

# INFORMATIQUE

Durée : 4 heures

## Consignes :

- Veuillez à numéroté vos copies en bas à droite sous le format numéro\_de\_la\_page / nombre\_total\_de\_pages.
- Ne pas utiliser de correcteur blanc.
- Des points bonus/malus de présentation pourront éventuellement être accordés.
- Veuillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- **L'usage du cours ou de notes est strictement interdit.**

*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.*

**Les questions pratiques seront corrigées par un testeur automatique.** Le non-respect des consignes notamment en ce qui concerne les noms et les spécifications des fonctions entraînera donc automatiquement la note de 0 à la question.

Les questions pratiques marquées du symbole (☹☹) seront aussi lues intégralement lors de la correction du devoir. Des points pourront leur être accordés même si la fonction est fausse ou n'est pas aboutie. L'aspect compréhension de l'algorithmique sera donc évalué indépendamment de la syntaxe des langages pour ces questions, ainsi que la clarté du code et de ses annotations. **Des points pourront éventuellement être accordés à des pseudo-codes papiers sur ces questions.**

Lorsque le candidat écrira une fonction, il pourra également définir des fonctions auxiliaires. 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.

On identifiera dans cet énoncé 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`).

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  $\mathcal{O}(f(n, m))$  où  $n$  et  $m$  sont les tailles des arguments de la fonction, et  $f$  une expression la plus simple possible. Les calculs de complexité seront justifiés succinctement.

Ce sujet est constitué de trois parties **indépendantes**.

- La Partie I est un extrait de CCINP à coder en Ocaml, qui implémente la solution à une petite énigme de traversée de rivière.
- La Partie II est à coder en C et vise à tester l'existence d'une chaîne de dominos.
- La Partie III est à coder en OCaml et vise à implémenter le calcul des attracteurs en OCaml.

**Ne retournez pas la page avant d'y être invités.**

# DS pratique : Manipulations diverses

## I - Partie I : CCINP 2023 - Traversée de rivière

Cette partie comporte des questions nécessitant un code OCaml. En OCaml, 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 `@`).

**Le code de cette partie est à écrire dans un unique fichier `randonneurs.ml`.**

Un fichier compilé `corrigeRandonneurs.cmo` est fourni, ainsi que sa documentation `corrigeRandonneurs.mli`. Elle contient des corrigés de chaque fonction demandée, que vous pouvez librement appeler dans vos fichiers (utile notamment pour sauter une question et avoir tout de même accès à la fonction codée dans cette question). Attention cependant, si vous utilisez le corrigé d'une fonction dans cette même fonction, le testeur automatique le détectera et mettra la note de la question à 0.

Pour utiliser le fichier compilé dans `utop`, vous pouvez utiliser la commande `#load "corrigeRandonneurs.cmo"`. Un `makefile` est fourni si vous souhaitez travailler en compilé.

Dans une vallée des Alpes, un passage à gué fait de cailloux permet de traverser la rivière. Deux groupes de randonneurs arrivent simultanément sur les berges gauche et droite de cette rivière et veulent la traverser. Le chemin étant très étroit, une seule personne peut se trouver sur chaque caillou de ce chemin (**figure 1**). Un randonneur sur la berge gauche peut avancer d'un caillou (vers la droite sur la **figure 1**) et sauter par dessus le randonneur devant lui (un caillou à droite) si le caillou où il atterrit est libre. De même, chaque randonneur de la berge droite peut avancer d'un caillou (vers la gauche sur la **figure 1**) et sauter par dessus le randonneur devant lui, dans la mesure où le caillou sur lequel il atterrit est libre. Une fois engagés, les randonneurs ne peuvent pas faire marche arrière. De plus, pour simplifier, on suppose qu'une fois tous les randonneurs sur le chemin, il ne reste qu'un caillou de libre.



**Figure 1** - Les randonneurs et le chemin de cailloux

Le chemin de cailloux est défini par un tableau d'entiers :

```
type chemin_caillou = int array
```

Dans ce tableau, un randonneur venant de la berge de gauche est représenté par un 1, un randonneur issu de la berge de droite par un 2 et un caillou libre par un 0.

**Question 1** Ecrire une fonction de signature `caillou_vide : chemin_caillou -> int` qui détermine la position du caillou inoccupé.

**Question 2** Ecrire une fonction de signature `echange : chemin_caillou -> int -> int -> chemin_caillou` qui permute les valeurs codées sur deux cailloux. Le tableau d'entiers initial représentant le chemin n'est pas modifié. On pourra utiliser ici la fonction `copy` du module `Array`.

**Question 3** Ecrire une fonction de signature `randonneurG_avance : chemin_caillou -> bool` qui teste si, parmi les randonneurs venant de la berge de gauche, il en existe un qui puisse avancer (vers la droite).

**Question 4** Ecrire une fonction de signature `randonneurG_saute : chemin_caillou -> bool` qui teste si, parmi les randonneurs venant de la berge de gauche, il en existe un qui puisse sauter (vers la droite) au-dessus d'un randonneur.

On supposera dans la suite les fonctions de signature `randonneurD_avance : chemin_caillou -> bool` et `randonneurD_saute : chemin_caillou -> bool` écrites de manière similaire pour les randonneurs venant de la berge de droite. Ces fonctions sont disponibles dans la bibliothèque compilée fournie.

**Question 5** Ecrire une fonction de signature `mouvement_chemin : chemin_caillou -> chemin_caillou list` qui, en fonction de l'état du chemin, calcule la liste des états suivants possibles après les opérations suivantes (si elles sont permises) :

- (i). déplacement d'un randonneur venant de la berge de gauche,
- (ii). déplacement d'un randonneur venant de la berge de droite,
- (iii). saut d'un randonneur venant de la berge de gauche,
- (iv). saut d'un randonneur venant de la berge de droite.

**Question 6** Ecrire une fonction `accessible : chemin_caillou -> chemin_caillou -> bool` telle que l'appel `accessible chemin1 chemin2` renvoie vrai si depuis le chemin1 on peut atteindre le chemin2, où les chemins sont dorénavant **quelconques** (il n'y a toujours qu'un seul caillou vide).

*Indication : On effectuera un parcours (en gardant le graphe implicite) à l'aide d'une des deux structures `Stack` ou `Queue` d'OCaml. On utilisera éventuellement un dictionnaire (via `Hashtbl`) pour vérifier si une configuration a déjà été vue ou pour retenir les prédécesseurs des sommets.*

**Remarque :** En particulier, la complexité doit être linéaire en la taille du graphe. Si ce n'est pas le cas, des points seront retirés.

**Question 7** Ecrire une fonction `passage_optimal : chemin_caillou list -> chemin_caillou list -> chemin_caillou list` telle que l'appel `passage chemin1 chemin2` renvoie la liste des configurations permettant de passer du chemin1 au chemin2 **en un nombre minimal de mouvements de randonneurs**.

**Question 8** Ecrire une fonction de signature `passage : int -> int -> chemin_caillou list`, utilisant la question précédente, telle que l'appel `passage nG nD` résout le problème de passage de  $nG$  randonneurs venant de la berge de gauche et de  $nD$  randonneurs venant de la berge de droite. Par exemple, `passage 3 2` permet de passer de `[1; 1; 1; 0; 2; 2]` à `[2; 2; 0; 1; 1; 1]`. On renverra la liste des configurations permettant de passer de l'état initial à l'état final.

On donne la syntaxe OCaml pour créer une liste de  $N$  entiers  $i : List.init N (fun x -> i)$ .

Par exemple, `List.init 5 (fun x -> 2)` s'évalue en `[2; 2; 2; 2; 2]`.

### Solution de l'Exercice 1

Voir fichier `randonneurs.ml`.

Le sujet donne ce type, et nous pouvons commencer par donner un exemple de tel chemin, avec  $nG = 3$  randonneurs à gauche, et  $nD = 2$  randonneurs à droite, et un caillou vide entre eux.

`let ex = [1; 1; 1; 0; 2; 2]`

**Question 1** La fonction `caillou_vide` calcule la position du dernier caillou libre (ou vide, ou inoccupé). Comme il est supposé qu'il n'y a qu'un seul caillou libre, cela suffit. Notons que l'on pourrait utiliser des exceptions paramétriques pour interrompre le calcul plus tôt, ou une fonction récursive, mais cette version devrait suffire amplement.

**Question 2** On fera attention à ne pas modifier le tableau d'entrée, mais à utiliser un `Array.copy`.

**Question 3** RAS

**Question 4** RAS

**Question 5** *Remarque :* on va recalculer à chaque fois la position du caillou vide, ce qui se fait en temps linéaire. Il aurait donc été opportun d'inclure la position du caillou vide directement dans la configuration, pour y avoir systématiquement accès en temps constant.

**Question 6** On va générer le chemin initial avec la fonction `init`, puis explorer depuis cette position initiale, avec la fonction précédente.

On note que le sujet donne la syntaxe OCaml pour `List.init` (qui était attendu pour construire l'état initial du jeu), sauf que l'état du jeu est codé par un tableau (`type chemin_caillou = int array`). Cela peut être source supplémentaire de confusion pour certains candidats.



## II - Partie II : CCINP sujet zéro (2022) - Dominos

Cette partie comporte des questions nécessitant un code C.

**Le code de cette partie est à écrire dans un unique fichier `dominos.c`.**

Dans toute la suite, on suppose disposer, via `stdbool.h`, d'un type `bool` avec deux constantes `true` et `false`.

Un fichier compilé `corrigeDominos.o` est fourni, ainsi que son header `corrigeDominos.h`. Elle contient des corrigés de chaque fonction demandée, que vous pouvez librement appeler dans vos fichiers (utile notamment pour sauter une question et avoir tout de même accès à la fonction codée dans cette question). Attention cependant, si vous utilisez le corrigé d'une fonction dans cette même fonction, le testeur automatique le détectera et mettra la note de la question à 0.

Un domino  $D$  est une pièce rectangulaire contenant deux valeurs, de 0 à  $N$ , matérialisées par des points. Par exemple,  ou  sont deux dominos, représentant les couples  $(3, 5)$  et  $(4, 0)$ . Un domino est donc représenté par un couple d'entiers  $(i, j) \in \llbracket 0; N \rrbracket^2$ .

### 1) Structure de données

Dans cette section, on construit les structures de données utiles pour le premier problème. On définit le type structuré Domino par :

```
1 struct domino_s {
2     int x ;
3     int y ;
4 };
5 typedef struct domino_s Domino ;
```

On dispose d'un sac  $S$  contenant  $n$  dominos  $D_k = (i_k, j_k)$  pour  $k \in \llbracket 1; n \rrbracket$  et  $i_k, j_k \in \llbracket 0; N \rrbracket$ .

Une **chaîne** de dominos est une séquence de pièces telles que les valeurs voisines sur chaque paire de dominos consécutifs coïncident. Pour construire une chaîne, on ne peut utiliser qu'une fois une pièce présente dans le sac (elle est retirée du sac).

Par exemple, pour le sac  $(\text{img alt="domino (3,5)" data-bbox="308 580 373 603"}, \text{img alt="domino (5,4)" data-bbox="408 580 473 603"}, \text{img alt="domino (4,0)" data-bbox="508 580 573 603"}, \text{img alt="domino (0,3)" data-bbox="608 580 673 603"}, \text{img alt="domino (3,0)" data-bbox="708 580 773 603})$ , la séquence

     est une chaîne de 5 dominos.

Pour  $k \in \llbracket 0; n \rrbracket$ , une  $k$ -**chaîne** est une chaîne de longueur  $k$ . Par convention, la 0-chaîne est la chaîne vide.

On appelle **chaîne complète** une chaîne de longueur  $n$ , c'est-à-dire une chaîne qui utilise tous les dominos du sac.

La chaîne précédente est une 5-chaîne et est une chaîne complète pour la sac donné en exemple.

On souhaite gérer un sac et une chaîne comme une liste chaînée de dominos. On utilise donc le type `element` suivant permettant de stocker un domino et un pointeur vers l'élément suivant de la liste chaînée.

```
1 struct element_s {
2     Domino d;
3     struct element_s* suivant;
4 };
5 typedef struct element_s element;
```

On définit le type chaîne par `typedef element* chaine;`. Une chaîne est un pointeur vers le premier élément de la chaîne s'il existe; le pointeur NULL représente la chaîne vide.

On représente les sacs de la même manière : on définit donc le type sac par `typedef chaine sac;`.

**Question 1** Écrire une fonction de prototype `element* ajoutElement(element* l, Domino d)` qui ajoute le domino  $d$  à la chaîne ou au sac  $l$ . Cet ajout se fera en fin de liste chaînée.

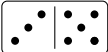
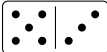
**Question 2** Écrire une fonction de prototype `element* retireElement(element* l, Domino d)` qui retire le domino  $d$  de la chaîne ou du sac  $l$  (s'il est présent). Cette fonction renvoie la chaîne obtenue.

**Question 3** Écrire une fonction de prototype `bool rechercheElement(element* l, Domino d)` qui recherche si le domino  $d$  est déjà dans la chaîne ou le sac  $l$ . La fonction renvoie `true` si c'est le cas, `false` sinon.

## 2) Existence d'une chaîne utilisant tous les dominos d'un sac

Dans ce premier problème, étant donné un sac contenant  $n$  dominos, on cherche à déterminer une chaîne complète, c'est-à-dire utilisant tous les dominos du sac, si elle existe.

**On suppose dans un premier temps que l'on ne peut pas effectuer de rotation de domino.**

Ainsi, le domino  ne pourra pas représenter le domino .

**Question 4** Écrire une fonction `bool possible(Domino Di, Domino Dj)` qui teste si il est possible de placer  $D_j$  à droite de  $D_i$ . La fonction renvoie `true` si c'est le cas, `false` sinon.

**On suppose maintenant qu'il est possible d'effectuer une rotation des dominos.**

Ainsi, si  $D_i = \begin{array}{|c|c|} \hline \bullet & \bullet \\ \hline \end{array}$  et  $D_j = \begin{array}{|c|c|} \hline \bullet & \bullet \\ \hline \end{array}$ , il n'est pas possible de placer directement  $D_j$  à droite de  $D_i$ , mais si on le retourne on obtient  $D_j = \begin{array}{|c|c|} \hline \bullet & \bullet \\ \hline \end{array}$  et le placement devient possible.

On souhaite donc écrire une fonction de prototype `bool possibleAvecRotation(Domino Di, Domino* Dj)`.

**Question 5** Pourquoi passe-t-on un pointeur sur  $D_j$ ? Spécifier de manière précise le rôle de cette fonction.

**Question 6** Écrire la fonction `possibleAvecRotation`.

Un algorithme de backtracking peut alors être envisagé pour résoudre ce problème.

**Question 7** Comment passer d'une  $k$ -chaîne à une  $k + 1$ -chaîne?

**Question 8** Proposer un algorithme, fondée sur un principe de backtracking, permettant de rechercher, si elle existe, une chaîne utilisant toutes les dominos du sac.

**Question 9 (Difficile! En particulier, difficile à déboguer. Ne passez pas plus de 30min sur cette question si vous n'aboutissez pas. Eventuellement, revenez-y à la fin...)**

Écrire en langage C le programme correspondant à votre algorithme, de prototype `element* chaineComplete(element* sac)`. (Utilisez une fonction intermédiaire qui effectue le retour sur trace, avec des arguments différents!)

*Indication : Il pourra être utile de coder une fonction intermédiaire `bout` qui va chercher le dernier maillon de la chaîne. Ou, pour une meilleure complexité, de recoder une fonction `ajoutElementTete` pour qu'elle ajoute le maillon en tête et non en bout, et construire la chaîne partielle à l'envers puis la renverser.*

**Question 10** Evaluer la complexité au pire des cas de votre algorithme. **Vous pouvez le faire à partir du pseudo-code!**

### Solution de l'Exercice 2

Voir fichier `dominos.c`.  
(corrigé d'après J.Carcenac)

**Question 1** Si la chaîne est vide, la chaîne est un pointeur vers un nouveau maillon (dans le tas). Si la chaîne n'est pas vide, on ajoute récursivement dans suivant.

**Remarques :**

- La fonction renvoie la chaîne (ce qui permet de gérer le cas de la chaîne vide).
- Il est possible d'écrire une fonction non récursive.

**Question 2** On suppose que si le domino est absent, il n'y a rien à faire.

Si la chaîne est vide : rien.

Si la chaîne contient le domino en tête, on renvoie suivant (et on pense à libérer la mémoire).

Si la chaîne contient un autre domino en tête, on supprime dans suivant **et on met à jour suivant**.

**Question 3** Avec des opérateurs paresseux et la récursivité, ou de manière plus lisible avec des if-then-else (les deux versions sont données).

**Question 4** Evitez le `if bool then return true; else return false; .`

**Question 5** On passe à la fonction un pointeur sur  $D_j$  pour qu'elle puisse modifier le domino  $D_j$ . Notez bien qu'une modification sur  $D_i$  n'aura ici pas d'effet du point de vue de ce qui appelle la fonction, **les paramètres sont passés par valeur** en C.

La fonction renvoie `true` et modifie `*Di` si la rotation est possible.

La fonction renvoie `false` sans modifier `*Di` sinon.

Remarque : On pourrait vouloir retourner le domino  $D_i$  aussi, ce qui n'est pas pris en compte par le sujet, vu sa question. Par exemple, ici, il n'est pas possible d'aboutir avec les dominos (2,3) et (2,1) alors que ce serait évidemment possible "dans la vraie vie"...

**Question 6** RAS.

**Question 7** On gère une chaîne de dominos encore dans le sac et une  $k$ -chaîne possible.

On peut passer à une  $k + 1$ -chaîne partielle en trouvant dans le sac un domino qui puisse se placer en bout de chaîne (avec éventuelle rotation).

**Question 8**

---

#### Algorithme 3 : `chercheChaîneComplete`

---

**Entrées :** Un sac et une chaîne partielle

**Sorties :** Vrai s'il est possible de compléter la chaîne partielle avec les dominos du sac, faux sinon

---

```

1 si le sac est vide alors
2 |   Afficher "Victoire!" + chaîne; renvoyer True;
3 sinon
4 |   pour chaque domino  $D$  du sac faire
5 |     Retirer le domino du sac;
6 |     si il se place en bout de chaîne alors
7 |       ajouter  $D$  en bout de chaînePartielle;
8 |       si chercheChaîneComplete(sac\D, chaînePartielle) alors
9 |         renvoyer true
10 |       ;
11 |     retirer  $D$  du bout de chaînePartielle;
12 |   sinon
13 |     si il se place en bout de chaîne avec rotation alors
14 |       ajouter  $D$  retourné en bout de chaînePartielle;
15 |       si chercheChaîneComplete(sac\D, chaînePartielle) alors
16 |         renvoyer true
17 |       retirer  $D$  retourné de chaînePartielle;
18 |   Replacer le domino  $D$  dans le sac.
19 renvoyer false
```

---

**Question 9** Il s'agit essentiellement d'implémenter le pseudo-code, assez littéralement. Voir code commenté.

**Question 10** Dans le pire cas, on peut potentiellement avoir à tester toutes les séquences possibles.

$2n$  possibilités pour le 1er domino ( $n$  dominos, avec rotations possibles).

$2(n - 1)$  possibilités pour le 2ème domino.

...

2 possibilités pour  $n$ -ième domino.

On obtient un total de  $2^n \cdot n!$  possibilités. Avec des ajouts/retraits dans le sac en  $\Theta(1)$  et dans la chaînePartielle,

on obtient une complexité en  $\Theta(2^n . n!)$ .



### III - Partie III : Jeux dans un graphe

Cette partie comporte des questions nécessitant un code OCaml.

Le code de cette partie est à écrire dans un unique fichier `jeux.ml`.

Un fichier compilé `corrigeJeux.cmo` est fourni, ainsi que sa documentation `corrigeJeux.mli`. Elle contient des corrigés de chaque fonction demandée, que vous pouvez librement appeler dans vos fichiers (utile notamment pour sauter une question et avoir tout de même accès à la fonction codée dans cette question). Attention cependant, si vous utilisez le corrigé d'une fonction dans cette même fonction, le testeur automatique le détectera et mettra la note de la question à 0.

Pour utiliser le fichier compilé dans `utop`, vous pouvez utiliser la commande `#load "corrigeJeux.cmo"`. Un `makefile` est fourni si vous souhaitez travailler en compilé.

Un **graphe** (fini et orienté) est un couple  $G = (S, A)$  tel que  $S$  est un ensemble fini non vide et  $A \subseteq S \times S$ . Pour  $s \in S$  un sommet, on note  $V(s)$  l'ensemble des **voisins** de  $s$ , c'est-à-dire  $V(s) = \{t \in S \mid (s, t) \in A\}$ . On appelle **chemin**  $\sigma$  une suite non vide, finie ou infinie, de sommets  $(s_i)_{0 \leq i < m}$ , avec  $m \in \mathbb{N}^* \cup \{\infty\}$ , telle que  $s_{i+1} \in V(s_i)$  si  $0 \leq i$  et  $i+1 < m$ . Si  $m \neq \infty$ , on dit que  $\sigma$  est **fini de longueur**  $m-1$ . Sinon, il est dit **infini**. Il est à noter qu'un chemin  $\sigma$  peut passer plusieurs fois par le même sommet. On notera  $\mathcal{C}(G)$  l'ensemble des chemins (finis ou infinis) de  $G$ .

Pour  $s \in S$ , le **nombre d'occurrences de  $s$  dans  $\sigma$** , noté  $|\sigma|_s$ , correspond aux nombres de fois qu'un sommet de  $T$  apparaît dans  $\sigma$ . Formellement,  $|\sigma|_s = \text{Card} \{i \in \llbracket 0, m \rrbracket \mid s_i = s\}$ , ce cardinal pouvant être fini ou infini. On note de même, pour  $T \subseteq S$ ,  $|\sigma|_T = \text{Card} \{i \in \llbracket 0, m \rrbracket \mid s_i \in T\}$ .

On considère un jeu à deux joueurs dans un graphe. Informellement, une partie se déroule comme suit : un jeton est placé sur un sommet du graphe  $G$  et déplacé par les joueurs de sommet en sommets, par une succession de coups. Un coup consiste à déplacer le jeton en suivant une arête : lorsque le jeton est sur un sommet  $s$ , le joueur à qui appartient  $s$  le déplace sur un voisin de  $s$ , et ainsi de suite. Une partie est un chemin traversé par le jeton.

Formellement, une **arène** est un triplet  $(G, S_1, S_2)$  tel que  $G = (S, A)$  est un graphe et  $S = S_1 \cup S_2$ ,  $S_1 \cap S_2 = \emptyset$ . Un **jeu** est un quadruplet  $(G, S_1, S_2, W)$  tel que  $(G, S_1, S_2)$  est une arène et  $W \subseteq \mathcal{C}(G)$ . Une **partie depuis un sommet initial**  $s$  est un chemin  $\sigma = (s_i)_{0 \leq i < m}$  tel que  $s_0 = s$ . On dit que la partie  $\sigma$  est **gagnée** par le joueur 1 si  $\sigma \in W$ . Sinon,  $\sigma$  est dite gagnée par le joueur 2.

Pour  $j \in \{1, 2\}$ , une **stratégie pour le joueur  $j$**  est une application  $f : S_j \rightarrow S$  telle que pour tout  $s \in S_j$ ,  $f(s) \in V(s)$ . Une partie  $\sigma = (s_i)_{0 \leq i < m}$  est dite une  **$f$ -partie** si pour tout  $s_i \in S_j$ , tel que  $i+1 < m$ ,  $s_{i+1} = f(s_i)$ . Une stratégie  $f$  pour le joueur  $j$  est dite **gagnante depuis  $s$**  si toute  $f$ -partie depuis  $s$  est gagnée par le joueur  $j$ . Un sommet  $s \in S$  est dit **gagnant** pour  $j$  s'il existe une stratégie gagnante pour  $j$  depuis  $s$ .

Un jeu est dit **positionnel** si tout sommet est gagnant pour 1 ou 2, c'est-à-dire s'il existe  $R_1, R_2 \subseteq S$  tels que  $S = R_1 \cup R_2$  et deux stratégies  $f_1$  et  $f_2$  telles que  $f_j$  est gagnante pour  $j$  depuis tout sommet de  $R_j$ , pour  $j \in \{1, 2\}$ . On remarquera que  $R_1$  (resp.  $R_2$ ) peut contenir à la fois des sommets de  $S_1$  et des sommets de  $S_2$ .

**On considèrera dans l'ensemble de cette partie que les graphes considérés sont sans puits, c'est-à-dire que pour tout  $s \in S$ ,  $V(s) \neq \emptyset$ .**

#### 1) Préliminaires

On considère l'arène de la figure 1, où  $S_1 = \{1\}$  et  $S_2 = \{0, 2\}$  (les sommets de  $S_1$  sont représentés par des cercles, ceux de  $S_2$  par des carrés).

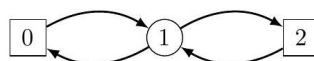


FIGURE 1 – Une arène  $(G, S_1, S_2)$ .

**Question 1** On suppose que  $W$  est l'ensemble des chemins ne visitant pas le sommet 0 :  $W = \{\sigma \in \mathcal{C}(G) \mid |\sigma|_0 = 0\}$ . Le jeu  $(G, S_1, S_2, W)$  est-il positionnel ? Si oui, donner les ensembles  $R_1$  et  $R_2$  et les stratégies  $f_1$  et  $f_2$  correspondants. Sinon, justifier.



**Question 2** Même question si  $W = \{\sigma \in \mathcal{C}(G) \mid |\sigma|_0 = \infty \text{ et } |\sigma|_2 = \infty\}$ .

**Question 3** Montrer que si un jeu est positionnel, alors les ensembles  $R_1$  et  $R_2$  sont disjoints.

### Solution de l'Exercice 3

**Question 1** On remarque qu'indépendamment de  $W$ , il n'existe qu'une seule stratégie pour le joueur 2 :  $f_2 : s \in S_2 \mapsto 1$ . De plus, il existe deux stratégies pour le joueur 1 :  $f_1 : 1 \mapsto 2$  et  $f'_1 : 1 \mapsto 0$ . Si  $W = \{\sigma \in \mathcal{C}(G) \mid |\sigma|_0 = 0\}$ . Le jeu  $(G, S_1, S_2, W)$  est positionnel. En effet, On pose  $R_1 = \{1, 2\}$  et  $R_2 = \{0\}$ , et on considère les stratégies  $f_1$  et  $f_2$ .

- toute partie depuis 0 est gagnée par 2, car un chemin commençant par 0 ne peut pas être dans  $W$ . On en déduit que  $f_2$  est bien une stratégie gagnante pour 2 depuis  $R_2$
- soit  $\sigma = (s_i)_{0 \leq i < m}$  une  $f_1$ -partie depuis 1 ou 2. Alors  $\sigma$  alterne entre les sommets 1 et 2. Plus formellement :
  - \* si  $s_0 = 1$ , alors pour tout  $0 \leq i < m$ ,  $s_i = 1 + (i \bmod 2)$ ;
  - \* si  $s_0 = 2$ , alors pour tout  $0 \leq i < m$ ,  $s_i = 2 - (i \bmod 2)$ .

Dans les deux cas,  $f_1$  est bien une stratégie gagnante pour 1 depuis  $R_1$ .

**Question 2** Si  $W = \{\sigma \in \mathcal{C}(G) \mid |\sigma|_0 = \infty \text{ et } |\sigma|_2 = \infty\}$ , alors le jeu n'est pas positionnel. En effet, toute  $f_1$ -partie ne passera qu'au plus une fois par le sommet 0, et toute  $f'_1$ -partie ne passera qu'au plus une fois par le sommet 2. On en déduit qu'aucune stratégie n'est gagnante pour 1, quel que soit le sommet de départ. De plus, si on pose  $\sigma = (0, 1, 2, 1, 0, 1, 2, \dots)$ , alors  $\sigma$  est une  $f_2$ -partie qui n'est pas gagnée par 2. On en déduit que le jeu n'est pas positionnel.

**Question 3** Supposons que le jeu est positionnel et soient  $f_1$  et  $f_2$  les stratégies correspondantes. Supposons par l'absurde que  $R_1 \cap R_2 \neq \emptyset$  et soit  $s \in R_1 \cap R_2$ . Sans perte de généralité, supposons  $s \in S_1$ . On définit  $\sigma = (s_i)_{i \in \mathbb{N}}$  par :

- $s_0 = s$
- pour  $i \in \mathbb{N}$ ,  $s_{i+1} = \begin{cases} f_1(s_i) & \text{si } s_i \in S_1 \\ f_2(s_i) & \text{sinon} \end{cases}$

Alors par définition,  $\sigma$  est une  $f_1$ -partie et une  $f_2$ -partie. Elle doit être donc gagnante pour 1 et pour 2, car  $s \in R_1 \cap R_2$ . On conclut par l'absurde.

## 2) Jeux d'accessibilité

On suppose dans cette partie que  $T \subseteq S$  et que  $W = \{\sigma \in \mathcal{C}(G) \mid |\sigma|_T > 0\}$ . On représente un graphe  $G = (S, A)$  en OCaml par un tableau de listes d'adjacence.

**OCaml** `1 type graphe = int list array`

Si  $g$  est une variable de type `graphe` correspondant à un graphe  $G = (S, A)$ , alors :

- $S = \llbracket 0, n-1 \rrbracket$ , où  $n = \text{Array.length } g$ ;
- pour  $s \in S$ ,  $g.(s)$  est une liste contenant les éléments de  $V(s)$  (sans doublons, dans un ordre arbitraire).

**Question 4** Écrire une fonction `transpose : graphe -> graphe` qui prend en argument un graphe  $G = (S, A)$  et renvoie son graphe transposé  $G^T = (S, A')$  où  $A' = \{(s, t) \mid (t, s) \in A\}$ . On garantira une complexité en  $\mathcal{O}(|S| + |A|)$  mais on ne demande pas de le justifier.

Une partie  $T \subseteq S$  sera représentée par un tableau de booléens `tab` de taille  $n$  tel que `tab.(s)` vaut `true` si et seulement si  $s \in T$ .

**Question 5** Dans cette question, on suppose que le jeu est à un seul joueur, c'est-à-dire que  $S_2 = \emptyset$ . Écrire une fonction `strategie : graphe -> bool array -> int array` qui prend en argument un graphe  $G$  et une partie  $T \subseteq S$  et renvoie un tableau `f` de taille  $n$  tel que pour tout  $s \in S$  :

- si  $s \in T$ , alors `f.(s) = n`;

- sinon, s'il existe une stratégie gagnante  $f_1$  depuis  $s$ , alors  $f \cdot (s) = f_1(s)$ ;
- sinon,  $f \cdot (s) = -1$ .

On garantira une complexité en  $\mathcal{O}(|S| + |A|)$  et on demande de justifier cette complexité.

On suppose pour la suite que  $S_1 \neq \emptyset$  et  $S_2 \neq \emptyset$ . Pour  $X \subseteq S$ , on définit par induction l'ensemble  $\text{Attr}_i(X)$ , pour  $i \in \mathbb{N}$ , par :

- $\text{Attr}_0(X) = X$ ;
- $\text{Attr}_{i+1}(X) = \text{Attr}_i(X) \cup \{s \in S_1 \mid V(s) \cap \text{Attr}_i(X) \neq \emptyset\} \cup \{s \in S_2 \mid V(s) \subseteq \text{Attr}_i(X)\}$ .

**Question 6** Donner une description en français de  $\text{Attr}_i(X)$  en termes d'existence de chemin dans le cas où  $S_2 = \emptyset$ .

**Question 7** Montrer qu'il existe  $k \in \mathbb{N}$  tel que  $\bigcup_{i \in \mathbb{N}} \text{Attr}_i(X) = \text{Attr}_k(X)$ . On notera  $\text{Attr}(X)$  cet ensemble pour la suite et on l'appellera **attracteur de  $X$** .

**Question 8** Montrer que le jeu est positionnel. Plus précisément :

1. Montrer que le joueur 1 a une stratégie gagnante depuis  $R_1 = \text{Attr}(T)$ .
2. Montrer que le joueur 2 a une stratégie gagnante depuis  $R_2 = S \setminus \text{Attr}(T)$ .

On rappelle que les files peuvent être utilisées en OCaml avec les commandes suivantes, toutes en complexité  $\mathcal{O}(1)$  :

- `Queue.create : unit -> 'a Queue.t` permet de créer une file vide ;
- `Queue.is_empty : 'a Queue.t -> bool` permet de tester si une file est vide ;
- `Queue.push : 'a -> 'a Queue.t -> unit` ajoute un élément à la fin d'une file ;
- `Queue.pop : 'a Queue.t -> 'a` enlève un élément au début d'une file et le renvoie.

**Question 9** Écrire une fonction `attracteur : graphe -> bool array -> bool array -> bool array` qui prend en argument un graphe  $G$ , un tableau représentant  $S_1 \subseteq S$  et un tableau représentant une partie  $T \subseteq S$  et renvoie un tableau `attr` de taille  $n$  représentant  $\text{Attr}(T)$ . Donner la complexité de la fonction.

### Solution de l'Exercice 4

Voir fichier `jeux.ml`

**Question 4** Un classique : pour chaque voisin  $t$  d'un sommet  $s$ , on ajoute  $s$  comme voisin de  $t$  dans le graphe transposé.

**Question 5** On remarque qu'il existe une stratégie gagnante depuis un sommet  $s$  si et seulement s'il existe un chemin de  $s$  à un sommet  $t \in T$ . Pour construire la stratégie, on va plutôt travailler dans le graphe transposé et trouver un chemin de  $t$  à  $s$ . On écrit pour cela une fonction auxiliaire `dfs : int -> int -> unit` qui lance un parcours depuis le sommet  $t$  en indiquant que  $f(t) = s$  (ou  $n$  si  $t \in T$ ). On lance ensuite un parcours depuis chaque sommet de  $T$ .

(Voir code)

**Complexité** : Le calcul de  $G^T$  se fait en temps  $\mathcal{O}(|A| + |S|)$ . De plus, on remarque qu'on fait au plus un appel à `List.iter` pour chaque liste d'adjacence, c'est-à-dire que le nombre total d'appels récurrents à `dfs` est au plus  $\mathcal{O}(|A|)$ . Étant donnée la taille de la boucle `for`, on a bien une complexité totale en  $\mathcal{O}(|A| + |S|)$ .

**Question 6**  $\text{Attr}_i(X)$  est l'ensemble des sommets  $s \in S$  tel qu'il existe un chemin de taille au plus  $i$  de  $s$  à un sommet de  $X$ .

**Question 7** Dans le cas général,  $\text{Attr}_i(X)$  est l'ensemble des sommets  $s \in S$  tel que le joueur 1 peut forcer une partie depuis  $s$  à arriver dans  $X$  en  $i$  coups ou moins.

Par définition, on remarque que la suite  $(\text{Attr}_i(X))_{i \in \mathbb{N}}$  est croissante. De plus, tous les  $\text{Attr}_i(X)$  sont inclus dans  $S$ . Étant donné que  $S$  est fini, on en déduit que la suite est ultimement stationnaire, ce qui conclut.

**Question 8**

1. On définit la stratégie  $f_1$  de la manière suivante : si  $s \in R_1 \cap S_1$ , alors soit  $i_0 = \min \{i \in \mathbb{N} \mid s \in \text{Attr}_i(T)\}$ . On pose  $f_1(s) \in V(s) \cap \text{Attr}_{i_0-1}(T)$  choisi arbitrairement (avec la convention  $\text{Attr}_{-1}(T) = S$ ). Par hypothèse, cet ensemble est non vide si  $i_0 > 0$ . Soit alors  $\sigma = (s_i)_{0 \leq i < m}$  une  $f_1$ -partie depuis un sommet  $s \in R_1$ . Soit  $0 \leq i < m$  tel que  $i + 1 < m$  et  $j$

l'indice minimal tel que  $s_i \in \text{Attr}_j(T)$ . Montrons que  $s_{i+1} \in \text{Attr}_{j-1}(T)$  :

- si  $s_i \in S_1$ , alors  $s_{i+1} = f_1(s_i) \in V(s_i) \cap \text{Attr}_{j-1}(T)$ ;
- sinon, si  $s_i \in S_2$ , alors  $V(s_i) \subseteq \text{Attr}_{j-1}(T)$ , donc  $s_{i+1} \in V(s_i) \cap \text{Attr}_{j-1}(T)$ .

On en déduit par une récurrence rapide que  $s_{i_0} \in T$ , donc  $\sigma$  est gagnante pour 1.

2. On définit la stratégie  $f_2$  de la manière suivante : si  $s \in R_2 \cap S_2$ , alors par hypothèse,  $V(s) \subsetneq \text{Attr}(T)$ . On pose alors  $f_2(s) \in V(s) \cap R_2$  qui est bien non vide.  
Soit alors  $\sigma = (s_i)_{0 \leq i < m}$  une  $f_2$ -partie depuis un sommet  $s \in R_2$ . Montrons par récurrence que pour  $0 \leq i < m$ ,  $s_i \in R_2$  :


- $s_0 = s \in R_2$  par hypothèse ;
- supposons le résultat vrai pour  $0 \leq i < m$  tel que  $i + 1 < m$  et distinguons :
  - \* si  $s_i \in S_1$ , alors par définition de  $\text{Attr}(T)$ ,  $V(s_i) \cap \text{Attr}(T) = \emptyset$ . On en déduit que  $s_{i+1} \in V(s_i) \cap R_2$
  - \* si  $s_i \in S_2$ , alors  $s_{i+1} = f_2(s_i) \in V(s_i) \cap R_2$ .

On conclut par récurrence que  $\sigma$  est gagnante pour 2, car  $R_2 \cap T = \emptyset$ .

**Question 9** On utilise ici une sorte de parcours en largeur dans  $G^T$ , où la « distance » d'un sommet  $s$  à  $T$  correspond à l'indice minimal  $i_0$  tel que  $s \in \text{Attr}_{i_0}(T)$ . Pour ce faire, on crée une file et un tableau `attr`. On écrit également une fonction auxiliaire `traiter : int -> unit` qui vérifie pour un sommet  $t$  donné s'il appartient à  $\text{Attr}_{i+1}(T)$ , en supposant que `attr` contient tous les sommets de  $\text{Attr}_i(T)$ . Pour ce faire, on utilise deux tableaux `deg_attr` et `degre` contenant respectivement, pour un sommet  $t$ , le nombre de voisins de  $t$  dans  $\text{Attr}_i(T)$  et le nombre de voisins dans  $S$ . La fonction `ajout_sommet` se contente d'ajouter un sommet  $s$  à l'attracteur et dans la file. Elle s'occupe également d'augmenter de 1 le nombre de voisins dans  $\text{Attr}_i(T)$  de chacun des prédécesseurs de  $s$  (donc des voisins de  $s$  dans  $G^T$ ). Dès lors, la boucle `while` consiste à traiter tous les prédécesseurs d'un sommet nouvellement ajouté à  $\text{Attr}(T)$ .  
(Voir code.)

**Complexité :** On remarque qu'un appel à `ajout_sommet s` a une complexité en  $\mathcal{O}(\deg_+(s))$ . De plus, un tel appel ne sera effectué qu'une seule fois par sommet, car un appel à `ajout_sommet s` n'est fait que dans la fonction `traiter`, qui nécessite que `attr.(s)` vaut `false` (et `attr.(s)` vaut `true` après l'appel à `ajout_sommet s`). Les autres opérations étant en  $\mathcal{O}(|S| + |A|)$ , on en déduit que c'est la complexité totale de la fonction.

### 3) (Bonus théorique) Jeux de Büchi

 Cette partie est à faire en bonus (après tout le reste) et ne vaudra pas beaucoup de points bonus.

On suppose dans cette partie que  $T \subseteq S$  et que  $W = \{\sigma \in \mathcal{C}(G) \mid |\sigma|_T = \infty\}$ , c'est-à-dire qu'une partie est gagnée par le joueur 1 si elle visite infiniment souvent un sommet de  $T$ .

