulement s'il l'est également en calcul des séquents ;

— un séquent $\Gamma \vdash \Delta$ est dérivable en calcul des séquents si et seulement si le séquent $\Gamma \vdash \bigvee_{A \in \Delta} A$ l'est également en déduction naturelle.

Afin d'observer plus concrètement cette équivalence, et de constater le côté mécanique de la recherche de preuve en calcul des séquents, nous pouvons redémontrer des séquents démontrés précédemment en déduction naturelle.

Question 16. Donner des dérivations des séquents des questions 1 et 2 en calcul des séquents, à savoir :

16.a) $\vdash \neg\neg A \leftrightarrow A$

16.b) $\vdash \neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$.

Pour simplifier notre étude du calcul des séquents, nous nous limiterons aux formules de l'ensemble $\mathcal{F}_1$ défini en Figure 2, ce qui nous permettra de nous passer des règles pour la disjonction et l'implication ($\vee_g$, $\vee_d$, $\to_g$ et $\to_d$).

Nous étendons la notion de séquent valide aux séquents symétriques de la manière suivante : $\Gamma \vdash \Delta$ est valide si et seulement si $\Gamma \vDash \bigvee_{A \in \Delta} A$, ce que nous noterons également $\Gamma \vDash \Delta$. Nous

admettons que le calcul des séquents est un système déductif correct et complet, i.e. qu'un séquent $\Gamma \vdash \Delta$ est dérivable si et seulement s'il est valide.

Nous pouvons maintenant démontrer qu'il n'est plus possible de se tromper lors de la recherche de preuve.

Question 17. Démontrer que les règles du calcul des séquents sont inversibles.

Nous souhaitons maintenant implémenter un algorithme de recherche de preuve en calcul des séquents. Pour ce faire, nous commençons par nous donner un type pour les formules.

```
type 'a formule =
  | Faux
  | Var of 'a
  | Non of 'a formule
  | Et of 'a formule * 'a formule
```

Nous programmons tout d'abord une fonction utilitaire.

Question 18. Écrire une fonction `est_atomique : 'a formule -> bool` qui détermine si une formule est atomique, i.e. si elle ne contient aucun connecteur $\neg$ ou $\wedge$.

Nous nous donnons ensuite un type représentant les séquents : nous utilisons un enregistrement contenant une liste `hyp` de formules qui constituent les hypothèses du séquent et une liste `concl` de formules qui constituent sa conclusion.

```
type 'a sequent =
  {hyp : 'a formule list; concl : 'a formule list}
```

Nous rappelons que la construction d'un élément de ce type dont les champs valent l_1 et l_2 se fait à l'aide de la syntaxe `{hyp =` l_1`; concl =` l_2`}` et que si `s` est un élément de ce type, on peut lire ses champs à l'aide des expressions `s.hyp` et `s.concl`.

Nous supposerons disposer des deux fonctions utilitaires suivantes sur les listes :

— une fonction `extrait : 'a -> 'a list -> 'a list` telle que `extrait x l` supprime la première occurrence de `x` dans `l` s'il en existe une et lève l'exception `Not_found` sinon ;

— une fonction `element_commun : 'a list -> 'a list -> 'a option` qui renvoie un élément commun aux deux listes passées en argument, sous la forme `Some x`, s'il en existe un, et `None` sinon.

Nous appelons formule principale d'une règle d'inférence la formule située dans la conclusion de cette règle et qui en permet l'instanciation. Le tableau suivant indique pour chaque règle de la figure 3 quelle est sa formule principale.

Règle	Formule principale
ax	A (à gauche et à droite de $\vdash$)
$\bot_g$	$\bot$
$\neg_g, \neg_d$	$\neg A$
$\wedge_g, \wedge_d$	$A \wedge B$
$\vee_g, \vee_d$	$A \vee B$
$\rightarrow_g, \rightarrow_d$	$A \rightarrow B$

Nous représentons enfin une dérivation de la manière suivante : nous utilisons un type somme dont chaque constructeur correspond à une règle d'inférence. Chaque constructeur est paramétré par la formule principale de la règle d'inférence, ainsi que par des dérivations de ses prémisses. Nous faisons une exception pour le constructeur `Faux_g`, car l'unique formule permettant d'instancier la règle $\bot_g$ est $\bot$.

```
type 'a derivation =
  | Axiome of 'a formule
  | Faux_g
  | Non_g of 'a formule * 'a derivation
  | Non_d of 'a formule * 'a derivation
  | Et_g of 'a formule * 'a derivation
  | Et_d of 'a formule * 'a derivation * 'a derivation
```

