DEVOIR SURVEILLÉ VIII

Informatique

Durée : 4 heures

Consignes :

- Veillez à numéroter 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.

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 fichier (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 atterit 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]`.

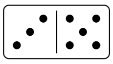
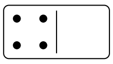
II - Partie II : CCINP sujet zéro (2022) - Dominos

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 fichier (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 $(i,j) \in [\![0;N]\!]^2$.

1) Structure de données

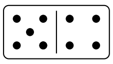
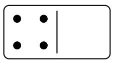
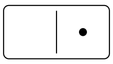
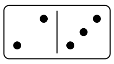
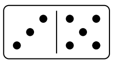
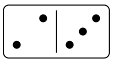
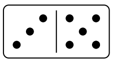
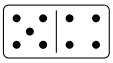
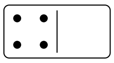
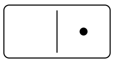
Dans cette section, on construit les structures de données utiles pour le premier problème. On définit le type structuré Domino par :

C
```c
struct domino_s {
    int x ;
    int y ;
};
typedef struct domino_s Domino ;
```

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

Définition 1

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 (, , , , ), la séquence      est une chaîne de 5 dominos.

Définition 2

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

Définition 3

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.

C
```c
struct element_s {
    Domino d;
    struct element_s* suivant;
};
typedef struct element_s element;
```

On définit le type `chaine` 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.

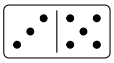
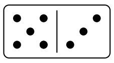
Question 2 Écrire une fonction de prototype `element* retireElement(element* l, Domino d)` qui retire le domino `d` de la chaîne ou du sac `l` (s'il est présent). Cette fonction renvoie la chaîne obtenue.

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.

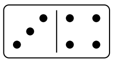
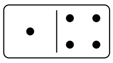
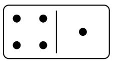
2) Existence d'une chaîne utilisant tous les dominos d'un sac

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 $D_i =$  et $D_j =$ , il n'est pas possible de placer directement D_j à droite de D_i, mais si on le retourne on obtient $D_j =$  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 débuguer. Ne passez pas plus de 30min sur cette question si vous n'aboutissez pas. Eventuellement, revenez-y à la fin...)**
Ecrire 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 !**

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 fichier (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** σ une suite non vide, finie ou infinie, de sommets $(s_i)_{0 \leqslant i < m}$, avec $m \in \mathbb{N}^* \cup \{\infty\}$, telle que $s_{i+1} \in V(s_i)$ si $0 \leqslant i$ et $i+1 < m$. Si $m \neq \infty$, on dit que σ est **fini de longueur** $m-1$. Sinon, il est dit **infini**. Il est à noter qu'un chemin σ 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 σ**, noté $|\sigma|_s$, correspond aux nombres de fois qu'un sommet de T apparaît dans σ. Formellement, $|\sigma|_s = \text{Card}\,\{i \in [\![0, m[\![\mid s_i = s\}$, ce cardinal pouvant être fini ou infini. On note de même, pour $T \subseteq S$, $|\sigma|_T = \text{Card}\,\{i \in [\![0, m[\![\mid s_i \in T\}$.

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 $\sigma = (s_i)_{0 \leqslant i < m}$ tel que $s_0 = s$. On dit que la partie σ est **gagnée** par le joueur 1 si $\sigma \in W$. Sinon, σ est dite gagnée par le joueur 2 .

Pour $j \in \{1, 2\}$, une **stratégie pour le joueur** j est une application $f : S_j \to S$ telle que pour tout $s \in S_j$, $f(s) \in V(s)$. Une partie $\sigma = (s_i)_{0 \leqslant i < m}$ 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 stratégie 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).

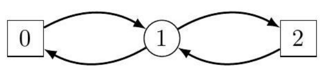


Figure 1 – Une arène (G, S_1, S_2).

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 $W = \left\{\sigma \in \mathcal{C}(G) \middle| |\sigma|_0 = \infty \text{ et } |\sigma|_2 = \infty\right\}$.

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

2) Jeux d'accessibilité

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

```ocaml
type graphe = int list array
```

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

- $S = [\![0, n-1]\!]$, où $n =$ `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 $s \in S$.

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.(s)` $= f_1(s)$;
- sinon, `f.(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.

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 $W = \left\{\sigma \in \mathcal{C}(G) \mid |\sigma|_T = \infty\right\}$, 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 Déduire 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 $\text{Attr}(E(T))\backslash E(T)$.