**Question 10** Dans cette question, on suppose que le jeu est à un seul joueur, c'est-à-dire que  $S_2 = \emptyset$ . On considère  $s \in S$ . Donner une condition nécessaire et suffisante en termes d'existence de chemin et de cycle pour qu'il existe une stratégie gagnante depuis  $s$ . Avec quelle complexité temporelle peut-on calculer tous les sommets  $s$  tels qu'il existe une stratégie gagnante depuis  $s$ ? Détailler.

On suppose pour la suite que  $S_1 \neq \emptyset$  et  $S_2 \neq \emptyset$ . Pour  $X \subseteq S$ , on définit par induction l'ensemble  $\text{Attr}_i^+(X)$ , pour  $i \in \mathbb{N}$ , par :

- $\text{Attr}_0^+(X) = \emptyset$ ;
- $\text{Attr}_{i+1}^+(X) = \text{Attr}_i^+(X) \cup \{s \in S_1 \mid V(s) \cap (\text{Attr}_i^+(X) \cup X) \neq \emptyset\} \cup \{s \in S_2 \mid V(s) \subseteq \text{Attr}_i^+(X) \cup X\}$ .

On pose de plus  $\text{Attr}^+(X) = \bigcup_{i \in \mathbb{N}} \text{Attr}_i^+(X)$ .

**Question 11** Si  $X \subseteq S$ , montrer que :

1. tout sommet de  $\text{Attr}^+(X) \cap S_1$  a un voisin dans  $\text{Attr}(X)$ ;
2. tout sommet de  $\text{Attr}^+(X) \cap S_2$  a tous ses voisins dans  $\text{Attr}(X)$  ;  
où  $\text{Attr}(X)$  a été défini à la partie 1.2.

On définit  $E_0(T) = T$  et pour  $i \in \mathbb{N}$ ,  $E_{i+1}(T) = \text{Attr}^+(E_i(T)) \cap T$ . On pose  $E(T) = \bigcap_{i \in \mathbb{N}} E_i(T)$ .

**Question 12** Dédurre de la question précédente une stratégie pour le joueur 1 gagnante depuis  $\text{Attr}(E(T))$ .

*Indication : on pourra commencer par montrer que la suite  $(E_i)_{i \in \mathbb{N}}$  est décroissante puis définir la stratégie par ses restrictions sur les ensembles  $E(T)$  et  $\text{Attr}(E(T)) \setminus E(T)$ .*

**Question 13** Montrer que le joueur 2 a une stratégie gagnante depuis  $S \setminus \text{Attr}(E(T))$ .

### Solution de l'Exercice 5

**Question 10** Il existe une stratégie gagnante depuis  $s$  s'il existe  $t \in T$  tel qu'il existe un chemin de  $s$  à  $t$  et un cycle (non vide) contenant  $t$ . En utilisant l'algorithme de Kosaraju, on peut calculer toutes les composantes fortement connexes du graphe en complexité  $\mathcal{O}(|S| + |A|)$ . Dès lors, un sommet  $t \in T$  conviendra si sa composante fortement connexe est de taille  $> 1$ , ou s'il existe une boucle  $(t, t) \in A$  (ce qui peut se tester pendant l'algorithme). On peut alors faire un parcours dans  $G^T$  depuis tous les sommets  $t$  qui conviennent pour trouver tous les sommets  $s$  cherchés. La complexité totale est en  $\mathcal{O}(|S| + |A|)$ .

**Question 11** Montrons un résultat intermédiaire, à savoir que pour  $i \in \mathbb{N}$ ,  $\text{Attr}_i^+(X) \cup X \subseteq \text{Attr}_i(X)$ , par récurrence sur  $i$  :

- c'est vrai pour  $i = 0$  car  $\text{Attr}_0^+(X) \cup X = X = \text{Attr}_0(X)$ ;
- supposons le résultat établi pour  $i \in \mathbb{N}$  fixé. Soit alors  $s \in \text{Attr}_{i+1}^+(X)$ . Si  $s \in \text{Attr}_i^+(X)$ , alors par hypothèse,  $s \in \text{Attr}_i(X) \subseteq \text{Attr}_{i+1}(X)$ . Sinon, distinguons :
  - \* si  $s \in S_1$ , alors  $V(s) \cap (\text{Attr}_i^+(X) \cup X) \neq \emptyset$ . On en déduit que  $V(s) \cap \text{Attr}_i(X) \neq \emptyset$ , c'est-à-dire que  $s \in \text{Attr}_{i+1}(X)$ ;
  - \* si  $s \in S_2$ , alors  $V(s) \subseteq \text{Attr}_i^+(X) \cup X \subseteq \text{Attr}_i(X)$ , c'est-à-dire  $s \in \text{Attr}_{i+1}(X)$ .

Comme  $X \subseteq \text{Attr}_{i+1}(X)$ , on en déduit que  $\text{Attr}_{i+1}^+(X) \cup X \subseteq \text{Attr}_{i+1}(X)$ .

On conclut par récurrence. On peut maintenant répondre à la question :

1. soit  $s \in \text{Attr}^+(X) \cap S_1$  et soit  $i \in \mathbb{N}^*$  l'indice minimal tel que  $s \in \text{Attr}_i^+(X)$ . Par hypothèse,  $V(s) \cap (\text{Attr}_{i-1}^+(X) \cup X) \neq \emptyset$ . On en déduit qu'il existe un voisin de  $s$  dans  $\text{Attr}_{i-1}^+(X) \cup X \subseteq \text{Attr}_{i-1}(X) \subseteq \text{Attr}(X)$ .
2. soit  $s \in \text{Attr}^+(X) \cap S_2$  et soit  $i \in \mathbb{N}^*$  l'indice minimal tel que  $s \in \text{Attr}_i^+(X)$ . Par hypothèse,  $V(s) \subseteq \text{Attr}_{i-1}^+(X) \cup X \subseteq \text{Attr}_{i-1}(X) \subseteq \text{Attr}(X)$ .

Notons que l'inclusion réciproque étant vraie par définition de  $\text{Attr}^+(X)$ , on a :

$$\text{Attr}^+(X) = \{s \in S_1 \mid V(s) \cap \text{Attr}(X) \neq \emptyset\} \cup \{s \in S_2 \mid V(s) \subseteq \text{Attr}(X)\}$$

**Question 12** Reformulons en français ce que veulent dire ces ensembles :

- $\text{Attr}_i^+(X)$  est l'ensemble des sommets  $s \in S$  tel que le joueur 1 peut forcer une partie depuis  $s$  à arriver dans  $X$  en un nombre de coups compris entre 1 et  $i$ ;
- $\text{Attr}^+(X)$  est l'ensemble des sommets  $s \in S$  tel que le joueur 1 peut forcer une partie depuis  $s$  à arriver dans  $X$  en un nombre de coups strictement positif;
- $E_i(T)$  est l'ensemble des sommets  $t \in T$  tel que le joueur 1 peut forcer une partie depuis  $t$  à passer  $i + 1$  fois par un sommet de  $T$ ;
- $E(T)$  est l'ensemble des sommets  $t \in T$  tel que le joueur 1 peut forcer une partie depuis  $t$  à passer une infinité de fois par un sommet de  $T$ .

Remarquons tout d'abord que si  $X \subseteq Y$ , alors  $\text{Attr}^+(X) \subseteq \text{Attr}^+(Y)$ . Montrons que  $E_{i+1}(T) \subseteq E_i(T)$  par récurrence sur  $i \in \mathbb{N}$  :

- pour  $i = 0$ ,  $E_1(T) = \text{Attr}^+(T) \cap T \subseteq T = E_0(T)$ ;
- supposons le résultat vrai pour  $i \in \mathbb{N}$  fixé. Alors  $E_{i+2}(T) = \text{Attr}^+(E_{i+1}(T)) \cap T \subseteq \text{Attr}^+(E_i(T)) \cap T = E_{i+1}(T)$ .

On conclut par récurrence.

Comme les  $E_i(T)$  sont des ensembles finis, on en déduit qu'il existe  $k \in \mathbb{N}$  tel que  $E_{k+1}(T) = E_k(T) = E(T)$ . On définit alors la stratégie  $f_1$  suivante pour  $s \in \text{Attr}(E(T)) \cap S_1$  :

- si  $s \in E(T)$ , alors  $s \in E_{k+1}(T) = \text{Attr}^+(E_k(T)) \cap T$ . On en déduit que  $V(s) \cap \text{Attr}(E_k(T)) \neq \emptyset$ , donc et on pose  $f(s)$  un voisin de  $s$  dans  $\text{Attr}(E(T))$ ;
- si  $s \notin E(T)$ , alors par la question 8, il existe une stratégie  $f$  telle que toute  $f$ -partie depuis  $s$  rencontre  $E(T)$ . On pose  $f_1(s) = f(s)$ .

La stratégie  $f_1$  est bien gagnante pour 1 : depuis un sommet de  $E(T)$ , on peut forcer un coup dans  $\text{Attr}(E(T))$ , et depuis un sommet de  $\text{Attr}(E(T)) \setminus E(T)$ , la stratégie permet de revenir dans  $E(T)$  en un nombre fini de coups. Une  $f_1$ -partie passe donc une infinité de fois par un sommet de  $E(T) \subseteq T$ .

**Question 13** On pose, par convention,  $E_{-1}(T) = S$ . Remarquons que pour  $s \notin \text{Attr}(E(T))$ , il existe  $i \in \mathbb{N}$  tel que  $s \in \text{Attr}(E_{i-1}(T))$  et  $s \notin \text{Attr}(E_i(T))$  (sinon  $s \in \text{Attr}(E(T))$ ). Pour  $s \in S_2$ , on définit la stratégie  $f_2$  en distinguant :

- si  $i = 0$ , alors  $s \in S \setminus \text{Attr}(T)$ , donc il existe une stratégie  $f$  pour le joueur 2 tel qu'une  $f$ -partie n'atteint jamais un sommet de  $T$ . On pose  $f_2(s) = f(s)$ ;
- sinon, si  $s \in T$ , alors  $s \in E_{i-1}(T)$ , et on a  $V(s) \setminus \text{Attr}(E_{i-1}(T)) \neq \emptyset$  (sinon  $s \in \text{Attr}^+(E_{i-1}(T)) \cap T = E_i(T)$ ). On pose  $f_2(s)$  un tel sommet.
- sinon,  $V(s) \setminus \text{Attr}(E_i(T)) \neq \emptyset$ , car  $s \notin \text{Attr}(E_i(T))$ . On pose  $f_2(s)$  un tel sommet.

Montrons que  $f_2$  est une stratégie gagnante pour 2 depuis tout sommet  $s$  de  $S \setminus \text{Attr}(E(T))$ . Soit  $\sigma = (s_i)_{i \in \mathbb{N}}$  une  $f_2$ -partie depuis  $s$ . Il existe un indice  $i_0 \in \mathbb{N}$  tel que  $s \in \text{Attr}(E_{i_0-1}(T))$  et  $s \notin \text{Attr}(E_{i_0}(T))$ . Alors  $\sigma$  passe au plus  $i_0$  fois par un sommet de  $T$  : tant qu'on ne passe pas par  $T$ , on reste dans un  $\text{Attr}(E_{j-1}(T)) \setminus \text{Attr}(E_j(T))$ , et dès qu'on passe par  $T$ , par construction, le sommet suivant sera dans  $\text{Attr}(E_{j-2}(T)) \setminus \text{Attr}(E_{j-1}(T))$ .

# INFORMATIQUE

**Durée : 4 heures**

**Consignes :**

- Veuillez à numéroté vos copies en bas à droite sous le format `numéro_de_la_page / nombre_total_de_pages`.
- Ne pas utiliser de correcteur blanc.
- Des points bonus/malus de présentation pourront éventuellement être accordés.
- Veuillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- **L'usage du cours, de notes ou de tout appareil électronique est strictement interdit.**

*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.*

Le sujet est constitué de quatre parties indépendantes. Il est recommandé de ne pas accorder plus de 30min à la partie 0 et pas plus de 45min à la partie I, mais il est recommandé de commencer par les traiter, car elles vaudront proportionnellement plus de points que le reste.

Les questions de programmation doivent être traitées en langage OCaml ou C selon ce qui est demandé par l'énoncé. En OCaml, 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. En C, on supposera que les bibliothèques `stdlib.h` et `stdbool.h` ont été chargées.

Le fonctionnement des programmes non triviaux doit être expliqué. En particulier, il est attendu des candidats qu'ils justifient au moins brièvement la correction de leurs programmes quand celle-ci n'est pas évidente, notamment en explicitant des variants et/ou des invariants.

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 auxiliaires, mais devra préciser leurs rôles ainsi que les types et significations 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.

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`).

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  $O(f(n, m))$  où  $n$  et  $m$  sont les tailles des arguments de la fonction, et  $f$  une expression la plus simple possible. Les calculs de complexité seront justifiés succinctement.

*Le sujet comporte 7 pages.*

**Ne retournez pas la page avant d'y être invités.**

## Partie 0 - Un peu de NP-complétude...

Le but de cette partie est de montrer que le problème 2-PARTITION est NP-complet.

On définit le problème 2-PARTITION comme suit :

**Instance :** un ensemble  $E$  de  $n$  entiers (quelconques)  $a_1, a_2, \dots, a_n$

**Question :** Existe-t-il un sous-ensemble  $E' \subseteq E$  tel que  $\sum_{a \in E'} a = \sum_{a \notin E'} a$  ?

**Question 1** Montrer que 2-PARTITION est dans NP.

On rappelle que le problème SUBSET-SUM est NP-complet. Il est défini comme suit :

**Instance :** un ensemble fini  $S$  d'entiers positifs  $s_1, \dots, s_m$  et un entier  $t$

**Question :** Existe-t-il un sous-ensemble  $S' \subseteq S$  tel que  $\sum_{x \in S'} x = t$  ?

**Question 2** En déduire que 2-PARTITION est NP-difficile.

*Une indication pour cette question se trouve à la toute fin du sujet. Il n'y a aucun bonus/malus à l'utiliser ou non, mais si vous voulez vous préparer aux concours les plus difficiles, commencez par chercher par vous-mêmes.*

**Question 3** Conclure.

### Solution de l'Exercice 1

**Question 1** On donne comme certificat pour une instance positive un sous-ensemble  $E'$  tel que  $\sum_{a \in E'} a = \sum_{a \notin E'} a$ . Ce certificat est de taille polynomiale en la taille de l'instance car  $|E'| \leq |E|$ .

A l'aide de ce certificat, on peut vérifier en temps polynomial qu'une instance est positive : étant donné  $a_1, a_2, \dots, a_n$  et  $E'$ , on calcule en temps polynomial  $\sum_{a \in E'} a$  et  $\sum_{a \notin E'} a$ , par exemple en un parcours de  $\{a_1, \dots, a_n\}$ , et on vérifie l'égalité.

**Question 2** On réduit polynomialement SUBSET-SUM à 2-PARTITION. Ainsi on aura :

$\text{SUBSET-SUM} \leq_p \text{2-PARTITION}$  et comme SUBSET-SUM est NP-dur, on pourra en déduire que 2-PARTITION l'est également.

Soit  $(S, t)$  une instance de SUBSET-SUM. On construit une instance de 2-PARTITION en prenant l'ensemble des entiers de  $S$  auxquels on ajoute les entiers  $2t$  et  $\sum_{i=1}^m s_i$  : on pose  $E = S \cup \{2t, \sum_{i=1}^m s_i\}$ .

On peut construire  $E$  en temps polynomial (la somme  $\sum_{i=1}^m s_i$  se calcule en temps linéaire, de même pour  $2t$ ).

Montrons que ces deux instances sont équivalentes, i.e. que  $(S, t)$  est une instance positive de SUBSET-SUM si et seulement si  $E$  est une instance positive de 2-PARTITION.

$\Rightarrow$  : Supposons que  $(S, t)$  est une instance positive de SUBSET-SUM. Alors il existe  $S' \subseteq S$  tel que  $\sum_{x \in S'} x = t$ .

On pose  $E' = S' \cup \{\sum_{i=1}^m s_i\}$ . On a alors  $E \setminus E' = S \setminus S' \cup \{2t\}$  et :

$$\sum_{a \in E'} a = \sum_{x \in S'} x + \sum_{i=1}^m s_i = \sum_{i=1}^m s_i + t$$

$$\text{et } \sum_{a \notin E'} a = \sum_{s \notin S'} s + 2t = (\sum_{s \notin S'} s + t) + t = (\sum_{s \notin S'} s + \sum_{x \in S'} x) + t = \sum_{i=1}^m s_i + t.$$

Donc  $E$  est une instance positive de 2-PARTITION.

$\Leftarrow$  : Réciproquement, si  $E$  est une instance positive de 2-PARTITION alors il existe  $E' \subseteq E$  tel que  $\sum_{a \in E'} a = \sum_{a \notin E'} a$ .

Or comme  $\sum_{a \in E} a = \sum_{s \in S} s + 2t + \sum_{i=1}^m s_i = 2(\sum_{i=1}^m s_i + t)$ , on sait que  $E'$  et  $E \setminus E'$  somment à  $\sum_{i=1}^m s_i + t$  chacun.

Sans perte de généralité, quitte à inverser  $E'$  et  $E \setminus E'$ , supposons que  $E'$  contient l'élément  $\sum_{i=1}^m s_i$ . Alors comme tous les entiers de  $S$  sont positifs,  $E'$  ne contient pas  $2t$  (sinon la somme de ses éléments dépasserait

$$\sum_{i=1}^m s_i + t). \text{ Considérons alors } S' = E' \setminus \left\{ \sum_{i=1}^m s_i \right\}.$$

Par ce qui précède, on sait que  $S' \subseteq S$  (car  $S'$  ne contient ni  $2t$  ni  $\sum_{i=1}^m s_i$ ).

De plus,  $S'$  somme à  $\sum_{i=1}^m s_i + t - \sum_{i=1}^m s_i = t$ . Donc  $(S, t)$  est une instance positive de SUBSET-SUM.

**Question 3** 2-PARTITION appartient à NP et est NP-difficile, donc 2-PARTITION est bien NP-complet.



## Partie I - Application directe du cours

**Question 4** Montrer que si  $\mathcal{L}$  est un langage régulier,  $Suff(\mathcal{L})$  est un langage régulier, où  $Suff(\mathcal{L})$  est l'ensemble des suffixes du langage  $\mathcal{L}$ .

**Question 5** Donner un automate reconnaissant l'ensemble des mots sur  $\{a, b\}$  commençant par  $b$ , finissant par  $b$ , et ne contenant pas le facteur  $bb$ .

**Question 6** Montrer qu'un nombre est multiple de 3 si et seulement si la somme de ses chiffres en base 10 est elle aussi multiple de 3.

**Question 7** Existe-t-il une expression régulière pour l'ensemble des écritures décimales de multiples de 3 ?

**Question 8** Démontrer que si un langage est reconnaissable il vérifie le lemme de l'étoile.

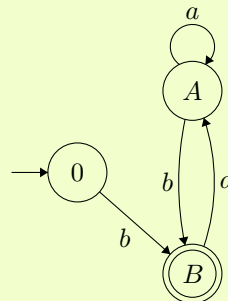
### Solution de l'Exercice 2

**Question 4** On peut le montrer par induction sur la structure d'expression régulière :

- Si  $\mathcal{L} = \emptyset$  ou  $\mathcal{L} = \{\epsilon\}$  ou  $\mathcal{L} = \{a\}$  pour un certain  $a \in \Sigma$ , le résultat est immédiat
- Si  $\mathcal{L} = \mathcal{L}(e_1 + e_2)$  avec  $\mathcal{L}(e_1)$  et  $\mathcal{L}(e_2)$  qui vérifient le résultat, comme  $Suff(\mathcal{L}) = Suff(\mathcal{L}(e_1)) \cup Suff(\mathcal{L}(e_2))$ , par hypothèse d'induction on a le résultat
- Si  $\mathcal{L} = \mathcal{L}(e_1 e_2)$  avec  $\mathcal{L}(e_1)$  et  $\mathcal{L}(e_2)$  qui vérifient le résultat, alors si  $\mathcal{L}(e_1)$  est vide le résultat est immédiat, et sinon  $Suff(\mathcal{L}) = Suff(\mathcal{L}(e_2)) \cup Suff(\mathcal{L}(e_1))\mathcal{L}(e_2)$  donc on a le résultat grâce à l'hypothèse d'induction. Si  $\mathcal{L} = \mathcal{L}(e_0^*)$  avec  $\mathcal{L}(e_0)$  qui vérifie le résultat, alors si  $\mathcal{L}(e_0) \neq \emptyset$  on a  $Suff(\mathcal{L}) = Suff(\mathcal{L}(e_0))\mathcal{L}^*$  donc on a le résultat grâce à l'hypothèse d'induction et sinon c'est immédiat ( $\emptyset^* = \{\epsilon\}$ )

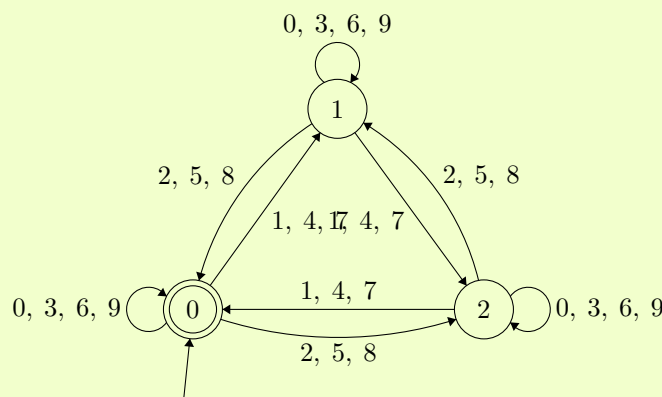
On a donc bien montré le résultat pour tout langage régulier.

**Question 5** On propose l'automate donné figure ?? (construction d'un automate local, mais sinon ça marche aussi très bien en construisant d'un côté les mots commençant et finissant par  $b$ , de l'autre les mots ne contenant pas  $bb$ , puis en faisant l'intersection des deux, par exemple)



**Question 6**  $10 \equiv 1 \pmod{3}$  donc en écrivant  $X$  comme  $d_n \dots d_0$  en base 10, on a  $X = \sum_{k=0}^n d_k 10^k \equiv \sum_{k=0}^n d_k \pmod{3}$

**Question 7** Oui, mais pour le démontrer on va plutôt montrer qu'il s'agit d'un langage reconnaissable (puis utiliser le théorème de Kleene) : l'automate donné figure ?? reconnaît ce langage (à chaque fois qu'on lit un nouveau digit  $d$ , on fait  $\times 10 + d$  sur le résultat intermédiaire, et on considère le résultat modulo 3)



**Question 8** cf cours. On considère un automate qui reconnaît le langage, il a  $N$  états, donc quand on lit un mot de  $N$  lettres on passe deux fois par le même état, i.e. on passe par un cycle dans l'automate, qu'on peut donc répéter autant de fois que souhaité avant de rejoindre un état final.

Vérifier des propriétés sur les langages reconnus par des automates ou, de manière équivalente, des propriétés sur des programmes sans mémoire, est un enjeu crucial en informatique : qu'il s'agisse de montrer la correction de programmes sans mémoire, de rechercher des failles de sécurité, ou même de trouver des optimisations, beaucoup de secteurs n'échappent pas à ces considérations. On se propose donc dans ce devoir d'explorer quelques techniques pour manipuler informatiquement les automates et explorer leurs comportements

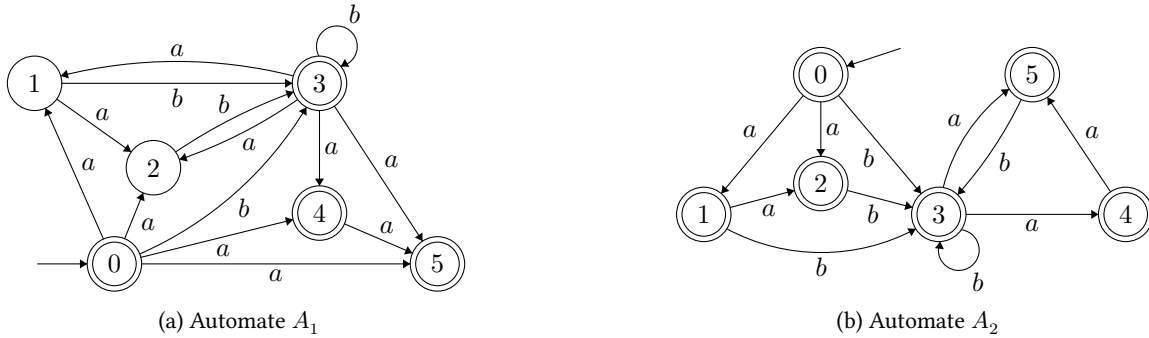
Dans tout le sujet, si l'alphabet n'est pas précisé, il s'agira par défaut de  $\Sigma = \{a, b\}^*$

## Partie II - Minimisation d'automates

### 1) Je le trouve petit, tout petit, minuscule!

Hein ? comment ? m'accuser d'un pareil ridicule ?<sup>1</sup>

On s'intéresse dans cette partie à trouver un automate déterministe le plus petit possible reconnaissant le même langage qu'un automate déterministe donné. On travaillera sur les deux automates suivants :



**Question 9** L'automate  $A_1$  de la figure 1a est-il déterministe ? Est-il complet ?

**Question 10** Déterminer et compléter l'automate  $A_1$  de la figure 1a .

**Question 11** Déterminer et compléter de même l'automate  $A_2$  de la figure 1b .

Pour un automate fini déterministe complet  $\mathcal{A} = (Q, \Sigma, q_i, F, \delta)$ , on définit une suite de relations  $(\sim_n)_{n \in \mathbb{N}}$  sur  $Q \times Q$  par récurrence de la manière suivante :

$$\begin{aligned} \forall p, q \in Q, p \sim_0 q &\Leftrightarrow (p \in F \Leftrightarrow q \in F) \\ \forall n \in \mathbb{N}, \forall p, q \in Q, p \sim_{n+1} q &\Leftrightarrow (p \sim_n q \text{ et } \forall a \in \Sigma, \delta(p, a) \sim_n \delta(q, a)) \end{aligned}$$

**Question 12** Démontrer que  $\sim_0$  est une relation d'équivalence sur  $Q$ .

**Question 13** En déduire que pour tout  $n \in \mathbb{N}$ ,  $\sim_n$  est une relation d'équivalence sur  $Q$ .

Dans toute la suite, on décrira une relation d'équivalence sur un ensemble fini indexé par des entiers par la donnée de ses classes, et on désignera une classe par le plus petit index appartenant à cette classe. Ainsi, le tableau donné figure 2 représente une relation d'équivalence sur l'ensemble des sommets d'un automate numérotés de 0 à 6, avec une classe contenant 0, 2 et 3, une classe contenant 1 et 5, et une classe contenant 4 et 6. Evidemment, si on a plusieurs relations à représenter sur le même ensemble, on peut se contenter de faire plusieurs lignes de relations.

numéro	0	1	2	3	4	5	6
relation	0	1	0	0	4	1	4

FIGURE 2 – Exemple de représentation d'une relation

**Question 14** Donner  $\sim_n$  pour  $n$  allant de 0 à 5 pour l'automate obtenu à la question 1.10 (on renommara les sommets de 0 à  $|Q| - 1$  pour simplifier). Que remarque-t-on ?

**Question 15** Démontrer que pour tout  $n \in \mathbb{N}$ ,  $\sim_n = \sim_{n+1} \Rightarrow \sim_{n+1} = \sim_{n+2}$ .

**Question 16** En déduire que s'il existe  $n \in \mathbb{N}$  tel que  $\sim_n = \sim_{n+1}$  alors pour tout  $k \in \mathbb{N}$ ,  $\sim_{n+k} = \sim_n$

**Question 17** Démontrer que pour tout  $n \in \mathbb{N}$ ,  $\sim_{n+1} \subset \sim_n$ . En déduire qu'il existe  $n_0 \in \mathbb{N}$  tel que pour tout  $n \geq n_0$ ,  $\sim_n = \sim_{n_0}$

1. (Edmond Rostand, *Cyrano de Bergerac*)

On note  $\sim = \sim_{n_0}$  (avec  $n_0$  construit à la question précédente), et on cherche désormais à montrer que l'automate obtenu en quotientant l'automate de départ par  $\sim$  reconnaît le même langage : pour  $q \in Q$ , on note  $\bar{q}$  la classe d'équivalence de  $q$  pour  $\sim$ , et on définit  $\mathcal{A}/\sim = (\{\bar{q} \mid q \in Q\}, \Sigma, \bar{q}_i, \{\bar{q} \mid q \in F\}, \{(\bar{q}, a, \bar{q}') \mid (q, a, q') \in \delta\})$

**Question 18** Calculer  $\sim$  pour les automates obtenus aux questions 10 et 11 puis les automates quotients.

**Question 19** Soit  $\mathcal{A}$  un automate fini déterministe complet. Démontrer que  $\mathcal{A}/\sim$  est déterministe et  $\mathcal{L}(\mathcal{A}) \subset \mathcal{L}(\mathcal{A}/\sim)$

Pour  $\mathcal{A} = (Q, \Sigma, q_i, F, \delta)$  un automate fini déterministe complet et  $q \in Q$ , on définit  $\mathcal{A}(q) = (Q, \Sigma, q, F, \delta)$  l'automate obtenu à partir de  $\mathcal{A}$  en désignant  $q$  comme sommet initial. On se fixe  $\mathcal{A}$  un AFD pour les questions suivantes.

**Question 20** Démontrer que pour tout  $n \in \mathbb{N}$  et  $p, q \in Q$ , si  $p \sim_n q$  alors  $\mathcal{L}(\mathcal{A}(p)) \cap \Sigma^n = \mathcal{L}(\mathcal{A}(q)) \cap \Sigma^n$ .

**Question 21** Démontrer la réciproque.

**Question 22** En déduire que pour tous  $p, q \in Q$ ,  $p \sim q \Leftrightarrow \mathcal{L}(\mathcal{A}(p)) = \mathcal{L}(\mathcal{A}(q))$ .

**Question 23** En déduire que  $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}/\sim)$

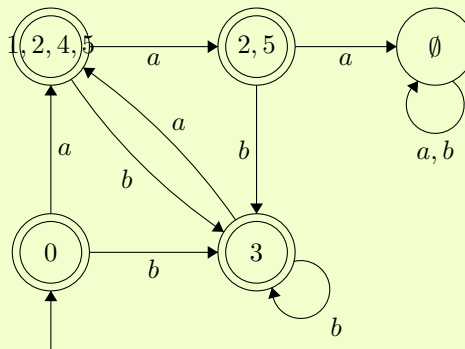
**Question 24** En déduire que les automates des figures 1a et 1b reconnaissent le même langage.

**Question 25** Proposer une démarche pour trouver la plus petite expression régulière décrivant un langage régulier donné (on ne demande pas une approche optimisée<sup>2</sup>).

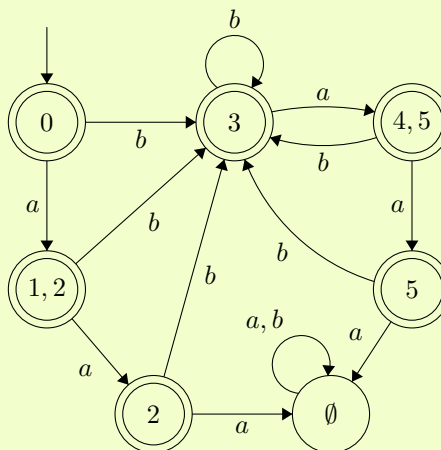
### Solution de l'Exercice 3

**Question 9** Il n'est pas déterministe car depuis 0 on peut rejoindre plusieurs états en lisant  $a$ , et il n'est pas complet car depuis 4 on ne peut pas lire de  $b$ .

**Question 10** On construit l'automate des parties accessible, donné ci-dessous :



**Question 11** On construit l'automate des parties accessible, donné ci-dessous :



2. et vu qu'il s'agit d'un problème PSPACE-complet, il y a de bonnes chances que la solution proposée, même naïve, soit en fait optimale

**Question 12** Il suffit de démontrer qu'elle est réflexive ( $\forall p \in Q, p \in F \Rightarrow p \in F$ , ok), symétrique (soient  $p, q \in F, p \sim_0 q \Rightarrow (p \in F \Leftrightarrow q \in F) \Rightarrow (q \in F \Leftrightarrow p \in F) \Rightarrow q \sim_0 p$ , ok) et transitive (soient  $p, q, r \in Q, p \sim_0 q$  et  $q \sim_0 r \Rightarrow (p \in F \Leftrightarrow q \in F \text{ et } q \in F \Leftrightarrow r \in F) \Rightarrow (p \in F \Leftrightarrow r \in F) \Rightarrow p \sim_0 r$ , ok).

**Question 13** On montre le résultat par récurrence sur  $n$  :

- L'initialisation a déjà été traitée à la question précédente.
  - On suppose le résultat vrai pour un certain  $n \in \mathbb{N}$  : soient  $p, q, r \in Q$  :
    - $p \sim_{n+1} p \Leftrightarrow p \sim_n p$  et  $\forall a \in \Sigma, \delta(p, a) \sim_n \delta(p, a)$ , ce qui est toujours vérifié car d'après HR,  $\sim_n$  est une relation d'équivalence, donc réflexive.
    - La symétrie et la transitivité fonctionnent exactement de la même manière en utilisant l'HR pour dire que  $\sim_n$  est réflexive puis transitive, au besoin en utilisant l'associativité et la commutativité du  $\text{et}$  et .
- Ainsi,  $\sim_{n+1}$  est une relation d'équivalence, ce qui conclut la récurrence.

**Question 14** Le résultat est donné figure 14. On remarque que la suite semble stationnaire à partir du rang 3

Sommets	0	1	2	3	4
$\sim_0$	0	0	0	0	4
$\sim_1$	0	0	2	0	4
$\sim_2$	0	1	2	0	4
$\sim_3$	0	1	2	0	4
$\sim_4$	0	1	2	0	4
$\sim_5$	0	1	2	0	4

**Question 15** Soit  $n \in \mathbb{N}$  tel que  $\sim_n = \sim_{n+1}$  : soient  $p, q \in Q$  :

$$\begin{aligned}
 p \sim_{n+2} q &\Leftrightarrow p \sim_{n+1} q \text{ et } \forall a \in \Sigma, \delta(p, a) \sim_{n+1} \delta(q, a) \\
 &\Leftrightarrow p \sim_n q \text{ et } \forall a \in \Sigma, \delta(p, a) \sim_n \delta(q, a) \\
 &\Leftrightarrow p \sim_{n+1} q
 \end{aligned}$$

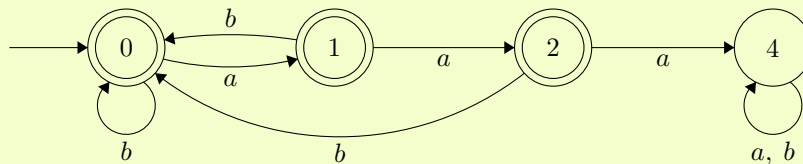
donc  $\sim_{n+1} = \sim_{n+2}$

**Question 16** On suppose qu'il existe un tel  $n \in \mathbb{N}$ . Montrons par récurrence forte sur  $k \in \mathbb{N}^*$  que  $\sim_n = \sim_{n+k}$  :

- L'initialisation ( $k = 1$ ) est triviale
- On suppose avoir le résultat pour tous les rangs inférieurs à un certain  $k > 0$  : en particulier, on a  $\sim_{n+k-1} = \sim_{n+k} = \sim_n$  donc  $\sim_n = \sim_{n+k} = \sim_{n+k+1}$  (question précédente), ce qui conclut la récurrence

**Question 17** Par définition,  $p \sim_{n+1} q \Rightarrow p \sim_n q$  donc si  $(p, q) \in \sim_{n+1}$ , on a  $(p, q) \in \sim_n$  d'où l'inclusion. Ainsi,  $(\sim_n)_{n \in \mathbb{N}}$  est une suite décroissante d'entiers naturels, donc elle ne peut être strictement décroissante : il existe  $n_0 \in \mathbb{N}$  tel que  $\sim_{n_0} = \sim_{n_0+1}$  et la question précédente nous donne alors le résultat

**Question 18** On obtient dans les deux cas le même automate, donné ci-dessous :



**Question 19** On a par définition de  $\sim$  :  $p \sim q \Leftrightarrow p \sim q$  et  $\forall a \in \Sigma, \delta(p, a) \sim \delta(q, a)$ , donc pour  $q_1, q_2 \in Q$  et  $a \in \Sigma$  tels que  $q_1 \sim q_2$ , on a en particulier  $\delta(q_1, a) \sim \delta(q_2, a)$  donc  $|\{q' \mid \exists q_2 \in \bar{q}_1 : (q_2, a, q') \in \delta\}| \leq 1$  : l'automate est donc bien déterministe.

De plus, si  $u \in \mathcal{L}(\mathcal{A})$ , il existe  $q_0 \xrightarrow{u_1} q_1 \dots \xrightarrow{u_{|u|}} q_{|u|}$  un chemin acceptant d'étiquette  $u$  dans  $\mathcal{A}$ , et ainsi, par construction,  $\bar{q}_0 \xrightarrow{u_1} \bar{q}_1 \dots \xrightarrow{u_{|u|}} \bar{q}_{|u|}$  est un chemin acceptant d'étiquette  $u$  dans  $\mathcal{A}/\sim$ , d'où le résultat.

**Question 20** On montre le résultat par récurrence :

- Pour  $n = 0$ , on a le résultat, car on ne reconnaît  $\epsilon$  que si on commence dans un état final

- On suppose le résultat vrai pour un certain  $n$  : en particulier, on a  $p \sim_{n+1} q \Rightarrow \forall a \in \Sigma, \delta(p, a) \sim_n \delta(q, a)$ , donc en prenant  $u = av \in \mathcal{L}(\mathcal{A}(p))$  de longueur  $n+1$  (avec  $v$  de longueur  $n$ ), on a  $v \in \mathcal{L}(\mathcal{A}(\delta(p, a))) = \mathcal{L}(\mathcal{A}(\delta(q, a)))$  donc  $u = av \in \mathcal{L}(\mathcal{A}(q))$  (et symétriquement), ce qui conclut la récurrence.

**Question 21** On montre le résultat comme précédemment, par récurrence :

- L'initialisation est la même qu'à la question précédente
- On suppose le résultat vrai pour un certain  $n$ , on se fixe  $p, q \in Q$  tels que  $\mathcal{L}(\mathcal{A}(p)) \cap \Sigma^{n+1} = \mathcal{L}(\mathcal{A}(q)) \cap \Sigma^{n+1}$ . En particulier, ça implique que pour tout  $a \in \Sigma$ ,  $\mathcal{L}(\mathcal{A}(\delta(p, a))) \cap \Sigma^n = \mathcal{L}(\mathcal{A}(\delta(q, a))) \cap \Sigma^n$ , et on conclut en appliquant l'hypothèse de récurrence, quantifiée avec un  $\forall$  sur les états.

**Question 22** Dans un sens,  $p \sim q \Rightarrow \forall n \in \mathbb{N}, p \sim_n q$ , ce qui nous donne d'après la question précédente  $\forall n \in \mathbb{N}, \mathcal{L}(\mathcal{A}(p)) \cap \Sigma^n = \mathcal{L}(\mathcal{A}(q)) \cap \Sigma^n$ , on fait l'union de toutes les égalités et on factorise, ce qui fait apparaître  $\cap \Sigma^*$  qu'on peut simplifier, d'où le résultat. Dans l'autre sens, il suffit d'appliquer le résultat précédent avec  $n_0$  tel que défini dans la définition de  $\sim$ .