Par exemple, l'expression `Faux_g` représente n'importe quelle dérivation de la forme

$$\frac{}{\Gamma, \bot \vdash \Delta}\bot_g$$

où Γ et Δ sont des ensembles de formules quelconques, tandis que l'expression

```
Et_d (Et (Var "p", Var "q"),
      Axiome (Var "p"),
      Non_d (Non (Var "q"), Axiome (Var "q")))
```

représente les dérivations de la forme

$$\frac{\frac{}{\Gamma, p \vdash p, \neg q, \Delta}ax \qquad \frac{\frac{}{\Gamma, p, q \vdash q, \Delta}ax}{\Gamma, p \vdash q, \neg q, \Delta}\neg_d}{\Gamma, p \vdash p \wedge q, \neg q, \Delta}\wedge_d$$

Une expression du type `'a derivation` peut représenter plusieurs dérivations différentes, puisqu'il peut exister dans le séquent dérivé des formules dont une telle expression ne fait pas mention (celles des ensembles Γ et Δ dans les exemples ci-avant). Nous nous proposons alors de vérifier si un séquent donné admet bien une dérivation représentée par une telle expression.

Question 19. Écrire une fonction `est_valide : 'a derivation -> 'a sequent -> bool` qui détermine si une dérivation est un arbre de preuve valide pour un séquent donné. On veillera à écrire une fonction de complexité $\mathcal{O}(nm)$, où n est la taille de la dérivation et m la taille du séquent.

Nous proposons enfin l'algorithme récursif suivant, qui construit une dérivation du séquent $\Gamma \vdash \Delta$ passé en argument s'il en existe une et provoque une erreur sinon :

1. S'il existe dans Γ une formule A non atomique, appliquer la règle correspondant au connecteur logique principal de A, puis construire récursivement une dérivation des prémisses de cette règle.
2. Sinon, s'il existe dans Δ une formule non atomique, procéder de même en appliquant la règle correspondante et en démontrant récursivement les prémisses de cette règle.
3. Sinon, appliquer l'une des deux règles $\bot_g$ et ax si possible et échouer sinon.

Question 20. Écrire une fonction `prouve : 'a sequent -> 'a derivation` qui implémente cet algorithme de recherche de preuve. L'échec de l'algorithme sera signifié par la levée d'une exception.

Question 21. Démontrer la terminaison de cet algorithme.

Question 22. Démontrer la correction de cet algorithme, c'est-à-dire qu'il parvient bien à construire une dérivation du séquent $\Gamma \vdash \Delta$ en entrée s'il est dérivable et qu'il échoue sinon.

Question 23. Donner un majorant de la complexité en termes de nombre d'appels récursifs de cet algorithme et montrer que cette borne est atteinte.

Partie III. Passage à la logique du premier ordre

Nous souhaitons maintenant étudier la mécanisation de la recherche de preuve en logique du premier ordre. Nous fixons donc un langage du premier ordre $\mathcal{L} = (\mathcal{S}_F, \mathcal{S}_R)$, où $\mathcal{S}_F$ est l'ensemble des symboles de fonction et $\mathcal{S}_R$ est l'ensemble des symboles de relation. Nous rappelons qu'un symbole de fonction d'arité 0 est parfois appelé constante et qu'un symbole de relation d'arité 0 est parfois appelé constante propositionnelle. Nous fixons un ensemble dénombrable $\mathcal{V}$ de variables, puis nous définissons inductivement l'ensemble $\mathcal{T}$ des termes en Figure 4a, et l'ensemble $\mathcal{F}$ des formules de la logique du premier ordre en Figure 4b.

Dans la suite, nous noterons $FV(t)$ l'ensemble des variables (libres) d'un terme $t \in \mathcal{T}$, notation que nous étendrons aux formules ($FV(A)$) et aux ensembles de formules ($FV(\Gamma)$). De même, nous noterons $t[\sigma]$ le résultat de l'application de la substitution σ aux variables de t et nous étendrons également cette notation aux formules et aux ensembles de formules. Nous noterons $x := t$ la substitution σ telle que $\sigma(x) = t$ et $\sigma(y) = y$ pour tout $y \neq x$.

Nous rappelons que, dans le calcul de $A[x := t]$, si une variable de t apparaît liée dans A, alors il est nécessaire de renommer cette variable liée avant d'appliquer la substitution afin d'éviter le phénomène de capture de variable. De manière générale, deux formules égales au renommage

$$\frac{x \in \mathcal{V}}{x \in \mathcal{T}} \qquad \frac{f \in \mathcal{S}_F \qquad f \text{ d'arité } n \qquad t_1, \ldots, t_n \in \mathcal{T}}{f(t_1, \ldots, t_n) \in \mathcal{T}}$$