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

Informatique

Durée : 4 heures

Consignes :

- Veillez à numéroter 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, 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.

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.

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

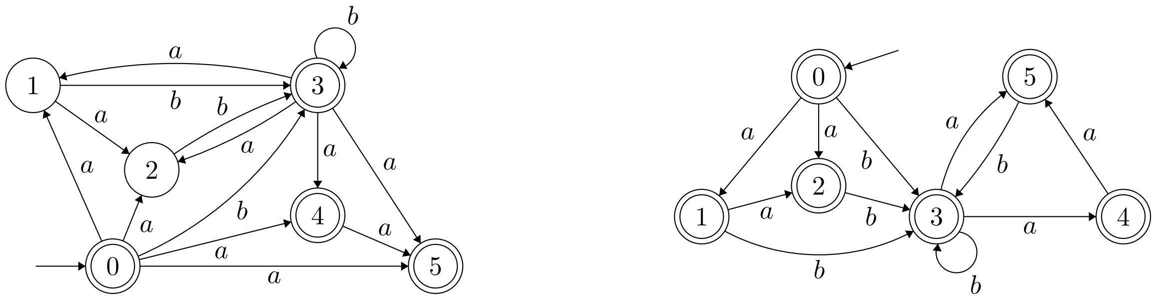
Dans tout le sujet, si l'alphabet n'est pas précisé, il s'agira par défaut de $\Sigma = \{a, b\}^*$

Partie II - Minimisation d'automates

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

Hein ? comment ? m'accuser d'un pareil ridicule ?[1]

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 :



(a) Automate A_1

(b) Automate A_2

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

Question 10 Déterminiser et compléter l'automate A_1 de la figure 1a .

Question 11 Déterminiser 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 :

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

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 renommera 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 [2]).

2) Morphismes d'automates

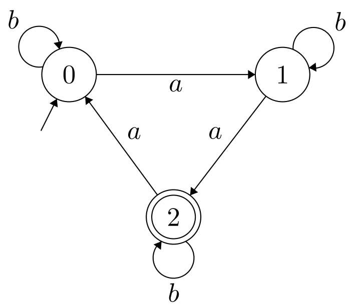


Figure 3

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

OCaml

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

Ainsi, par exemple, l'expression suivante [3] définit l'automate de la figure 3 :

OCaml

```
let a = {
  n = 3;
  init = 0;
  final = function | 2 -> true
                   | _ -> false;
  delta = function | 0 -> (function A -> 1 | B -> 0)
                   | 1 -> (function A -> 2 | B -> 1)
```

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

```ocaml
                  | 2 -> (function A -> 0 | B -> 2)
  }
```

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 $g : Q \to Q'$ 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
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).

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.

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 ${}^tM_a B$ 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 3).

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

si $k \leq 0$ **alors**
  renvoyer $A \times B$ *//Produit de deux entiers*
On note $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ et $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$
$M_1 \leftarrow Strassen(A_{11} + A_{22}, B_{11} + B_{22}, k-1)$
$M_2 \leftarrow Strassen(A_{21} + A_{22}, B_{11}, k-1)$
$M_3 \leftarrow Strassen(A_{11}, B_{12} - B_{22}, k-1)$
$M_4 \leftarrow Strassen(A_{22}, B_{21} - B_{11}, k-1)$
$M_5 \leftarrow Strassen(A_{11} + A_{12}, B_{22}, k-1)$
$M_6 \leftarrow Strassen(A_{21} - A_{11}, B_{11} + B_{12}, k-1)$
$M_7 \leftarrow Strassen(A_{12} - A_{22}, B_{21} + B_{22}, k-1)$
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 3 Algorithme de Strassen

Question 33 Justifier que l'algorithme 3 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[4] 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) [5]. 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.

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

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

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 $x \in E \Leftrightarrow ...$". Vous ne voulez pas écrire "Soit $u \in \mathcal{L}(A) \Leftrightarrow ...$" mais plutôt "Soit $u \in \Sigma^*$. $u \in \mathcal{L}(A) \Leftrightarrow ...$.

(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 utisera 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 :

- Veillez à numéroter 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, 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 grapiller 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 : $\forall u \in S \backslash D, \exists v \in D, (u, v) \in A$
(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.