**Question 23** L'inclusion directe est vérifiée. L'inclusion réciproque se montre en raisonnant sur la longueur du mot : le cas  $u = \epsilon$  se fait avec  $\epsilon$ , qu'on ne peut reconnaître que si  $q_i \in F$ , et le reste en considérant un mot  $u = av$  de longueur  $n+1$  reconnu par  $\mathcal{A}/\sim$  : il existe un chemin dans  $\mathcal{A}/\sim$  acceptant d'étiquette  $av$ , donc après la première transition, on doit lire  $v$  depuis un certain sommet en relation avec un sommet  $q'$  qu'on peut atteindre depuis  $q_i$  en lisant  $a$  : depuis ce sommet on peut accepter  $v$ , donc depuis  $q'$  également, ce qui permet de conclure.

**Question 24** L'automate de la figure 1a reconnaît le même langage que son automate des parties, lui-même reconnaissant le même langage que l'automate construit à la question précédente. De même pour celui de la figure 1b. On conclue donc que tous ces automates reconnaissent le même langage.

**Question 25** On pourrait se donner un ordre sur les expressions régulières (typiquement en associant à chaque symbole un chiffre, ça nous donne un nombre), puis énumérer toutes les expressions régulières jusqu'à l'expression donnée, en construisant à chaque fois l'automate de Glushkov associé, puis en le déterminisant puis en le quotientant par  $\sim$  on peut vérifier s'il reconnaît le même langage que celui décrit par l'expression régulière initiale.

## 2) Morphismes d'automates

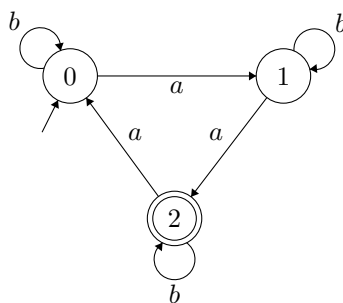


FIGURE 3

Cette partie est à réaliser en OCaml. On autorisera des crashes d'exécutions dans le cas d'automates non complets (on ne demande pas de rattraper les exceptions). On se donne les types suivants :

OCaml

```

1 type lettre = A | B
2 type etat = int
3 type mot = lettre list
4 type automate = {
5   n : int; (* nombre d'états *)
6   init : etat; (* état initial *)
7   final : etat -> bool; (* renvoie true si l'état est final et false sinon *)
8   delta : etat -> lettre -> etat (* fonction de transition *)
9 }

```

Ainsi, par exemple, l'expression suivante<sup>3</sup> définit l'automate de la figure 3 :

OCaml

```
1 let a = {
2   n = 3;
3   init = 0;
4   final = function | 2 -> true
5               | _ -> false;
6   delta = function | 0 -> (function A -> 1 | B -> 0)
7               | 1 -> (function A -> 2 | B -> 1)
8               | 2 -> (function A -> 0 | B -> 2)
9 }
```

Dans cette partie, on considérera travailler sur des automates obtenus avec la technique présentée dans la partie précédente.

**Question 26** Ecrire une expression représentant un des automates obtenus à la question 18 de la partie précédente.

**Question 27** Ecrire une fonction de signature : `val evalue : automate -> mot -> bool` telle que l'évaluation de l'expression `evalue auto m` soit `true` si `m` est accepté par l'automate et `false` (ou un crash) sinon.

La sous-section précédente proposait une façon de construire des automates reconnaissant un certain langage. On peut montrer (mais on ne le fera pas) que l'automate ainsi construit était l'automate des langages résiduels de l'automate de départ, dont la forme est unique à isomorphisme près. Même si on ne le montrera pas ici, on propose de commencer dans cette partie à étudier un peu la notion d'isomorphisme d'automates.

Soient  $\mathcal{A}_1 = (Q, \Sigma, q_i, F, \delta)$  et  $\mathcal{A}' = (Q', \Sigma, q'_i, F', \delta')$  deux automates déterministes. On dit que  $g : Q \rightarrow Q'$  réalise un morphisme d'automates de  $\mathcal{A}$  dans  $\mathcal{A}'$  si :

- $g(q_i) = q'_i$
- $g(F) \subset F'$
- Pour tous  $q \in Q, a \in \Sigma$  on a  $g(\delta(q, a)) = \delta'(g(q), a)$

Enfin, on appelle isomorphisme d'automates un morphisme d'automates bijectif dont la réciproque est aussi un morphisme d'automates.

**Question 28** Montrer que s'il existe un morphisme d'automates de  $\mathcal{A}$  dans  $\mathcal{A}'$  avec  $\mathcal{A}$  et  $\mathcal{A}'$  deux automates déterministes, alors  $\mathcal{L}(\mathcal{A}) \subset \mathcal{L}(\mathcal{A}')$ .

On cherche désormais à concevoir un algorithme permettant de déterminer s'il existe un morphisme d'automates entre deux automates. On représente par la suite un morphisme sous la forme d'un tableau d'états  $g$  indexés par les états, tel que pour tout  $q \in Q$ ,  $g.(q)$  contient un état  $q'$  si  $q'$  est l'image de  $q$  par l'isomorphisme  $g$  et contient (-1) si l'image de  $q$  n'est pas encore déterminée. On se donne donc le type supplémentaire suivant :

OCaml

```
1 type morphisme = etat array
```

**Question 29** Proposer une fonction de signature :

`val verifie : automate -> automate -> morphisme -> bool` telle que l'évaluation de `verifie a1 a2 g` renvoie `true` si  $g$  est un morphisme d'automate de  $a1$  dans  $a2$  et `false` sinon.

**Question 30** Proposer une fonction de signature : `val attribue : isomorphisme -> int -> int -> bool` telle que l'évaluation de `attribue g q q'` met `iso` à jour pour ajouter l'information que l'image de  $q$  est  $q'$  si  $q$  n'avait pas encore d'image, et vaut `true` s'il n'y a pas de problème ( $q$  n'avait pas encore d'image ou l'image de  $q$  était déjà  $q'$ ) et `false` en cas de conflit ( $q$  avait déjà une image différente de  $q'$ ).

Pour construire un morphisme d'automates de  $\mathcal{A}$  dans  $\mathcal{A}'$ , on se propose de procéder de la manière suivante : on effectue un parcours de l'automate  $\mathcal{A}$  depuis l'état initial  $q_i$  (qui doit être envoyé sur  $q'_i$ ). On se déplace en parallèle dans l'automate  $\mathcal{A}'$  en suivant les mêmes transitions que dans  $\mathcal{A}$  (i.e. étiquetées par les mêmes lettres) et on met à jour le morphisme  $g$  au fur et à mesure de ces déplacements. Si on trouve un conflit dans les images de  $g$  pendant le parcours, on peut conclure qu'il n'existe pas de morphisme entre  $\mathcal{A}$  et  $\mathcal{A}'$ . Sinon, on peut conclure que tout morphisme entre  $\mathcal{A}$  et  $\mathcal{A}'$  doit avoir les mêmes images que  $g$  (condition nécessaire).

Il ne reste alors plus qu'à vérifier que le  $g$  construit est un bien un morphisme (condition suffisante).

3. On rappelle que le mot-clé `function` est équivalent à `fun x -> match x with`



**Question 31** Proposer une fonction de signature :

`val parcours : automate -> automate -> etat array option` telle que `parcours a1 a2` effectue le parcours de  $\mathcal{A}$  expliqué ci-dessus et renvoie la fonction  $g$  construite pendant ce parcours, ou `None` si un conflit a été rencontré pendant sa construction.

**Question 32** En déduire une fonction de signature :

`val existe_morphisme : automate -> automate -> bool` tel que `existe_morphisme a1 a2` renvoie `true` s'il existe un morphisme de  $a1$  vers  $a2$  et `false` sinon.

### Solution de l'Exercice 4

**Question 26** On propose l'expression suivante :

OCaml : cor-ocaml.ml

```
1 let a15 = {n = 4; init = 0; final = (function 0 | 1 | 2 -> true | _ -> false);
2   delta = function | 0 -> (function | A -> 1 | B -> 0)
3               | 1 -> (function | A -> 2 | B -> 0)
4               | 2 -> (function | A -> 3 | B -> 0)
5               | 3 -> (function | A -> 3 | B -> 3)
6               | _ -> failwith "etat_invalide"
7 }
```

**Question 27** On lit le mot de manière récursive en parcourant la liste des lettres et en appliquant la fonction de transition. La version donnée ici est récursive terminale avec un accumulateur :

OCaml : cor-ocaml.ml

```
1 let evaluer a mot =
2   let rec lit q m = match m with
3     [] -> a.final q
4     | h::t -> lit (a.delta q h) t
5   in lit a.init mot
```

**Question 28** Il suffit d'appliquer la définition de morphisme : soit  $u \in \mathcal{L}(\mathcal{A})$ , il existe  $q_i \xrightarrow{u_1} q_1 \dots \xrightarrow{u_{|u|}} q_{|u|}$  un chemin acceptant d'étiquette  $u$  dans  $\mathcal{A}$ , donc par définition de morphisme,  $g(q_i) \xrightarrow{u_1} g(q_1) \dots \xrightarrow{u_{|u|}} g(q_{|u|})$  est un chemin dans  $\mathcal{A}'$ , et il est acceptant car  $g(q_i) = q'_i$  et  $q_{|u|} \in F$  donc  $g(q_{|u|}) \in F'$ .

**Question 29** On propose la fonction suivante :

OCaml : cor-ocaml.ml

```
1 let verifie a1 a2 g =
2   let rec verifie_etats q =
3     if q = a1.n then true (* cas de base *)
4     else g.(q) <> -1 (* image définie *)
5     && g.(a1.delta q A) = a2.delta g.(q) A (* transitions *)
6     && g.(a1.delta q B) = a2.delta g.(q) B
7     && (not (a1.final q) || a2.final g.(q)) (* q final => g.(q) final *)
8     && verifie_etats (q+1) (* appel récursif sur les états restants à vérifier *)
9   in g.(a1.init) = a2.init && (verifie_etats 0)
```

**Question 30** On propose la fonction suivante :

OCaml : cor-ocaml.ml

```
1 let attribue g q q' = match g.(q) with
2   | -1 -> g.(q) <- q'; true
3   | q2 when q' = q2 -> true
4   | _ -> false
```

**Question 31** On propose l'implémentation suivante :

OCaml : cor-ocaml.ml

```

1 exception Break (* on propose ici une solution avec une exception *)
2 let parcours a1 a2 =
3   let g = Array.make a1.n (-1) in
4   let vus = Array.make a1.n false in
5   let rec traite q q' =
6     vus.(q) <- true;
7     if (attribue g q q') then begin
8       if not vus.(a1.delta q A) then traite (a1.delta q A) (a2.delta q' A);
9       if not vus.(a1.delta q B) then traite (a1.delta q B) (a2.delta q' B)
10    end
11   else raise Break (* conflit d'images dans g : abandon *)
12   in try traite a1.init a2.init; Some g
13 with Break -> None

```

OCaml : cor-ocaml.ml

**Question 32** On propose l'implémentation suivante :

```

1 let existe_morphisme a1 a2 =
2   let g_opt = parcours a1 a2 in
3   match g_opt with
4   None -> false
5   | Some g -> verifie a1 a2 g

```

## Partie III - Automates et multiplication matricielle

Cette partie est à traiter en C. Par souci d'optimisation, on considérera une matrice comme un tableau à une dimension constitué de ses lignes les unes après les autres : ainsi,  $M_{ij}$  sera contenu dans  $m[i*n+j]$  si  $M$  est de dimension  $n \times n$ .

On s'intéresse dans cette partie à l'optimisation du produit matriciel : celui-ci a en effet une place de choix dans la manipulation pratique d'automates, surtout s'ils sont non déterministes, car alors en notant  $B = (b_i)_{0 \leq i \leq n-1} \in \{0,1\}^n$  le vecteur colonne listant les états sur lesquels on peut se trouver à un instant donné, on peut construire une famille de matrices (les matrices d'adjacence du graphe obtenu à partir de l'automate en ne conservant que les arêtes portant une certaine étiquette)  $(M_a)_{a \in \Sigma}$  telle que les coefficients non nuls dans  ${}^t M_a B$  correspondent aux états dans lesquels on peut se trouver à l'instant suivant si on a lu  $a$

Par exemple, les matrices pour l'automate donné figure 3 sont :  $M_a = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$ ,  $M_b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Un algorithme naïf pour effectuer une multiplication matricielle serait d'appliquer la formule  $c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$ , mais cet algorithme a une complexité en  $\Theta(n^3)$ . On cherche comment optimiser cette procédure au moyen de l'algorithme de Strassen (algorithme 4).

**Entrées :** Deux matrices  $A$  et  $B$  de taille carrée  $2^k$  (pour simplifier)

```

1 si  $k \leq 0$  alors
2   | renvoyer  $A \times B$  //Produit de deux entiers
3 On note  $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$  et  $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$ 
4  $M_1 \leftarrow \text{Strassen}(A_{11} + A_{22}, B_{11} + B_{22}, k-1)$ 
5  $M_2 \leftarrow \text{Strassen}(A_{21} + A_{22}, B_{11}, k-1)$ 
6  $M_3 \leftarrow \text{Strassen}(A_{11}, B_{12} - B_{22}, k-1)$ 
7  $M_4 \leftarrow \text{Strassen}(A_{22}, B_{21} - B_{11}, k-1)$ 
8  $M_5 \leftarrow \text{Strassen}(A_{11} + A_{12}, B_{22}, k-1)$ 
9  $M_6 \leftarrow \text{Strassen}(A_{21} - A_{11}, B_{11} + B_{12}, k-1)$ 
10  $M_7 \leftarrow \text{Strassen}(A_{12} - A_{22}, B_{21} + B_{22}, k-1)$ 
11 renvoyer  $\begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix}$ 
```

**Algorithme 4** Algorithme de Strassen

**Question 33** Justifier que l'algorithme 4 termine sur toute entrée.

**Question 34** Vérifier que pour  $k > 0$ , en supposant les calculs corrects sur les sous-matrices, on renvoie bien  $A \times B$ .

**Question 35** Démontrer que l'algorithme de Strassen est totalement correct.

**Question 36** En notant  $T(k)$  le nombre d'opérations arithmétiques réalisées pour un produit de matrices de taille  $2^k$ , démontrer que  $T(k+1) = 7T(k) + 18 \times 4^k$ .

**Question 37** En déduire que pour tout  $k \in \mathbb{N}$ , on a  $T(k) = 7^k T(0) + \sum_{i=0}^{k-1} 18 \times 7^i \times 4^{k-i}$ .

**Question 38** Conclure que pour tout  $k \in \mathbb{N}$ , on a  $T(k) = \theta(7^k)$ , puis qu'il est possible de faire une multiplication de matrices carrées de taille  $n$  en  $\mathcal{O}(n^{\log_2(7)})$ .

**Question 39** Ecrire une fonction de prototype `int *create_zero_matrix(int n);` qui permet de créer une nouvelle matrice carrée de taille  $n$  dont tous les coefficients sont nuls.

**Question 40** Ecrire une fonction de prototype : `int *sum_and_sub(int *A, int *B, int *C, int n);` qui, étant données trois matrices  $A, B$  et  $C$  de taille  $n$ , renvoie une nouvelle matrice contenant  $A + B - C$ . On s'autorise à passer en paramètre le pointeur nul pour  $B$  et  $C$  si on veut passer la matrice nulle comme paramètre.

**Question 41** Ecrire une fonction de prototype : `int *extract(int *A, int i, int j, int n, int m);` renvoyant une nouvelle matrice de taille  $m$  qui est le bloc de  $A$  commençant<sup>4</sup> en position  $(i, j)$ .  $A$  est de taille  $n$ . On ne s'occupera pas de s'assurer que ça ne déborde pas.

4. i.e. dont le coin en haut à gauche était

**Question 42** Ecrire une fonction de prototype : `void inscribe(int *A, int *B, int i, int j, int n, int m);` modifiant  $A$  en y écrivant un bloc  $B$  de taille  $m$  à partir de la position  $(i, j)$ <sup>5</sup>.  $A$  est de taille  $n$ . On ne s'occupera pas de s'assurer que ça ne déborde pas.

**Question 43** Ecrire une fonction de prototype : `int *strassen(int *A, int *B, int n);` réalisant la multiplication de deux matrices  $A$  et  $B$  de taille  $n$  et renvoyant le résultat. On supposera pour simplifier que  $n$  est toujours une puissance de 2 et on ne s'occupera pas de le vérifier.

### Solution de l'Exercice 5

**Question 33** On considère la suite des valeurs de  $k$ , qui sont strictement décroissantes à chaque appel récursif, et comme  $k$  est minoré par 0, on a le résultat.

**Question 34** Il suffit de développer les produits puis de simplifier pour vérifier qu'on retombe bien sur les mêmes expressions que celles données par une multiplication par blocs classique. C'est un peu fastidieux, mais rien de difficile.

**Question 35** La terminaison a été traitée deux questions plus tôt. On montre la correction par récurrence sur  $k$ , en quantifiant  $A$  et  $B$  par un  $\forall$  : l'initialisation ( $k = 0$ ) marche bien, car on est dans le cas  $k \leq 0$  donc on fait juste un produit de scalaires, ce qui est correct, et l'hérédité se fait en utilisant la question précédente, et en utilisant l'hypothèse de récurrence pour affirmer que les multiplications des blocs ont été faites correctement.

**Question 36** On effectue 7 appels récursifs, chacun réalisant  $T(k)$  opérations arithmétiques, et il reste 18 sommes et soustractions de matrices carrées de taille  $2^k$ , donc contenant  $4^k$  coefficients, soit  $18 \times 4^k$  opérations arithmétiques au total, d'où le résultat.

**Question 37** On procède par récurrence : c'est vrai pour  $k = 0$ , et pour l'hérédité il suffit de réinjecter l'expression de  $T(k)$  que nous donne l'hypothèse de récurrence dans l'expression de  $T(k + 1)$ .

**Question 38** En faisant sortir  $18 \times 4^k$  de la somme, on reconnaît une somme géométrique de raison  $7/4$  : on obtient donc que  $T(k) = \theta(7^k) + \theta(7^k - 4^k)$ , et  $4^k = o(7^k)$  d'où le résultat. Ensuite, pour une matrice de taille  $n$ , quitte à ajouter des lignes et colonnes nulles, on peut la placer dans une matrice de taille  $2^{\lceil \log_2(n) \rceil}$ . On exécute donc la multiplication en  $\theta(7^{\lceil \log_2(n) \rceil}) = \theta(7^{\log_2(n)+1}) = \theta(7^{\log_2(n)}) = \theta(n^{\log_2(7)})$ .

**Question 39** On propose l'implémentation suivante :

C : cor-c.c

```
1 int *create_zero_matrix(int n){
2     int *ret = (int *)malloc(n * n * sizeof(int));
3     for(int i = 0; i < n*n; i+=1)
4         ret[i] = 0;
5     return ret;
6 }
```

**Question 40** On propose l'implémentation suivante :

C : cor-c.c

```
1 int *sum_and_sub(int *A, int *B, int *C, int n){
2     int *ret = create_zero_matrix(n);
3     for(int i = 0; i < n * n; i+=1){
4         ret[i] = A[i];
5         if(B != NULL) ret[i] += B[i];
6         if(C != NULL) ret[i] -= C[i];
7         // et hors de question que quiconque m'écrive :
8         // ret[i] = A[i] + (B != NULL ? B[i] : 0) - (C != NULL ? C[i] : 0);
9     }
10    return ret;
11 }
```

**Question 41** On propose l'implémentation suivante :

5. i.e. réaliser l'inverse de ce qu'on a fait à la question précédente

C : cor-c.c

```

1 int *extract(int *A, int i, int j, int n, int m){
2     int *ret = create_zero_matrix(m);
3     for(int x = 0; x < n; x+=1){
4         for(int y = 0; y < n; y+=1){
5             ret[x * m + y] = A[(x+i)*n + (y+j)],
6         }
7     }
8     return ret;
9 }

```

**Question 42** On propose l'implémentation suivante :

C : cor-c.c

```

1 void inscribe(int *A, int *B, int i, int j, int n, int m){
2     for(int x = 0; x < n; x += 1){
3         for(int y = 0; y < n; y += 1){
4             A[(x+i)*n + (y+j)] = B[x * m + y];
5         }
6     }
7 }

```

**Question 43** On propose l'implémentation suivante :

C : cor-c.c

```

1  int *strassen(int A, int B, int n){
2      int *ret = create_zero_matrix(n);
3      if(n == 1){
4          ret[0] = A[0] * B[0];
5          return ret;
6      }
7      int t = n/2;
8      int *A11 = extract(A, 0, 0, t);
9      int *A12 = extract(A, 0, t, t);
10     int *A21 = extract(A, t, 0, t);
11     int *A22 = extract(A, t, t, t);
12     int *B11 = extract(B, 0, 0, t);
13     int *B12 = extract(B, 0, t, t);
14     int *B21 = extract(B, t, 0, t);
15     int *B22 = extract(B, t, t, t);
16     int *A11p22 = sum_and_sub(A11, A22, NULL, t);
17     int *B11p22 = sum_and_sub(B11, B22, NULL, t);
18     int *M1 = strassen(A11p22, B11p22, t);
19     int *A21p22 = sum_and_sub(A21, A22, NULL, t);
20     int *M2 = strassen(A21p22, B11, t);
21     int *B12m22 = sum_and_sub(B12, NULL, B22, t);
22     int *M3 = strassen(A11, B12m22, t);
23     int *B21m11 = sum_and_sub(B21, NULL, B11, t);
24     int *M4 = strassen(A22, B21m11, t);
25     int *A11p12 = sum_and_sub(A11, A12, NULL, t);
26     int *M5 = strassen(A11p12, B22, t);
27     int *A21m11 = sum_and_sub(A21, NULL, A11, t);
28     int *B11p12 = sum_and_sub(B11, B12, NULL, t);
29     int *M6 = strassen(A21m11, B11p12, t);
30     int *A12m22 = sum_and_sub(A12, NULL, A22, t);
31     int *B21p22 = sum_and_sub(B21, B22, NULL, t);
32     int *M7 = strassen(A12m22, B21p22, t);
33     int *interm1 = sum_and_sub(M1, M4, M5, t);
34     int *C11 = sum_and_sub(inter1, M7, t);
35     int *C12 = sum_and_sub(M3, M5, NULL, t);
36     int *C21 = sum_and_sub(M2, M4, NULL, t);
37     int *interm2 = sum_and_sub(M1, M3, M2, t);
38     int *C22 = sum_and_sub(inter2, M6, NULL, t);
39     inscribe(ret, C11, 0, 0, t);
40     inscribe(ret, C12, 0, t, t);
41     inscribe(ret, C21, t, 0, t);
42     inscribe(ret, C22, t, t, t);
43     free(A11); free(A12); free(A21); free(A22);
44     free(B11); free(B12); free(B21); free(B22);
45     free(C11); free(C12); free(C21); free(C22);
46     free(A11p22); free(A21p22); free(A11p12); free(A21m11); free(A12m22);
47     free(B11p22); free(B12m22); free(B21m11); free(B11p12); free(B21p22);
48     free(M1); free(M2); free(M3); free(M4); free(M5); free(M6); free(M7);
49     free(inter1); free(inter2);
50     return ret;
51 }

```

**Remarque :** La gestion de la mémoire est ce qui rend le code long et fastidieux, car il faut stocker tous les résultats intermédiaires. Il est possible de faire avec beaucoup moins de malloc moyennant une gestion habile de la mémoire allouée, mais cela nécessite une approche beaucoup plus subtile (et donc dangereuse pour l'intégrité des données manipulées). Une autre approche consisterait à ne jamais libérer la mémoire, ce qui est envisageable dans un langage comme caml ou java, mais cela aboutirait à des fuites de mémoire en C, et comme on se démène pour optimiser le calcul matriciel c'est qu'on compte en faire beaucoup, l'approche consistant à dire "de toutes façons on aura terminé de s'exécuter avant de crasher par manque de RAM" est impossible.

*Indication pour la question 2. (pour montrer que 2-PARTITION est NP-difficile) : considérer les entiers  $2t$  et  $\sum_{i=1}^m s_i$ .*



# Observations sur le DS Minimisation d'automates

## I - Application du cours

- (A) Q5. Le mot  $b$  est un mot qui commence par la lettre  $b$ , finit par la lettre  $b$  et ne contient pas le facteur  $bb$ . Il doit donc être reconnu par l'automate que vous proposez.
- (B) Q6. Cette question n'est pas une application du cours à proprement parler, c'est un lemme (court) pour la question suivante, très proche de ce qu'on a fait plusieurs fois en cours (en comptant le nombre de  $a$  modulo 3 par exemple, ou les écritures binaires multiples de 3...)
- (C) Mathématiquement, ça n'a pas de sens d'écrire quelque chose comme "Soit  $x \in E \Leftrightarrow \dots$ ". Vous ne voulez pas écrire "Soit  $u \in \mathcal{L}(A) \Leftrightarrow \dots$ " mais plutôt "Soit  $u \in \Sigma^*. u \in \mathcal{L}(A) \Leftrightarrow \dots$ ".
- (D) Dans une équivalence, il faut répéter les quantificateurs, sinon on perd le sens réciproque (les objets ne sont plus définis!). Ainsi, c'est correct de procéder par implication en disant "Supposons  $P$ . Alors il existe  $v$  tel que  $Q(v)$ , et on en déduit  $R(v)$ ". Mais c'est incorrect d'écrire :

$$\begin{aligned} P &\Leftrightarrow \exists v, Q(v) \\ &\Leftrightarrow R(v) \end{aligned}$$

Ici on ne peut clairement pas "remonter" l'équivalence car dans le sens  $\Leftarrow$ ,  $v$  n'est pas défini ! Il faut répéter :

$$\begin{aligned} P &\Leftrightarrow \exists v, Q(v) \\ &\Leftrightarrow \exists v, R(v) \end{aligned}$$

**Si vous avez un doute, procédez par double implication ! C'est beaucoup plus sûr !**

## II - Minimisation d'automates

- (E) Attention à vos raisonnements par équivalences et implications. Vous vous retrouvez souvent à dire autre chose que ce que vous voulez montrer. Par exemple, pour montrer la symétrie de  $\sim_0$ , vous affirmez :  $\forall p, (p \in F \Leftrightarrow p \in F) \Leftrightarrow p \sim_0 p$ . Mais ce n'est pas cette **équivalence** que vous voulez obtenir. Là, vous n'avez fait que redire la définition de  $\sim_0$  et vous n'avez rien prouvé. Une implication n'irait pas non plus. Vous voulez, pour n'importe quel  $p$ , obtenir  $p \sim_0 p$ . Vous devez donc partir du fait que  $(p \in F \Leftrightarrow p \in F)$  est vrai et **en déduire** que  $p \sim_0 p$ . On utilisera donc les mots clés "donc" ou "d'où" plutôt qu'une implication ou une équivalence.

## III - Multiplication de matrices (Strassen)

- (F) Q36 : il y avait une erreur dans l'énoncé, c'était  $T(k+1) = 7T(k) + 18.4^k$  qu'il fallait obtenir. Une matrice de taille  $n = 2^k$  possède  $n^2 = 4^k$  coefficients, et on fait les additions et soustractions coefficient par coefficient.
- (G) Q40 (sum\_and\_sub) : l'énoncé vous dit très clairement « On s'autorise à passer en paramètre le pointeur nul pour  $B$  et  $C$  si on veut passer la matrice nulle comme paramètre ». Tout accès aux coefficients  $B[i]$  ou  $C[i]$  doit donc vérifier que le pointeur n'est pas nul, autrement vous accédez à une zone mémoire inexistante !

# INFORMATIQUE

**Durée : 4 heures**

## Consignes :

- Veuillez à numéroté vos copies en bas à droite sous le format numéro\_de\_la\_page / nombre\_total\_de\_pages.
- Ne pas utiliser de correcteur blanc.
- Des points bonus/malus de présentation pourront éventuellement être accordés.
- Veuillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- **L'usage du cours, de notes ou de tout appareil électronique est strictement interdit.**

*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.*

Le sujet est constitué de trois parties complètement indépendantes, mais les parties du problème ne sont pas indépendantes entre elles. Ce sujet existe en deux versions : étoilée et non étoilée, **au choix**.

**Commencez par indiquer sur votre copie la version du sujet que vous traitez (étoilée ou non).** Dans le doute, il est conseillé de traiter le sujet non étoilé. **Vous ne devez en aucun cas choisir le sujet étoilé pour "éviter" les questions de cours. Un tel choix sera très fortement pénalisé : si vous montrez une faiblesse sur les parties de cours et que vous choisissez le sujet étoilé, votre note sera divisée par deux.**

Voici ce que vous devez traiter pour chaque sujet :

- **Sujet étoilé** : Partie 0, Partie I (questions 3 et 5), Problème (entier, même si ce n'est pas finissable).
- **Sujet non étoilé** : Partie 0, Partie I (tout), Problème (Parties I et II). Aucun point ne sera accordé aux parties III et IV du problème, il est inutile d'essayer d'aller grappiller des points dedans ! (sauf si vous avez convenablement traité l'entièreté du reste du sujet, auquel cas vous auriez effectivement dû choisir le sujet étoilé, et je vous autorise à le faire).

Les questions de programmation doivent être traitées en langage OCaml. 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.

**Tout code qui n'est ni expliqué ni commenté se verra attribué la note de 0 et ne sera pas lu. Même pour un programme simple, vous devez indiquer (brièvement) ce que vous faites, et comment. Les fonctions les plus complexes doivent être d'abord détaillées en Français et agrémentées de commentaires.**

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`).

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  $O(f(n, m))$  où  $n$  et  $m$  sont les tailles des arguments de la fonction, et  $f$  une expression la plus simple possible. Les calculs de complexité seront justifiés succinctement.

*Ce sujet comporte 9 pages (celle-ci comprise).*

**Ne retournez pas la page avant d'y être invités.**

## Partie 0 - Un peu de NP-complétude...

Le but de cette partie est de montrer que le problème DOMINATING SET est NP-complet.

On définit le problème DOMINATING SET comme suit :

**Instance :** un graphe  $G = (S, A)$  et un entier  $K \geq 3$

**Question :** Existe-t-il un ensemble dominant de taille au plus  $K$  dans  $G$ , c'est à dire un sous-ensemble  $D \subseteq S$  à au plus  $K$  éléments tel que :  $\forall u \in S \setminus D, \exists v \in D, (u, v) \in A$

(autrement dit, intuitivement  $D$  "couvre"/"touche" tous les sommets du graphe).

**Question 1** Montrer que DOMINATING SET est dans NP.

On rappelle que le problème VERTEX COVER est NP-complet. Il est défini comme suit :

**Instance :** un graphe non orienté  $G = (S, A)$  et un entier  $k$

**Question :** existe-t-il une partie  $S' \subseteq S$  de cardinal au plus  $k$  telle que toute arête de  $A$  ait au moins une extrémité dans  $S'$  ?

**Question 2** En déduire que DOMINATING SET est NP-difficile.

*Une indication pour cette question se trouve à la toute fin du sujet. Il n'y a aucun bonus/malus à l'utiliser ou non, mais si vous voulez vous préparer aux concours les plus difficiles, commencez par chercher par vous-mêmes.*

**Question 3** Conclure.

### Solution de l'Exercice 1

**Question 1** On donne comme certificat pour une instance positive un sous-ensemble dominant  $D$  de  $S$  de taille au plus  $K$ . Ce certificat est de taille polynomiale en la taille de l'instance car  $|D| \leq |S| \leq |A|$ .

A l'aide de ce certificat, on peut vérifier en temps polynomial qu'une instance est positive : étant donné  $(G, K)$  et  $D$ , on vérifie que  $D$  contient bien au plus  $K$  éléments et que pour chaque sommet  $u \in S$  du graphe, soit  $u \in D$  soit  $u$  possède un voisin dans  $D$  (temps polynomial pour chaque sommet, donc polynomial au total).

**Question 2** On réduit polynomialement VERTEX COVER à DOMINATING SET. Ainsi on aura :

$\text{VERTEX COVER} \leq_p \text{DOMINATING SET}$  et comme VERTEX COVER est NP-dur, on pourra en déduire que DOMINATING SET l'est également.

Soit  $(G, k)$  une instance de VERTEX COVER. On construit une instance  $(G', K)$  de DOMINATING SET selon l'idée suivante : on part de  $G$ , et on représente chaque arête  $(u, v)$  de  $G$  par un nouveau sommet nommé  $uv$  dans  $G'$ . On relie le nouveau sommet  $uv$  aux sommets  $u$  et  $v$  par deux arêtes, et ce sont les seuls voisins de  $uv$ .

On pose donc :

$$G' = (S \cup \{uv \mid (u, v) \in A\}, A \cup \{(uv, u) \mid (u, v) \in A\} \cup \{(uv, v) \mid (u, v) \in A\}).$$

**FAIRE UN SCHEMA SUR VOS COPIES !!!**

Notez que les sommets isolés de  $G$  ne participent jamais à couvrir des arêtes du graphe, mais sont nécessaires pour couvrir les sommets. Il faudra les compter dans le  $K$  de  $G'$  (nombre de sommets attendus). On pose donc :

$$K = k + |\{u \in S \mid u \text{ est isolé (pas de voisin) dans } G\}|$$

On peut construire  $(G', K)$  en temps polynomial en la taille de  $(G, k)$ , on rajoute au plus  $n^2$  sommets et  $2n^2$  arêtes à  $G$  et  $K \leq 2|S|$ . On note  $I$  l'ensemble des sommets isolés.

Montrons maintenant que ces deux instances sont équivalentes, i.e. que  $(G, k)$  est une instance positive de VERTEX COVER si et seulement si  $(G', K)$  est une instance positive de DOMINATING SET.

$\Rightarrow$ : Supposons que  $(G, k)$  est une instance positive de VERTEX COVER. Alors il existe  $S' \subseteq S$  de taille au plus  $k$  tel que chaque arête de  $A$  a une extrémité dans  $S'$ .

On pose alors  $D = S' \cup I$ . Cet ensemble contient bien au plus  $K$  éléments et c'est un ensemble dominant pour  $G'$ . En effet, soit  $u \in S$ , alors on a deux possibilités :

- $u$  est isolé, auquel cas il appartient à  $D$ .
- il existe un voisin  $v$  de  $u$  dans  $G$ , i.e.  $(u, v) \in A$ . Mais comme  $S'$  est couvrant,  $u \in S'$  ou  $v \in S'$ , ainsi  $u \in D$  ou  $\exists v \in D, (u, v) \in A$ .

$\Leftarrow$ : Réciproquement, si  $(G', K)$  est une instance positive de DOMINATING SET alors il existe  $D \subseteq S$  un ensemble dominant de  $G$  de taille au plus  $k + |I|$ . Un ensemble dominant contient nécessairement tous les sommets isolés (par définition), donc  $I \subseteq D$ , et on en déduit que  $D \setminus I$  est de taille au plus  $k$ .

On peut supposer sans perte de généralité que  $D$  ne contient aucun sommet  $uv$ . En effet, si  $D$  contient un sommet  $uv$  correspondant à une arête  $(u, v) \in A$ , alors on a deux cas :

- Si  $D$  contient  $u$  ou  $v$ , on enlève  $uv$  de  $D$  et on obtient un nouvel ensemble dominant de taille au plus  $k$  (tous les sommets restent couverts, FAITES UN SCHEMA).
- Sinon, on remplace  $uv$  par  $u$  (arbitrairement), et on obtient de même un nouvel ensemble dominant.  $u$  et  $v$  sont bien toujours couverts (ainsi que  $uv$ ), et ce sont les seuls voisins de  $uv$ .

On peut donc supposer que  $D \setminus I \subseteq S$ . Mais alors  $D \setminus I$  est un ensemble couvrant de  $G$ . Soit  $(u, v)$  une arête de  $G$ , alors comme  $uv \notin D$ , et les seuls voisins de  $uv$  sont  $u$  et  $v$ , on en déduit que  $u \in D \setminus I$  ou  $v \in D \setminus I$ .

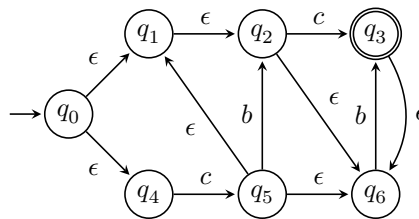
**Question 3** DOMINATING SET appartient à NP et est NP-difficile, donc DOMINATING SET est bien NP-complet.

## Partie I - Questions de cours

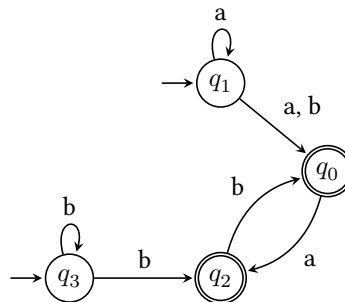
**Question 1** Comment sont définies les expressions régulières ?

**Question 2** Soient  $L_1$  et  $L_2$  deux langages rationnels. Montrer que  $L_1 \cap L_2$  est rationnel.  
On détaillera rigoureusement la construction de l'automate et on en donnera une preuve.

**Question 3** Éliminer les  $\epsilon$ -transitions de l'automate suivant, puis le déterminiser (on dessinera les deux automates obtenus), sur l'alphabet  $\{b, c\}$ . On veut un automate final complet.



**Question 4** En appliquant l'algorithme d'élimination des états, trouver une expression régulière dénotant le langage reconnu par l'automate suivant :



On dessinera tous les automates intermédiaires obtenus et on indiquera les états éliminés<sup>1</sup>.

**Question 5** Énoncer précisément le lemme de l'étoile.

Les langages suivants sont-ils rationnels sur l'alphabet  $\{a, b\}$  ? Justifier.

1.  $L_1 = \{a^n b^m \mid n < m\}$
2.  $L_2 = \{a^n b^m \mid n + m \leq 1024\}$
3.  $L_3 = \{a^n b^m \mid n \neq m\}$
4.  $L_4 = \{a^n b^m \mid n \equiv m \pmod{2}\}$

**Question 6** En utilisant l'algorithme de Berry-Sethi, trouver un automate reconnaissant le langage  $L$  dénoté par l'expression régulière suivante :  $(a|c)^*abb \mid (a|c)^*$ .

Comment s'appelle l'automate obtenu ?

**Question 7** On définit :

1. Même si ceci n'est pas demandé, vous devriez toujours le faire, pour gagner des points même en cas d'erreur à une étape.

- le **miroir** d'un mot  $w$ , noté  $\bar{w}$ , par 
$$\begin{cases} \bar{\epsilon} = \epsilon \\ \overline{a_1 \dots a_n} = a_n \dots a_1 \end{cases}$$
- le miroir d'un langage  $\mathcal{L}$  par  $\bar{\mathcal{L}} = \{\bar{w} \mid w \in \mathcal{L}\}$

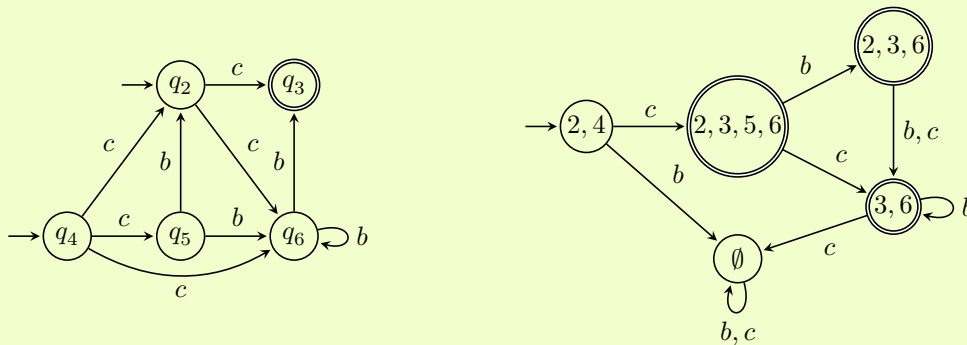
Montrer que si  $\mathcal{L}$  est reconnaissable, alors  $\bar{\mathcal{L}}$  l'est également. On pourra admettre les propriétés sur la fonction de transition étendue, à condition qu'elle soit rigoureusement énoncée.