(a) Définition inductive de l'ensemble des termes

$$\frac{R \in \mathcal{S}_R \qquad R \text{ d'arité } n \qquad t_1, \ldots, t_n \in \mathcal{T}}{R(t_1, \ldots, t_n) \in \mathcal{F}} \qquad \frac{}{\bot \in \mathcal{F}} \qquad \frac{A \in \mathcal{F}}{\neg A \in \mathcal{F}}$$

$$\frac{A \in \mathcal{F} \qquad B \in \mathcal{F}}{A \wedge B \in \mathcal{F}} \qquad \frac{A \in \mathcal{F} \qquad B \in \mathcal{F}}{A \vee B \in \mathcal{F}} \qquad \frac{A \in \mathcal{F} \qquad B \in \mathcal{F}}{A \to B \in \mathcal{F}}$$

$$\frac{A \in \mathcal{F} \qquad x \in \mathcal{V}}{\forall x\ A \in \mathcal{F}} \qquad \frac{A \in \mathcal{F} \qquad x \in \mathcal{V}}{\exists x\ A \in \mathcal{F}}$$

(b) Définition inductive des formules de la logique du premier ordre

FIGURE 4 – Termes et formules de la logique du premier ordre

près de leurs variables liées seront considérées identiques. Par exemple, si $P \in \mathcal{S}_R$ est d'arité 2, alors le calcul de $(\forall x\ P(x, y))\,[y := x]$ peut passer par le renommage de x en z, ce qui donne

$$(\forall x\ P(x, y))\,[y := x] = (\forall z\ P(z, y))\,[y := x] = \forall z\ P(z, x).$$

III.1 Calcul des séquents pour la logique du premier ordre

Nous ajoutons les règles de la figure 5 aux règles de la figure 3 afin de définir le système déductif du calcul des séquents pour la logique du premier ordre.

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\mathit{contr}_g \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\mathit{contr}_d$$

$$\frac{\Gamma, A[x := t] \vdash \Delta}{\Gamma, \forall x\ A \vdash \Delta}\forall_g \qquad \frac{\Gamma \vdash A, \Delta \qquad x \notin FV(\Gamma, \Delta)}{\Gamma \vdash \forall x\ A, \Delta}\forall_d$$

$$\frac{\Gamma, A \vdash \Delta \qquad x \notin FV(\Gamma, \Delta)}{\Gamma, \exists x\ A \vdash \Delta}\exists_g \qquad \frac{\Gamma \vdash A[x := t], \Delta}{\Gamma \vdash \exists x\ A, \Delta}\exists_d$$

FIGURE 5 – Règles supplémentaires du calcul des séquents pour la logique du premier ordre

Question 24. Proposer une dérivation du séquent $\vdash \exists x\ \forall y\ (R(y) \to R(x))$, où $R \in \mathcal{S}_R$ est d'arité 1. On pourra utiliser une règle de contraction contr_d.

Question 25. Justifier qu'il est impossible de dériver le séquent $\vdash \exists x\ \forall y\ (R(y) \to R(x))$ sans utiliser de règle de contraction.

En fait, les contractions ne sont utiles que pour l'usage des règles des quantificateurs et nous admettrons qu'il est possible de remplacer les quatre règles $\forall_g$, $\exists_d$, $contr_g$ et $contr_d$ par les deux règles suivantes :

$$\frac{\Gamma, A[x := t], \{\forall x\ A\} \vdash \Delta}{\Gamma, \forall x\ A \vdash \Delta}\forall_g^c \qquad \frac{\Gamma \vdash A[x := t], \{\exists x\ A\}, \Delta}{\Gamma \vdash \exists x\ A, \Delta}\exists_d^c$$

Dans les règles ci-avant, une formule est mise entre accolades si l'on peut choisir de ne pas la conserver dans une instance de la règle. Par exemple, la règle $\forall_g^c$ admet les deux instances suivantes :

$$\frac{\Gamma, A[x := t] \vdash \Delta}{\Gamma, \forall x\ A \vdash \Delta} \qquad \frac{\Gamma, A[x := t], \forall x\ A \vdash \Delta}{\Gamma, \forall x\ A \vdash \Delta}$$

Ainsi, dans toute la suite, un séquent sera dit dérivable en calcul des séquents s'il est dérivable dans le système déductif constitué des règles de la Figure 3 et de la Figure 6.

$$\frac{\Gamma, A[x := t], \{\forall x\ A\} \vdash \Delta}{\Gamma, \forall x\ A \vdash \Delta}\forall_g^c \qquad \frac{\Gamma \vdash A, \Delta \qquad x \notin FV(\Gamma, \Delta)}{\Gamma \vdash \forall x\ A, \Delta}\forall_d$$

$$\frac{\Gamma, A \vdash \Delta \qquad x \notin FV(\Gamma, \Delta)}{\Gamma, \exists x\ A \vdash \Delta}\exists_g \qquad \frac{\Gamma \vdash A[x := t], \{\exists x\ A\}, \Delta}{\Gamma \vdash \exists x\ A, \Delta}\exists_d^c$$

FIGURE 6 – Nouvelles règles pour les quantificateurs

Le système obtenu n'est cependant pas très pratique dans le cadre de l'automatisation de la recherche de preuve pour deux raisons : les nouvelles règle nécessitent de décider si l'on conserve la formule avec son quantificateur via une contraction, et il est nécessaire de choisir un terme pour la substitution. Le principe de la méthode des tableaux est de retarder le choix de ce terme jusqu'à l'usage des axiomes pour clore les branches de l'arbre de dérivation, afin d'exploiter les informations supplémentaires obtenues au cours de la dérivation. L'idée est de choisir la substitution de sorte à égaliser des formules à gauche et à droite des séquents afin d'obtenir des instances de la règle ax. Sa réalisation repose sur un outil appelé unification.

III.2 Unification

Pour simplifier, nous limiterons notre étude dans cette section à l'unification de termes (la généralisation aux formules est assez directe). Le principe de l'unification est alors d'égaliser des termes à l'aide d'une substitution. Nous disons que deux termes $t, u \in \mathcal{T}$ sont unifiables si et seulement s'il existe une substitution σ telle que $t[\sigma] = u[\sigma]$. Une telle substitution est appelée

unificateur pour t et u. Plus généralement, puisque nous voulons égaliser des formules atomiques pouvant contenir plusieurs termes, nous nous intéresserons au problème de l'unification simultanée de plusieurs couples de termes. Nous considérons alors le problème de décision suivant, que nous appellons UNIF : étant donné une liste $L = [t_i \stackrel{?}{=} u_i, i \in [\![1, n]\!]]$ d'équations entre termes, existe-t-il une substitution σ telle que pour tout $i \in [\![1, n]\!]$, $t_i[\sigma] = u_i[\sigma]$? Nous dirons encore qu'une telle substitution est un unificateur pour L.

Question 26.

26.a) Montrer que les termes $t = f(x, y)$ et $u = f(f(y, z), a)$, où $a \in \mathcal{S}_F$ est une constante et $f \in \mathcal{S}_F$ est d'arité 2, sont unifiables.

26.b) Que peut-on dire de la réponse au problème UNIF si l'on considère l'instance

$$L = [f(x, y) \stackrel{?}{=} f(f(y, z), a), f(z, y) \stackrel{?}{=} f(f(y, z), a)]?$$

Nous voulons maintenant démontrer que le problème UNIF est décidable. Nous nous proposons d'étudier l'algorithme d'unification de Martelli et Montanari. À partir de la liste L d'équations, cet algorithme construit une suite L_n de listes et une suite σ_n de substitutions de la manière suivante :

- $L_0 = L$ et σ_0 est la fonction identité ;
- L_{n+1} et σ_{n+1} sont calculés comme suit :
  - Si $L_n = [\ ]$: l'algorithme se termine et σ_n est un unificateur pour L.
  - Si $L_n = (f(t_1, \ldots, t_p) \stackrel{?}{=} g(u_1, \ldots, u_q)) :: L'$ avec $f, g \in \mathcal{S}_F$ et $t_1, \ldots, t_p, u_1, \ldots, u_q \in \mathcal{T}$:
    - si $f \neq g$, alors l'algorithme échoue : il n'y a pas d'unificateur pour L ;
    - si $f = g$ (et donc $p = q$), alors $L_{n+1} = [t_1 \stackrel{?}{=} u_1, \ldots, t_p \stackrel{?}{=} u_p]@L'$ et $\sigma_{n+1} = \sigma_n$.
  - Si $L_n = (x \stackrel{?}{=} x) :: L'$ avec $x \in \mathcal{V}$: $L_{n+1} = L'$ et $\sigma_{n+1} = \sigma_n$;
  - Si $L_n = (x \stackrel{?}{=} t) :: L'$ avec $x \in \mathcal{V}$ et $t \in \mathcal{T} \setminus \{x\}$, à symétrie de l'équation près :
    - si $x \in FV(t)$, alors l'algorithme échoue : il n'y a pas d'unificateur pour L ;
    - sinon, $\sigma_{n+1} = (x := t) \circ \sigma_n$ et $L_{n+1} = L'[x := t]$, où $[t_i \stackrel{?}{=} u_i, i \in [\![1, n]\!]][x := t] = [t_i[x := t] \stackrel{?}{=} u_i[x := t], i \in [\![1, n]\!]]$.

Question 27. Appliquer cet algorithme aux listes suivantes :

27.a) la liste L de la question **26.b)** ;

27.b) $\left[S(f(x, S(y))) \stackrel{?}{=} S(f(S(t), z))\right]$, où $S \in \mathcal{S}_F$ est d'arité 1.