### Solution de l'Exercice 2

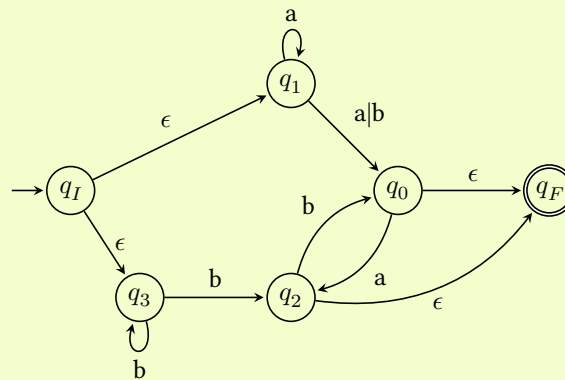
**Question 1** Cf cours. La définition est inductive, il y a trois cas de base (langage vide, mot vide, lettre) et trois règles inductives (concaténation, union et étoile de Kleene). Ce ne sont que des **symboles** à ce stade.

**Question 2** Cf preuve du cours. Il s'agit de construire l'automate produit avec  $F = F_1 \times F_2$ .

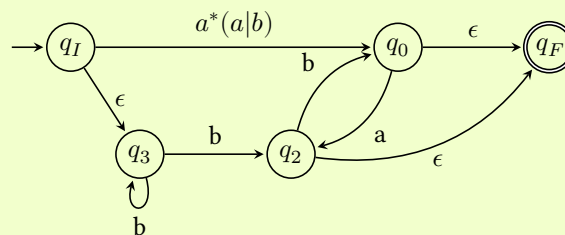
**Question 3** On obtient successivement les deux automates suivants :



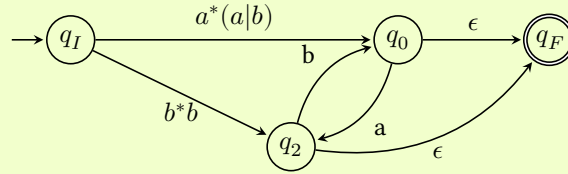
**Question 4** On commence par ajouter un état initial (et ses  $\epsilon$ -transitions) et un unique état final (et ses  $\epsilon$ -transitions) et on regroupe les transitions de sorte à avoir au plus une transition entre chaque paire d'états :



On élimine d'abord les états les plus simples. Commençons par éliminer  $q_1$ . Le seul état entrant est  $I$ , le seul état sortant est  $q_0$ , et on ajoute donc l'unique transition  $a^*(a|b)$  entre  $I$  et  $q_0$  :



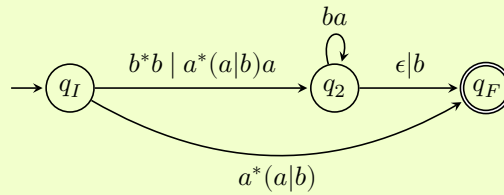
On élimine de même  $q_3$  aisément :



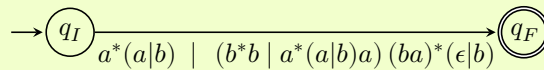
Éliminons maintenant  $q_0$  (choix arbitraire, les deux sont aussi délicat). On fait bien attention de n'oublier aucun couple d'états. Les états entrants sont  $q_I$  et  $q_2$ , les états sortants sont  $q_F$  et  $q_2$ , on doit donc adapter les transitions pour les quatre paires suivantes :

- $(q_I, q_F) : a^*(a|b)$
- $(q_I, q_2) : b^*b \mid a^*(a|b)a$
- $(q_2, q_F) : \epsilon|b$
- $(q_2, q_2) : ba$

On obtient donc :



Enfin, on élimine  $q_2$  :



On obtient donc l'expression régulière suivante :

$$a^*(a|b) \mid (b^*b \mid a^*(a|b)a)(ba)^*(\epsilon|b)$$

### Question 5

1. Non rationnel via le lemme de l'étoile en considérant  $u = a^N b^{N+1}$  (à rédiger).
2. Rationnel car fini.
3. Non rationnel : s'il l'était, son complémentaire, intersecté avec  $a^*b^*$ , c'est-à-dire  $\{a^n b^n \mid n \in \mathbb{N}\}$  le serait, ce qui n'est pas le cas.
4. Rationnel car égal à  $\{a^n b^m \mid n \text{ et } m \text{ pairs}\} \cup \{a^n b^m \mid n \text{ et } m \text{ impairs}\}$  pour lequel une expression est  $(a^2)^*(b^2)^* + a(a^2)^*b(b^2)^*$ . On peut aussi exhiber un automate à 4 états.

**Question 6** Cf cours. On linéarise, on construit l'automate local, puis on oublie les indices sur les transitions. L'expression régulière linéarisée est la suivante (on donne ici la variante avec les numérotations lettre par lettre, pour changer) :  $(a_1|c_1)^*a_2b_1b_2 \mid (a_3|c_2)^*$ .

Avec la numérotation globales :  $(a_1|c_2)^*a_3b_4b_5 \mid (a_6|c_7)^*$ .

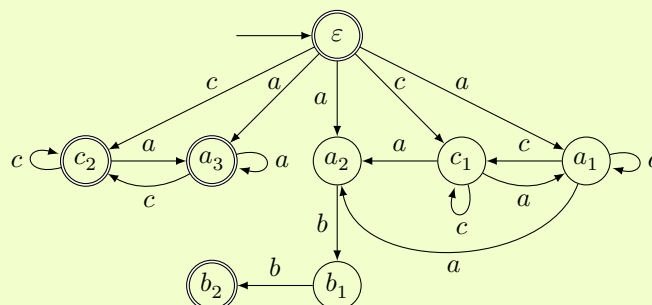
Les ensembles à calculer sont les suivants :

(premières lettres :)  $P(L) = \{a_1, c_1, a_2, a_3, c_2\}$

(dernières lettres :)  $S(L) = \{b_2, a_3, c_2\}$

(facteurs de longueur 2 :)  $F(L) = \{a_1a_1, a_1c_1, c_1c_1, c_1a_1, a_1a_2, \dots\}$  (on complète directement sur l'automate).

On obtient l'automate suivant.



N'oubliez pas de rendre l'état initial final !

Cet automate s'appelle l'**automate de Glushkov** associé à l'expression régulière.

### Question 7 Faire un schéma !

Soit  $\mathcal{A} = (Q, I, F, \delta)$  un automate (non nécessairement déterministe) reconnaissant  $L$ , on définit  $\mathcal{A}' = (Q, I', F', \eta)$  par :

- $I' = F$ ;
- $F' = I$ ;
- $\eta(q, a) = \{q' \in Q \mid q \in \delta(q', a)\}$ .

Autrement dit,  $\mathcal{A}'$  est obtenu à partir de  $\mathcal{A}$  en inversant toutes les flèches et en échangeant les états initiaux et les états acceptants. Pour tous  $q, q' \in Q$  et  $u \in \Sigma^*$ , on a  $q' \in \delta^*(q, u)$  si et seulement si  $q \in \eta^*(q', \bar{u})$  (par récurrence sur la longueur de  $u$ ). Ainsi :

$$\begin{aligned} u \in \bar{L} &\Leftrightarrow \bar{u} \in L \\ &\Leftrightarrow \exists q \in I \exists q' \in F, q' \in \delta^*(q, \bar{u}) \\ &\Leftrightarrow \exists q \in I \exists q' \in F, q \in \eta^*(q', u) \\ &\Leftrightarrow \exists q \in F' \exists q' \in I', q \in \eta^*(q', u) \\ &\Leftrightarrow u \in \mathcal{L}(\mathcal{A}') \end{aligned}$$

On a donc  $\bar{L} = \mathcal{L}(\mathcal{A}')$  :  $\bar{L}$  est reconnaissable.

#### Remarque :

- On aurait pu partir d'un automate  $\mathcal{A}$  déterministe, mais même dans ce cas l'automate  $\mathcal{A}'$  serait *a priori* non déterministe.

## Problème - Apprentissage d'un langage régulier

On s'intéresse dans cette partie à la possibilité pour un élève d'apprendre un langage régulier en interagissant avec un *enseignant*. L'apprentissage est ici « inductif » : l'enseignant ne transmet pas de règle à l'élève, mais est capable de répondre (correctement) à des questions posées par l'élève.

Plus précisément, on considère un langage régulier  $L$  sur un alphabet  $\Sigma$ .  $L$  est connu de l'enseignant mais pas de l'élève,  $\Sigma$  est en revanche connu des deux. Un enseignant est dit *minimalement compétent* s'il répond correctement aux deux types suivants de questions :

- les *requêtes d'appartenance*, où l'élève fournit un mot  $w$  et l'enseignant indique si  $w \in L$  ;
- les *conjectures*, où l'élève soumet une description d'un langage régulier  $X$ , et où l'enseignant :
  - répond *Correct* si  $X = L$  ;
  - fournit sinon un élément de la différence symétrique de  $X$  et  $L$ , élément que l'on appelle *contre-exemple*.

La description de  $X$  fournie par l'élève peut *a priori* être une expression régulière ou un automate fini, déterministe ou pas. Dans ce qui suit, l'élève fera en pratique ses conjectures sous la forme d'un automate fini déterministe complet  $A$  tel que  $\mathcal{L}(A) = X$ .

**Remarque :** On rappelle que la différence symétrique  $L \oplus X$  de  $L$  et  $X$  est l'ensemble  $(L \setminus X) \cup (X \setminus L)$  des mots qui appartiennent à  $L$  mais pas à  $X$ , ou inversement. On a  $L \oplus X = \emptyset$  si et seulement si  $L = X$ .

Le but du problème est l'étude d'un algorithme, appelé  $\mathcal{L}^*$ , qui permet à l'élève de déterminer exactement le langage  $L$  avec une complexité raisonnable (autant en nombre de requêtes à l'enseignant qu'en temps de calcul pour l'élève).

**Indications pour la programmation** Certaines fonctions du module `Queue` pourront être utiles : elles sont rap- pelées en annexe à la fin du sujet.

## I - Programmation de l'enseignant

On suppose désormais que l'alphabet  $\Sigma$  est de la forme  $[0 \dots n-1]$  pour un certain  $n$ . Une lettre de  $\Sigma$  sera donc un entier, et un mot une liste de lettres :



OCaml

```
1 type letter = int
2 type word = letter list
```

Pour représenter un automate fini déterministe complet, on utilise le type suivant :

OCaml

```
1 type dfa =
2   {q0 : int;
3     nb_letters : int;
4     nb_states : int;
5     accepting : bool array;
6     delta : int array array}
```

- L'alphabet  $\Sigma$  est  $[0 \dots \text{nb\_letters} - 1]$ .
- L'ensemble  $Q$  d'états est  $[0 \dots \text{nb\_states} - 1]$ .
- $q0$  indique l'état initial (et appartient donc à  $[0 \dots \text{nb\_states} - 1]$ );
- Pour  $0 \leq q < \text{nb\_states}$ ,  $\text{accepting}.(q)$  vaut `true` si l'état  $q$  est acceptant, `false` sinon.
- $\text{delta}$  est un tableau de longueur  $\text{nb\_states}$ , et pour  $0 \leq q < \text{nb\_states}$ ,  $\text{delta}.(q)$  est un tableau de longueur  $\text{nb\_letters}$  tel que, pour  $0 \leq x < \text{nb\_letters}$ ,  $\text{delta}.(q).(x)$  indique l'état  $\delta(q, x)$ .

#### Remarques :

- Tous les automates considérés dans le sujet seront supposés déterministes et complets.
- Les champs `nb_letters` et `nb_states` sont redondants (on pourrait calculer leur valeur à partir des dimensions du tableau `delta`); ils sont inclus pour clarifier le code.

**Question 1** Écrire une fonction de prototype `val delta_star : dfa -> int -> word -> int` telle que l'appel `delta_star auto q w` renvoie l'état  $\delta^*(q, w)$ , et indiquer sa complexité.

**Question 2** Soit  $A$  un automate à  $n$  états. Montrer que si  $\mathcal{L}(A)$  est non vide, alors il contient un mot de longueur strictement inférieure à  $n$ .

**Question 3** Écrire une fonction de prototype `val shortest_word : dfa -> word option` qui prend en entrée un automate  $A$  et renvoyant :

- `Some w`, où  $w$  est un mot de longueur minimale de  $\mathcal{L}(A)$ , s'il en existe un.
- `None` sinon (c'est-à-dire si  $\mathcal{L}(A)$  est vide).

**Question 4** Indiquer, en la justifiant rapidement, la complexité de la fonction `shortest_word`, en fonction de  $n = |Q|$  et  $p = |\Sigma|$ .

**Question 5** Étant donnés deux automates  $A$  et  $A'$  sur un même alphabet  $\Sigma$ , indiquer comment construire un automate  $B$  reconnaissant le langage  $\mathcal{L}(A) \oplus \mathcal{L}(A')$  (différence symétrique des deux langages).

**Question 6** Écrire une fonction de prototype `val symetric_difference : dfa -> dfa -> dfa` prenant en entrée deux automates  $A$  et  $A'$  sur un même alphabet, et renvoyant l'automate  $B$  de la question précédente.

**Question 7** Déterminer la complexité de la fonction `symetric_difference`.

On définit le type suivant pour représenter un enseignant minimalement compétent associé à un langage  $L$  :

OCaml

```
1 type teacher = {
2   nb_letters : int;
3   member : word -> bool;
4   counter_example : dfa -> word option;
5 }
```

- `nb_letters` spécifie l'alphabet  $\Sigma = [0 \dots \text{nb\_letters} - 1]$ .
- `member` prend en entrée un mot de  $\Sigma^*$  et renvoie un booléen indiquant s'il appartient au langage  $L$ .
- `counter_example auto` renvoie :
  - `None` si le langage reconnu par `auto` est égal à  $L$ ;
  - `Some w`, où  $w$  est un contre-exemple (élément de  $L \oplus \mathcal{L}(\text{auto})$ ), sinon.

**Question 8** Écrire une fonction de prototype `val create_teacher : dfa -> teacher` prenant en entrée un automate déterministe  $A$  tel que  $\mathcal{L}(A) = L$  et renvoyant un enseignant adapté. L'enseignant renvoyé fournira systématiquement des contre-exemples de longueur minimale (un tel enseignant sera dit *raisonnablement compétent*).

### Solution de l'Exercice 3

**Question 1** L'automate étant supposé complet, il n'y a vraiment aucune difficulté.

OCaml : cor.ml

```
1 let rec delta_star auto q word =
2   match word with
3   | [] -> q
4   | letter :: word' -> delta_star auto auto.delta.(q).(letter) word'
```

On effectue un simple parcours de liste avec des opérations en temps constant pour chaque élément, la complexité est en  $(|word|)$ .

**Question 2** Si  $\mathcal{L}(A)$  est non vide, alors il existe  $q_f \in F$  accessible depuis l'état initial  $q_0$ . Un chemin de longueur minimal de  $q_0$  à  $q_f$  est nécessairement élémentaire, et donc de longueur inférieure ou égale à  $|Q| - 1 = n - 1$ . L'étiquette de ce chemin fournit un mot de  $\mathcal{L}(A)$  de longueur strictement inférieure à  $n$ .

**Question 3** On effectue un parcours en largeur à partir de l'état initial jusqu'à atteindre un état final (ou terminer le parcours sans en rencontrer, auquel cas le langage est vide). On utilise une file dont les éléments sont des couples  $(q, w)$ , où  $q$  est un état et  $w$  un mot tel que  $q_I.\bar{w} = q$  (avec  $\bar{w}$  le miroir du mot  $w$ ).

OCaml : cor.ml

```
1 let shortest_word auto =
2   let queue = Queue.create () in
3   let seen = Array.make auto.nb_states false in
4   Queue.push (auto.q0, []) queue;
5   seen.(auto.q0) <- true;
6   let rec loop () =
7     if Queue.is_empty queue then None
8     else (
9       let (q, word) = Queue.pop queue in
10      if auto.accepting.(q) then Some (List.rev word)
11      else (
12        for letter = 0 to auto.nb_letters - 1 do
13          let q' = auto.delta.(q).(letter) in
14          if not seen.(q') then (
15            seen.(q') <- true;
16            Queue.push (q', letter :: word) queue
17          )
18        done;
19        loop ()
20      )
21    ) in
22   loop ()
```

**Question 4** Chaque opération effectuée sur la file est en temps constant, et la création du tableau `seen` en temps  $(|Q|)$ . Un état est ajouté au plus une fois dans la file, et l'on enlève un état de la file à chaque passage dans `loop` : on passe donc au plus  $|Q|$  fois. Or chaque passage s'effectue en temps  $(|\Sigma|)$  : on obtient donc une complexité en  $(|Q| + |Q| \cdot |\Sigma|) = (|Q| \cdot |\Sigma|)$ .

**Question 5** On construit un automate produit (déterministe complet)  $B$  comme suit :

- ensemble d'états  $Q \times Q'$  ;
- état initial  $(q_I, q'_I)$  ;
- fonction de transition  $(q, q').a = (q.a, q'.a)$  (toujours bien défini puisque  $A$  et  $A'$  sont déterministes complets) ;
- états acceptants  $F_B = \{(q, q') \in Q \times Q' \mid q \in F \text{ ou exclusif } q' \in F'\}$ .

**Question 6** En posant  $n_1 = |Q_1|$  et  $n_2 = |Q_2|$ , on numérote les états de l'automate produit par  $f(i, j) = i \cdot n_2 + j$ .

OCaml : cor.ml

```

1 let symmetric_difference a1 a2 =
2   let n1 = a1.nb_states in
3   let n2 = a2.nb_states in
4   let n = n1 * n2 in
5   let m = a1.nb_letters in
6   let f q1 q2 = q1 * n2 + q2 in
7   let delta = Array.make_matrix n m (-1) in
8   for q1 = 0 to n1 - 1 do
9     for q2 = 0 to n2 - 1 do
10      let q = f q1 q2 in
11      for letter = 0 to m - 1 do
12        let q' = f a1.delta.(q1).(letter) a2.delta.(q2).(letter) in
13        delta.(q).(letter) <- q'
14      done
15    done
16  done;
17  let accepting = Array.make n false in
18  let xor x y = (x && not y) || (not x && y) in
19  for q1 = 0 to n1 - 1 do
20    for q2 = 0 to n2 - 1 do
21      accepting.(f q1 q2) <- xor a1.accepting.(q1) a2.accepting.(q2)
22    done
23  done;
24  {delta; accepting; q0 = f a1.q0 a2.q0; nb_states = n; nb_letters = m}

```

**Question 7** On a les étapes principales suivantes :

- création de delta en  $(n_1 n_2)$ ;
- calcul de delta en  $(n_1 n_2 \cdot |\Sigma|)$ ;
- création de accepting en  $(n_1 n_2)$ ;
- calcul de accepting en  $(n_1 n_2)$ .

On obtient une complexité temporelles en  $(|Q_1| \cdot |Q_2| \cdot |\Sigma|)$ .

**Question 8** On propose l'implémentation suivante :

OCaml : cor.ml

```

1 let create_teacher (auto : dfa) : teacher =
2   let member w =
3     auto.accepting.(delta_star auto auto.q0 w) in
4   let counter_example auto' =
5     shortest_word (symetric_difference auto auto') in
6   {
7     member;
8     nb_letters = auto.nb_letters;
9     counter_example;
10  }

```

## II - Table d'observation

L'algorithme utilisé par l'élève va effectuer des requêtes d'appartenance dans un ordre bien défini, et organiser les résultats de ces requêtes dans une *table d'observation*, que l'on définit ci-dessous.

- Un ensemble  $X \subseteq \Sigma^*$  est dit *clos par préfixe* s'il vérifie la propriété suivante : si  $w \in X$ , alors tous les préfixes de  $w$  sont dans  $X$ .
- De même,  $X$  est *clos par suffixe* s'il contient tous les suffixes de  $w$  dès qu'il contient  $w$ .
- On remarquera que si  $X$  est non vide et clos par préfixe, alors  $\epsilon \in X$  (et de même si  $X$  est clos par suffixe).
- Une *table d'observation* est un triplet  $(S, E, f)$  tel que :
  - $S \subset \Sigma^*$  est un ensemble non vide et clos par préfixe (le «  $S$  » signifie *Start*, cet ensemble contient des débuts de mots);

- $E \subset \Sigma^*$  est un ensemble non vide et clos par suffixe (le «  $E$  » signifie *End*, cet ensemble contient des fins de mots);
- $f : (S \cup S\Sigma)E \rightarrow \{0, 1\}$ .
- Une table d'observation et un langage  $X$  sont dits *compatibles* si, pour tout couple  $s, e \in (S \cup S\Sigma) \times E$ , on a :
 
$$f(se) = 1 \Leftrightarrow se \in X.$$
- Une table d'observation et un automate  $A$  sont dits compatibles si la table est compatible avec  $\mathcal{L}(A)$ .
- À une application  $f$ , on peut associer une matrice  $T$  dont les lignes sont indexées par  $S \cup S\Sigma$  et les colonnes indexées par  $E$ . Pour un mot  $s \in S \cup S\Sigma$ , l'application partielle  $e \mapsto f(se)$  correspond alors à une « ligne » de cette matrice, et on la note  $\text{ligne}(s)$ . Autrement dit :

$$\text{ligne}(s) : \begin{cases} E \rightarrow \{0, 1\} \\ e \mapsto f(se) \end{cases}$$

*Exemple :*

La matrice  $T_0$  ci-dessous spécifie la fonction  $f$  d'une table d'observation  $(S, E, f)$  avec  $S = \{\epsilon, 0, 1, 10\}$  et  $E = \{10, 0, \epsilon\}$ .

		$\epsilon$	0	10
S	$\epsilon$	0	0	1
	0	0	0	1
	1	1	1	0
	10	1	0	1
$S\Sigma \setminus S$	00	0	0	1
	01	0	1	0
	11	0	0	1
	100	0	0	0
	101	1	1	0

FIGURE 1 – La table  $T_0$ .

On a ici  $(S \cup S\Sigma)E = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 010, 100, 101, 110, 0010, 0110, 1000, 1010, 1110, 10010, 10110\}$ . La première ligne indique que  $f(\epsilon) = 0$ ,  $f(0) = 0$  et  $f(10) = 1$ . La quatrième ligne indique que  $f(10) = 1$ ,  $f(100) = 0$  et  $f(1010) = 1$ .

La matrice  $T_0$  contient 27 coefficients, alors que  $|(S \cup S\Sigma)E| = 19$  : on a donc des informations redondantes. Par exemple, l'image de 10 est donnée à la fois dans la première ligne ( $\epsilon \cdot 10$ ), dans la troisième ligne ( $1 \cdot 0$ ) et dans la quatrième ligne ( $10 \cdot \epsilon$ ).

### Définition

- Une table d'observation est dite *cohérente* si, pour tous  $s, s' \in S$  tels que  $\text{ligne}(s) = \text{ligne}(s')$  et pour tout  $a \in \Sigma$ , on a  $\text{ligne}(sa) = \text{ligne}(s'a)$ .
- Elle est dite *close* si, pour tout  $t \in S\Sigma$ , il existe  $s \in S$  tel que  $\text{ligne}(t) = \text{ligne}(s)$ .

**Question 9** Montrer que la table de la figure 1 n'est ni close ni cohérente.

Dans toute la suite de cette partie, on suppose que  $(S, E, T)$  est une table d'observation close et cohérente, et l'on définit l'automate  $M(S, E, f) = (\Sigma, Q, q_0, F, \delta)$  par :

- $Q = \{\text{ligne}(s) \mid s \in S\}$ ;
- $q_0 = \text{ligne}(\epsilon)$ ;
- $F = \{\text{ligne}(s) \mid s \in S \text{ et } f(s) = 1\}$ ;
- $\delta(\text{ligne}(s), a) = \text{ligne}(sa)$  pour  $s \in S$  et  $a \in \Sigma$ .

**Question 10** Montrer que  $M(S, E, f)$  est correctement défini, et qu'il s'agit d'un automate déterministe complet.

**Question 11** On considère la table d'observation close et cohérente ci-dessous. Donner l'automate  $A = M(S, E, f)$  associé, et déterminer le langage  $\mathcal{L}(A)$  reconnu par cet automate. Justifier que la table est compatible avec  $A$ .

		$\epsilon$	0
S	$\epsilon$	0	0
	0	0	1
	1	0	0
	00	1	1
$S\Sigma \setminus S$	01	0	0
	10	0	1
	11	0	0
	000	1	1
	001	0	0

FIGURE 2 – La table  $T_1$ .

**Question 12** Montrer que pour tout  $s \in S \cup S\Sigma$ , on a  $\delta^*(q_0, s) = \text{ligne}(s)$ .

**Question 13** Montrer que  $M(S, E, f)$  est compatible avec  $(S, E, f)$ .

**Question 14** Soit  $A' = (\Sigma, Q', q'_0, F', \delta')$  compatible avec  $(S, E, f)$ . Montrer que  $|Q'| \geq |Q|$ .

**Définition** Deux automates (déterministes et complets)  $A_1 = (\Sigma, Q_1, q_0^1, F_1, \delta_1)$  et  $A_2 = (\Sigma, Q_2, q_0^2, F_2, \delta_2)$  sont dits *isomorphes* s'il existe une application  $\varphi : Q_1 \rightarrow Q_2$  telle que :

- (1)  $\varphi$  est bijective;
- (2)  $\varphi(q_0^1) = q_0^2$ ;
- (3)  $\varphi(F_1) = F_2$ ;
- (4)  $\varphi(\delta_1(q, a)) = \delta_2(\varphi(q), a)$  pour  $q \in Q_1$  et  $a \in \Sigma$ .

**Question 15** Montrer que si  $A' = (\Sigma, Q', q'_0, F', \delta')$  est un automate compatible avec  $(S, E, F)$  tel que  $|Q'| \leq |Q|$ , alors  $A'$  est isomorphe à  $M(S, E, f)$ .

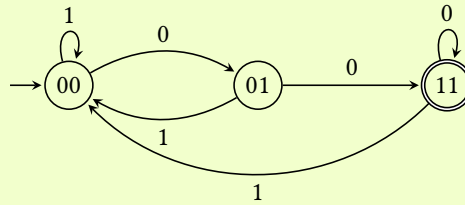
#### Solution de l'Exercice 4

**Question 9** On a  $\text{ligne}(01) \neq \text{ligne}(s)$  pour tout  $s \in S$ , donc la table n'est pas close. De plus,  $\text{ligne}(\epsilon) = \text{ligne}(0)$  mais  $\text{ligne}(1) \neq \text{ligne}(01)$ , donc la table n'est pas cohérente.

#### Question 10

- $Q$  est bien défini.
- $q_0$  est bien défini, puisque  $S$  clos par préfixe et donc  $\epsilon \in S$ .
- si  $\text{ligne}(s) = \text{ligne}(s')$  avec  $s, s' \in S$ , alors :
  - $f(s)$  et  $f(s')$  sont bien définis puisque  $\epsilon \in E$  (qui est clos par suffixe) et égaux puisque  $\text{ligne}(s) = \text{ligne}(s')$ , donc  $F$  est bien défini;
  - pour toute lettre  $a \in \Sigma$ ,  $\text{ligne}(sa) = \text{ligne}(s'a)$  puisque la table est cohérente, et il existe  $s'' \in S$  tel que  $\text{ligne}(sa) = \text{ligne}(s'')$  puisque la table est complète. Donc  $\delta$  est bien définie (et l'automate est complet et déterministe).

**Question 11** On obtient l'automate suivant :



Le langage reconnu est dénoté par  $(0|1)^*00$ . On vérifie que la table est compatible avec ce langage :

- on a  $f(u) = 1$  pour  $u \in \{00, 000, 100, 0000\}$ ;
- on a  $f(u) = 0$  pour  $u \in \{\epsilon, 0, 1, 10, 01, 010, 11, 110, 001, 0010\}$ .

On a donc bien  $f(u) = 1 \Leftrightarrow u \in \mathcal{L}(A)$  pour  $u \in (S \cup S\Sigma)E$  : l'automate et la table sont compatibles.

**Question 12** Par récurrence sur  $|s|$  :

- si  $s = \epsilon$ , alors  $\delta^*(q_0, \epsilon) = q_0$  par définition, et on a bien  $q_0 = \text{ligne}(\epsilon)$ ;

- sinon,  $s = ua$  et  $\delta^*(q_0, ua) = \delta(\delta^*(q_0, u), a)$ . On a nécessairement  $u \in S$  (évident si  $s \in S\Sigma$ , par fermeture par préfixe de  $S$  si  $s \in S$ ), on peut donc appliquer l'hypothèse de récurrence :

$$\begin{aligned}\delta^*(q_0, s) &= \delta(\delta^*(q_0, u), a) \\ &= \delta(\text{ligne}(u), a) \\ &= \delta(\text{ligne}(ua))\end{aligned}\quad \text{par définition de } \delta$$

On a donc bien  $\delta^*(q_0, s) = \text{ligne}(s)$  pour tout  $s \in S \cup S\Sigma$ .

**Question 13** Notons  $L_A$  le langage reconnu par  $M(S, E, f)$ . Notons  $H_n$  : « pour tout  $s \in S \cup S\Sigma$  et pour tout  $e \in E$  tel que  $|e| \leq n$ , on a  $f(se) = 1 \Leftrightarrow se \in L_A$  ».

- Dans tous les cas, comme la table est close, on peut considérer  $s' \in S$  tel que  $\text{ligne}(s') = \text{ligne}(s)$ . On a alors par définition  $f(se) = f(s'e)$ .
- Pour  $n = 0$ , on a  $e = \epsilon$  et

$$\begin{aligned}se \in L_A &\Leftrightarrow s \in L_A \\ &\Leftrightarrow \delta^*(q_0, s) \in F \\ &\Leftrightarrow \text{ligne}(s) \in F && \text{d'après la question précédente} \\ &\Leftrightarrow \text{ligne}(s') \in F \\ &\Leftrightarrow f(s') = 1 \\ &\Leftrightarrow f(s'e) = 1 \\ &\Leftrightarrow f(se) = 1\end{aligned}$$

- sinon,  $e = ae'$  avec  $a \in \Sigma$ .

$$\begin{aligned}se \in L_A &\Leftrightarrow \delta^*(q_0, sae') \in F \\ &\Leftrightarrow \delta^*(\delta^*(q_0, s), ae') \in F \\ &\Leftrightarrow \delta^*(\text{ligne}(s), ae') \in F && \text{d'après la question précédente} \\ &\Leftrightarrow \delta^*(\text{ligne}(s'), ae') \in F \\ &\Leftrightarrow \delta^*(\text{ligne}(s'a), e') \in F && \text{par définition de } \delta \\ &\Leftrightarrow \delta^*(\delta^*(q_0, s'a), e') \in F && \text{d'après la question précédente, avec } s'a \in S\Sigma \\ &\Leftrightarrow \delta^*(q_0, s'ae') \in F && \text{par définition de } \delta^* \\ &\Leftrightarrow s'ae' \in L_A\end{aligned}$$

Comme  $s' \in S$ , on a  $s'a \in S \cup S\Sigma$  et comme  $E$  est clos par suffixe, on a  $e' \in E$  (et  $|e'| < |e|$ ) : on peut donc appliquer l'hypothèse de récurrence à  $s'a$  et  $e'$ . On obtient donc :

$$\begin{aligned}se \in L_A &\Leftrightarrow s'ae' \in L_A \\ &\Leftrightarrow f(s'ae') = 1 && \text{par hypothèse de récurrence} \\ &\Leftrightarrow f(s'e) = 1 \\ &\Leftrightarrow f(se) = 1\end{aligned}$$

On a prouvé que  $\forall s \in S \cup S\Sigma, \forall e \in E, se \in L_A \Leftrightarrow f(se) = 1$  : la table et l'automate sont compatibles.

**Question 14** Soient  $n = |Q|$  et  $s_1, \dots, s_n \in S$  tels que les  $\text{ligne}(s_i)$  soient deux à deux distincts, et  $q'_i = \delta'^*(q'_0, s_i)$  pour  $1 \leq i \leq n$ . Pour  $i \neq j$ , il existe  $e \in E$  tel que  $f(s_i e) \neq f(s_j e)$  (puisque  $\text{ligne}(s_i) \neq \text{ligne}(s_j)$ ). Comme  $A'$  est compatible avec  $(S, E, f)$ , on a alors  $\delta'^*(q'_0, s_i e) \in F'$  et  $\delta'^*(q'_0, s_j e) \notin F'$  (ou inversement), et donc  $\delta'^*(q'_0, s_i e) \neq \delta'^*(q'_0, s_j e)$ . On en déduit  $q'_i \neq q'_j$  : on a donc au moins  $n$  états distincts dans  $Q'$ . Donc  $|Q'| \geq |Q|$ .

**Question 15** Pour commencer, on remarque qu'on a nécessairement  $|Q| = |Q'|$  d'après la question précédente. Supposons  $\delta'^*(q'_0, s) = \delta'^*(q'_0, s')$  pour  $s, s' \in S$ . On a alors  $se \in \mathcal{L}(A') \Leftrightarrow s'e \in \mathcal{L}(A')$  pour tout mot  $e$ , et en particulier pour tout  $e \in E$ . Comme  $A'$  est compatible avec  $(S, E, f)$ , cela implique que  $f(se) = f(s'e)$  pour tout  $e \in E$ , et donc que  $\text{ligne}(s) = \text{ligne}(s')$ . Par conséquent, l'application

$$\varphi : \begin{cases} Q & \rightarrow Q' \\ \text{ligne}(s) & \mapsto \delta'^*(q'_0, s) \end{cases}$$

est bien définie (l'image ne dépend pas du choix du représentant). D'après la question précédente, elle est injective, et donc bijective par égalité des cardinaux. Il reste à montrer qu'il s'agit d'un isomorphisme.

- Pour l'état initial,  $\varphi(q_0) = \varphi(\text{ligne}(\epsilon)) = \delta'^*(q'_0, \epsilon) = q'_0$ .
- Pour les états acceptants :

$$\begin{aligned}
 \text{ligne}(s) \in F &\Leftrightarrow s \in \mathcal{L}(A) && \text{par définition de } F \\
 &\Leftrightarrow f(s) = 1 && \text{puisque } A \text{ est compatible avec } (S, E, f) \\
 &\Leftrightarrow s \in \mathcal{L}(A') && \text{puisque } A' \text{ est compatible avec } (S, E, f) \\
 &\Leftrightarrow \delta'^*(q'_0, s) \in F' \\
 &\Leftrightarrow \varphi(\text{ligne}(s)) \in F'
 \end{aligned}$$

Donc  $\varphi(F) = F'$ .

- Pour les transitions, soit  $s \in S$  et  $a \in \Sigma$ . Il existe  $s' \in S$  tel que  $\text{ligne}(sa) = \text{ligne}(s')$  (la table est close), et :

$$\begin{aligned}
 \varphi(\delta(\text{ligne}(s), a)) &= \varphi(\text{ligne}(sa)) && \text{par définition de } \delta \\
 &= \varphi(\text{ligne}(s'))
 \end{aligned}$$

D'autre part :

$$\begin{aligned}
 \delta'(\varphi(\text{ligne}(s)), a) &= \delta'(\delta'^*(q'_0, s), a) \\
 &= \delta'^*(q'_0, sa)
 \end{aligned}$$

Par surjectivité de  $\varphi$ , il existe  $s'' \in S$  tel que  $q'_0.sa = \varphi(\text{ligne}(s'')) = q'_0.s''$ . Pour  $e \in E$ , on a  $q'_0.sae = q'_0.s''ae$ , donc  $f(s''e) = 1 \Leftrightarrow f(sae) = 1$  (puisque  $A'$  est compatible avec la table), c'est-à-dire  $\text{ligne}(s'') = \text{ligne}(sa) = \text{ligne}(s')$ . Finalement,  $q'_0.sa = \varphi(\text{ligne}(s'')) = \varphi(\text{ligne}(s'))$ , ce qui permet de conclure.

### III - Algorithme $\mathcal{L}^*$

L'algorithme  $\mathcal{L}^*$  utilisé par l'élève maintient à jour une table d'observation  $(S, E, f)$ , où  $f$  est en fait stockée sous la forme d'une matrice  $T$ . À chaque fois qu'il ajoute des mots à  $S$  ou à  $E$ , l'algorithme effectue des requêtes d'appartenance pour étendre  $f$  (en ajoutant des lignes ou des colonnes à la matrice  $T$ ).

```

1   $S \leftarrow \{\epsilon\}$ 
2   $E \leftarrow \{\epsilon\}$ 
3  Calculer  $f$  à l'aide de requêtes d'appartenance.
4  tant que Vrai faire
5      tant que  $(S, E, f)$  n'est pas close ou pas cohérente faire
6          si  $(S, E, f)$  n'est pas close alors
7              Trouver  $s \in S$  et  $a \in \Sigma$  tels que  $\text{ligne}(sa) \neq \text{ligne}(s')$  pour tout  $s' \in S$ .
8               $S \leftarrow S \cup \{sa\}$ 
9              Étendre  $f$  à  $(S \cup S\Sigma)E$  en effectuant des requêtes d'appartenance.
10         sinon si  $(S, E, f)$  n'est pas cohérente alors
11             Trouver  $s, s' \in S, a \in \Sigma$  et  $e \in E$  tels que  $\text{ligne}(s) = \text{ligne}(s')$  et  $f(sae) \neq f(s'ae)$ .
12              $E \leftarrow E \cup \{ae\}$ 
13             Étendre  $f$  à  $(S \cup S\Sigma)E$  en effectuant des requêtes d'appartenance.
14      $M \leftarrow M(S, E, f)$ 
15     Soumettre à l'enseignant la conjecture  $M$ .
16     si l'enseignant répond par un contre-exemple  $w$  alors
17         Ajouter  $w$  et ses préfixes à  $S$ .
18         Étendre  $f$  à  $(S \cup S\Sigma)E$  en effectuant des requêtes d'appartenance.
19     sinon
20         renvoyer  $M$ 

```

Algorithme 5 Apprentissage  $\mathcal{L}^*$

**Question 16** Justifier que les lignes commençant par « Trouver...tels que... » (lignes 7 et 11) ne peuvent échouer et que  $S$  (respectivement  $E$ ) reste toujours clos par préfixe (respectivement par suffixe).

**Question 17** Donner l'ensemble des requêtes d'appartenance que l'élève effectue aux lignes 3, 9, 13 et 18.

On définit :

- $A_m$  un automate déterministe complet minimal (en nombre d'états) pour le langage  $L$  – on note  $n$  son nombre d'états;
- $|(S, E, f)| \stackrel{df}{=} |\{\text{ligne}(s) \mid s \in S\}|$  le nombre de lignes *distinctes* correspondant à des mots de  $S$  dans la matrice  $T$  (où  $(S, E, f)$  est une table d'observation).

**Question 18** Montrer que  $|(S, E, f)|$  croît strictement à chaque passage dans la boucle interne.

**Question 19** Montrer que  $|(S, E, f)|$  croît strictement entre une conjecture (incorrecte) et la conjecture suivante.

**Question 20** Soit  $(S, E, f)$  une table d'observation, dont on ne suppose ni qu'elle est close ni qu'elle est cohérente. Montrer que si  $A$  est un automate (déterministe et complet) compatible avec  $(S, E, f)$ , alors  $A$  possède au moins  $|(S, E, f)|$  états.

**Question 21** Montrer que l'algorithme  $\mathcal{L}^*$  termine et renvoie un automate isomorphe à  $A_m$ .

On suppose à présent que l'enseignant utilisé est celui programmé à la partie I, construit à partir de l'automate minimal  $A_m$  du langage  $L$ .

**Question 22** Majorer en fonction de  $n$  la longueur maximale d'un contre-exemple  $w$  donné en réponse à une conjecture incorrecte.

**Question 23** Majorer en fonction de  $n$  et  $|\Sigma|$  le cardinal de  $S$  et de  $E$ .

**Question 24** Majorer en fonction de  $n$  le nombre de requêtes d'appartenance et de conjectures effectuées par l'algorithme  $\mathcal{L}^*$ .