Nous nous proposons d'implémenter cet algorithme avant d'en étudier les propriétés. Nous définissons le type des termes de la manière suivante :

```
type ('a, 'b) terme =
  | Var of 'a
  | App of 'b * ('a, 'b) terme list
```

Ainsi, un terme est soit une variable, représentée par un élément du type `'a`, soit l'application d'un symbole de fonction, représenté par un élément du type `'b`, à une liste de termes. Par exemple, l'expression `Var "x"` pourra représenter la variable x et le terme $f(x, y)$ pourra être représenté par `App ("f", [Var "x"; Var "y"])`.

Nous représentons alors une substitution comme une liste associant des termes à des éléments du type `'a` et une instance du problème UNIF comme une liste de couple de termes à unifier.

```
type ('a, 'b) subst = ('a * ('a, 'b) terme) list

type ('a, 'b) instance = (('a, 'b) terme * ('a, 'b) terme) list
```

Nous commençons par implémenter deux fonctions utilitaires.

Question 28. Écrire des fonctions :

28.a) `est_variable : 'a -> ('a, 'b) terme -> bool` telle que `est_variable x t` calcule le booléen $x \in FV(t)$;

28.b) `substitue : 'a -> ('a, 'b) terme -> ('a, 'b) terme -> ('a, 'b) terme` telle que `substitue x u t` calcule le terme $t[x := u]$.

Ces deux fonctions devront être de complexité linéaire en la taille du terme t passé en paramètre.

Nous pouvons alors implémenter l'algorithme d'unification.

Question 29. Écrire une fonction `unifie : ('a, 'b) instance -> ('a, 'b) subst` qui implémente l'algorithme de Martelli et Montanari. Cette fonction lèvera une exception dans les cas d'échec de l'algorithme.

Question 30. Montrer que l'algorithme d'unification termine toujours, soit en signalant l'échec de l'unification, soit avec $L_n = [\ ]$.

Question 31. Soit $(x_n)_{n\in\mathbb{N}} \in \mathcal{V}^{\mathbb{N}}$ une suite de variables deux à deux distinctes et $(t_n)_{n\in\mathbb{N}}, (u_n)_{n\in\mathbb{N}} \in \mathcal{T}^{\mathbb{N}}$ les suites de termes définies par :

— $t_0 = u_0 = x_0$;

— pour tout $n \in \mathbb{N}$, $t_{n+1} = f(t_n, x_n)$ et $u_{n+1} = f(x_n, u_n)$.

Montrer que l'algorithme d'unification appliqué à l'équation $t_n \stackrel{?}{=} u_n$ produit une substitution σ telle que la taille de $t_n[\sigma]$ soit en $\Theta(2^n)$. En déduire une instance telle que l'algorithme d'unification est de complexité exponentielle en la taille de son entrée.

Question 32. Démontrer que pour tout $n \in \mathbb{N}$ tel que L_n est définie et pour toute substitution σ, σ est un unificateur pour L si et seulement s'il existe une substitution σ' qui est un unificateur pour L_n et telle que $\sigma = \sigma' \circ \sigma_n$.

Question 33. Conclure quant à la correction de l'algorithme d'unification.

III.3 Méthode des tableaux

Nous revenons maintenant à la recherche de preuve en calcul des séquents. Comme énoncé précédemment, le principe de la méthode des tableaux est de retarder l'instanciation des variables dans les règles $\forall_g^c$ et $\exists_d^c$ afin d'appliquer un algorithme d'unification à l'ensemble des feuilles de l'arbre de dérivation pour vérifier s'il est possible d'utiliser la règle ax au niveau de ces feuilles.

Cela nécessite toutefois d'être prudent. En effet, une même formule quantifiée peut être exploitée plusieurs fois via les règles $\forall_g^c$ et $\exists_d^c$, via la contraction intégrée à ces règles. Ainsi, si l'on veut pouvoir faire plusieurs instanciations avec des termes différents, il ne faut pas conserver la variable associée au quantificateur que l'on retire dans cette règle, mais la remplacer par une nouvelle variable n'apparaissant pas ailleurs dans l'arbre de dérivation. Cela revient donc à utiliser les deux règles suivantes à la place des règles $\forall_g^c$ et $\exists_d^c$:

$$\frac{\Gamma, A[x := ?x], \{\forall x\ A\} \vdash \Delta}{\Gamma, \forall x\ A \vdash \Delta}\forall_g^d \qquad \frac{\Gamma \vdash A[x := ?x], \{\exists x\ A\}, \Delta}{\Gamma \vdash \exists x\ A, \Delta}\exists_d^d$$

où $?x$ est une variable en attente d'instanciation, appelée métavariable, et qui ne doit apparaître nulle part ailleurs dans la dérivation en cours de construction.

Par exemple, si $?x$ et $?y$ sont deux noms différents, il sera possible d'enchaîner deux applications de la règle $\forall_g^d$ portant sur la même formule $\forall x\ A$ de la façon suivante :

$$\cfrac{\cfrac{\Gamma, A[x := ?x], A[x := ?y] \vdash \Delta}{\Gamma, A[x := ?x], \forall x\ A \vdash \Delta}}{\Gamma, \forall x\ A \vdash \Delta}$$

La méthode des tableaux se résume alors en ces deux étapes :

1. appliquer des règles d'inférences jusqu'à l'obtention de séquents ne contenant que des formules atomiques dans toutes les branches (nous mettons de côté le problème de la contraction dans les règles $\forall_g^d$ et $\exists_d^d$, que nous évoquerons plus tard), en fermant les branches avec la règle $\bot_g$ si possible ;
2. utiliser un algorithme d'unification pour essayer d'obtenir une substitution σ ne s'appliquant qu'aux métavariables et telle que, pour tout séquent $\Gamma \vdash \Delta$ dans les branches encore ouvertes de l'arbre obtenu à l'étape précédente, la règle ax peut être appliquée au séquent $\Gamma[\sigma] \vdash \Delta[\sigma]$. En cas de succès, le séquent initial est considéré comme démontré. Sinon, il n'est pas démontrable.

Question 34. Appliquer la méthode des tableaux au séquent $\exists y\ \forall x\ P(x, y) \vdash \forall x\ \exists y\ P(x, y)$.

Cependant, avec les règles d'inférence dont nous disposons, la méthode des tableaux peut valider un séquent qui n'est pas démontrable.

Question 35. Appliquer la méthode des tableaux au séquent $\forall x\ \exists y\ P(x, y) \vdash \exists y\ \forall x\ P(x, y)$. Pourquoi la substitution produite est-elle invalide ?

Pour résoudre ce problème, nous proposons d'utiliser une technique appelée skolémisation et dont le principe est le suivant : dans une règle $\forall_d$ ou $\exists_g$, au lieu de vérifier une contrainte supplémentaire sur la variable quantifiée, nous encodons sa possible dépendance aux autres variables libres de la formule en lui substituant un terme dépendant de ces variables, c'est-à-dire que nous remplaçons les règles $\forall_d$ et $\exists_g$ par les règles suivantes :

$$\frac{\Gamma \vdash A[x := f(?x_1, \ldots, ?x_n)], \Delta}{\Gamma \vdash \forall x\ A, \Delta}\forall_d^s \qquad \frac{\Gamma, A[x := f(?x_1, \ldots, ?x_n)] \vdash \Delta}{\Gamma, \exists x\ A \vdash \Delta}\exists_g^s$$

où $?x_1, \ldots, ?x_n$ sont les métavariables de $\forall x\ A$ (ou $\exists x\ A$ dans $\exists_g^s$) et f est un symbole de fonction qui n'apparaît nulle part ailleurs dans la dérivation en cours de construction. S'il n'y a pas de métavariables, le terme $f(?x_1, \ldots, ?x_n)$ est bien-sûr remplacé par une constante c qui répond à la même contrainte que f.

Ainsi, les règles de dérivation utilisées dans la méthode des tableaux sont les règles de la Figure 3 et celles de la Figure 7.

$$\frac{\Gamma, A[x := ?x], \{\forall x\ A\} \vdash \Delta}{\Gamma, \forall x\ A \vdash \Delta}\forall_g^d \qquad \frac{\Gamma \vdash A[x := f(?x_1, \ldots, ?x_n)], \Delta}{\Gamma \vdash \forall x\ A, \Delta}\forall_d^s$$

$$\frac{\Gamma, A[x := f(?x_1, \ldots, ?x_n)] \vdash \Delta}{\Gamma, \exists x\ A \vdash \Delta}\exists_g^s \qquad \frac{\Gamma \vdash A[x := ?x], \{\exists x\ A\}, \Delta}{\Gamma \vdash \exists x\ A, \Delta}\exists_d^d$$

FIGURE 7 – Règles des quantificateurs pour la méthode des tableaux

Question 36. Reprendre les questions 34 et 35 avec ces nouvelles règles. Le problème de la question 35 est-il résolu ?

Question 37. En supposant que l'on veuille utiliser l'algorithme de Martelli et Montanari pour l'unification, expliquer comment réaliser la seconde étape de la méthode des tableaux. Donner l'ordre de grandeur du nombre d'appels à l'algorithme de Martelli et Montanari en fonction de la taille du séquent initial, dans le pire cas si l'on n'utilise pas la contraction dans les règles $\forall_g^d$ et $\exists_d^d$.