**Question 25** Justifier, en esquisant une réalisation très simple de la structure de table d'observation, que l'algorithme  $\mathcal{L}^*$  peut être implémenté avec une complexité polynomiale en  $n$ .

### Solution de l'Exercice 5

#### Question 16

- La ligne 7 n'est exécutée que si la table n'est pas close, ce qui garantit l'existence de  $s \in S$  et  $a \in \Sigma$  tel que  $\text{ligne}(sa) \notin \{\text{ligne}(s') \mid s' \in S\}$ .
- De même, la ligne 11 n'est exécutée que si la table n'est pas cohérente. Dans ce cas, il existe  $s, s' \in S$  et  $a \in \Sigma$  tels que  $\text{ligne}(s) = \text{ligne}(s')$  et  $\text{ligne}(sa) \neq \text{ligne}(s'a)$ , ce qui signifie qu'il existe  $e \in E$  tel que  $f(sae) \neq f(s'ae)$ .
- Montrons que «  $S$  est clos par préfixe » est un invariant.
  - On initialise  $S$  à  $\{\epsilon\}$  qui est clos par préfixe.
  - Quand on exécute la ligne 8, on ajoute  $sa$  à  $S$ . Ses préfixes sont :
    - $s$ , qui est dans  $S$  par définition;
    - les préfixes propres de  $s$ , qui sont dans  $S$  d'après l'invariant.
 L'invariant est donc conservé.
  - Quand on exécute la ligne 17, on ajoute tous les préfixes de  $w$  en même temps que  $w$ , donc l'invariant est conservé.

Donc  $S$  reste clos par préfixe.

- De même pour  $E$  : le seul ajout est  $ae$  avec  $e \in E$ , donc tous les suffixes propres de  $ae$  sont déjà dans  $E$ .

#### Question 17

**Ligne 3** On teste pour  $\epsilon$  et pour les  $a \in \Sigma$ .

**Ligne 9** On teste pour les  $sabe$  avec  $b \in \Sigma$  et  $e \in E$ .

**Ligne 13** On teste pour les  $sae$  avec  $s \in S \cup S\Sigma$ .

**Ligne 17** On teste pour les  $se$  et  $sae$  avec  $s$  préfixe de  $w$ ,  $a \in \Sigma$  et  $e \in E$ .



**Question 18**

- Si l'on passe dans la première branche,  $|(S, E, f)|$  augmente de un puisqu'on ajoute  $sa$  à  $S$  et que  $\text{ligne}(sa)$  n'était égal à aucun  $\text{ligne}(s)$  avec  $s \in S$ .
- Si l'on passe dans la deuxième branche,  $\text{ligne}(s)$  est séparé en deux (puisque  $\text{ligne}(s) \neq \text{ligne}(s')$  après l'ajout de  $ae$  à  $E$ ). Donc  $|(S, E, f)|$  augment au moins de un (potentiellement plus, d'autres lignes pouvant être séparées).

Donc  $|(S, E, f)|$  augmente strictement à chaque passage.

**Question 19** Considérons  $A = M(S, E, f)$  l'automate de la conjecture infructueuse et  $A' = M(S', E', f')$  l'automate de la conjecture suivante.  $A'$  est compatible avec  $(S, E, f)$  (puisque  $(S', E', f')$  ne fait qu'étendre cette table), et n'est pas équivalent à  $A$  (puisque  $w \in \mathcal{L}(A) \oplus \mathcal{L}(A')$ ). Il ne peut donc pas être isomorphe à  $A$ , donc d'après la question 15 il a strictement plus d'états que  $A$ . Ce nombre d'états est exactement le nombre de lignes distinctes, donc  $|(S, E, f)|$  croît strictement entre deux conjectures successives.

**Question 20** Soit  $A = (\Sigma, Q, q_I, F, \delta)$  déterministe complet compatible avec  $(S, E, f)$ , et  $s, s' \in S$  tels que  $\text{ligne}(s) \neq \text{ligne}(s')$ . Il existe donc  $e \in E$  tel que  $f(se) \neq f(s'e)$ . Comme  $A$  est compatible avec  $(S, E, f)$ , cela signifie que  $se \in \mathcal{L}(A)$  et  $s'e \notin \mathcal{L}(A)$  (ou inversement). On a donc  $q_I.se \neq q_I.s'e$  (où  $q_I$  est l'état initial de  $A$ ), et donc  $q_I.s \neq q_I.s'$ . On a donc  $|Q| \geq |(S, E, f)|$ .

**Question 21** Remarquons déjà que, si l'algorithme termine, alors il renvoie nécessairement un automate qui reconnaît  $L$ .

$A_m$  est compatible avec  $(S, E, f)$  à tout moment de l'exécution, donc d'après la question 20 on a  $|(S, E, f)| \leq n$  tout au long de l'algorithme. Comme de plus  $|(S, E, f)|$  croît strictement à chaque passage dans la boucle interne et dans la boucle externe d'après les questions 18 et 19,  $n - |(S, E, f)|$  est un variant (entier, positif, strictement décroissant). Donc l'algorithme termine.

L'automate renvoyé reconnaît  $L$  : il a donc au moins  $n$  états par minimalité de  $A_m$ . D'après ce qui précède, il a en fait exactement  $n$  états, et la question 15 montre qu'il est isomorphe à  $A_m$ .

**Remarque :** De manière générale, et comme nous l'avons vu en cours, l'automate (déterministe et complet) minimal pour un langage donné est unique à isomorphisme près : il est donc normal que l'automate renvoyé soit isomorphe à  $A_m$ .

**Question 22** D'après ce qui précède, l'automate  $M(S, E, f)$  de la conjecture possède un nombre  $p \leq n$  d'états. L'automate produit construit pour la différence symétrique possède donc au plus  $n^2$  états, et reconnaît un langage non vide (puisque la conjecture est incorrecte). L'enseignant renvoie un mot de longueur minimale dans la différence symétrique, donc d'après la question 2 on a  $|w| \leq n^2$ .

**Question 23**

- Initialement,  $|S| = |E| = 1$ .
- À chaque passage dans la boucle interne, soit  $|E|$  soit  $|S|$  augmente de un, et  $|(S, E, f)|$  augmente au moins de un.
- À chaque conjecture incorrecte,  $|S|$  augmente de au plus  $n^2$  d'après la question précédente,  $|E|$  reste inchangé et  $|(S, E, f)|$  augmente au moins de un.
- Comme  $|(S, E, f)|$  est majoré par  $n$ , le nombre de conjectures plus le nombre de passages dans la boucle interne est majoré par  $n$ .

On en déduit que  $|S| \leq n^3$  et  $|E| \leq n$ .

**Question 24** En combinant la question 17 et la question précédente, on voit que :

- à chaque passage ligne 9, on fait au plus  $|E| \cdot |\Sigma| \leq n \cdot |\Sigma|$  requêtes ;
- à chaque passage ligne 13, on fait au plus  $|S \cup S\Sigma| \leq n^3 + |\Sigma|n^3$  requêtes ;
- à chaque passage ligne 18, on fait au plus  $|E| \leq n$  requêtes pour chaque  $ua$  avec  $u$  préfixe de  $w$  et  $a \in \Sigma$ . Il y a au plus  $n^2$  préfixes, donc  $n^2|\Sigma|$  mots  $ua$  et au plus  $n^3|\Sigma|$  requêtes au total.

Le nombre total de passages dans toutes ces lignes étant majoré par  $n$ , le nombre de requêtes d'appartenance est majoré par  $n^4|\Sigma|$ .

**Question 25** Les mots de  $S$  et de  $E$  sont tous de longueur  $(n^2)$  : en effet, les mots ajoutés ligne 17 sont de longueur  $\leq n^2$ , et chaque passage ligne 8 ou 12 augmente la longueur maximale d'au plus un. Donc chaque

requête d'appartenance se fait en temps  $(n^2)$ , et le coût total des requêtes d'appartenance est polynomial (en  $n$  et  $\Sigma$ ).

Chaque conjecture demande la construction d'un automate produit puis un parcours de cet automate, en temps total  $(n^2\Sigma)$  : il y a au plus  $n$  conjectures, donc à nouveau polynomial en  $n$  et  $\Sigma$ .

Pour déterminer si  $(S, E, f)$  est close, il suffit, pour chaque  $s \in S$  et  $a \in \Sigma$ , de parcourir tous les  $s' \in S$  pour voir si  $\text{ligne}(s) = \text{ligne}(s')$  : si l'on a simplement stocké la matrice codant la table, cela se fait en temps  $(|S|^2 \cdot |\Sigma| \cdot |E|)$ , à nouveau polynomial en  $n$  et  $|\Sigma|$  d'après ce qui précède.

Le calcul de complexité est essentiellement le même pour la cohérence de  $(S, E, f)$ . Les autres opérations sont des ajouts de ligne et colonne (au pire, on peut recopier toute la table dans une nouvelle table plus grande à chaque fois) et la construction de l'automate, qui se fait clairement en temps polynomial en  $|S|$ ,  $|E|$  et  $\Sigma$ , et donc en  $n$  et  $|\Sigma|$ .

Finalement, l'algorithme est bien en temps polynomial en  $n$  et  $|\Sigma|$ , pour n'importe quel choix raisonnable de structure de données pour la table.

## IV - Programmation de l'élève

### 1) Construction de l'automate

On suppose pour l'instant que l'on dispose d'une structure de table d'observation, avec l'interface suivante (toutes les fonctions ne sont pas nécessairement immédiatement utiles) :

OCaml

```
1 (* Le type d'une table d'observation *)
2 type t
3
4 (* Renvoie le nombre de lignes *distinctes* de la table. *)
5 (* Les lignes sont numérotées de 0 à nb_rows - 1. *)
6 val nb_rows : t -> int
7
8 (* Renvoie la taille de l'alphabet *)
9 val nb_letters : t -> int
10
11 (* Renvoie le numéro de la ligne correspondant à un mot de  $S \cup S\Sigma^*$ . *)
12 val get_row_number : t -> word -> int
13
14 (* Prend en entrée un mot  $w \in S \cup S\Sigma^*$  et un mot  $e \in E$  *)
15 (* et renvoie  $f(w \cdot e)$ . *)
16 val compute_f : t -> word -> word -> bool
17
18 (* Applique une fonction g successivement à tous les mots de l'ensemble  $S\Sigma^*$ . *)
19 val iter_s : t -> (word -> unit) -> unit
20
21 (* Applique une fonction g successivement à tous les mots de l'ensemble  $S\Sigma^*$ . *)
22 val iter_sa : t -> (word -> unit) -> unit
```

On suppose que ces fonctions sont regroupées dans un module `Obs` : ainsi, on pourra appeler la fonction `nb_rows` par `Obs.nb_rows` et son type sera `Obs.t -> int`.

**Question 26** Écrire une fonction `construct_auto : Obs.t -> dfa` qui prend en entrée une table d'observation  $(S, E, f)$ , supposée close et cohérente, et renvoie l'automate  $M(S, E, f)$ .

### 2) Test de complétude et de fermeture

On définit les deux exceptions suivantes :

OCaml

```
1 exception Incomplete of word
2 exception Inconsistent of word
```

**Question 27** Écrire une fonction `check_complete : Obs.t -> unit` qui prend en entrée une table et lève l'exception `Incomplete w`, où  $w$  est un mot de  $S\Sigma$  tel que  $\text{ligne}(w) \neq \text{ligne}(u)$  pour tout  $u \in S$ , si la table n'est pas close.

On ajoute à la signature du module `Obs` la fonction suivante :

**OCaml** `1 val separate_rows : Obs.t -> word -> word -> word option`

L'appel `Obs.separate_rows table u u'`, dont le comportement n'est défini que si  $u$  et  $u'$  appartiennent à  $S \cup S\Sigma$ , renvoie :

- `None` si  $\text{ligne}(u) = \text{ligne}(u')$ ;
- `Some w`, où  $w$  est un mot de  $E$  tel que  $f(uw) \neq f(u'w)$  sinon.

**Question 28** Écrire une fonction `check_consistent : Obs.t -> unit` qui prend en entrée une table d'observation et :

- ne fait rien si la table est cohérente;
- si la table est incohérente, lève l'exception `Inconsistent w`, où  $w$  est un mot de la forme  $aw'$  tel que :
  - $a \in \Sigma$ ;
  - $w' \in E$ ;
  - il existe  $u, u' \in S$  tels que  $\text{ligne}(u) = \text{ligne}(u')$  et  $f(uaw') \neq f(u'aw')$ .

### 3) Algorithme $\mathcal{L}^*$

On ajoute à la signature du module `Obs` les deux fonctions suivantes :

**OCaml** `1 val add_to_s : t -> word -> (word -> bool) -> unit  
2 val add_to_e : t -> word -> (word -> bool) -> unit`

- Ces deux fonctions prennent en argument une table d'observation  $t$ , un mot  $w$  et une fonction `member` permettant d'effectuer une requête d'appartenance au langage  $L$  que l'on cherche à apprendre (`member u` renvoie `true` si et seulement si  $u \in L$ ).
- La fonction `add_to_s` ajoute le mot  $w$  passé en argument, ainsi que tous ses préfixes, à l'ensemble  $S$ . Elle met également à jour la fonction  $f$  en ajoutant les lignes nécessaires pour le nouvel ensemble  $S \cup S\Sigma$  (et en effectuant les requêtes d'appartenance nécessaires pour remplir ces lignes).
- La fonction `add_to_e` ajoute le mot  $w$  passé en argument à  $E$ . Elle met également à jour la fonction  $f$  en ajoutant la nouvelle colonne correspondant à  $w$  (et en effectuant les requêtes d'appartenance nécessaires pour remplir cette colonne).

**Question 29** Écrire une fonction `make_complete_and_coherent : table -> teacher -> unit` prenant en entrée une table et un enseignant, et modifiant la table pour la rendre close et cohérente. Cette fonction doit donc réaliser la boucle interne des lignes 5 à 13 de l'algorithme 5.

On ajoute à la signature du module `Obs` la fonction suivante :

**OCaml** `1 val initial_table : teacher -> t`

Cette fonction renvoie la table d'observation pour  $S = E = \{\epsilon\}$  avec l'enseignant fourni (elle effectue donc l'initialisation des lignes 1 à 3 de l'algorithme 5).

**Question 30** Écrire une fonction `learn : teacher -> dfa` qui prend en entrée un enseignant pour un langage  $L$  et renvoie un automate déterministe minimal reconnaissant  $L$ , en suivant l'algorithme  $\mathcal{L}^*$ .

### Solution de l'Exercice 6

**Question 26** On suit exactement la définition de l'énoncé.

OCaml : cor.ml

```

1 let construct_auto table =
2   let n = Obs.nb_rows table in
3   let m = Obs.nb_letters table in
4
5   (* Détermination des états acceptants *)
6   let accepting = Array.make n false in
7   let process_word word =
8     let i = Obs.get_row_number table word in
9     accepting.(i) <- Obs.compute_f table word [] in
10  Obs.iter_s table process_word;
11
12  (* Calcul des transitions *)
13  let delta = Array.make_matrix n m (-1) in
14  let add_transitions word =
15    let q = Obs.get_row_number table word in
16    for letter = 0 to m - 1 do
17      let q' = Obs.get_row_number table (word @ [letter]) in
18      delta.(q).(letter) <- q'
19    done in
20  Obs.iter_s table add_transitions;
21
22  (* État initial *)
23  let q0 = Obs.get_row_number table [] in
24
25  {accepting; q0; delta; nb_states = n; nb_letters = m}

```

**Question 27** On itère sur tous les mots de  $S$  pour déterminer les numéros de ligne  $i$  tels qu'aucun  $s \in S$  ne vérifie  $\text{numero}(\text{ligne}(s)) = i$  : c'est le rôle du tableau `present`. D'autre part, on remplit le tableau `preimage` avec des éléments de  $S\Sigma$  de manière à avoir  $\text{numero}(\text{ligne}(\text{preimage}[i])) = i$  pour tout  $i$ . On parcourt ensuite le tableau `present` :

- si toutes les cases valent `true`, la table est complète;
- sinon, on trouve un numéro de ligne  $i$  qui ne correspond à aucun mot de  $S$ , et l'on lève l'exception `Incomplete u` avec  $u \in S\Sigma$  tel que  $\text{numero}(\text{ligne}(u)) = i$ .

OCaml : cor.ml

```

1 let check_complete table =
2   let n = Obs.nb_rows table in
3   let present = Array.make n false in
4   let preimage = Array.make n [] in
5   let process_s_word s =
6     present.(Obs.get_row_number table s) <- true in
7   Obs.iter_s table process_s_word;
8   let process_sa_word sa =
9     preimage.(Obs.get_row_number table sa) <- sa in
10  Obs.iter_sa table process_sa_word;
11  for i = 0 to n - 1 do
12    if not present.(i) then raise (Incomplete preimage.(i))
13  done

```

**Question 28** On définit une fonction `check : word -> word -> unit` qui prend en entrée deux mots  $s$  et  $s'$  et :

- ne fait rien si  $\text{ligne}(sa) = \text{ligne}(s'a)$  pour toute lettre  $a \in \Sigma$ ;
- sinon, lève l'exception `Inconsistent w`, avec  $w \in E$  tel que  $\text{ligne}(saw) \neq \text{ligne}(s'aw)$ .

Il faut ensuite appeler cette fonction sur toutes les paires  $\{s, s'\}$  d'éléments de  $S$  telles que  $\text{ligne}(s) = \text{ligne}(s')$ . Pour ce faire :

- on construit un tableau `preimages` tel que `preimages.(i)` contiennent la liste des mots  $s \in S$  tels que  $\text{ligne}(s) = i$ ;
- on appelle `check_class` sur chacune de ces listes, ce qui a pour effet de vérifier que, en notant  $s$  le premier mot de la liste, on a  $\text{ligne}(sa) = \text{ligne}(s'a)$  pour tout  $s'$  de la liste et  $a \in \Sigma$ , et de lever l'exception voulue sinon.

OCaml : cor.ml

```

1 let check_consistent (table : Obs.t) =
2   let rec check s s' =
3     for letter = 0 to Obs.nb_letters table - 1 do
4       match Obs.separate_rows table (s @ [letter]) (s' @ [letter]) with
5       | None -> ()
6       | Some word -> raise (Inconsistent (letter :: word))
8     done in
9   let preimages = Array.make (Obs.nb_rows table) [] in
10  let process_word s =
11    let i = Obs.get_row_number table s in
12    preimages.(i) <- s :: preimages.(i) in
13  Obs.iter_s table process_word;
14  let check_class = function
15    | s :: rest -> List.iter (check s) rest
16    | _ -> () in
17  Array.iter check_class preimages

```

**Question 29** C'est très simple si l'on a compris la manière dont les deux fonctions précédentes ont été conçues ; essayer d'utiliser une boucle while, en revanche, risque de rendre le code très compliqué.

OCaml : cor.ml

```

1 let rec make_complete_and_coherent table teacher =
2   try
3     check_complete table;
4     check_consistent table
5   with
6   | Incomplete word ->
7     Obs.add_to_s table word teacher.member;
8     make_complete_and_coherent table teacher
9   | Inconsistent word ->
10    Obs.add_to_e table word teacher.member;
11    make_complete_and_coherent table teacher

```

**Question 30** Tout le travail a déjà été fait.

OCaml : cor.ml

```

1 let learn teacher =
2   let table = Obs.initial_table teacher.nb_letters teacher.member in
3   let rec outer_loop () =
4     make_complete_and_coherent table teacher;
5     let auto = construct_auto table in
6     match teacher.counter_example auto with
7     | None -> auto
8     | Some w ->
9       Obs.add_to_s table w teacher.member;
10      outer_loop () in
11   outer_loop ()

```

Ce sujet est directement adapté de l'article *Learning Regular Sets from Queries and Counter-Examples*, publié en 1987 par Dana Angluin, professeure à Yale et contributrice majeure au développement de l'apprentissage automatique (adaptation par M.Bianquis). Cet article a eu un impact assez important, et de nombreuses variantes de l'algorithme  $L^*$  ont ensuite été développées.

Vous êtes invités à réfléchir à des implémentations possibles du module Obs, basées par exemple sur des *tries* pour réaliser des dictionnaires dont l'ensemble des clés est un ensemble clos par préfixe (ou suffixe) de mots.

## Annexe

## Module Queue

Toutes les fonctions ci-dessous ont une complexité en (1). L'utilisation d'autres fonctions du module est autorisée, à condition de rappeler leur spécification; cependant, les fonctions fournies suffisent pour traiter le sujet.

OCaml

```
1 (* Crée une file vide *)
2 Queue.create : unit -> 'a Queue.t
3
4 (* Test de vacuité *)
5 Queue.is_empty : 'a Queue.t -> bool
6
7 (* Ajout d'un élément *)
8 Queue.push : 'a -> 'a Queue.t -> unit
9
10 (* Extraction de l'élément le plus ancien *)
11 (* Lève l'exception Queue.Empty si la file est vide *)
12 Queue.pop : 'a Queue.t -> 'a
```

*Indication pour la question 2. (pour montrer que DOMINATING SET est NP-difficile) : ajouter un nouveau sommet  $uv$  dans  $G$  pour chaque arête  $(u, v)$  présente dans le graphe.*

# Observations sur le DS Minimisation d'automates

## I - NP-complétude

- (A) Un ensemble dominant (pour Dominating Set) et un ensemble couvrant (pour Vertex Cover) ne sont pas la même chose ! On peut donner des exemples d'ensembles qui sont couvrants mais pas dominants et inversement. Par exemple, dans une clique à 3 sommets (par exemple un triangle), il suffit de prendre un unique sommet pour dominer le graphe. Tout le monde a alors un voisin dans  $D$ . Mais il faut en prendre  $n - 1$  pour couvrir toutes ses arêtes.
- Inversement, si  $G$  est un graphe ne contenant aucune arête, l'ensemble vide est couvrant, mais absolument pas dominant. Un ensemble dominant doit contenir tous les sommets du graphe.
- (B) Q2 : Quand on réduit VERTEX COVER à DOMINATING SET, il faut construire une fonction qui part d'une instance QUELCONQUE de VERTEX COVER (pas une instance positive !) et qui construit une instance de DOMINATING SET, **de sorte que** l'instance de départ *était* positive SSI son image est positive.
- En particulier, ça doit aussi marcher pour les instances négatives ! Si on part d'une instance négative, l'instance qu'on a construite doit être négative.

## II - Application du cours

- Q3 : Il y a une erreur dans mon corrigé (je ne l'ai vue qu'à la fin des corrections de copies),  $q_6 \in E(q_0)$  et est donc bien un état initial. (Une partie des points de la question a été offerte dans le calcul des notes pour compenser.)
- (C) Q3 : Attention, d'abord on lit une lettre, et ensuite on prend l' $\epsilon$ -fermeture. Des états comme  $q_1$  et  $q_0$  n'ont donc aucune transition sortante, et on peut les éliminer de l'automate.
- (D) Q5. 2. Ce langage est fini, il est donc rationnel ! On l'a montré dans le cours, vous pouvez penser à un langage fini comme à l'union finie de ses mots, et vous savez construire un automate qui reconnaît un unique mot  $u = u_1 \dots u_n$ . Attention à ce que vous écrivez ! Tout raisonnement à base de "de même que dans la question précédente" est grossièrement faux. Ici le lemme de l'étoile est bien respecté par ce langage : il suffit de prendre  $n \geq 1025$ , et on a aucun mot de longueur plus grande que  $n$  dans le langage. Toute preuve pour montrer le contraire ne peut qu'échouer.
- (E) Q4 : La plupart du temps, vous éliminez correctement  $q_1$  et  $q_3$ , mais vous faites beaucoup d'erreurs sur l'élimination de  $q_0$  et  $q_2$  ! Revoyez l'algorithme, il faut commencer par énumérer proprement tous les prédécesseurs et tous les successeurs pour ne pas oublier de transition à ajouter. Souvent, vous oubliez  $(q_2, q_2)$  et  $(q_I, q_F)$  en éliminant  $q_0$ .
- (F) Q5. 1. Attention, quand vous utilisez le lemme de l'étoile, vous ne pouvez **pas** choisir les mots  $x, y$  et  $z$  vous-mêmes pour aboutir à une absurdité ! Le lemme de l'étoile vous affirme juste qu'il en existe qui vérifient la propriété, mais ce n'est pas parce qu'un triplet  $(x, y, z)$  précis ne marche pas que c'est absurde. Il faut trouver une absurdité quels que soient les mots  $x, y, z$  qui vérifient les propriétés !

# INFORMATIQUE (VERSION NORMALE)

Durée : 4 heures

## Consignes :

- Veuillez à numéroté vos copies en bas à droite sous le format `numéro_de_la_page / nombre_total_de_pages`.
- Ne pas utiliser de correcteur blanc.
- Des points bonus/malus de présentation pourront éventuellement être accordés.
- Veuillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- **L'usage du cours ou de notes est strictement interdit.**

*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.*

Le sujet est constitué de trois parties. La Partie I est complètement indépendante du reste du sujet, qui forme un problème cohérent. Toutes les parties peuvent être traitées indépendamment les unes des autres. Plus une partie est loin du début du sujet, plus elle demande d'autonomie de la part du candidat et de ses codes. Ce sujet existe en deux versions : étoilée et non étoilée, **au choix**. Avec ce sujet sont fournis des fichiers de code et leurs headers associés, qui comprennent les fichiers à rendre (à trous).

**Consigne de rendu :** Votre rendu se fera sous la forme d'un dossier compressé portant votre nom de famille **uniquement**, sans espace, et contenant deux **dossiers** appelés `OCaml` et `C`. Ces dossiers contiendront chacun un **unique** fichier : `couplages.ml` et `capacite.c` respectivement. Vous devez rendre **les deux fichiers**, même si vous n'avez pas touché à l'un d'eux.

**Commencez par indiquer sur votre copie la version du sujet que vous traitez (étoilée ou non).**

**Les questions pratiques seront corrigées par un testeur automatique.** Le non-respect des consignes notamment en ce qui concerne les noms et les spécifications des fonctions entraînera donc automatiquement la note de 0 à la question.

Lorsque le candidat écrira une fonction, il pourra également définir des fonctions auxiliaires. 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.

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`).

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  $O(f(n, m))$  où  $n$  et  $m$  sont les tailles des arguments de la fonction, et  $f$  une expression la plus simple possible. Les calculs de complexité seront justifiés succinctement.

*Ce sujet comporte 8 pages (celle-ci comprise).*

**Ne retournez pas la page avant d'y être invités.**



## Définitions et notations

- Un *graphe non orienté* est un couple  $(S, A)$  où  $A \subseteq \mathcal{P}_2(S)$  (ensemble des parties à deux éléments de  $S$ ).
- S'il n'y a pas d'ambiguïté, une arête  $\{x, y\}$  pourra être notée  $xy$ .
- On notera  $G + xy$  le graphe  $G$  auquel on a ajouté l'arête  $xy$  et  $G - xy$  le graphe  $G$  auquel on a enlevé l'arête  $xy$ .
- Un *graphe pondéré* est un triplet  $(S, A, f)$  où  $f : A \rightarrow \mathbb{R}$  est une fonction dite de *pondération*.

## Partie 0 - Application du cours

Cette partie est purement théorique et est à traiter en premier.

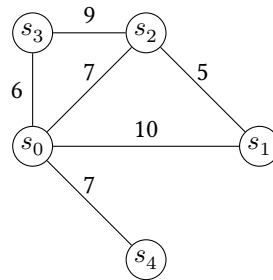


FIGURE 1 – Le graphe  $G_0$ .

**Question 1** Dessiner sans justifier un arbre couvrant de  $G_0$  de poids minimal.

**Question 2**

1. Donner un pseudo-code générique pour l'algorithme de Kruskal. On ne demande pas d'utiliser une structure union-find pour cette question (mais son utilisation est acceptée).
2. Quelle est la complexité de cet algorithme, en admettant qu'on peut tester et mettre à jour les composantes connexes en temps  $O(\log^*(n))$ ?

**Question 3** Décrire brièvement (sans preuve) une méthode pour trouver un ordre topologique des sommets d'un graphe orienté acyclique. Cet ordre est-il unique ? Si oui, le prouver. Si non, donner un contre-exemple.

**Question 4** Soit  $G = (S, A)$  un graphe non orienté. On fixe une numérotation  $\{a_1, \dots, a_p\}$  des arêtes, et l'on note  $G_i = (S, \{a_1, \dots, a_i\})$  pour  $0 \leq i \leq p$  (le graphe  $G_0$  n'a donc aucune arête). Montrer par récurrence que le graphe  $G_i$  possède au moins  $n - i$  composantes connexes, et en déduire que si  $G$  est connexe, alors  $|A| \geq |S| - 1$ .

**Question 5** Avec les notations précédentes, montrer que si  $G_i$  possède au moins  $n - i + 1$  composantes connexes, alors  $G_i$  possède un cycle. En déduire que si  $G$  est acyclique, alors  $|A| \leq |S| - 1$ .

**Question 6** En déduire l'équivalence entre les trois propriétés suivantes, pour  $G = (S, A)$  un graphe non orienté :

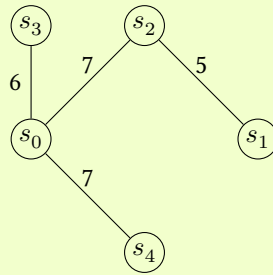
- (a)  $G$  est un arbre ;
- (b)  $G$  est connexe et  $|A| = |S| - 1$  ;
- (c)  $G$  est sans cycle et  $|A| = |S| - 1$ .

Pour les deux questions suivantes, on suppose que  $G = (S, A, f)$  est un graphe pondéré avec une fonction de pondération  $f$  injective (deux arêtes distinctes ne peuvent donc pas avoir le même poids).

**Question 7** Montrer que si  $G$  est connexe,  $G$  possède un unique arbre couvrant de poids minimal.

Si  $X$  est une partie de  $S$ , on note  $E(X)$  l'ensemble des arêtes  $xy \in A$  telles que  $x \in X$  et  $y \notin X$ .

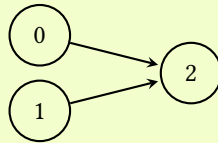
**Question 8** Soit  $X \subset S$  telle que  $X \neq \emptyset$  et  $\bar{X} \neq \emptyset$  (où  $\bar{X}$  est le complémentaire de  $X$  dans  $S$ ). Montrer que si  $T$  est un arbre couvrant minimal de  $G$ , alors  $T$  contient l'arête de poids minimal de  $E(X)$ .

**Solution de l'Exercice 1****Question 1****Question 2**

- cf Cours. Il y a deux pseudo-codes possible (un sans union-find et un avec).
- Notons  $p = |A|$ . Le tri des arêtes se fait en temps  $O(p \log p)$ . La boucle principale s'exécute  $p$  fois en temps  $O(\log^*(n))$  pour chaque itération (un test d'appartenance aux composantes connexes et une mise à jour des composantes connexes). Donc la boucle s'exécute en temps  $O(p \log^*(n))$ , dominé par le tri des arêtes. Ainsi cet algorithme a une complexité temporelle en  $O(p \log p) = O(p \log n)$ .

**Question 3** On renvoie les sommets du graphe dans l'ordre des dates de fin de parcours **décroissantes** lors d'un parcours en profondeur récursif.

Cet ordre n'est pas unique. Par exemple dans le DAG ci-dessous :



les ordres 0,1,2 et 1,0,2 sont des ordres topologiques valides.

**Question 4**

Par récurrence :

- pour  $i = 0$ ,  $G_0$  n'a pas d'arêtes, donc  $n = n - 0$  composantes connexes ;
- si on suppose le résultat vrai pour  $i$ , alors ajouter l'arête  $a_{i+1}$  peut diminuer le nombre de composantes connexes d'au plus 1.

On conclut par récurrence. Dès lors,  $G$  possède au moins  $n - k$  composantes connexes. Si  $G$  est connexe, on en déduit que  $n - k \leq 1$ , soit  $|S| - 1 \leq |A|$ .

**Question 5** Dans la récurrence précédente, si le nombre de composantes connexes ne diminue pas lors d'une étape c'est que l'arête ajoutée relie deux sommets de la même composante et crée donc un cycle.

Par conséquent, si  $G$  est acyclique chacune des  $p$  étapes a diminué le nombre de composantes, donc  $G$  a au plus  $n - p$  composantes connexes. Ce nombre de composantes valant au moins 1, on a donc  $n - p \geq 1$ , c'est-à-dire  $|S| - 1 \geq |A|$ .

**Question 6** En notant  $c$  le nombre de composantes connexes de  $G$ ,  $n$  le nombre de sommets et  $p$  le nombre d'arêtes, les arguments précédents montrent en fait que  $n - p \geq c$  avec égalité si et seulement si  $G$  est acyclique. Dès lors :

- (a)  $\Rightarrow$  (b)  $G$  est connexe donc  $c = 1$  et acyclique donc  $n - p = c$ , on a bien  $n - p = 1$ .
- (b)  $\Rightarrow$  (c)  $G$  est connexe donc  $c = 1$  et  $n - p = 1$ , donc  $n - p = c$  donc  $G$  est acyclique.
- (c)  $\Rightarrow$  (a)  $G$  est acyclique donc  $n - p = c$  et d'autre part  $n - p = 1$ , donc  $c = 1$  et  $G$  est connexe (et donc un arbre).

**Question 7** Notons  $T = (S, B)$  et  $T^* = (S, B^*)$ . Supposons  $T \neq T^*$  et soit  $a$  l'arête de poids minimal de  $B \Delta B^*$  (différence symétrique de  $B$  et  $B^*$ ). Sans perte de généralité,  $a \in B \setminus B^*$ . En rajoutant  $a$  à  $T^*$ , on crée un cycle. Ce cycle contient une arête  $a^* \in B^* \setminus B$  (sinon il y aurait un cycle dans  $T$ ). En supprimant cette arête, on obtient un nouvel arbre  $T'$ . Or par hypothèse,  $f(a) < f(a^*)$ , donc  $f(T') < f(T^*)$ , ce qui contredit la minimalité.

**Question 8** Pour commencer, on remarque que  $E(X)$  est non vide puisque  $G$  est connexe (s'il admet un arbre couvrant). Soient  $T$  un arbre couvrant de  $G$  et  $a = xy$  l'arête de poids minimal de  $E(X)$ , supposons  $a \notin T$ . En posant  $T' = T + a$ . Alors  $T'$  possède un cycle (il a trop d'arêtes) et ce cycle passe par  $a$ . Comme  $x \in X$  et  $y \in \bar{X}$ , le cycle contient nécessairement au moins une autre arête de  $E(X)$ , que l'on note  $a'$ . On a alors  $T'' = T + a - a'$  qui est connexe et vérifie  $p = n - 1$ , donc est un arbre couvrant. Or par définition de  $a$  et injectivité de  $f$  on a  $f(a) < f(a')$ , d'où  $f(T'') < f(T)$ , ce qui contredit le caractère minimal de  $T$ . On conclut par l'absurde.

## Partie 1 - Couplage maximum dans un graphe biparti

Cette partie est à traiter en OCaml.

Dans cette partie, on cherche à implémenter en OCaml la recherche d'un couplage de cardinalité maximale dans un graphe biparti, selon la méthode vue en cours.

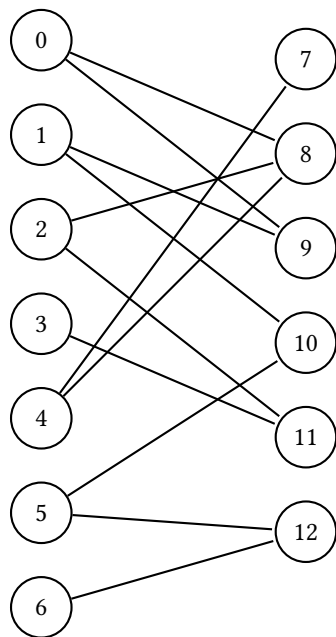
On travaillera dans cette partie avec les types suivants :

OCaml

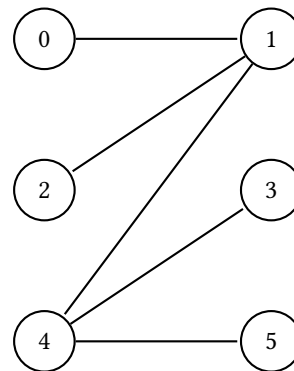
```
1 type sommet = int
2 type graphe = sommet list array
3
4 type arete = {x : sommet; y : sommet}
5
6 type couplage = arete list
7 type graphe_biparti = { g : graphe; partition : bool array }
```

- On représente un graphe par listes d'adjacence.
- Une arête est donnée par ses deux extrémités (notées  $x$  et  $y$ ).
- Un couplage est donné sous la forme d'une liste d'arêtes.
- Enfin, un graphe biparti est la donnée d'un graphe (qui doit être biparti) et d'une bipartition de ce graphe, sous la forme d'un tableau de booléens. Si  $S = X \sqcup Y$ , alors les sommets  $s \in S$  appartenant à l'ensemble  $X$  vérifient `partition.(s) = true` et ceux appartenant à  $Y$  vérifient `partition.(s) = false`.

On fournit un fichier `couplages.ml` dans lequel les deux graphes bipartis suivants sont déjà fournis (pour vous aider à tester) :



(a) Graphe biparti gb exemple du cours



(b) Graphe biparti gb2 tel que le meilleur couplage vérifie  $|C| = 2$

Un fichier corrigé déjà compilé `corrige.cmo` est fourni pour vous permettre de continuer après une question manquée. Vous pouvez librement utiliser les fonctions du corrigé à tout moment, mais vous n'aurez bien sûr pas les points pour une fonction `f` si cette fonction `f` fait appel à sa propre version corrigée `f_cor` (directement ou indirectement, par une fonction intermédiaire). Dans tous les autres cas, il n'y a aucun malus à utiliser le corrigé, y compris si une autre fonction `g` appelle `f_cor` (vous ne perdrez pas les points pour `f`).

Pour utiliser une fonction `f_cor` du corrigé, il faut avoir compilé et chargé le module `Corrige`. Vous avez donc deux façon d'exécuter votre fichier `couplages.ml` :

- En version compilée : un `Makefile` vous est fourni. Tapez simplement `make` en ligne de commande.
- Dans `utop` : commencez par taper `#load "corrige.cmo";;` dans `utop` (il n'y a besoin de le faire qu'une seule fois par ouverture de `utop`). Ensuite, vous pourrez utiliser votre fichier `couplages.ml` normalement, comme d'habitude, avec `#use "couplages.ml";;`.

## 1) Fonctions intermédiaires utiles

Commençons par quelques fonctions intermédiaires dont nous aurons besoin pour travailler sur les couplages.

1. Écrire une fonction `est_dans_couplage : arete -> arete list -> bool` telle que `est_dans_couplage ar c` renvoie `true` si l'arête `ar` est présente dans le couplage `c` et `false` sinon.  
**Attention :** une arête entre deux sommets  $u$  et  $v$  peut être représentée de deux manières suivantes, dans le sens  $u \rightarrow v$  ou  $v \rightarrow u$ . Dans les deux cas, on veut renvoyer `true`.
2. Écrire une fonction `difference_symetrique : arete list -> arete list -> arete list` telle que `difference_symetrique c1 c2` renvoie l'ensemble d'arêtes  $c1 \Delta c2$ , sous la forme d'une liste d'arêtes (sans doublons).  
*Indication : si besoin, vous pouvez commencer par une fonction intermédiaire `prive_de` qui renvoie `c \setminus c'`.*
3. Écrire une fonction `est_couvert : sommet -> arete list -> bool` telle que `est_couvert s c` renvoie `true` si le sommet `s` est couvert par le couplage `c` et `false` sinon, c'est à dire si le sommet `s` donné est une extrémité d'une des arêtes du couplage.

## 2) Graphe de couplage et recherche de chemin augmentant

L'algorithme de recherche de couplage maximum consiste à trouver successivement des chemins augmentants dans le graphe biparti pour améliorer successivement un couplage initialement vide. On utilise pour cela le graphe d'un couplage  $G_C$ . Construisons ce graphe en OCaml.

4. Écrire une fonction `graphe_de_couplage : graphe_biparti -> arete list -> graphe` telle que `graphe_de_couplage gb c` renvoie le graphe  $G_C$  du couplage `c` dans le graphe biparti `gb`.  
 Le sommet  $s$  ajouté sera d'indice  $n$  et le sommet  $t$  ajouté sera d'indice  $n + 1$ , avec  $n = |S|$ .  
**Remarque :** le graphe  $G_C$  renvoyé est orienté, et on ne demande pas d'en renvoyer une bipartition, seulement le graphe.

Dans le graphe du couplage  $G_C$ , on souhaite trouver un chemin de  $s$  à  $t$  pour en déduire un chemin augmentant de  $C$  dans le graphe d'origine.

5. Écrire une fonction `arbre_parours : graphe -> sommet -> sommet array` telle que `arbre_parours g s` renvoie l'arbre issu d'un parcours de graphe (de votre choix) depuis  $s$ , sous la forme d'un tableau `pred` des prédécesseurs (parents) de chaque sommet dans le parcours. La racine aura elle-même pour prédécesseur, et les sommets inaccessibles auront la valeur `-1` pour prédécesseur.
6. Écrire une fonction `chemin : graphe -> sommet -> sommet -> arete list` telle que `chemin g s t` renvoie un chemin reliant les sommets  $s$  à  $t$  dans le graphe `g`, s'il en existe, sous la forme d'une liste des arêtes à emprunter successivement depuis  $s$  pour atteindre  $t$  (**dans le bon sens, x vers y**). S'il n'en existe pas, on renverra la liste vide `[]`.

## 3) Recherche de couplage maximum

Il ne reste plus qu'à implémenter l'algorithme de recherche de couplage de cardinalité maximale. Rappelons le pseudo-code (très générique) de cet algorithme :

---

**Algorithme 4 :** Couplage maximum dans un graphe

---

**Entrées :** Un graphe  $G = (S, A)$  non orienté.

**Sorties :** Un couplage  $C \subseteq A$  de cardinal maximal.

---

- 1  $C \leftarrow \emptyset$
  - 2 **tant que** il existe un chemin  $c$  augmentant pour  $C$  **faire**
  - 3    $C \leftarrow C \Delta c$
  - 4 **renvoyer**  $C$
- 

7. Écrire une fonction `couplage_maximum_biparti : graphe_biparti -> arete list` telle que `couplage_maximum_biparti gb` renvoie un couplage de cardinalité maximale du graphe biparti `gb` donné en entrée, sous la forme d'une liste d'arêtes (sans doublons).
8. Testez votre fonction sur les graphes `gb` et `gb2`. Combien d'arêtes trouvez-vous dans vos couplages ? Justifier que ces couplages sont bien optimaux pour ces deux exemples.

## Partie 2 - Chemin de largeur maximale

Dans toute cette partie, on suppose que le graphe  $G = (S, A, f)$  est non orienté et connexe, et on représente toujours un graphe par listes d'adjacence.

On appelle *capacité* d'un chemin  $\sigma = (x_0, \dots, x_n)$ , et l'on note  $c(\sigma)$ , le minimum des poids de ses arêtes :

$$c(\sigma) = \min_{0 \leq i < n-1} f(x_i x_{i+1})$$

On note  $c_G^*(x, y)$  la capacité maximale entre  $x$  et  $y$  dans  $G$ , c'est-à-dire la valeur maximale de  $c(\sigma)$  pour  $\sigma$  un chemin de  $G$  reliant  $x$  à  $y$ . On appelle *goulot maximal* de  $x$  à  $y$  un chemin de  $x$  à  $y$  de capacité maximale.

**Remarque :**

- Si l'on imagine que le poids d'une arête représente une capacité de réseau (routier, informatique...), alors il est assez naturel de considérer que la capacité d'un chemin est sa largeur (puisque l'on est contraint par le tronçon de plus faible capacité). On cherche donc un chemin de capacité maximale entre deux sommets.

**Question 1** Déterminer, sans justification, un goulot maximal entre les sommets  $s_0$  et  $s_8$  du graphe  $G_1$  ci-dessous, ainsi que la valeur de  $c_{G_1}^*(s_0, s_8)$ .

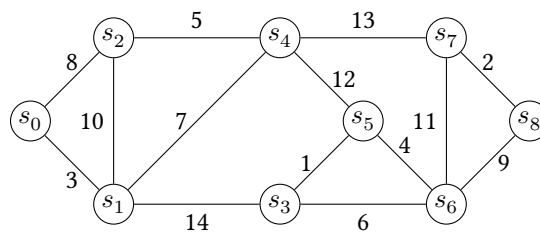


FIGURE 3 – Le graphe pondéré  $G_1$ .

Cette partie s'intéresse à plusieurs méthodes pour trouver un goulot maximal entre deux sommets donnés  $x$  et  $y$  d'un graphe  $G$ , et sa capacité.

**Question 2** Montrer que si  $c_G^*(x, y)$  est connu, alors on peut déterminer un goulot optimal entre  $x$  et  $y$  en temps linéaire en la taille de  $G$ . On décrira brièvement une méthode en Français et on justifiera brièvement sa complexité.

### Solution de l'Exercice 2

**Question 1** Le chemin  $(s_0, s_2, s_1, s_4, s_7, s_6, s_8)$  est un goulot maximal, de capacité 7.

**Question 2** On commence par retirer toutes les arêtes de  $G$  de poids inférieur strictement à  $c_G^*(x, y)$  (en temps linéaire en passant une première fois sur le graphe), puis on obtient un chemin entre  $x$  et  $y$  dans ce graphe filtré par un simple parcours de graphe en temps linéaire. Le chemin obtenu ne passe que par des arêtes de poids  $c_G^*(x, y)$  ou plus, donc il est de capacité maximale (on ne peut pas faire mieux).

## I - Première implémentation sous-optimale

### 1) Représentation des données et première fonction

On représente les graphes en C sous forme de tableau de listes d'adjacence avec les structures suivantes :

```
1 struct edge {
2     int x;
3     int y;
4     double weight;
5 };
6 typedef struct edge edge_t;
7
8 struct graph {
9     int n;
10    int* degrees;
11    edge_t** adj;
```

```
12 };
```

```
13 typedef struct graph graph_t;
```

- Pour un graphe  $G$ , représenté par un objet  $g$  de type `graph_t` :
  - $g.n$  est le nombre de sommets de  $G$ , les sommets étant numérotés  $0, \dots, n-1$ ;
  - $g.degrees$  est un tableau de  $n$  entiers tel que  $g.degrees[i]$  soit le degré du sommet  $i$ ;
  - $g.adj$  est un tableau de taille  $n$ , tel que  $g.adj[i]$  soit un tableau de taille  $g.deg[i]$ .
- Pour un sommet  $u$ , les éléments du tableau  $g.adj[u]$  correspondent aux arêtes incidentes au sommet  $u$ . Si  $g.adj[u][j]$  est égal à  $e$ , alors  $e.x$  sera égal à  $u$  et  $e.y$  indiquera l'autre extrémité de l'arête.  $e.weight$  correspond au poids de l'arête.
- Ainsi, chaque arête  $uv$  du graphe sera représentée deux fois dans la structure :
  - une fois comme élément de  $g.adj[u]$ , avec le champ  $x$  égal à  $u$  et le champ  $y$  égal à  $v$ ;
  - une fois comme élément de  $g.adj[v]$ , avec le champ  $x$  égal à  $v$  et le champ  $y$  égal à  $u$ .

Ces types, ainsi que quelques fonctions utiles de manipulation de graphes, sont fournis dans le fichier `graph.c` et son header associé `graph.h`. Vous n'avez pas à recopier ces types dans vos fichiers de code ! **Un fichier Makefile est fourni** pour gérer la compilation des différents fichiers donnés, et vous pouvez directement aller coder dans `capacite.c`.

**Un fichier corrigé compilé `corrige.o` et son header sont également fournis.** Comme pour la partie OCaml, vous pouvez librement utiliser les fonctions du corrigé n'importe quand, mais vous n'aurez pas les points accordés à une fonction  $f$  si celle-ci fait appel à son corrigé `f_cor`.

**Question 3** Écrire une fonction `int nb_edges(graph_t* g)` prenant en entrée un pointeur vers un graphe  $G$  et renvoyant son nombre d'arêtes.

## 2) Première résolution

On se propose d'abord de calculer tous les  $c_G^*(s, t)$  d'un coup. On fixe un graphe  $G$ . Notons  $m_{i,j}^k$  la capacité maximale d'un chemin reliant  $i$  à  $j$  dans  $G$  en passant par les sommets intermédiaires  $0$  à  $k-1$  uniquement.

**Question 4** Quelle valeur  $m_{i,j}^k$  cherche-t-on à calculer pour répondre au problème initial ?

**Question 5** Montrer que pour tout  $i, j$  sommets,  $m_{i,j}^0 = \begin{cases} +\infty & \text{si } i = j \\ -\infty & \text{si } i \neq j \text{ et il n'y a pas d'arêtes entre } i \text{ et } j \\ f(ij) & \text{sinon} \end{cases}$

**Question 6** Montrer que pour tout  $k \in \llbracket 0; n-1 \rrbracket$  et pour tout  $i, j$ , on a  $m_{i,j}^{k+1} = \max(m_{i,j}^k, \min(m_{i,k}^k, m_{k,j}^k))$ .

**Question 7** Écrire une fonction `double** matrice_initiale(graph_t* g)` qui renvoie la matrice  $(m_{i,j}^0)_{i,j}$  pour le graphe  $g$  dont on donne un pointeur.

*Indication : on pourra utiliser la valeur INFINITY de la bibliothèque math.h.*

**Question 8** Écrire une fonction `double** copy_matrice (double** m, int n)` qui renvoie une copie profonde de la matrice  $m$  de dimensions  $n \times n$  donnée en argument.

**Question 9** En déduire une fonction `double capacite_max(graph_t* g, int x, int y)` qui calcule la capacité maximale entre deux sommets dans un graphe.

*Attention à bien libérer la mémoire au fur et à mesure. On pourra écrire une fonction intermédiaire.*

**Question 10** Quelle est la complexité en temps de votre algorithme ? Et en espace ?

**Question 11** Quel algorithme (au programme de MP2I-MPI) connaissez-vous qui fonctionne sur le même principe ? A quelle famille d'algorithmes appartient-il, ainsi que celui que vous avez écrit ?

**Solution de l'Exercice 3**

**Question 4** On cherche à calculer  $m_{x,y}^n$ , qui correspond à  $c_G^*(x, y)$ . C'est la capacité maximale d'un chemin entre  $x$  et  $y$  en passant par tous les sommets intermédiaires possibles (0 à  $n - 1$ ).

**Question 5** On n'autorise à passer par aucun sommet intermédiaire. Le seul chemin possible entre  $i$  et  $j$  est donc le chemin vide (si  $i = j$ ) ou le chemin  $i \rightarrow j$ . La capacité maximale d'un chemin vide  $m_{i,i}^0$  est  $+\infty$  (on peut faire passer autant d'information qu'on veut, et plus généralement un minimum d'ensemble vide est le neutre du minimum).

Si  $i \neq j$  et qu'il n'y a pas d'arête entre  $i$  et  $j$ , on a  $m_{i,j}^0 = -\infty$  car il n'existe aucun chemin entre  $i$  et  $j$  (par convention, le neutre du max est  $-\infty$ ). Sinon, le seul et unique chemin entre  $i$  et  $j$  est de capacité  $f(i, j)$ .

**Question 6** Montrons que  $m_{i,j}^{k+1} = \max(m_{i,j}^k, \min(m_{i,k}^k, m_{k,j}^k))$ .

On compare le chemin de plus grande capacité possible qu'on puisse former sans passer par le sommet intermédiaire  $k$  (i.e.  $m_{i,j}^k$ ) et en passant par le sommet  $k$ , auquel cas le chemin de plus grande capacité obtainable est composé des chemins de plus grande capacité entre  $i$  et  $k$  et entre  $k$  et  $j$  (i.e.  $m_{i,k}^k$  et  $m_{k,j}^k$ ), et la largeur de ce chemin est le minimum des arêtes qui le composent, soit le minimum des largeurs des deux chemins.

**Question 7** Code.

**Question 8** Code.

**Question 9** Code.

**Question 10** On a seulement besoin de retenir une matrice dans laquelle on réécrit les coefficients au fur et à mesure (et éventuellement une matrice intermédiaire), soit du  $O(n^2)$  en mémoire.

On calcule itérativement chaque matrice, il y en a  $n$  au total. Chaque coefficient se calcule en temps  $O(1)$ , et chaque matrice contient  $n^2$  coefficients, d'où un temps de  $O(n^3)$  au total.

**Question 11** On s'est inspiré ici de l'algorithme de Floyd-Warshall. Ces deux algorithmes sont des exemples de programmation dynamique (bottom-up).

## II - Arbre couvrant maximal : algorithme de Prim

La méthode précédente est très très peu efficace pour calculer un goulot maximal entre deux sommets précis. On propose ici une nouvelle méthode, utilisant la notion d'arbre couvrant maximal.

**Question 12** Soit  $(s, t) \in S^2$  et  $T = (S, B)$  un arbre couvrant de poids **maximal** de  $G$ . Montrer qu'un chemin de  $s$  à  $t$  dans  $T$  est un goulot maximal de  $s$  à  $t$  dans  $G$ .

On pourrait adapter l'algorithme de Kruskal pour qu'il renvoie un arbre couvrant de poids maximal en parcourant les arêtes dans l'ordre inverse. Mais dans cette partie, on se propose d'étudier un autre algorithme de construction d'un arbre couvrant minimal, que l'on utilisera pour renvoyer un arbre couvrant maximal. Il s'agit de l'*algorithme de Prim* suivant :

---

### Algorithme 4 : Algorithme de Prim

---

**Entrées :** Un graphe pondéré non orienté connexe  $G = (S, A, \rho)$ .

**Sorties :** Un arbre couvrant minimal  $T$  de  $G$ .

```

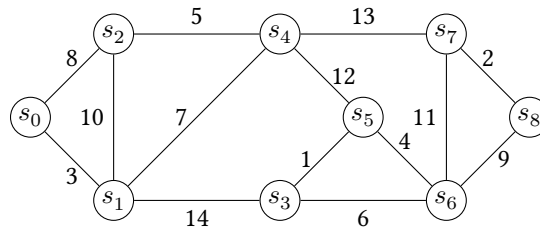
1  $F \leftarrow \emptyset$ 
2  $X \leftarrow \{x_0\}$  (un sommet quelconque de  $G$ )
3 tant que  $X \neq S$  faire
4   Trouver  $xy \in A$  de poids minimal telle que  $x \in X$  et  $y \in S \setminus X$ .
5    $F \leftarrow F \cup \{xy\}$ 
6    $X \leftarrow X \cup \{y\}$ 
7 renvoyer  $T = (X, F)$ 
```

---

**Question 13** Appliquer l'algorithme de Prim au graphe  $G_1$  de la figure 4 en partant du sommet  $s_0$ . On ne demande pas la description de l'exécution détaillée mais simplement une représentation graphique de l'arbre obtenu.

On admet ici que l'algorithme de Prim est totalement correct (même si ce serait un bon entraînement pour vous de le prouver !). Pour implémenter efficacement l'algorithme de Prim, on va utiliser une structure de file de priorité. Ces



FIGURE 4 – Le graphe pondéré  $G_1$ .

files de priorité contiendront des arêtes (des objets de type `edge_t`) et utiliseront les poids comme priorités. La structure a déjà été implémentée en utilisant un tas binaire, et la bibliothèque vous est fournie avec l'interface suivante :

```

1 // Création d'une file vide.
2 // L'argument capacity indique le nombre maximal
3 // d'arêtes que la file pourra contenir.
4 // La complexité de cette fonction est en  $O(\text{capacity})$ .
5 heap_t *heap_create(int capacity);
6
7 // Libération des ressources associées à une file
8 // Complexité  $O(1)$ 
9 void heap_free(heap_t *heap);
10
11 // Détermine si la file est vide
12 // Complexité  $O(1)$ 
13 bool heap_is_empty(heap_t *heap);
14
15 // Ajoute une arête à la file.
16 void heap_push(heap_t *heap, edge_t pair);
17
18 // Renvoie l'arête de poids minimal présente dans la file,
19 // et la supprime de la file.
20 // Erreur si la file est vide.
21 edge_t heap_extract_min(heap_t *heap);

```

**Question 14** Écrire une fonction `edge_t* prim(graph_t* g);` prenant en entrée un graphe  $G = (S, A, f)$ , supposé connexe, et renvoyant un arbre couvrant maximal de  $G$  sous la forme d'un tableau de  $|S| - 1$  arêtes. On demande une bonne complexité.

*Indication : on pourra ajouter dans une file de priorité les arêtes partant des sommets couverts au fur et à mesure qu'on les couvre. Avant de traiter une arête, on vérifiera qu'on n'a pas déjà couvert sa deuxième extrémité. La question vous laisse volontairement autonome sur l'implémentation de cette fonction. N'hésitez pas à utiliser des fonctions intermédiaires!*

**Question 15** Rappeler rapidement le principe de l'insertion d'un élément dans un tas binaire et de l'extraction du minimum, et en déduire les complexités des fonctions `heap_push` et `heap_extract_min` (en supposant bien sûr qu'elles sont correctement implémentées).

#### Question 16

Quelle est la complexité de l'algorithme de Prim que vous avez implémenté? Comparer avec celle de l'algorithme de Kruskal.

**Question 17** Comment trouver un arbre couvrant maximal à partir de l'algorithme de Prim?

Il ne resterait alors plus qu'à effectuer un parcours dans un arbre maximal pour trouver un goulot maximal.

### Solution de l'Exercice 4

**Question 12** Notons  $\sigma = (s = s_0, s_1, \dots, s_k = t)$  un chemin de  $s$  à  $t$  dans  $T$ . Supposons par l'absurde que  $\sigma$  n'est pas un goulot maximal de  $s$  à  $t$  dans  $G$  et soit  $\sigma^*$  un goulot maximal de  $s$  à  $t$  dans  $G$ . Par hypothèse,  $c(\sigma) < c(\sigma^*)$ . Soit  $a$  l'arête de poids minimal de  $\sigma$ . Alors  $(S, B \setminus \{a\})$  contient deux composantes connexes. Il existe une arête  $a'$  de  $\sigma^*$  qui relie ces deux composantes connexes (car  $\sigma^*$  est un chemin qui relie ces deux composantes). De plus,  $f(a') > f(a)$  (car  $f(a') \geq c(\sigma^*) > c(\sigma) = f(a)$ ), donc  $a' \neq a$ . On en déduit, comme dans la première partie, que

$T' = (S, B \cup a' \setminus \{a\})$  est un arbre couvrant, de poids  $f(T') = f(T) + f(a') - f(a) > f(T)$ , ce qui contredit la maximalité de  $T$ .

**Question 13** TODO

**Question 14** Code.

**Question 15** Cf cours de MP2I sur les tas pour tous les détails.

**Insertion** : On ajoute l'élément à insérer en fin de tableau, dans la première case disponible, puis on fait une percolation vers le haut pour rétablir la structure de tas : tant que l'élément inséré est plus petit que son parent, on l'échange avec son parent.

**Extraction du minimum** : On retient la racine et on la remplace par le dernier élément du tableau, puis on fait une percolation vers le bas pour rétablir l'ordre du tas : tant que l'élément qu'on a initialement mis à la racine est plus grand qu'un de ses enfants, on l'échange avec son plus petit enfant.

Ces deux opérations sont en temps propositionnel à la hauteur du tas, qui est un tas binaire donc un arbre binaire complet. On a donc  $h = \lfloor \log n \rfloor$  et la complexité de ces opérations est en  $O(\log n)$ .

**Question 16** La complexité de l'algorithme de Prim est en  $O(|A| \log |S|)$ . En effet, on ajoute une arête à  $X \subseteq S$  à chaque itération donc on effectue au plus  $|S|$  itérations de la boucle Tant que, et chaque itération est responsable de l'ajout d'un sommet  $y$  à  $X$ . On trouve l'arête de poids minimal à l'aide d'une extraction de la file de priorité en temps  $O(\log |A|) = O(\log |S|)$  au plus. Mais l'ajout d'un sommet  $y$  à  $X$  nous force à mettre à jour la file de priorité en traitant chaque arête incidente au sommet ajouté, en temps  $O(\deg(y) \cdot \log(|S|))$ . Au total, on obtient donc une complexité :

$$O\left(\sum_{y \in S} (1 + \deg(y)) \cdot \log |S|\right) = O((|A| + |S|) \log |S|)$$

Comme le graphe est connexe, on sait que  $|A| \geq |S| - 1$  et on peut simplifier en  $O(|A| \log |S|)$ .

**Question 17** Pour trouver un arbre couvrant maximal de  $G$ , il suffit d'inverser tous les poids des arêtes dans  $G$ , i.e. de trouver un arbre couvrant maximal de  $(S, A, -f)$ . On peut aussi adapter l'algorithme en utilisant une file-max pour trouver à chaque étape une arête de poids maximal.

### III - Implémentation en temps linéaire

On peut faire encore mieux ! Il est possible de trouver un goulot maximal entre deux sommets d'un graphe en temps linéaire en la taille du graphe.

Dans toute cette partie, on suppose que la fonction de pondération est injective. Pour  $G = (S, A, f)$  un graphe non orienté pondéré et  $X \subseteq S$ , on appelle opération de fusion de  $X$  dans  $G$  l'opération qui consiste à remplacer  $G$  par le graphe  $G' = (S', A', f')$  où :

- $S' = S \setminus X \cup \{x\}$ , où  $x$  est un nouveau sommet ;
- $A' = (P_2(S \setminus X) \cap A) \cup \{\{s, x\} \mid s \in S \setminus X, \exists t \in X, s, t \in A\}$  ;
- pour  $a \in A'$  :
  - si  $a \in A$ ,  $f'(a) = f(a)$  ;
  - sinon,  $a = \{s, x\}$  et  $f'(a) = \max\{f(s, t) \mid t \in X\}$ .

Autrement dit, on fusionne les sommets de  $X$  en un seul, on laisse les arêtes qui relient  $X$  à un autre sommet, et on garde le poids maximal lorsqu'il y a plusieurs choix.

On considère l'algorithme suivant :

---

#### Algorithme 5 : Algorithme BSP (Bottleneck Shortest Path)

---

**Entrées** : Un graphe pondéré non orienté connexe  $G = (S, A, f)$ , une source  $s \in S$  et une destination  $t \in S$ .

- 1 **tant que**  $|A| > 1$  **faire**
  - 2      $M \leftarrow$  médiane de  $\{f(a) \mid a \in A\}$ .
  - 3     Supprimer de  $A$  les arêtes  $a$  de poids  $f(a) < M$ .
  - 4     **si** il n'existe pas de chemin de  $s$  à  $t$  **alors**
  - 5         Poser  $X_1, \dots, X_k$  les composantes connexes de  $G$ .
  - 6         Rajouter les arêtes supprimées avant le test.
  - 7         Fusionner  $X_1, \dots, X_k$  dans  $G$ .
  - 8 **renvoyer**  $f(a)$  où  $a$  est l'unique arête de  $G$ .
-

On admet qu'un passage dans la boucle Tant que peut se réaliser en temps  $O(|A|)$ , où  $A$  désigne ici l'ensemble des arêtes à l'entrée dans la boucle (qui est donc modifié d'un passage dans la boucle au suivant).

**Question 18** Montrer que l'ensemble de l'algorithme a une complexité linéaire en  $|A|$ , où  $A$  est l'ensemble initial des arêtes.

**Question 19** Montrer que l'algorithme renvoie la capacité d'un goulot maximal de  $s$  à  $t$  dans  $G$ .

#### Solution de l'Exercice 4

**Question 18** Remarquons que le nombre d'arêtes est environ divisé par deux à chaque itération :

- si la suppression des arêtes laisse un chemin de  $s$  à  $t$ , on a supprimé  $\lfloor \frac{|A|}{2} \rfloor$  arêtes (celles de poids inférieur à la médiane);
- sinon, on n'a gardé que ces arêtes là, et supprimé les autres, soit supprimé  $\lceil \frac{|A|}{2} \rceil$  arêtes.

Le nombre de passages dans la boucle est donc de l'ordre de  $\log_2 |A|$  au maximum, et la complexité totale est en

$$O\left(\sum_{k=0}^{+\infty} \frac{|A|}{2^k}\right) = O(|A|).$$

**Question 19** Montrons qu'il existe un goulot maximal de  $s$  à  $t$  dont l'arête de poids minimal n'est jamais supprimée :

- si la suppression des arêtes laisse un chemin de  $s$  à  $t$ , alors les arêtes supprimées ne font pas partie d'un goulot maximal (car la capacité des chemins restants est strictement plus grande que le poids des arêtes supprimées);
- sinon, le goulot maximal passe nécessairement par l'une de ces arêtes, et l'une d'entre elles est de poids minimal dans le goulot maximal. Le fait de fusionner les composantes ne change pas la suite du problème.

# INFORMATIQUE (VERSION ÉTOILÉE)

Durée : 4 heures

## Consignes :

- Veillez à numéroté vos copies en bas à droite sous le format `numéro_de_la_page / nombre_total_de_pages`.
- Ne pas utiliser de correcteur blanc.
- Des points bonus/malus de présentation pourront éventuellement être accordés.
- Veillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- **L'usage du cours ou de notes est strictement interdit.**

*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.*

Le sujet est constitué de trois parties. La Partie I est complètement indépendante du reste du sujet, qui forme un problème cohérent. Toutes les parties peuvent être traitées indépendamment les unes des autres. Plus une partie est loin du début du sujet, plus elle demande d'autonomie de la part du candidat et de ses codes. Ce sujet existe en deux versions : étoilée et non étoilée, **au choix**. Avec ce sujet sont fournis des fichiers de code et leurs headers associés, qui comprennent les fichiers à rendre (à trous).

**Consigne de rendu :** Votre rendu se fera sous la forme d'un dossier compressé portant votre nom de famille **unique**ment, sans espace, et contenant deux **dossiers** appelés `OCaml` et `C`. Ces dossiers contiendront chacun un **unique** fichier : `couplages.ml` et `capacite.c` respectivement. Vous devez rendre **les deux fichiers**, même si vous n'avez pas touché à l'un d'eux.

**Commencez par indiquer sur votre copie la version du sujet que vous traitez (étoilée ou non).**

**Les questions pratiques seront corrigées par un testeur automatique.** Le non-respect des consignes notamment en ce qui concerne les noms et les spécifications des fonctions entraînera donc automatiquement la note de 0 à la question.

Lorsque le candidat écrira une fonction, il pourra également définir des fonctions auxiliaires. 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.

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`).

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  $O(f(n, m))$  où  $n$  et  $m$  sont les tailles des arguments de la fonction, et  $f$  une expression la plus simple possible. Les calculs de complexité seront justifiés succinctement.

*Ce sujet comporte 8 pages (celle-ci comprise).*

## Ne retournez pas la page avant d'y être invités.

## Définitions et notations

- Un *graphe non orienté* est un couple  $(S, A)$  où  $A \subseteq \mathcal{P}_2(S)$  (ensemble des parties à deux éléments de  $S$ ).
- S'il n'y a pas d'ambiguïté, une arête  $\{x, y\}$  pourra être notée  $xy$ .
- On notera  $G + xy$  le graphe  $G$  auquel on a ajouté l'arête  $xy$  et  $G - xy$  le graphe  $G$  auquel on a enlevé l'arête  $xy$ .
- Un *graphe pondéré* est un triplet  $(S, A, f)$  où  $f : A \rightarrow \mathbb{R}$  est une fonction dite de *pondération*.

## Partie 0 - Application du cours

Cette partie est purement théorique et est à traiter en premier.

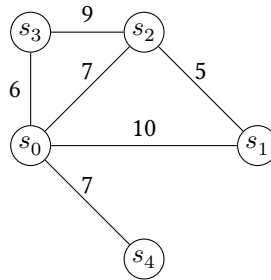


FIGURE 1 – Le graphe  $G_0$ .

**Question 1** Soit  $G = (S, A)$  un graphe non orienté. On fixe une numérotation  $\{a_1, \dots, a_p\}$  des arêtes, et l'on note  $G_i = (S, \{a_1, \dots, a_i\})$  pour  $0 \leq i \leq p$  (le graphe  $G_0$  n'a donc aucune arête). Montrer par récurrence que le graphe  $G_i$  possède au moins  $n - i$  composantes connexes, et en déduire que si  $G$  est connexe, alors  $|A| \geq |S| - 1$ .

**Question 2** Avec les notations précédentes, montrer que si  $G_i$  possède au moins  $n - i + 1$  composantes connexes, alors  $G_i$  possède un cycle. En déduire que si  $G$  est acyclique, alors  $|A| \leq |S| - 1$ .

**Question 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 avec une fonction de pondération  $f$  injective (deux arêtes distinctes ne peuvent donc pas avoir le même poids).

**Question 4** Montrer que si  $G$  est connexe,  $G$  possède un unique arbre couvrant de poids minimal.

Si  $X$  est une partie de  $S$ , on note  $E(X)$  l'ensemble des arêtes  $xy \in A$  telles que  $x \in X$  et  $y \notin X$ .

**Question 5** Soit  $X \subset S$  telle que  $X \neq \emptyset$  et  $\bar{X} \neq \emptyset$  (où  $\bar{X}$  est le complémentaire de  $X$  dans  $S$ ). Montrer que si  $T$  est un arbre couvrant minimal de  $G$ , alors  $T$  contient l'arête de poids minimal de  $E(X)$ .

### Solution de l'Exercice 1

#### Question 1

Par récurrence :

- pour  $i = 0$ ,  $G_0$  n'a pas d'arêtes, donc  $n = n - 0$  composantes connexes;
- si on suppose le résultat vrai pour  $i$ , alors ajouter l'arête  $a_{i+1}$  peut diminuer le nombre de composantes connexes d'au plus 1.

On conclut par récurrence. Dès lors,  $G$  possède au moins  $n - k$  composantes connexes. Si  $G$  est connexe, on en déduit que  $n - k \leq 1$ , soit  $|S| - 1 \leq |A|$ .

**Question 2** Dans la récurrence précédente, si le nombre de composantes connexes ne diminue pas lors d'une étape c'est que l'arête ajoutée relie deux sommets de la même composante et crée donc un cycle.

Par conséquent, si  $G$  est acyclique chacune des  $p$  étapes a diminué le nombre de composantes, donc  $G$  a au plus

$n - p$  composantes connexes. Ce nombre de composantes valant au moins 1, on a donc  $n - p \geq 1$ , c'est-à-dire  $|S| - 1 \geq |A|$ .

**Question 3** En notant  $c$  le nombre de composantes connexes de  $G$ ,  $n$  le nombre de sommets et  $p$  le nombre d'arêtes, les arguments précédents montrent en fait que  $n - p \geq c$  avec égalité si et seulement si  $G$  est acyclique. Dès lors :

(a)  $\Rightarrow$  (b)  $G$  est connexe donc  $c = 1$  et acyclique donc  $n - p = c$ , on a bien  $n - p = 1$ .

(b)  $\Rightarrow$  (c)  $G$  est connexe donc  $c = 1$  et  $n - p = 1$ , donc  $n - p = c$  donc  $G$  est acyclique.

(c)  $\Rightarrow$  (a)  $G$  est acyclique donc  $n - p = c$  et d'autre part  $n - p = 1$ , donc  $c = 1$  et  $G$  est connexe (et donc un arbre).

**Question 4** Notons  $T = (S, B)$  et  $T^* = (S, B^*)$ . Supposons  $T \neq T^*$  et soit  $a$  l'arête de poids minimal de  $B \Delta B^*$  (différence symétrique de  $B$  et  $B^*$ ). Sans perte de généralité,  $a \in B \setminus B^*$ . En rajoutant  $a$  à  $T^*$ , on crée un cycle. Ce cycle contient une arête  $a^* \in B^* \setminus B$  (sinon il y aurait un cycle dans  $T$ ). En supprimant cette arête, on obtient un nouvel arbre  $T'$ . Or par hypothèse,  $f(a) < f(a^*)$ , donc  $f(T') < f(T^*)$ , ce qui contredit la minimalité.

**Question 5** Pour commencer, on remarque que  $E(X)$  est non vide puisque  $G$  est connexe (s'il admet un arbre couvrant). Soient  $T$  un arbre couvrant de  $G$  et  $a = xy$  l'arête de poids minimal de  $E(X)$ , supposons  $a \notin T$ . En posant  $T' = T + a$ . Alors  $T'$  possède un cycle (il a trop d'arêtes) et ce cycle passe par  $a$ . Comme  $x \in X$  et  $y \in \bar{X}$ , le cycle contient nécessairement au moins une autre arête de  $E(X)$ , que l'on note  $a'$ . On a alors  $T'' = T + a - a'$  qui est connexe et vérifie  $p = n - 1$ , donc est un arbre couvrant. Or par définition de  $a$  et injectivité de  $f$  on a  $f(a) < f(a')$ , d'où  $f(T'') < f(T)$ , ce qui contredit le caractère minimal de  $T$ . On conclut par l'absurde.

## Partie 1 - Couplage maximum dans un graphe biparti

Cette partie est à traiter en OCaml.

Dans cette partie, on cherche à implémenter en OCaml la recherche d'un couplage de cardinalité maximale dans un graphe biparti, selon la méthode vue en cours.

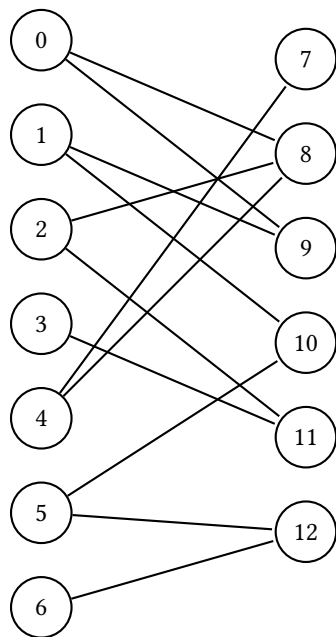
On travaillera dans cette partie avec les types suivants :

OCaml

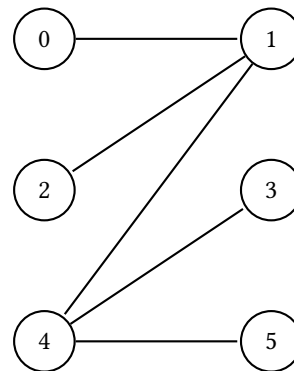
```
1 type sommet = int
2 type graphe = sommet list array
3
4 type arete = {x : sommet; y : sommet}
5
6 type couplage = arete list
7 type graphe_biparti = { g : graphe; partition : bool array }
```

- On représente un graphe par listes d'adjacence.
- Une arête est donnée par ses deux extrémités (notées  $x$  et  $y$ ).
- Un couplage est donné sous la forme d'une liste d'arêtes.
- Enfin, un graphe biparti est la donnée d'un graphe (qui doit être biparti) et d'une bipartition de ce graphe, sous la forme d'un tableau de booléens. Si  $S = X \sqcup Y$ , alors les sommets  $s \in S$  appartenant à l'ensemble  $X$  vérifient `partition.(s) = true` et ceux appartenant à  $Y$  vérifient `partition.(s) = false`.

On fournit un fichier `couplages.ml` dans lequel les deux graphes bipartis suivants sont déjà fournis (pour vous aider à tester) :



(a) Graphe biparti gb exemple du cours



(b) Graphe biparti gb2 tel que le meilleur couplage vérifie  $|C| = 2$

Un fichier corrigé déjà compilé `corrige.cmo` est fourni pour vous permettre de continuer après une question manquée. Vous pouvez librement utiliser les fonctions du corrigé à tout moment, mais vous n'aurez bien sûr pas les points pour une fonction `f` si cette fonction `f` fait appel à sa propre version corrigée `f_cor` (directement ou indirectement, par une fonction intermédiaire). Dans tous les autres cas, il n'y a aucun malus à utiliser le corrigé, y compris si une autre fonction `g` appelle `f_cor` (vous ne perdrez pas les points pour `f`).

Pour utiliser une fonction `f_cor` du corrigé, il faut avoir compilé et chargé le module `Corrige`. Vous avez donc deux façon d'exécuter votre fichier `couplages.ml` :

- En version compilée : un `Makefile` vous est fourni. Tapez simplement `make` en ligne de commande.
- Dans `utop` : commencez par taper `#load "corrige.cmo";;` dans `utop` (il n'y a besoin de le faire qu'une seule fois par ouverture de `utop`). Ensuite, vous pourrez utiliser votre fichier `couplages.ml` normalement, comme d'habitude, avec `#use "couplages.ml";;`.

## 1) Fonctions intermédiaires utiles

Commençons par quelques fonctions intermédiaires dont nous aurons besoin pour travailler sur les couplages.

1. Écrire une fonction `est_dans_couplage : arete -> arete list -> bool` telle que `est_dans_couplage ar c` renvoie `true` si l'arête `ar` est présente dans le couplage `c` et `false` sinon.  
**Attention :** une arête entre deux sommets  $u$  et  $v$  peut être représentée de deux manières suivantes, dans le sens  $u \rightarrow v$  ou  $v \rightarrow u$ . Dans les deux cas, on veut renvoyer `true`.
2. Écrire une fonction `difference_symetrique : arete list -> arete list -> arete list` telle que `difference_symetrique c1 c2` renvoie l'ensemble d'arêtes  $c1 \Delta c2$ , sous la forme d'une liste d'arêtes (sans doublons).
3. Écrire une fonction `est_couvert : sommet -> arete list -> bool` telle que `est_couvert s c` renvoie `true` si le sommet `s` est couvert par le couplage `c` et `false` sinon, c'est à dire si le sommet `s` donné est une extrémité d'une des arêtes du couplage.

## 2) Graphe de couplage et recherche de chemin augmentant

L'algorithme de recherche de couplage maximum consiste à trouver successivement des chemins augmentants dans le graphe biparti pour améliorer successivement un couplage initialement vide. On utilise pour cela le graphe d'un couplage  $G_C$ . Construisons ce graphe en OCaml.

4. Écrire une fonction `graphe_de_couplage : graphe_biparti -> arete list -> graphe` telle que `graphe_de_couplage gb c` renvoie le graphe  $G_C$  du couplage `c` dans le graphe biparti `gb`.  
 Le sommet  $s$  ajouté sera d'indice  $n$  et le sommet  $t$  ajouté sera d'indice  $n + 1$ , avec  $n = |S|$ .  
**Remarque :** le graphe  $G_C$  renvoyé est orienté, et on ne demande pas d'en renvoyer une bipartition, seulement le graphe.

Dans le graphe du couplage  $G_C$ , on souhaite trouver un chemin de  $s$  à  $t$  pour en déduire un chemin augmentant de  $C$  dans le graphe d'origine.

5. Écrire une fonction `arbre_parours : graphe -> sommet -> sommet array` telle que `arbre_parours g s` renvoie l'arbre issu d'un parcours de graphe (de votre choix) depuis  $s$ , sous la forme d'un tableau `pred` des prédécesseurs (parents) de chaque sommet dans le parcours (**dans le bon sens, x vers y**). La racine aura elle-même pour prédécesseur, et les sommets inaccessibles auront la valeur `-1` pour prédécesseur.
6. Écrire une fonction `chemin : graphe -> sommet -> sommet -> arete list` telle que `chemin g s t` renvoie un chemin reliant les sommets  $s$  à  $t$  dans le graphe `g`, s'il en existe, sous la forme d'une liste des arêtes à emprunter successivement depuis  $s$  pour atteindre  $t$ . S'il n'en existe pas, on renverra la liste vide `[]`.