Nous voulons maintenant démontrer la complétude de la méthode des tableaux : si un séquent est prouvable en calcul des séquents, alors la méthode des tableaux appliquée à ce séquent réussit. Nous nous limiterons pour cela, suivant le même principe qu'en section 1, à des formules n'utilisant pas les connecteurs $\vee$, $\rightarrow$ et $\exists$.

Question 38. Montrer que si un séquent $\Gamma \vdash \Delta$ admet une dérivation, alors pour toute substitution σ, le séquent $\Gamma[\sigma] \vdash \Delta[\sigma]$ est dérivable avec une dérivation de même taille.

Question 39. Soit $\Gamma \vdash \Delta$ un séquent et σ une substitution telle que $\Gamma[\sigma] \vdash \Delta[\sigma]$ est dérivable en calcul des séquents. Montrer que pour tout ensemble fini E de formules, il existe une exécution réussie de la méthode des tableaux appliquée à $\Gamma \vdash \Delta$ et que parmi toutes les substitutions possibles pour l'unification dans les feuilles de l'arbre construit lors de cette exécution, il en existe une de la forme $\sigma \circ \tau$, où τ est une substitution laissant invariantes les formules de E.

Question 40. Conclure quant à la complétude de la méthode des tableaux.

Pour résoudre le problème de l'usage ou non de la contraction dans les règles $\forall_g^d$ et $\exists_d^d$, une technique classique consiste à décorer les quantificateurs avec des entiers strictement positifs : un quantificateur Q^n, où $Q \in \{\forall, \exists\}$ signifie que les $n-1$ premiers usages de la règle associée à ce quantificateur se feront avec contraction et que le suivant se fera sans contraction. Par exemple, la règle $\forall_g^d$ est remplacée par les deux règles suivantes :

$$\frac{\Gamma, A[x := ?x], \forall^{n-1} x\ A \vdash \Delta \qquad n > 1}{\Gamma, \forall^n x\ A \vdash \Delta} \qquad\qquad \frac{\Gamma, A[x := ?x] \vdash \Delta}{\Gamma, \forall^1 x\ A \vdash \Delta}$$

Une idée simple pour la recherche de preuve avec la méthode des tableaux consiste alors à faire une recherche avec tous les quantificateurs décorés par l'entier 1 et, en cas d'échec, à recommencer en incrémentant les décorations. Il est évidemment possible d'utiliser des approches plus subtiles afin de gagner en efficacité.

* *
*

EAE INF 3

Annexe : formules et règles de la déduction naturelle pour le calcul propositionnel classique

$$\frac{p \in \mathcal{P}}{p \in \mathcal{F}_0} \qquad \frac{}{\bot \in \mathcal{F}_0} \qquad \frac{A \in \mathcal{F}_0}{\neg A \in \mathcal{F}_0}$$

$$\frac{A \in \mathcal{F}_0 \qquad B \in \mathcal{F}_0}{A \wedge B \in \mathcal{F}_0} \qquad \frac{A \in \mathcal{F}_0 \qquad B \in \mathcal{F}_0}{A \vee B \in \mathcal{F}_0} \qquad \frac{A \in \mathcal{F}_0 \qquad B \in \mathcal{F}_0}{A \to B \in \mathcal{F}_0}$$

(a) Définition inductive des formules du calcul propositionnel

$$\frac{}{\Gamma, A \vdash A}\,ax \qquad \frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\,RAA \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\,\neg_i \qquad \frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\,\neg_e$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,\wedge_i \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,\wedge_{e,g} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,\wedge_{e,d}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,\vee_{i,g} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,\vee_{i,d} \qquad \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\,\vee_e$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\,\to_i \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\,\to_e$$

(b) Règles d'inférence de la déduction naturelle classique

FIGURE 1 – Formules et règles de la déduction naturelle pour le calcul propositionnel classique

$$\frac{p \in \mathcal{P}}{p \in \mathcal{F}_1} \qquad \frac{}{\bot \in \mathcal{F}_1} \qquad \frac{A \in \mathcal{F}_1}{\neg A \in \mathcal{F}_1} \qquad \frac{A \in \mathcal{F}_1 \qquad B \in \mathcal{F}_1}{A \wedge B \in \mathcal{F}_1}$$

FIGURE 2 – Définition inductive de $\mathcal{F}_1$

Annexe : règles du calcul des séquent pour le calcul propositionnel classique et types OCaml associés

$$\frac{}{\Gamma, A \vdash A, \Delta}\,ax \qquad \frac{}{\Gamma, \bot \vdash \Delta}\,\bot_g$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\,\neg_g \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\,\neg_d$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,\wedge_g \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\,\wedge_d$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\,\vee_g \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}\,\vee_d$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}\,\to_g \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}\,\to_d$$