## 3) Recherche de couplage maximum

Il ne reste plus qu'à implémenter l'algorithme de recherche de couplage de cardinalité maximale. Rappelons le pseudo-code (très générique) de cet algorithme :

---

**Algorithme 4 :** Couplage maximum dans un graphe

---

**Entrées :** Un graphe  $G = (S, A)$  non orienté.

**Sorties :** Un couplage  $C \subseteq A$  de cardinal maximal.

- 1  $C \leftarrow \emptyset$
  - 2 **tant que** *il existe un chemin  $c$  augmentant pour  $C$*  **faire**
  - 3    $C \leftarrow C \Delta c$
  - 4 **renvoyer**  $C$
- 

7. Écrire une fonction `couplage_maximum_biparti : graphe_biparti -> arete list` telle que `couplage_maximum_biparti gb` renvoie un couplage de cardinalité maximale du graphe biparti `gb` donné en entrée, sous la forme d'une liste d'arêtes (sans doublons).
8. Testez votre fonction sur les graphes `gb` et `gb2`. Combien d'arêtes trouvez-vous dans vos couplages? Justifier que ces couplages sont bien optimaux pour ces deux exemples.



## Partie 2 - Chemin de largeur maximale

Dans toute cette partie, on suppose que le graphe  $G = (S, A, f)$  est non orienté et connexe, et on représente toujours un graphe par listes d'adjacence.

On appelle *capacité* d'un chemin  $\sigma = (x_0, \dots, x_n)$ , et l'on note  $c(\sigma)$ , le minimum des poids de ses arêtes :

$$c(\sigma) = \min_{0 \leq i < n-1} f(x_i x_{i+1})$$

On note  $c_G^*(x, y)$  la capacité maximale entre  $x$  et  $y$  dans  $G$ , c'est-à-dire la valeur maximale de  $c(\sigma)$  pour  $\sigma$  un chemin de  $G$  reliant  $x$  à  $y$ . On appelle *goulot maximal* de  $x$  à  $y$  un chemin de  $x$  à  $y$  de capacité maximale.

**Remarque :**

- Si l'on imagine que le poids d'une arête représente une capacité de réseau (routier, informatique...), alors il est assez naturel de considérer que la capacité d'un chemin est sa largeur (puisque l'on est contraint par le tronçon de plus faible capacité). On cherche donc un chemin de capacité maximale entre deux sommets.

**Question 1** Déterminer, sans justification, un goulot maximal entre les sommets  $s_0$  et  $s_8$  du graphe  $G_1$  ci-dessous, ainsi que la valeur de  $c_{G_1}^*(s_0, s_8)$ .

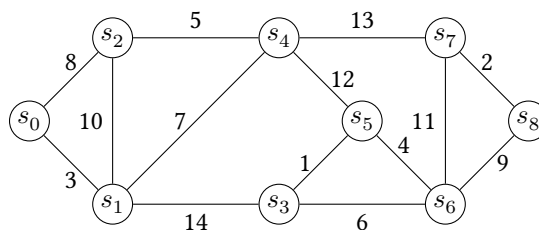


FIGURE 3 – Le graphe pondéré  $G_1$ .

Cette partie s'intéresse à plusieurs méthodes pour trouver un goulot maximal entre deux sommets donnés  $x$  et  $y$  d'un graphe  $G$ , et sa capacité.

**Question 2** Montrer que si  $c_G^*(x, y)$  est connu, alors on peut déterminer un goulot optimal entre  $x$  et  $y$  en temps linéaire en la taille de  $G$ . On décrira brièvement une méthode en Français et on justifiera brièvement sa complexité.

### Solution de l'Exercice 2

**Question 1** Le chemin  $(s_0, s_2, s_1, s_4, s_7, s_6, s_8)$  est un goulot maximal, de capacité 7.

**Question 2** On commence par retirer toutes les arêtes de  $G$  de poids inférieur strictement à  $c_G^*(x, y)$  (en temps linéaire en passant une première fois sur le graphe), puis on obtient un chemin entre  $x$  et  $y$  dans ce graphe filtré par un simple parcours de graphe en temps linéaire. Le chemin obtenu ne passe que par des arêtes de poids  $c_G^*(x, y)$  ou plus, donc il est de capacité maximale (on ne peut pas faire mieux).

## I - Première implémentation sous-optimale

### 1) Représentation des données et première fonction

On représente les graphes en C sous forme de tableau de listes d'adjacence avec les structures suivantes :

```
1 struct edge {
2     int x;
3     int y;
4     double weight;
5 };
6 typedef struct edge edge_t;
7
8 struct graph {
9     int n;
10    int* degrees;
11    edge_t** adj;
```

```
12 };
13 typedef struct graph graph_t;
```

- Pour un graphe  $G$ , représenté par un objet  $g$  de type `graph_t` :
  - $g.n$  est le nombre de sommets de  $G$ , les sommets étant numérotés  $0, \dots, n-1$ ;
  - $g.degrees$  est un tableau de  $n$  entiers tel que  $g.degrees[i]$  soit le degré du sommet  $i$ ;
  - $g.adj$  est un tableau de taille  $n$ , tel que  $g.adj[i]$  soit un tableau de taille  $g.deg[i]$ .
- Pour un sommet  $u$ , les éléments du tableau  $g.adj[u]$  correspondent aux arêtes incidentes au sommet  $u$ . Si  $g.adj[u][j]$  est égal à  $e$ , alors  $e.x$  sera égal à  $u$  et  $e.y$  indiquera l'autre extrémité de l'arête.  $e.weight$  correspond au poids de l'arête.
- Ainsi, chaque arête  $uv$  du graphe sera représentée deux fois dans la structure :
  - une fois comme élément de  $g.adj[u]$ , avec le champ  $x$  égal à  $u$  et le champ  $y$  égal à  $v$ ;
  - une fois comme élément de  $g.adj[v]$ , avec le champ  $x$  égal à  $v$  et le champ  $y$  égal à  $u$ .

Ces types, ainsi que quelques fonctions utiles de manipulation de graphes, sont fournis dans le fichier `graph.c` et son header associé `graph.h`. Vous n'avez pas à recopier ces types dans vos fichiers de code ! **Un fichier Makefile est fourni** pour gérer la compilation des différents fichiers donnés, et vous pouvez directement aller coder dans `capacite.c`.

**Un fichier corrigé compilé `corrige.o` et son header sont également fournis.** Comme pour la partie OCaml, vous pouvez librement utiliser les fonctions du corrigé n'importe quand, mais vous n'aurez pas les points accordés à une fonction  $f$  si celle-ci fait appel à son corrigé `f_cor`.

## 2) Première résolution

On se propose d'abord de calculer tous les  $c_G^*(s, t)$  d'un coup, à la manière de l'algorithme de Floyd-Warshall. On fixe un graphe  $G$ . Notons  $m_{i,j}^k$  la capacité maximale d'un chemin reliant  $i$  à  $j$  dans  $G$  en passant par les sommets intermédiaires  $0$  à  $k-1$  uniquement.

**Question 3** Quelle valeur  $m_{i,j}^k$  cherche-t-on à calculer pour répondre au problème initial ?

**Question 4** Montrer que pour tout  $i, j$  sommets,  $m_{i,j}^0 = \begin{cases} +\infty & \text{si } i = j \\ -\infty & \text{si } i \neq j \text{ et il n'y a pas d'arêtes entre } i \text{ et } j \\ f(ij) & \text{sinon} \end{cases}$

**Question 5** Trouver une formule de récurrence : exprimer  $m_{i,j}^{k+1}$  en fonction des  $m_{i',j'}^k$ , avec  $k' < k$ . Justifier.

**Question 6** En déduire une fonction `double capacite_max(graph_t* g, int x, int y)` qui calcule la capacité maximale entre deux sommets dans un graphe.

*Indication : on pourra utiliser la valeur INFINITY de la bibliothèque math.h.*

*Attention à bien libérer la mémoire au fur et à mesure. On pourra écrire des fonctions intermédiaires...*

**Question 7** Quelle est la complexité en temps de votre algorithme ? Et en espace ?

**Question 8** A quelle famille d'algorithmes cet algorithme appartient-il, ainsi que celui de Floyd-Warshall ?

### Solution de l'Exercice 3

**Question 4** On cherche à calculer  $m_{x,y}^n$ , qui correspond à  $c_G^*(x, y)$ . C'est la capacité maximale d'un chemin entre  $x$  et  $y$  en passant par tous les sommets intermédiaires possibles ( $0$  à  $n-1$ ).

**Question 5** On n'autorise à passer par aucun sommet intermédiaire. Le seul chemin possible entre  $i$  et  $j$  est donc le chemin vide (si  $i = j$ ) ou le chemin  $i \rightarrow j$ . La capacité maximale d'un chemin vide  $m_{i,i}^0$  est  $+\infty$  (on peut faire passer autant d'information qu'on veut, et plus généralement un minimum d'ensemble vide est le neutre du minimum).

Si  $i \neq j$  et qu'il n'y a pas d'arête entre  $i$  et  $j$ , on a  $m_{i,j}^0 = -\infty$  car il n'existe aucun chemin entre  $i$  et  $j$  (par convention, le neutre du max est  $-\infty$ ). Sinon, le seul et unique chemin entre  $i$  et  $j$  est de capacité  $f(ij)$ .

**Question 6** Montrons que  $m_{i,j}^{k+1} = \max(m_{i,j}^k, \min(m_{i,k}^k, m_{k,j}^k))$ .

On compare le chemin de plus grande capacité possible qu'on puisse former sans passer par le sommet intermédiaire  $k$  (i.e.  $m_{i,j}^k$ ) et en passant par le sommet  $k$ , auquel cas le chemin de plus grande capacité obtainable est composé des chemins de plus grande capacité entre  $i$  et  $k$  et entre  $k$  et  $j$  (i.e.  $m_{i,k}^k$  et  $m_{k,j}^k$ ), et la largeur de ce chemin est le minimum des arêtes qui le composent, soit le minimum des largeurs des deux chemins.

**Question 7** Code.

**Question 8** On a seulement besoin de retenir une matrice dans laquelle on réécrit les coefficients au fur et à mesure (et éventuellement une matrice intermédiaire), soit du  $O(n^2)$  en mémoire.

On calcule itérativement chaque matrice, il y en a  $n$  au total. Chaque coefficient se calcule en temps  $O(1)$ , et chaque matrice contient  $n^2$  coefficients, d'où un temps de  $O(n^3)$  au total.

**Question 9** Ces deux algorithmes sont des exemples de programmation dynamique (bottom-up).

## II - Arbre couvrant maximal : algorithme de Prim

La méthode précédente est très très peu efficace pour calculer un goulot maximal entre deux sommets précis. On propose ici une nouvelle méthode, utilisant la notion d'arbre couvrant maximal.

**Question 10** Soit  $(s, t) \in S^2$  et  $T = (S, B)$  un arbre couvrant de poids **maximal** de  $G$ . Montrer qu'un chemin de  $s$  à  $t$  dans  $T$  est un goulot maximal de  $s$  à  $t$  dans  $G$ .

On pourrait adapter l'algorithme de Kruskal pour qu'il renvoie un arbre couvrant de poids maximal en parcourant les arêtes dans l'ordre inverse. Mais dans cette partie, on se propose d'étudier un autre algorithme de construction d'un arbre couvrant minimal, que l'on utilisera pour renvoyer un arbre couvrant maximal. Il s'agit de *l'algorithme de Prim* suivant :

---

### Algorithme 4 : Algorithme de Prim

---

**Entrées :** Un graphe pondéré non orienté connexe  $G = (S, A, \rho)$ .

**Sorties :** Un arbre couvrant minimal  $T$  de  $G$ .

```

1  $F \leftarrow \emptyset$ 
2  $X \leftarrow \{x_0\}$  (un sommet quelconque de  $G$ )
3 tant que  $X \neq S$  faire
4   Trouver  $xy \in A$  de poids minimal telle que  $x \in X$  et  $y \in S \setminus X$ .
5    $F \leftarrow F \cup \{xy\}$ 
6    $X \leftarrow X \cup \{y\}$ 
7 renvoyer  $T = (X, F)$ 
```

---

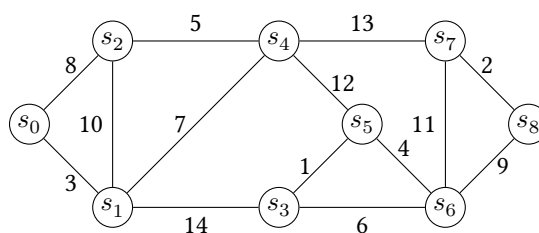


FIGURE 4 – Le graphe pondéré  $G_1$ .

**Question 11** Appliquer l'algorithme de Prim au graphe  $G_1$  de la figure 4 en partant du sommet  $s_0$ . On ne demande pas la description de l'exécution détaillée mais simplement une représentation graphique de l'arbre obtenu.

On admet ici que l'algorithme de Prim est totalement correct (même si ce serait un bon entraînement pour vous de le prouver!). Pour implémenter efficacement l'algorithme de Prim, on va utiliser une structure de file de priorité. Ces files de priorité contiendront des arêtes (des objets de type `edge_t`) et utiliseront les poids comme priorités. La structure a déjà été implémentée en utilisant un tas binaire, et la bibliothèque vous est fournie avec l'interface suivante :

```

1 // Création d'une file vide.
2 // L'argument capacity indique le nombre maximal
3 // d'arêtes que la file pourra contenir.
4 // La complexité de cette fonction est en  $O(\text{capacity})$ .
```

```

5 heap_t *heap_create(int capacity);
6
7 // Libération des ressources associées à une file
8 // Complexité O(1)
9 void heap_free(heap_t *heap);
10
11 // Détermine si la file est vide
12 // Complexité O(1)
13 bool heap_is_empty(heap_t *heap);
14
15 // Ajoute une arête à la file.
16 void heap_push(heap_t *heap, edge_t pair);
17
18 // Renvoie l'arête de poids minimal présente dans la file,
19 // et la supprime de la file.
20 // Erreur si la file est vide.
21 edge_t heap_extract_min(heap_t *heap);

```

**Question 12** Écrire une fonction `edge_t* prim(graph_t* g);` prenant en entrée un graphe  $G = (S, A, f)$ , supposé connexe, et renvoyant un arbre couvrant minimal de  $G$  sous la forme d'un tableau de  $|S| - 1$  arêtes. On demande une bonne complexité.

*Indication : on pourra ajouter dans une file de priorité les arêtes partant des sommets couverts au fur et à mesure qu'on les couvre. Avant de traiter une arête, on vérifiera qu'on n'a pas déjà couvert sa deuxième extrémité. La question vous laisse volontairement autonome sur l'implémentation de cette fonction. N'hésitez pas à utiliser des fonctions intermédiaires!*

**Question 13** Rappeler rapidement le principe de l'insertion d'un élément dans un tas binaire et de l'extraction du minimum, et en déduire les complexités des fonctions `heap_push` et `heap_extract_min` (en supposant bien sûr qu'elles sont correctement implémentées).

#### Question 14

Quelle est la complexité de l'algorithme de Prim que vous avez implémenté? Comparer avec celle de l'algorithme de Kruskal.

**Question 15** Comment trouver un arbre couvrant maximal à partir de l'algorithme de Prim?

**Question 16** Écrire une fonction `edge_t* arbre_max(graph_t* g)` prenant en entrée un graphe  $G = (S, A, f)$ , supposé connexe, et renvoyant un arbre couvrant **maximal** de  $G$  sous la forme d'un tableau de  $|S| - 1$  arêtes. On demande la même complexité totale que l'algorithme de Prim.

**Question 17** En déduire une fonction `double cmax(graph_t* g, int x, int y)` qui calcule la capacité maximale entre deux sommets dans un graphe avec une meilleure complexité que précédemment.

### Solution de l'Exercice 4

**Question 12** Notons  $\sigma = (s = s_0, s_1, \dots, s_k = t)$  un chemin de  $s$  à  $t$  dans  $T$ . Supposons par l'absurde que  $\sigma$  n'est pas un goulot maximal de  $s$  à  $t$  dans  $G$  et soit  $\sigma^*$  un goulot maximal de  $s$  à  $t$  dans  $G$ . Par hypothèse,  $c(\sigma) < c(\sigma^*)$ . Soit  $a$  l'arête de poids minimal de  $\sigma$ . Alors  $(S, B \setminus \{a\})$  contient deux composantes connexes. Il existe une arête  $a'$  de  $\sigma^*$  qui relie ces deux composantes connexes (car  $\sigma^*$  est un chemin qui relie ces deux composantes). De plus,  $f(a') > f(a)$  (car  $f(a') \geq c(\sigma^*) > c(\sigma) = f(a)$ ), donc  $a' \neq a$ . On en déduit, comme dans la première partie, que  $T' = (S, B \cup a' \setminus \{a\})$  est un arbre couvrant, de poids  $f(T') = f(T) + f(a') - f(a) > f(T)$ , ce qui contredit la maximalité de  $T$ .

**Question 13** TODO

**Question 14** Code.

**Question 15** Cf cours de MP2I sur les tas pour tous les détails.

**Insertion :** On ajoute l'élément à insérer en fin de tableau, dans la première case disponible, puis on fait une percolation vers le haut pour rétablir la structure de tas : tant que l'élément inséré est plus petit que son parent, on l'échange avec son parent.

**Extraction du minimum :** On retient la racine et on la remplace par le dernier élément du tableau, puis on fait

une percolation vers le bas pour rétablir l'ordre du tas : tant que l'élément qu'on a initialement mis à la racine est plus grand qu'un de ses enfants, on l'échange avec son plus petit enfant.

Ces deux opérations sont en temps proportionnel à la hauteur du tas, qui est un tas binaire donc un arbre binaire complet. On a donc  $h = \lfloor \log n \rfloor$  et la complexité de ces opérations est en  $O(\log n)$ .

**Question 16** La complexité de l'algorithme de Prim est en  $O(|A| \log |S|)$ . En effet, on ajoute une arête à  $X \subseteq S$  à chaque itération donc on effectue au plus  $|S|$  itérations de la boucle Tant que, et chaque itération est responsable de l'ajout d'un sommet  $y$  à  $X$ . On trouve l'arête de poids minimal à l'aide d'une extraction de la file de priorité en temps  $O(\log |A|) = O(\log |S|)$  au plus. Mais l'ajout d'un sommet  $y$  à  $X$  nous force à mettre à jour la file de priorité en traitant chaque arête incidente au sommet ajouté, en temps  $O(\deg(y) \cdot \log(|S|))$ . Au total, on obtient donc une complexité :

$$O\left(\sum_{y \in S} (1 + \deg(y)) \cdot \log |S|\right) = O((|A| + |S|) \log |S|)$$

Comme le graphe est connexe, on sait que  $|A| \geq |S| - 1$  et on peut simplifier en  $O(|A| \log |S|)$ .

**Question 17** Pour trouver un arbre couvrant maximal de  $G$ , il suffit d'inverser tous les poids des arêtes dans  $G$ , i.e. de trouver un arbre couvrant maximal de  $(S, A, -f)$ . On peut aussi adapter l'algorithme en utilisant une file-max pour trouver à chaque étape une arête de poids maximal.

### III - Implémentation en temps linéaire

On peut faire encore mieux ! Il est possible de trouver un goulot maximal entre deux sommets d'un graphe en temps linéaire en la taille du graphe.

Dans toute cette partie, on suppose que la fonction de pondération est injective. Pour  $G = (S, A, f)$  un graphe non orienté pondéré et  $X \subseteq S$ , on appelle opération de fusion de  $X$  dans  $G$  l'opération qui consiste à remplacer  $G$  par le graphe  $G' = (S', A', f')$  où :

- $S' = S \setminus X \cup \{x\}$ , où  $x$  est un nouveau sommet ;
- $A' = (P_2(S \setminus X) \cap A) \cup \{\{s, x\} \mid s \in S \setminus X, \exists t \in X, s, t \in A\}$  ;
- pour  $a \in A'$  :
  - si  $a \in A$ ,  $f'(a) = f(a)$  ;
  - sinon,  $a = \{s, x\}$  et  $f'(a) = \max\{f(s, t) \mid t \in X\}$ .

Autrement dit, on fusionne les sommets de  $X$  en un seul, on laisse les arêtes qui relient  $X$  à un autre sommet, et on garde le poids maximal lorsqu'il y a plusieurs choix.

On considère l'algorithme suivant :

---

#### Algorithme 5 : Algorithme BSP (Bottleneck Shortest Path)

---

**Entrées :** Un graphe pondéré non orienté connexe  $G = (S, A, f)$ , une source  $s \in S$  et une destination  $t \in S$ .

```

1 tant que  $|A| > 1$  faire
2    $M \leftarrow$  médiane de  $\{f(a) \mid a \in A\}$ .
3   Supprimer de  $A$  les arêtes  $a$  de poids  $f(a) < M$ .
4   si il n'existe pas de chemin de  $s$  à  $t$  alors
5     Poser  $X_1, \dots, X_k$  les composantes connexes de  $G$ .
6     Rajouter les arêtes supprimées avant le test.
7     Fusionner  $X_1, \dots, X_k$  dans  $G$ .
8 renvoyer  $f(a)$  où  $a$  est l'unique arête de  $G$ .
```

---

On admet qu'un passage dans la boucle Tant que peut se réaliser en temps  $O(|A|)$ , où  $A$  désigne ici l'ensemble des arêtes à l'entrée dans la boucle (qui est donc modifié d'un passage dans la boucle au suivant).

**Question 18** Montrer que l'ensemble de l'algorithme a une complexité linéaire en  $|A|$ , où  $A$  est l'ensemble initial des arêtes.

**Question 19** Montrer que l'algorithme renvoie la capacité d'un goulot maximal de  $s$  à  $t$  dans  $G$ .

#### Solution de l'Exercice 4

**Question 18** Remarquons que le nombre d'arêtes est environ divisé par deux à chaque itération :

- si la suppression des arêtes laisse un chemin de  $s$  à  $t$ , on a supprimé  $\lfloor \frac{|A|}{2} \rfloor$  arêtes (celles de poids inférieur

à la médiane);

- sinon, on n’a gardé que ces arêtes là, et supprimé les autres, soit supprimé  $\lceil \frac{|A|}{2} \rceil$  arêtes.

Le nombre de passages dans la boucle est donc de l’ordre de  $\log_2 |A|$  au maximum, et la complexité totale est en

$$O\left(\sum_{k=0}^{+\infty} \frac{|A|}{2^k}\right) = O(|A|).$$

**Question 19** Montrons qu’il existe un goulot maximal de  $s$  à  $t$  dont l’arête de poids minimal n’est jamais supprimée :

- si la suppression des arêtes laisse un chemin de  $s$  à  $t$ , alors les arêtes supprimées ne font pas partie d’un goulot maximal (car la capacité des chemins restants est strictement plus grande que le poids des arêtes supprimées);
- sinon, le goulot maximal passe nécessairement par l’une de ces arêtes, et l’une d’entre elles est de poids minimal dans le goulot maximal. Le fait de fusionner les composantes ne change pas la suite du problème.

# INFORMATIQUE

Durée : 4 heures

## Consignes :

- Veuillez à numéroté vos copies en bas à droite sous le format numéro\_de\_la\_page / nombre\_total\_de\_pages.
- Ne pas utiliser de correcteur blanc.
- Des points bonus/malus de présentation pourront être accordés.
- Veuillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- **L'usage du cours, de notes ou de tout appareil électronique est strictement interdit.**

*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.*

**Articulation des parties** La partie I est essentiellement indépendante des autres. Les parties II à VIII se suivent et sont destinées à être traitées dans l'ordre : même en admettant les résultats, il est conseillé de lire la partie  $n$  avant de passer à la partie  $n + 1$ .

**Programmation** L'ensemble du sujet est à traiter dans le langage OCaml. Toutes les fonctions des modules `List` et `Array` peuvent être librement utilisées, ainsi que les celles du module initialement ouvert de la bibliothèque standard (les fonctions qui s'écrivent sans préfixe, comme `incr`, `min`...). Les fonctions du module `List`, **en particulier** `List.iter`, permettent de simplifier l'écriture d'une grande partie du code demandé. Sauf mention explicite du sujet, l'utilisation de fonctions issues d'autres modules est interdite.

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 auxiliaires, mais devra préciser leurs rôles ainsi que les types et significations de leurs arguments (éventuellement grâce à un choix judicieux de noms d'identifiants). 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.

**Tout code qui n'est ni expliqué ni commenté se verra attribué la note de 0 et ne sera pas lu. Même pour un programme simple, vous devez indiquer (brièvement) ce que vous faites, et comment. Les fonctions les plus complexes doivent être d'abord détaillées en Français et agrémentées de commentaires.**

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`).

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  $O(f(n, m))$  où  $n$  et  $m$  sont les tailles des arguments de la fonction, et  $f$  une expression la plus simple possible. Les calculs de complexité seront justifiés succinctement.

*Ce sujet comporte 9 pages (celle-ci comprise).*

**Ne retournez pas la page avant d'y être invités.**

# ANALYSE SYNTAXIQUE LL(1)

**Notations** Si  $G = (\Sigma, V, P, S)$  est une grammaire hors-contexte et  $X \in V$ , on notera  $L_G(X)$  (voire  $L(X)$  quand il n'y a aucun risque d'ambiguïté) l'ensemble des mots de  $\Sigma^*$  engendrés par  $X$ . Formellement,  $L_G(X) = \{u \in \Sigma^* \mid X \Rightarrow^* u\}$ .

## I - Préliminaires

On considère la grammaire hors-contexte  $G_0 = (\Sigma, V_0, P_0, S)$  définie par :

- $\Sigma = \{1, +, \times, (, )\}$ ;
- $V_0 = \{S\}$ ;
- $P_0 = \{S \rightarrow S + S \mid S \times S \mid (S) \mid 1\}$ .

**Question 1** Montrer que le mot  $u = (1 + 1) \times 1$  appartient au langage  $L(G_0)$  en exhibant un arbre de dérivation pour  $u$ , ainsi qu'une dérivation gauche.

**Question 2** Montrer que la grammaire  $G_0$  est ambiguë.

**Question 3** En dehors de son caractère ambigu, quel défaut cette grammaire présente-t-elle si on souhaite écrire un analyseur syntaxique descendant ?

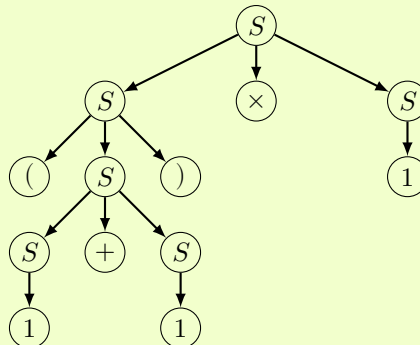
**Question 4** Montrer que pour  $u \in L(G)$ ,  $|u|_1 = |u|_+ + |u|_\times + 1$ . En déduire que  $1 \times (+1) \notin L(G_0)$ .

### Solution de l'Exercice 1

**Question 1** On a la dérivation gauche suivante :

$$S \Rightarrow S \times S \Rightarrow (S) \times S \Rightarrow (S + S) \times S \Rightarrow (1 + S) \times S \Rightarrow (1 + 1) \times S \Rightarrow (1 + 1) \times 1$$

Elle correspond à l'arbre de dérivation :



**Question 2** Le mot  $1 + 1 + 1$  est ambigu puisqu'il possède deux dérivation gauches (et donc deux arbres de dérivation) :

- $S \Rightarrow S + S \Rightarrow 1 + S \Rightarrow 1 + S + S \Rightarrow 1 + 1 + S \Rightarrow 1 + 1 + 1$ ;
- $S \Rightarrow S + S \Rightarrow S + S + S \Rightarrow 1 + S + S \Rightarrow 1 + 1 + S \Rightarrow 1 + 1 + 1$ .

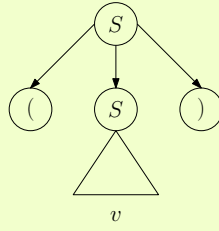
La grammaire est donc ambiguë.

**Question 3** Cette grammaire est réursive à gauche : une règle  $S \rightarrow S + S$  donne un appel récursif sur le même symbole  $S$  sans avoir rien consommé de l'entrée.

**Question 4** On procède par récurrence la propriété  $H_k$  suivante : « si  $u$  admet un arbre de dérivation de hauteur au plus  $k$ , alors  $|u|_1 = |u|_+ + |u|_\times + 1$  ».

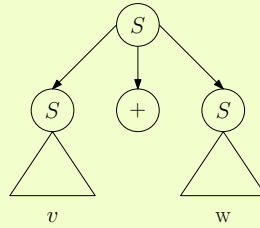
- $k$  ne peut être nul.
- Si  $k$  vaut 1, alors l'arbre est nécessairement  $(S; 1)$  et donc  $u = 1$ . On a bien  $|u|_1 = 1 = |u|_+ + |u|_\times + 1$ .
- Sinon, on distingue suivant la racine :
  - premier cas :





On peut appliquer l'hypothèse de récurrence à  $v$  et conclure directement (puisque  $u = (v)$ ).

– deuxième cas :



On peut appliquer l'hypothèse de récurrence à  $v$  et  $w$ , et l'on a  $u = v + w$ , donc :

$$\begin{aligned}
 |u|_1 &= |v|_1 + |w|_1 \\
 &= |v|_+ + |v|_\times + 1 + |w|_+ + |w|_\times + 1 \\
 &= \underbrace{|v|_+ + |w|_+ + 1}_{|u|_+} + \underbrace{|v|_\times + |w|_\times + 1}_{|u|_\times} \\
 &= |u|_+ + |u|_\times + 1
 \end{aligned}
 \quad H_k$$

– le troisième cas, où la règle à la racine est  $S \rightarrow S \times S$  se traite de la même manière.

On conclut par récurrence, et l'on constate ensuite que pour  $u = 1 \times (+1)$  on a  $|u|_1 = 2$  et  $|u|_+ + |u|_\times + 1 = 3$ , donc  $u \notin L(G)$ .

## Programmation

Pour une grammaire hors-contexte  $G = (\Sigma, V, P, S)$ , on représente  $\Sigma$  comme un alphabet de *tokens*, c'est-à-dire des objets d'un type `token` prédéfini (il n'est pas nécessaire de savoir comment est défini ce type pour traiter les questions).  $V$  sera représenté comme l'ensemble  $[0 \dots |V| - 1]$ , c'est-à-dire que chaque lettre de  $V$  correspondra à un entier. Par convention, on posera  $S = 0$ . Un élément de  $\Sigma \cup V$  sera appelé *symbole* et sera représenté par le type :

OCaml

```
1 type symbole = T of token | V of int
```

On représente un mot de  $(\Sigma \cup V)^*$  comme une liste de symboles, c'est-à-dire un objet de type `symbole list`. On définit :

OCaml

```
1 type mot = symbole list
```

Une grammaire  $G$  est représentée par un tableau `g` de longueur  $|V|$  tel que `g.(i)` soit la liste des mots  $\alpha$  tels que  $X_i \rightarrow \alpha$  soit une production de la grammaire. On définit donc le type `grammaire` :

OCaml

```
1 type grammaire = mot list array
```

**Exemple** On considère la grammaire  $G_1$  définie sur  $\Sigma = \{a, b, c\}$  par :

- $S \rightarrow SA \mid A \mid B$ ;
- $A \rightarrow AC \mid CC \mid a$ ;
- $B \rightarrow b$ ;
- $C \rightarrow c \mid \varepsilon$ .

En supposant que `a`, `b` et `c` sont des objets de type `token`, cette grammaire peut être représentée par :

**OCaml**

```

1 let g1 = [| [[V 0; V 1]; [V 1]; [V 2]];
2           [[V 1; V 3]; [V 3; V 3]; [T a]];
3           [[T b]];
4           [[T c]; []] |]

```

**Variables globales** Pour alléger, on supposera dans la suite du sujet qu'un certain nombre de variables sont définies globalement au lieu d'être passées en argument à chaque fonction. En particulier, on suppose dispose d'une variable  $g$  globale représentant une grammaire  $G = (\Sigma, V, P, S)$ .

## II - Détermination de $\Sigma$

**Question 5** Écrire une fonction `fusion : 'a list -> 'a list -> 'a list` prenant en entrée deux listes  $u$  et  $v$  supposées strictement croissantes (au sens de la relation d'ordre usuelle  $\leq$  de OCaml) et renvoyant la liste strictement croissante dont l'ensemble des éléments est l'union de l'ensemble des éléments de  $u$  et de celui de  $v$ .

**Terminal**

```

1 fusion [0; 3; 4] [2; 3; 5; 6];;
2 - : int list = [0; 2; 3; 4; 5; 6]

```

**Question 6** En déduire une fonction `tri_unique : 'a list -> 'a list` prenant en entrée une liste  $u$  et renvoyant la liste strictement croissante ayant le même ensemble d'éléments que  $u$ . On demande une complexité en  $O(|u| \log |u|)$ , en supposant que les comparaisons se font en temps constant (mais on ne demande pas ici de la justifier).

**Terminal**

```

1 tri_unique [4; 1; 2; 4; 0; 5; 2];;
2 - : int list = [0; 1; 2; 4; 5]

```

**Remarque :** Une telle fonction (`List.sort_uniq`) existe dans la bibliothèque standard, mais on demande ici de la reprogrammer.

**Question 7** Justifier la complexité de la fonction `tri_unique` de la question précédente.

**Question 8** Écrire une fonction `calcul_sigma` qui renvoie la liste de tokens correspondant à  $\Sigma$ , l'alphabet terminal de la grammaire  $G$ . On supposera que  $\Sigma$  est limité à l'ensemble des tokens apparaissant effectivement dans le membre droit d'une des productions de la grammaire. On renverra une liste sans doublon,

**OCaml**

```

1 val calcul_sigma : unit -> token list

```

On suppose dans la suite que l'on a défini une variable globale `sigma` par :

**OCaml**

```

1 let sigma = calcul_sigma ()

```

### Solution de l'Exercice 2

**Question 5** On adapte l'algorithme de fusion pour éviter de créer des doublons.

**OCaml : corrige.ml**

```

1 let rec fusion u v =
2   match u, v with
3   | [], w | w, [] -> w
4   | x :: xs, y :: ys ->
5     if x = y then x :: fusion xs ys
6     else if x < y then x :: fusion xs v
7     else y :: fusion u ys

```

**Question 6** Adaptation immédiate du tri fusion :

```

1 let rec separe = function
2   | [] -> ([], [])
3   | [x] -> ([x], [])
4   | x :: y :: xs ->
5     let u, v = separe xs in (x :: u, y :: v)
6
7 let rec tri_unique u =
8   match u with
9   | [] | [_] -> u
10  | _ ->
11    let a, b = separe u in
12    fusion (tri_unique a) (tri_unique b)

```

**Question 7** Pour cette question, je vous conseille plutôt de tracer l'arbre des appels récursifs pour additionner les complexités étage par étage (à partir de la formule  $T(n) = 2T(n/2) + O(n)$ ), comme d'habitude! Je laisse le corrigé écrit par mon collègue ci-dessous :

La fonction fusion a une complexité en  $O(|u| + |v|)$  et separe une complexité en  $O(|u|)$ . En notant  $T(n)$  la complexité de tri\_unique pour une liste de taille  $n$ , on a donc :

- $O(n)$  pour l'appel à separe;
- $2T(n/2)$  pour les appels récursifs;
- $O(n)$  pour l'appel à fusion.

Pour  $n$  une puissance de 2, on en déduit successivement :

$$\begin{aligned} \frac{T(2^i)}{2^i} &\leq \frac{T(2^{i-1})}{2^{i-1}} + A && A \text{ constante réelle} \\ \sum_{i=1}^k \left( \frac{T(2^i)}{2^i} - \frac{T(2^{i-1})}{2^{i-1}} \right) &\leq \sum_{i=1}^k A \\ T(2^k) &\leq Ak2^k + 2^k T(1) \\ T(2^k) &= O(k2^k) \end{aligned}$$

Pour un  $n$  quelconque, on a  $n \leq 2^{\lceil \log_2 n \rceil}$  et l'on obtient donc  $T(n) = O(n \log n)$ .

**Question 8** Une solution possible :

```

1 let calcul_sigma () =
2   let u = ref [] in
3   let traite_symbole = function
4     | V _ -> ()
5     | T a -> u := T a :: !u in
6   let traite_regle droite = List.iter traite_symbole droite in
7   let traite_variable regles = List.iter traite_regle regles in
8   Array.iter traite_variable g;
9   tri_unique !u

```

### III - Symboles accessibles

On dit qu'un symbole  $X \in V$  est *accessible* s'il peut apparaître dans une dérivation depuis  $S$ , c'est-à-dire s'il existe  $\alpha, \beta \in (\Sigma \cup V)^*$  tels que  $S \Rightarrow^* \alpha X \beta$ . On définit le graphe orienté  $Acc_G$  comme suit :

- l'ensemble des sommets est l'ensemble  $V$  des variables de la grammaire  $G$ ;
- il y a un arc de  $X$  vers  $Y$  si et seulement si la grammaire contient au moins une règle de la forme  $X \rightarrow \alpha Y \beta$  avec  $\alpha, \beta \in (\Sigma \cup V)^*$ .

**Question 9** Expliquer comment calculer l'ensemble des symboles accessibles de  $G$  à partir du graphe  $Acc_G$ .

**Question 10** Écrire une fonction `graphe_accessible` calculant le graphe  $Acc_G$ , sous forme d'un tableau de listes d'adjacence.

OCaml

```
1 val graphe_accessible : unit -> int list array
```

**Question 11** Écrire une fonction `calcul_accessibles` qui renvoie un tableau `acc` de booléens de longueur  $|V|$  tel que `acc.(i)` est vrai si et seulement si  $X_i$  est accessible.

OCaml

```
1 val calcul_accessibles : unit -> bool array
```

Dans toute la suite, on supposera que toutes les variables de la grammaire sont accessibles.

### Solution de l'Exercice 3

**Question 9** L'ensemble des symboles accessibles correspond exactement à l'ensemble des sommets accessibles depuis le symbole initial dans le graphe  $Acc_G$ .

**Question 10** La fonction `traite_symbole` prend un symbole en entrée et l'ajoute à la liste d'adjacence s'il s'agit d'une variable. On élimine les doublons une fois la liste d'adjacence créée.

OCaml : corrige.ml

```
1 let construit_graphe () =
2   let n = Array.length g in
3   let graphe = Array.make n [] in
4   for i = 0 to n - 1 do
5     let traite_symbole = function
6       | T _ -> ()
7       | V j ->
8         graphe.(i) <- j :: graphe.(i) in
9     let traite_regle = List.iter traite_symbole in
10    List.iter traite_regle g.(i);
11    graphe.(i) <- tri_unique graphe.(i)
12  done;
13  graphe
```

**Question 11** On effectue ensuite un simple parcours de graphe en profondeur :

OCaml : corrige.ml

```
1 let calcul_accessibles () =
2   let graphe = construit_graphe () in
3   let n = Array.length g in
4   let acc = Array.make n false in
5   let rec dfs i =
6     if not acc.(i) then begin
7       acc.(i) <- true;
8       List.iter dfs graphe.(i)
9     end in
10  dfs 0;
11  acc
```

## IV - Symboles nuls

Pour  $X \in V$ , on dit que  $X$  est *nul* si  $\epsilon \in L(X)$ . On définit  $NUL(X)$  comme le prédicat associé (autrement dit,  $NUL(X) = \text{vrai si } X \Rightarrow^* \epsilon$ ,  $NUL(X) = \text{faux sinon}$ ).

**Question 12** Déterminer, en justifiant brièvement,  $NUL(X)$  pour chaque variable  $X$  de la grammaire  $G_1$ .

On considère l'algorithme suivant :

```

Entrées : Une grammaire  $G = (\Sigma, V, P, S)$ 
1  $\mathcal{N} \leftarrow \emptyset$ 
2  $\text{fini} \leftarrow \text{faux}$ 
3 tant que  $\neg \text{fini}$  faire
4    $\text{fini} \leftarrow \text{vrai}$ 
5   pour chaque  $X \rightarrow X_1 \dots X_n \in P$  faire //  $X \rightarrow \epsilon$  donne  $n = 0$ 
6     si  $X \notin \mathcal{N}$  et  $X_i \in \mathcal{N}$  pour tout  $i \in [1 \dots n]$  alors
7        $\mathcal{N} \leftarrow \mathcal{N} \cup \{X\}$ 
8      $\text{fini} \leftarrow \text{faux}$ 
9 renvoyer  $\mathcal{N}$ 

```

**Algorithme 6** Calcul des symboles nuls

**Question 13** Montrer que l'algorithme 6 termine.

**Question 14** Montrer que l'ensemble  $\mathcal{N}$  renvoyé est exactement l'ensemble des symboles nuls de la grammaire.

**Question 15** Écrire une fonction `calcul_nul` qui prend en entrée une grammaire et renvoie un tableau de booléens nul de longueur  $|V|$  tel que `nul.(i)` soit égal à `true` si et seulement si  $\text{NUL}(X_i)$  est vrai.

On rappelle l'existence de la fonction `List.for_all` de type  $( 'a \rightarrow \text{bool} ) \rightarrow 'a \text{ list} \rightarrow \text{bool}$ , qui permet de tester si un prédicat est vérifié par tous les éléments d'une liste, et de la fonction `List.exists` de même signature.

OCaml

```

1 val calcul_nul : unit -> bool array

```

On suppose à présent avoir créé une variable globale `nul` en exécutant :

OCaml

```

1 let nul = calcul_nul ()

```

**Question 16** Déterminer la complexité temporelle de la fonction précédente dans le pire cas en fonction de  $|P|$ ,  $|V|$  et  $n_{\max}$ , où  $n_{\max}$  est la taille maximale d'un mot  $\alpha$  tel qu'il existe une règle  $X \rightarrow \alpha \in P$ .

On étend la définition de  $\text{NUL}$  à l'ensemble des mots  $\alpha \in (\Sigma \cup V)^*$  : un mot  $\alpha$  est dit *nul* si  $\alpha \Rightarrow^* \epsilon$ . On admet que pour  $\alpha = x_1 \dots x_n \in (\Sigma \cup V)^*$  :

$$\text{NUL}(\alpha) = \bigwedge_{i=1}^n (x_i \in V \wedge \text{NUL}(x_i))$$

**Question 17** Écrire une fonction `mot_nul` qui prend en entrée un mot de  $(\Sigma \cup V)^*$  et détermine si ce mot est nul.

OCaml

```

1 val mot_nul : mot -> bool

```

### Solution de l'Exercice 4

**Question 12** On a immédiatement  $L(B) = \{b\}$ , et donc  $\text{NUL}(B) = \text{faux}$ . Pour les autres variables :

- $C \Rightarrow \epsilon$ , donc  $\text{NUL}(C) = \text{vrai}$ ;
- $A \Rightarrow CC \Rightarrow C \Rightarrow \epsilon$ , donc  $\text{NUL}(A) = \text{vrai}$ ;
- $S \Rightarrow A \Rightarrow^* \epsilon$ , donc  $\text{NUL}(S) = \text{vrai}$ .

**Question 13** La seule question vient de la boucle **tant que**, pour laquelle  $|V \setminus \mathcal{N}|$  fournit un variant :

- c'est un entier positif ou nul;
- après chaque passage dans la boucle, soit on a ajouté au moins un élément à  $\mathcal{N}$  (et la quantité a donc strictement décro), soit *fini* est faux (et l'on sort donc de la boucle).

**Question 14** On définit la suite d'ensembles  $\mathcal{N}_i$  par :

- $\mathcal{N}_0 = \emptyset$
- $\mathcal{N}_{i+1} = \mathcal{N}_i \cup \{X \in V \mid \exists X \rightarrow X_1 \dots X_n \in P, X_1, \dots, X_n \in \mathcal{N}_i\}$

À la lecture du code, il est immédiat que :

- après  $i$  passages dans la boucle,  $\mathcal{N} = \mathcal{N}_i$ ;
- on renvoie le premier  $\mathcal{N}_i$  vérifiant  $\mathcal{N}_i = \mathcal{N}_{i+1}$ .

Or d'autre part :

- une récurrence immédiate montre que, si  $X \in \mathcal{N}_i$ , alors  $\text{NUL}(X) = \text{vrai}$  – on en déduit que  $\text{NUL}(X) = \text{vrai}$  pour tous les éléments de l'ensemble  $\mathcal{N}$  final;
- de même, on a immédiatement  $\mathcal{N}_{i+1} = \mathcal{N}_i \Rightarrow \forall j \geq i, \mathcal{N}_j = \mathcal{N}_i$ . Ainsi, si  $X \in \mathcal{N}_i$  pour un  $i$  quelconque, alors  $X$  est dans l'ensemble renvoyé.

Il reste à montrer que si  $X$  est nul, alors  $X \in \mathcal{N}_i$  pour un certain  $i$ . On montre par récurrence sur  $i$  que si  $X \Rightarrow^i \epsilon$ , alors  $X \in \mathcal{N}_i$ .

- Si  $i = 1$ , alors  $X \rightarrow \epsilon \in P$ , et l'on a bien  $X \in \mathcal{N}_1$  (avec  $n = 0$  dans la définition).
- Sinon, on a  $X \Rightarrow \alpha \Rightarrow^{i-1} \epsilon$ . On a nécessairement  $\alpha \in V^*$  (puisqu'il faut obtenir  $\epsilon$  à la fin), donc  $X \Rightarrow X_1 \dots X_n \Rightarrow^{i-1} \epsilon$ . D'après l'hypothèse de récurrence, chaque  $X_j$  est dans  $\mathcal{N}_{i-1}$ , donc  $X$  est dans  $\mathcal{N}_i$  par définition de  $\mathcal{N}_i$ .

On conclut que l'ensemble  $\mathcal{N} = \bigcup_{i \geq 0} \mathcal{N}_i$  renvoyé est bien l'ensemble des symboles nuls.

**Question 15** On suit l'algorithme proposé, avec une fonction `test : symbole -> bool` qui permet de tester si un symbole est annulable.

```

1 let calcul_nul () =
2   let m = Array.length g in
3   let nul = Array.make m false in
4   let fini = ref false in
5   let test = function
6     | T _ -> false
7     | V x -> nul.(x) in
8   while not !fini do
9     fini := true;
10    for x = 0 to m - 1 do
11      if not nul.(x) && List.exists (List.for_all test) g.(x) then
12        (nul.(x) <- true; fini := false)
13    done
14  done;
15  nul

```

**Question 16** La boucle principale est exécutée au plus  $|V|$  fois. À chaque tour de boucle, on examine toutes les règles, et chaque règle est traitée en temps  $O(n_{\max})$ . Au total, on a donc du  $O(|V| \cdot |P| \cdot n_{\max})$ .

**Question 17** Il suffit d'appliquer la formule :

```

1 let rec mot_nul = function
2   | [] -> true
3   | T a :: _ -> false
4   | V x :: q -> nul.(x) && mot_nul q

```

## V - Ensembles **PREMIER**( $\alpha$ )

Pour  $\alpha \in (\Sigma \cup V)^*$ , on définit **PREMIER**( $\alpha$ ) comme l'ensemble des symboles terminaux qui peuvent apparaître au début d'un mot obtenu depuis une dérivation de  $\alpha$ . Formellement :

$$\text{PREMIER}(\alpha) = \{a \in \Sigma \mid \exists \beta \in (\Sigma \cup V)^*, \alpha \Rightarrow^* a\beta\}.$$

**Question 18** Déterminer, sans justifier, les ensembles **PREMIER**( $X$ ) pour  $X \in V$  et **PREMIER**( $\alpha$ ) pour chaque  $\alpha$  apparaissant du côté droit d'une règle de production de la grammaire  $G_1$ .

**Question 19** Pour une grammaire quelconque, que vaut **PREMIER**( $\varepsilon$ ) ? Que vaut **PREMIER**( $a$ ) pour  $a \in \Sigma$  ? On ne demande pas de justification.

**Question 20** Pour  $x \in \Sigma \cup V$  et  $\alpha \in (\Sigma \cup V)^*$ ,  $\alpha \neq \varepsilon$ , exprimer **PREMIER**( $x\alpha$ ) en fonction de **PREMIER**( $x$ ), **PREMIER**( $\alpha$ ) et **NUL**( $x$ ).

**Question 21** En s'inspirant de l'algorithme 6, décrire en pseudo-code un algorithme permettant de calculer **PREMIER**( $X$ ) pour  $X \in V$ .

On suppose disposer d'une structure de données persistante de type `TokenSet.t` correspondant à des ensembles (sans doublon) de tokens. On manipule cette structure avec les primitives suivantes (il n'est pas nécessaire de toutes les utiliser) :

- `empty`, constante qui correspond à l'ensemble vide ;
- `mem` qui détermine si un token donné est présent dans l'ensemble ;
- `add` qui ajoute un token à un ensemble donné et renvoie le nouvel ensemble ;
- `union` qui calcule l'union de deux ensembles ;
- `inter` qui calcule l'intersection de deux ensembles ;
- `cardinal` qui renvoie le cardinal d'un ensemble ;
- `subset` qui teste si le premier ensemble est inclus dans le deuxième ;
- `to_list` qui renvoie une liste contenant les éléments de l'ensemble.

OCaml

```
1 val empty : TokenSet.t
2 val mem : token -> TokenSet.t -> bool
3 val add : token -> TokenSet.t -> TokenSet.t
4 val union : TokenSet.t -> TokenSet.t -> TokenSet.t
5 val inter : TokenSet.t -> TokenSet.t -> TokenSet.t
6 val cardinal : TokenSet.t -> int
7 val subset : TokenSet.t -> TokenSet.t -> bool
8 val to_list : TokenSet.t -> token list
```

**Question 22** Écrire une fonction `calcul_premier` qui renvoie un tableau `premier` de taille  $|V|$  tel que pour  $X \in V$ , `premier.(x)` contient l'ensemble correspondant à **PREMIER**( $X$ ).

OCaml

```
1 val calcul_premier : unit -> TokenSet.t array
```

Pour la suite, on supposera créée une variable globale `premier` définie par :

OCaml

```
1 let premier = calcul_premier ()
```

**Question 23** Écrire une fonction `premier_mot` qui prend en argument un mot  $u \in (\Sigma \cup V)^*$  et renvoie un ensemble de tokens correspondant à **PREMIER**( $u$ ).

OCaml

```
1 val premier_mot : mot -> TokenSet.t
```

## Solution de l'Exercice 5

**Question 18** On donne le résultat sous forme de tableau dans l'ordre permettant de faciliter les calculs :

$\alpha$	$B$	$C$	$A$	$S$	$SA$	$AC$	$CC$
$\text{PREMIER}(\alpha)$	$b$	$c$	$a, c$	$a, b, c$	$a, b, c$	$a, c$	$c$

**Question 19**  $\text{PREMIER}(\epsilon) = \emptyset$ , car  $\epsilon \Rightarrow^* u$  implique  $u = \epsilon$  (donc ne commence pas par une lettre de  $\Sigma$ ).  
 $\text{PREMIER}(a\alpha) = \{a\}$ , car  $a\alpha \Rightarrow^* u$  implique que  $u$  commence par  $a$ .

**Question 20** On distingue selon que  $x$  est annulable ou pas :

- si  $\text{NUL}(x)$ , alors  $\text{PREMIER}(x\alpha) = \text{PREMIER}(x) \cup \text{PREMIER}(\alpha)$ ;
- sinon,  $\text{PREMIER}(x\alpha) = \text{PREMIER}(x)$ .

**Question 21** L'idée est de mettre à jour des ensembles tant que possible et de s'arrêter quand il n'y a plus de modification. On passe d'une lettre à la suivante tant que les lettres sont annulables.

**Entrées :** Une grammaire  $G = (\Sigma, V, P, S)$

```

1 pour chaque  $x \in V$  faire
2    $\text{PREMIER}(X) \leftarrow \emptyset$ .
3 Fini  $\leftarrow \perp$ 
4 tant que  $\neg \text{Fini}$  faire
5    $\text{Fini} \leftarrow \top$ 
6   pour chaque  $X \rightarrow x_1 \dots x_n \in P$  faire
7      $i \leftarrow 1$ 
8     tant que  $i \leq n$  faire
9       si  $\text{PREMIER}(x_i) \not\subseteq \text{PREMIER}(X)$  alors
10         $\text{PREMIER}(X) \leftarrow \text{PREMIER}(X) \cup \text{PREMIER}(x_i)$ 
11         $\text{Fini} \leftarrow \perp$ 
12      si  $x_i \in V$  et  $\text{NUL}(x_i)$  alors
13         $i \leftarrow i + 1$ 
14      sinon
15         $i \leftarrow n + 1$ 

```

**Question 22** On traduit l'algorithme donné ci-dessus. La fonction `traiter_regle` prend en entrée le mot  $x_1 \dots x_n$  (membre droit d'une règle) et fait les opérations correspondant au corps de la boucle **pour** de l'algorithme. Pour savoir si l'ensemble a été modifié, on compare les valeurs initiale et finale de son cardinal.

```

1 let calcul_premier () =
2   let premier = Array.make m empty in
3   let fini = ref false in
4   while not !fini do
5     fini := true;
6     for x = 0 to m - 1 do
7       let c = cardinal premier.(x) in
8       let rec traiter_regle = function
9         | []      -> ()
10        | T a :: _ -> premier.(x) <- add a premier.(x)
11        | V y :: q -> premier.(x) <- union premier.(x) premier.(y);
12          if nul.(y) then traiter_regle q in
13       List.iter traiter_regle g.(x);
14       if cardinal premier.(x) <> c then fini := false
15     done
16   done;
17   premier

```

**Question 23** On utilise la question 20.



OCaml : corrige.ml

```

1 let rec premier_mot = function
2   | []      -> empty
3   | T a :: _ -> add a empty
4   | V x :: q ->
5     if nul.(x) then union premier.(x) (premier_mot q)
6     else premier.(x)

```

## VI - Ensembles SUIVANT( $X$ )

Pour  $X \in V$ , on définit  $\text{SUIVANT}(X)$  comme l'ensemble des symboles terminaux qui peuvent suivre  $X$  dans un mot obtenu par une dérivation depuis  $S$ . Formellement :

$$\text{SUIVANT}(X) = \{a \in \Sigma \mid \exists \alpha, \beta \in (\Sigma \cup V)^*, S \Rightarrow^* \alpha X a \beta\}$$

On suppose de plus l'existence d'un token particulier **EOF** (pour *End Of File*). S'il existe  $\alpha \in (\Sigma \cup V)^*$  tel que  $S \Rightarrow^* \alpha X$ , alors  $\text{SUIVANT}(X)$  contient EOF. En particulier,  $\text{SUIVANT}(S)$  contient toujours EOF.

**Question 24** Déterminer, sans justifier, les ensembles  $\text{SUIVANT}(X)$  pour chaque  $X \in V$  dans la grammaire  $G_1$ .

**Question 25** Soit  $X \in V$ . On considère l'ensemble des occurrences de  $X$  dans les membres de droite des règles, et on les met sous la forme  $X_i \rightarrow \alpha_i X \beta_i$  avec  $\alpha_i, \beta_i \in (\Sigma \cup V)^*$ . Notons qu'une règle comme  $Y \rightarrow aXbXc$  apparaîtra deux fois (une avec  $\alpha = a$  et  $\beta = bXc$ , l'autre avec  $\alpha = aXb$  et  $\beta = c$ ).

Donner, en la justifiant, une formule permettant de calculer  $\text{SUIVANT}(X)$  en fonction des  $X_i, \alpha_i$  et  $\beta_i$ .

**Question 26** Écrire une fonction `calcul_suivant` qui renvoie un tableau suivant de longueur  $|V|$  tel que `suivant.(i)` soit l'ensemble  $\text{SUIVANT}(X_i)$ .

OCaml

```
1 val calcul_suivant : unit -> TokenSet.t array
```

On suppose dans la suite avoir créé une variable globale `suivant` par :

OCaml

```
1 let suivant = calcul_suivant ()
```

### Solution de l'Exercice 6

**Question 24** On a  $\text{SUIVANT}(X) = \{\text{EOF}, a, c\}$  pour tout  $X \in V$ .

**Question 25** On a :  $\text{SUIVANT}(X) = \bigcup_{i=1}^k \text{PREMIER}(\beta_i) \cup \bigcup_{\substack{i=1 \\ \text{NUL}(\beta_i)=\top}}^k \text{SUIVANT}(X_i)$ . Montrons la double inclusion.

- Soit  $a \in \text{SUIVANT}(X)$ . Alors il existe  $\alpha, \beta \in (\Sigma \cup V)^*$  tels que  $S \Rightarrow^* \alpha X a \beta$ . Quitte à réordonner les dérivations immédiates, il existe donc  $i \in [1 \dots k]$ ,  $\gamma, \mu \in (\Sigma \cup V)^*$  tels que  $S \Rightarrow^* \gamma X_i \mu \Rightarrow \gamma \alpha_i X \beta_i \mu$ , avec  $\gamma \alpha_i \Rightarrow^* \alpha$  et  $\beta_i \mu \Rightarrow^* a \beta$ . On distingue alors :
  - si dans la dérivation,  $\beta_i \Rightarrow^* \varepsilon$ , alors  $\mu \Rightarrow^* a \beta$ , et on a bien  $\text{NUL}(\beta_i)$  et  $S \Rightarrow^* \gamma X_i a \beta$ , donc  $a \in \text{SUIVANT}(X_i)$ ;
  - sinon,  $\beta_i \Rightarrow^* a \beta'_i$  et  $a \in \text{PREMIER}(\beta_i)$ .
- Soit  $a \in \text{PREMIER}(\beta_i)$ . Alors il existe  $\beta'_i \in (\Sigma \cup V)^*$  tel que  $\beta_i \Rightarrow^* a \beta'_i$ . De plus, on a supposé que  $X_i$  peut apparaître dans une dérivation depuis  $S$ . Il existe donc  $\alpha, \beta \in (\Sigma \cup V)^*$  tel que  $S \Rightarrow^* \alpha X_i \beta$ . On en déduit que :

$$S \Rightarrow^* \alpha X_i \beta \Rightarrow \alpha \alpha_i X \beta_i \beta \Rightarrow^* \alpha \alpha_i X a \beta'_i \beta$$

ce qui montre que  $a \in \text{SUIVANT}(X)$ .

- Soit  $a \in \text{SUIVANT}(X_i)$  tel que  $\text{NUL}(\beta_i)$ . Alors il existe  $\alpha, \beta \in (\Sigma \cup V)^*$  tels que  $S \Rightarrow^* \alpha X_i a \beta$ . On a alors :

$$S \Rightarrow^* \alpha X_i a \beta \Rightarrow \alpha \alpha_i X \beta_i a \beta \Rightarrow \alpha \alpha_i X a \beta$$

car  $\beta_i \Rightarrow^* \varepsilon$ . On en déduit que  $a \in \text{SUIVANT}(X)$ .

**Question 26** On procède à nouveau en mettant à jour les ensembles jusqu'à ce qu'il n'y ait plus de modifications. La fonction `union_changement` renvoie l'union de ses deux arguments  $E_1$  et  $E_2$ , et met `fini` à `false` si  $E_1 \cup E_2 \neq E_1$ .

OCaml : corrige.ml

```
1 let calcul_suivant () =
2   let suivant = Array.make m empty in
3   let fini = ref false in
4   suivant.(0) <- add EOF empty;
5   let union_changement e1 e2 =
6     if not (subset e2 e1) then fini := false;
7     union e1 e2 in
8   while not !fini do
9     fini := true;
10    for x = 0 to m - 1 do
11      let rec traiter_regle = function
12        | []      -> ()
13        | T _ :: q -> traiter_regle q
14        | V y :: q ->
15          suivant.(y) <- union_changement suivant.(y) (premier_mot q);
16          if nul_mot q then
17            suivant.(y) <- union_changement suivant.(y) suivant.(x);
18            traiter_regle q in
19      List.iter traiter_regle g.(x)
20    done
21  done;
22  suivant
```

## VII - Grammaires LL(1)

On dit qu'une grammaire  $G = (\Sigma, V, P, S)$  est LL(1) (pour *Left to right, Leftmost derivation, 1 token*) si et seulement si pour toute paire de règles  $X \rightarrow \alpha \mid \beta$  :

- $\text{PREMIER}(\alpha) \cap \text{PREMIER}(\beta) = \emptyset$ ;
- si  $\text{NUL}(\beta)$ , alors  $\neg \text{NUL}(\alpha)$  et  $\text{PREMIER}(\alpha) \cap \text{SUIVANT}(X) = \emptyset$ .

**Question 27** Justifier que la grammaire  $G_1$  n'est pas LL(1).

On considère la grammaire  $G_2 = (\Sigma, V_2, P_2, S)$  définie par :

- $\Sigma = \{1, +, \times, (, )\}$ ;
- $V_2 = \{S, A, B, C, D\}$ ;
- $P_2$  contient les règles de production :
  - $S \rightarrow BA$ ;
  - $A \rightarrow +BA \mid \varepsilon$ ;
  - $B \rightarrow DC$ ;
  - $C \rightarrow \times DC \mid \varepsilon$ ;
  - $D \rightarrow 1 \mid (S)$ .

**Question 28** Déterminer les valeurs de  $\text{NUL}(X)$ ,  $\text{PREMIER}(X)$  et  $\text{SUIVANT}(X)$  pour chaque valeur de  $X \in V_2$  puis montrer que la grammaire  $G_2$  est LL(1).

**Question 29** Écrire une fonction `est_LL1` qui teste si la grammaire  $g$  est LL(1).

OCaml

```
1 val est_LL1 : unit -> bool
```

### Solution de l'Exercice 7

**Question 27** On considère les règles  $S \rightarrow SA \mid A$ . On remarque que  $\text{PREMIER}(SA) \cap \text{PREMIER}(A) = \{a, c\} \neq \emptyset$ . On en déduit que  $G_1$  n'est pas LL(1).

**Question 28** On calcule les ensembles NUL, PREMIER et SUIVANT pour chaque variable :

$X$	$S$	$A$	$B$	$C$	$D$
$\text{NUL}(X)$	$\perp$	$\top$	$\perp$	$\top$	$\perp$
$\text{PREMIER}(X)$	1, (	+	1, (	$\times$	1, (
$\text{SUIVANT}(X)$	), EOF	), EOF	+, ), EOF	+, ), EOF	+, $\times$ ), EOF

On remarque que pour les trois paires de règles  $X \rightarrow \alpha \mid \beta$ , on a bien  $\text{PREMIER}(\alpha) \cap \text{PREMIER}(\beta) = \emptyset$  et  $\text{NUL}(\alpha)$  et  $\text{NUL}(\beta)$  ne sont jamais vrais simultanément. De plus,  $\text{PREMIER}(+BA) \cap \text{SUIVANT}(A) = \emptyset$  et  $\text{PREMIER}(\times DC) \cap \text{SUIVANT}(C) = \emptyset$ . Comme  $\text{PREMIER}(\varepsilon) = \emptyset$ , on en déduit que  $G_2$  est bien LL(1).

**Question 29** On commence par calculer les trois tableaux. L'appel `comparer_regles x alpha beta` fait la vérification pour une paire de règles  $X \rightarrow \alpha \mid \beta$  indiquée par l'énoncé. On pense à vérifier également en inversant le rôle de  $\alpha$  et  $\beta$ . La fonction `verifier` permet de vérifier pour l'ensemble des paires possibles (on compare un élément à chacun des suivants dans la liste).

```

1 let est_LL1 () =
2   let b = ref true in
3   let comparer_regles x alpha beta =
4     let nalpha = nul_mot alpha in
5     let nbeta = nul_mot beta in
6     let palpha = premier_mot alpha in
7     let pbeta = premier_mot beta in
8     let b1 = (inter palpha pbeta = empty) in
9     let b2 = not nbeta || (inter palpha suivant.(x) = empty) in
10    let b3 = not nalpha || (inter pbeta suivant.(x) = empty) in
11    b := !b && b1 && b2 && b3 in
12  for x = 0 to m - 1 do
13    let rec verifier = function
14      | [] -> ()
15      | alpha :: q -> List.iter (comparer_regles x alpha) q;
16      verifier q in
17    verifier g.(x)
18  done;
19  !b

```

## VIII - Table d'analyse syntaxique

Une *table d'analyse syntaxique* d'une grammaire  $G = (\Sigma, V, P, S)$  est un outil permettant de faciliter les calculs dans une analyse syntaxique descendante. L'idée est la suivante :

- chaque ligne de la table correspond à une variable  $X \in V$ ;
- chaque colonne de la table correspond à une lettre (ou terminal, ou token)  $a \in \Sigma$ ;
- la case à la ligne  $X$  et à la colonne  $a$  contient les règles de production qui peuvent être appliquée lorsque le prochain token à lire est  $a$  et que l'on essaie une dérivation gauche commençant par  $X$ .

Formellement, en notant  $TAS$  cette table, pour  $X \in V$  et  $a \in \Sigma$  :

$$X \rightarrow \alpha \in TAS(X, a) \iff a \in \text{PREMIER}(\alpha) \vee (\text{NUL}(\alpha) \wedge a \in \text{SUIVANT}(X))$$

Par exemple, voici la table d'analyse syntaxique de la grammaire  $G_1$  :

	$a$	$b$	$c$	EOF
$S$	$S \rightarrow SA \mid A$	$S \rightarrow B$	$S \rightarrow SA \mid A$	$S \rightarrow SA \mid A$
$A$	$A \rightarrow AC \mid a$		$A \rightarrow CC$	$A \rightarrow AC \mid CC$
$B$		$B \rightarrow b$		
$C$			$C \rightarrow c$	$C \rightarrow \varepsilon$

**Question 30** Construire la table d'analyse syntaxique de la grammaire  $G_2$ .

**Question 31** Montrer que si  $G$  est une grammaire LL(1), alors chaque case de sa table d'analyse syntaxique contient au plus une règle de production.

**Question 32** En déduire qu'une grammaire LL(1) n'est pas ambiguë.

On représente une table d'analyse syntaxique d'une grammaire LL(1) par une table de hachage dont les clés sont des couples (variable, token). La valeur associée à une clé  $(X, a)$  correspond à la liste de mots  $[\alpha_1; \alpha_2; \dots; \alpha_k]$  telle que les  $X \rightarrow \alpha_i$  sont les règles de productions apparaissant dans la case  $(X, a)$  de la table d'analyse syntaxique. On rappelle les fonctions usuelles pour manipuler les tables de hachage :

- `('a, 'b) Hashttbl.t` est le type des tables de hachage ayant des clés de type 'a et des valeurs de type 'b;
- `Hashttbl.create` permet de créer une nouvelle table vide (l'argument entier précise la taille initiale du tableau sous-jacent, mais n'a pas vraiment d'importance puisqu'il est redimensionnable);
- `Hashttbl.add` permet d'ajouter une association à une table;
- `Hashttbl.mem` permet de tester la présence d'une clé;
- `Hashttbl.find` et `Hashttbl.find_opt` permettent de récupérer la valeur associée à une clé. La première renvoie directement la valeur et lève une exception si la clé n'est pas présente, la deuxième renvoie `Some v` si la valeur associée est `v`, `None` si la clé est absente.

OCaml

```
1 val Hashttbl.create : int -> ('a, 'b) Hashttbl.t
2 val Hashttbl.add : ('a, 'b) Hashttbl.t -> 'a -> 'b -> unit
3 val Hashttbl.mem : ('a, 'b) Hashttbl.t -> 'a -> bool
4 val Hashttbl.find : ('a, 'b) Hashttbl.t -> 'a -> 'b
5 val Hashttbl.find_opt : ('a, 'b) Hashttbl.t -> 'a -> 'b
```

**Question 33** Écrire une fonction `table_analyse` qui renvoie la table d'analyse syntaxique de  $g$ .

Remarque : on pensera à rajouter le token EOF à l'alphabet.

OCaml

```
1 val table_analyse : unit -> (int * token, symbole list list) Hashttbl.t
```

Pour la suite, on supposera avoir défini une variable globale `tas` par :

OCaml

```
1 let tas = table_analyse ()
```

On représente un arbre de dérivation par le type suivant :

OCaml

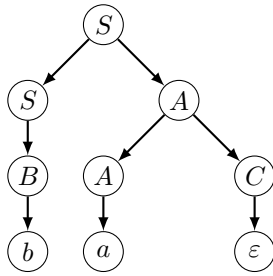
```
1 type arbre_syntaxe =
2 | Epsilon
3 | F of token
4 | N of int * arbre_syntaxe list
```

Dans un nœud interne `N (i, enfants)`, l'entier `i` indique le numéro de la variable correspondante. On a représenté ci-dessous un exemple d'arbre de dérivation et de l'objet OCaml associé pour le mot  $u = ba$  dans la grammaire  $G_1$ , en supposant que  $S, A, B$  et  $C$  sont respectivement numérotés 0, 1, 2 et 3 :

**Question 34** Écrire une fonction `analyse_syntaxique` qui prend en argument un mot  $u \in \Sigma^*$  et renvoie un arbre de dérivation de  $u$  dans  $G$ . La fonction lèvera une exception d'erreur de syntaxe, qui sera définie avant la fonction, si  $u \notin L(G)$ . On supposera que le mot  $u$  termine par le token spécial EOF (et que ce token n'apparaît nulle part ailleurs).

OCaml

```
1 val analyse_syntaxique : mot -> arbre_syntaxe
```



OCaml

```

1 N(0, [N(0, [N(2, [F b])])]);
2       N(1, [N(1, [F a])]);
3       N(3, [Epsilon])])])
  
```

Il existe de très nombreux outils permettant de créer automatiquement un parser à partir de la description d'une grammaire (en particulier `bison`, `yacc` ...). Ces outils sont en général limités à des classes spécifiques de grammaire, pour lesquelles l'analyse syntaxique peut être réalisée en temps linéaire (en la taille de la chaîne à analyser). C'est le cas des grammaires LL(1) (ou plus généralement LL(k)), qui sont utilisées par beaucoup de ces outils. La principale autre possibilité est d'utiliser une grammaire LR ou LALR, pour laquelle on génère automatiquement un analyseur syntaxique ascendant.

### Solution de l'Exercice 8

**Question 30** On obtient :

	1	+	×	(	)	EOF
<i>S</i>	$S \rightarrow BA$			$S \rightarrow BA$		
<i>A</i>		$A \rightarrow +BA$			$A \rightarrow \epsilon$	$A \rightarrow \epsilon$
<i>B</i>	$B \rightarrow DC$			$B \rightarrow DC$		
<i>C</i>		$C \rightarrow \epsilon$	$C \rightarrow \times DC$		$C \rightarrow \epsilon$	$C \rightarrow \epsilon$
<i>D</i>	$D \rightarrow 1$			$D \rightarrow (S)$		

**Question 31** On suppose qu'une des cases  $TAS(X, a)$  de la table d'analyse syntaxique contient deux règles  $X \rightarrow \alpha \mid \beta$ . On distingue les cas :

- si  $a \in \text{PREMIER}(\alpha)$  et  $a \in \text{PREMIER}(\beta)$ , alors  $\text{PREMIER}(\alpha) \cap \text{PREMIER}(\beta) \neq \emptyset$ , donc  $G$  n'est pas LL(1);
- si  $a \in \text{PREMIER}(\alpha)$ ,  $\text{NUL}(\beta)$  et  $a \in \text{SUIVANT}(X)$ , alors  $\text{PREMIER}(\alpha) \cap \text{SUIVANT}(X) \neq \emptyset$ , donc  $G$  n'est pas LL(1);
- le cas symétrique au précédent se traite de la même manière;
- sinon,  $\text{NUL}(\alpha)$  et  $\text{NUL}(\beta)$  et  $a \in \text{SUIVANT}(X)$ , et  $G$  n'est pas LL(1).

**Question 32** Supposons que  $G$  est ambiguë et soient  $u = a_1 \dots a_n$  un mot ambigu pour  $G$ . Il existe donc deux dérivations gauches de  $u$ . Notons  $X$  la première variable de gauche dans les deux dérivations à laquelle on applique une règle différente. Il existe donc  $i \in [0 \dots n]$ ,  $\alpha, \beta, \gamma \in (\Sigma \cup V)^*$  tels que  $S \Rightarrow_g^* a_1 \dots a_i X \gamma$  et  $X \gamma \Rightarrow_g \alpha \gamma \Rightarrow_g^* a_{i+1} \dots a_n$  et  $X \gamma \Rightarrow_g \beta \gamma \Rightarrow_g^* a_{i+1} \dots a_n$ . Distinguons les cas :

- si  $i = n$ , alors  $\text{NUL}(\alpha)$  et  $\text{NUL}(\beta)$ , donc  $G$  n'est pas LL(1);
- sinon, montrons que  $TAS(X, a_{i+1})$  contient  $X \rightarrow \alpha$  et  $X \rightarrow \beta$ . En effet, soit  $\alpha \Rightarrow_g^* a_{i+1} \alpha'$  auquel cas  $a_{i+1} \in \text{PREMIER}(\alpha)$ , ou  $\text{NUL}(\alpha)$ , auquel cas  $\gamma \Rightarrow_g^* a_{i+1} \dots a_n$  et donc  $S \Rightarrow_g^* a_1 \dots a_i X a_{i+1} \dots a_n$  donc  $a \in \text{SUIVANT}(X)$ . On raisonne de même pour  $\beta$ . On conclut par la question précédente que  $G$  n'est pas LL(1).

En conclusion, si  $G$  est LL(1), alors  $G$  n'est pas ambiguë.

**Question 33** On suit la formule de construction de la table. On fait le calcul de  $\Sigma$  en rajoutant le token spécial EOF.

```

1 let table_analyse () =
2   let sigma = Array.of_list (EOF :: calcul_sigma ()) in
3   let p = Array.length sigma in
4   let tas = Hashtbl.create 1 in
5   for x = 0 to m - 1 do
6     for a = 0 to p - 1 do
7       let ajout_regle alpha =
8         if mem sigma.(a) (premier_mot alpha)
9         || (nul_mot alpha && mem sigma.(a) suivant.(x)) then
10          Hashtbl.add tas (x, sigma.(a)) alpha in
11       List.iter ajout_regle g.(x)
12     done
13   done;
14   tas

```

**Question 34** On commence par définir l'exception avant d'écrire la fonction. Pour faire la lecture de  $u$  au fur et à mesure de la dérivation, on crée une référence contenant le mot  $u$  (la référence porte le même nom que l'argument, mais ce n'est pas gênant pour la suite). On écrit alors une fonction `derive : symbole -> arbre_syntaxique` qui construit l'arbre dans l'ordre préfixe selon le principe suivant :

- si  $u = av$  et le symbole est  $a$ , alors on crée une feuille  $a$  et on modifie  $u$  en  $v$ ;
- si  $u = av$  et le symbole est  $X$ , on considère la règle  $X \rightarrow \alpha$  dans  $TAS(X, a)$ , et on distingue :
  - si  $\alpha = \epsilon$ , on crée une feuille  $\epsilon$ ;
  - sinon, on crée récursivement la liste des arbres des symboles de  $\alpha$ , et on crée un nœud  $X$  ayant pour fils les éléments de cette liste.

Dès lors, il suffit de calculer la dérivation de  $S$ , c'est-à-dire  $V \ 0$ . On vérifie, avant de renvoyer l'arbre, que le mot  $u$  a bien été lu en entier.

```

1 exception Erreur_syntaxe
2
3 let analyse_syntaxique u =
4   let u = ref u in
5   let rec derive symb = match !u, symb with
6     | a :: v, T b when a = b -> u := v; F a
7     | a :: v, V x when Hashtbl.mem tas (x, a) ->
8       let gamma = Hashtbl.find tas (x, a) in
9       if gamma = [] then N(x, [Epsilon])
10      else N(x, List.map derive gamma)
11     | _ -> raise Erreur_syntaxe in
12   let arbre = derive (V 0) in
13   if !u <> [EOF] then raise Erreur_syntaxe else arbre

```

# Observations sur le DS Analyse syntaxique LL(1)

## I - Détermination de $\Sigma$

(A) Erreur de code très (**trop !!**) fréquente dans vos copies : il ne **sert à rien** d'écrire `h::l ; suite` . Je vous rappelle que :

- **Les listes OCaml sont immuables**, on ne peut pas modifier une liste ! Vous vouliez plutôt utiliser une **référence de liste**, ici. Si `l` est une référence de liste, vous pouvez écrire `l := h :: (!l) ; suite` . Là, on modifie vraiment la mémoire.
- L'opérateur `;` en OCaml permet d'évaluer ce qui précède, **OUBLIER LE RESULTAT** et évaluer ce qui suit. Par conséquent, ce qui vient avant un `;` devrait toujours avoir le type `unit` et agir par effet de bord. Or `h::l` est une simple liste. Il ne sert à rien d'écrire `h::l ; suite` de la même manière qu'il ne servirait à rien d'écrire `3 ; suite` .

(B) Q6 : On vous demandait ici de coder un **tri fusion**, bien sûr ! C'était (subtilement) indiqué par le fait qu'on vous fasse coder l'étape de fusion juste avant...

Pour le tri fusion, il faut **séparer la liste de taille  $n$  en deux listes de taille  $n/2$** , trier récursivement les deux sous-listes, puis les fusionner. Il vous restait à **coder l'étape de séparation** ! Prenez un minimum d'initiatives sur les fonctions auxiliaires ! Tout n'a pas à vous être explicitement demandé...

Vous avez été nombreux à isoler uniquement le premier élément, trier séparément le reste (liste de taille  $n - 1$ ) et insérer l'élément isolé dans la liste avec une fusion, par exemple en écrivant :

```
h::t -> fusion [h] (tri_unique t) .
```

Ceci est un **tri par insertion**, qui se fait en  $\Theta(n^2)$ . Vous devez connaître vos tris classiques ! On vous demandait un tri en temps  $O(n \log n)$  ici !!

## II - Préliminaires

(C) Q4 : Vous avez été nombreux à vouloir faire une induction pour montrer qu'un mot  $u$  généré par la grammaire  $G$  vérifie  $|u|_1 = |u|_+ + |u|_\times + 1$ . Ceci n'a pas de sens :

- Si vous n'avez pas précisé **sur quoi** vous faites une induction, vous montrez que vous ne comprenez pas ce que vous faites et que vous écrivez un raisonnement "au hasard".

**Il faut toujours indiquer sur quelle variable ou objet on fait une induction ou une récurrence.**

- Si vous avez voulu faire une induction sur  $u \in L(G)$  ou sur  $G$ , alors vous avez mal compris les objets que vous manipulez. En effet, une grammaire n'est PAS définie par induction, et l'ensemble des mots reconnus par une grammaire  $L(G)$  **n'est PAS non plus défini par induction**. (Allez revoir les définitions du cours !)

Ici, ce qu'on sait d'un mot  $u \in L(G)$  est qu'il existe une dérivation  $S \Rightarrow^* u$  (c'est comme ça que c'est défini !). Il faut donc plutôt travailler par **récurrence sur la longueur d'une dérivation** de  $u$  ! On peut aussi s'en sortir avec une récurrence sur  $|u|$  ou sur la hauteur de l'arbre de dérivation associé. Dans tous les cas, on fera une disjonction de cas sur la première règle appliquée dans la dérivation.

## III - Symboles nuls

(D) Q13 : Quand on vous demande de **montrer** qu'un algorithme termine, il ne s'agit bien sûr **pas d'agiter les mains ou de paraphraser l'algorithme** (on sait que vous savez lire), mais la plupart du temps d'**exhiber un variant**,

c'est à dire un entier strictement décroissant minoré ou un entier strictement croissant majoré!

Dans de très rares cas, on pourra avoir besoin de trouver une suite strictement décroissante selon un ordre bien fondé, de manière plus générale. Ce n'était pas le cas ici.

(E)

(F)