FIGURE 3 – Règles d'inférence du calcul des séquents classique

```
type 'a formule =
  | Faux
  | Var of 'a
  | Non of 'a formule
  | Et of 'a formule * 'a formule

type 'a sequent =
  {hyp : 'a formule list; concl : 'a formule list}

type 'a derivation =
  | Axiome of 'a formule
  | Faux_g
  | Non_g of 'a formule * 'a derivation
  | Non_d of 'a formule * 'a derivation
  | Et_g of 'a formule * 'a derivation
  | Et_d of 'a formule * 'a derivation * 'a derivation
```

Annexe : termes, formules et règles du calcul des séquents pour la logique du premier ordre

$$\frac{x \in \mathcal{V}}{x \in \mathcal{T}} \qquad \frac{f \in \mathcal{S}_F \qquad f \text{ d'arité } n \qquad t_1, \ldots, t_n \in \mathcal{T}}{f(t_1, \ldots, t_n) \in \mathcal{T}}$$

(a) Définition inductive de l'ensemble des termes

$$\frac{R \in \mathcal{S}_R \qquad R \text{ d'arité } n \qquad t_1, \ldots, t_n \in \mathcal{T}}{R(t_1, \ldots, t_n) \in \mathcal{F}} \qquad \frac{}{\bot \in \mathcal{F}} \qquad \frac{A \in \mathcal{F}}{\neg A \in \mathcal{F}}$$

$$\frac{A \in \mathcal{F} \qquad B \in \mathcal{F}}{A \wedge B \in \mathcal{F}} \qquad \frac{A \in \mathcal{F} \qquad B \in \mathcal{F}}{A \vee B \in \mathcal{F}} \qquad \frac{A \in \mathcal{F} \qquad B \in \mathcal{F}}{A \rightarrow B \in \mathcal{F}}$$

$$\frac{A \in \mathcal{F} \qquad x \in \mathcal{V}}{\forall x\ A \in \mathcal{F}} \qquad \frac{A \in \mathcal{F} \qquad x \in \mathcal{V}}{\exists x\ A \in \mathcal{F}}$$

(b) Définition inductive des formules de la logique du premier ordre

FIGURE 4 – Termes et formules de la logique du premier ordre

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta} contr_g \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta} contr_d$$

$$\frac{\Gamma, A[x := t] \vdash \Delta}{\Gamma, \forall x\ A \vdash \Delta} \forall_g \qquad \frac{\Gamma \vdash A, \Delta \qquad x \notin FV(\Gamma, \Delta)}{\Gamma \vdash \forall x\ A, \Delta} \forall_d$$

$$\frac{\Gamma, A \vdash \Delta \qquad x \notin FV(\Gamma, \Delta)}{\Gamma, \exists x\ A \vdash \Delta} \exists_g \qquad \frac{\Gamma \vdash A[x := t], \Delta}{\Gamma \vdash \exists x\ A, \Delta} \exists_d$$

FIGURE 5 – Règles supplémentaires du calcul des séquents pour la logique du premier ordre

Annexe : variantes pour les règles du calcul des séquents relatives aux quantificateurs

$$\frac{\Gamma, A[x := t], \{\forall x\ A\} \vdash \Delta}{\Gamma, \forall x\ A \vdash \Delta}\forall_g^c \qquad \frac{\Gamma \vdash A, \Delta \qquad x \notin FV(\Gamma, \Delta)}{\Gamma \vdash \forall x\ A, \Delta}\forall_d$$

$$\frac{\Gamma, A \vdash \Delta \qquad x \notin FV(\Gamma, \Delta)}{\Gamma, \exists x\ A \vdash \Delta}\exists_g \qquad \frac{\Gamma \vdash A[x := t], \{\exists x\ A\}, \Delta}{\Gamma \vdash \exists x\ A, \Delta}\exists_d^c$$

FIGURE 6 – Nouvelles règles pour les quantificateurs

$$\frac{\Gamma, A[x := ?x], \{\forall x\ A\} \vdash \Delta}{\Gamma, \forall x\ A \vdash \Delta}\forall_g^d \qquad \frac{\Gamma \vdash A[x := f(?x_1, \ldots, ?x_n)], \Delta}{\Gamma \vdash \forall x\ A, \Delta}\forall_d^s$$

$$\frac{\Gamma, A[x := f(?x_1, \ldots, ?x_n)] \vdash \Delta}{\Gamma, \exists x\ A \vdash \Delta}\exists_g^s \qquad \frac{\Gamma \vdash A[x := ?x], \{\exists x\ A\}, \Delta}{\Gamma \vdash \exists x\ A, \Delta}\exists_d^d$$

FIGURE 7 – Règles des quantificateurs pour la méthode des tableaux