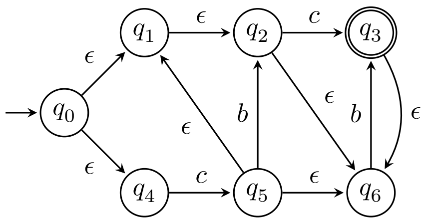
Question 3 Conclure.

Partie I - Questions de cours

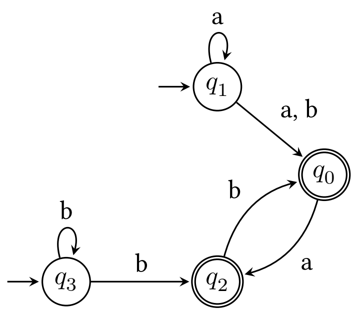
Question 1 Comment sont définies les expressions régulières ?

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 ϵ-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 [1].

Question 5 Énoncer précisément le lemme de l'étoile.
Les langages suivants sont-ils rationnels sur l'alphabet $\{a, b\}$? Justifier.

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.

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. $L_4 = \{a^n b^m \mid n \equiv m \bmod 2\}$

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

Comment s'appelle l'automate obtenu ?

Question 7 On définit :

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

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

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 Σ. L est connu de l'enseignant mais pas de l'élève, Σ 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 $\mathcal{L}^\star$, 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 rappelées en annexe à la fin du sujet.

I - Programmation de l'enseignant

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

OCaml
```ocaml
type letter = int
type word = letter list
```

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

OCaml
```ocaml
type dfa =
  {q0 : int;
   nb_letters : int;
   nb_states : int;
   accepting : bool array;
   delta : int array array}
```

- L'alphabet Σ est $[0 \dots \texttt{nb\_letters} - 1]$.
- L'ensemble Q d'états est $[0 \dots \texttt{nb\_states} - 1]$.
- `q0` indique l'état initial (et appartient donc à $[0 \dots \texttt{nb\_states} - 1]$) ;
- Pour $0 \leq q < \texttt{nb\_states}$, `accepting.(q)` vaut `true` si l'état q est acceptant, `false` sinon.
- `delta` est un tableau de longueur `nb_states`, et pour $0 \leq q < \texttt{nb\_states}$, `delta.(q)` est un tableau de longueur `nb_letters` tel que, pour $0 \leq x < \texttt{nb\_letters}$, `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 Σ, 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
```ocaml
type teacher = {
  nb_letters : int;
  member : word -> bool;
  counter_example : dfa -> word option;
}
```

- `nb_letters` spécifie l'alphabet $\Sigma = [0 \dots \texttt{nb\_letters} - 1]$.
- `member` prend en entrée un mot de Σ^* 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 $L \oplus \mathcal{L}(\texttt{auto})$), 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*).

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);
  - $f : (S \cup S\Sigma)E \to \{0, 1\}$.
- 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 $\texttt{ligne}(s)$. Autrement dit :
$$\texttt{ligne}(s) : \begin{cases} E \to \{0, 1\} \\ e \mapsto f(se) \end{cases}$$

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 $E = \{10, 0, \epsilon\}$.

		ϵ	0	10
S	ϵ	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 $\texttt{ligne}(s) = \texttt{ligne}(s')$ et pour tout $a \in \Sigma$, on a $\texttt{ligne}(sa) = \texttt{ligne}(s'a)$.
- Elle est dite *close* si, pour tout $t \in S\Sigma$, il existe $s \in S$ tel que $\texttt{ligne}(t) = \texttt{ligne}(s)$.

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 :

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

		ϵ	0
S	ϵ	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 $\delta^*(q_0, s) = \texttt{ligne}(s)$.

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 $\varphi : Q_1 \to Q_2$ telle que :

(1) φ 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 $M(S, E, f)$.

III - Algorithme $\mathcal{L}^\star$

L'algorithme $\mathcal{L}^\star$ 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).

$S \leftarrow \{\epsilon\}$
$E \leftarrow \{\epsilon\}$
Calculer f à l'aide de requêtes d'appartenance.
tant que *Vrai* **faire**
  tant que (S, E, f) *n'est pas close ou pas cohérente* **faire**
    si (S, E, f) *n'est pas close* **alors**
      Trouver $s \in S$ et $a \in \Sigma$ tels que $\texttt{ligne}(sa) \neq \texttt{ligne}(s')$ pour tout $s' \in S$.
      $S \leftarrow S \cup \{sa\}$
      Étendre f à $(S \cup S\Sigma)E$ en effectuant des requêtes d'appartenance.
    sinon si (S, E, f) *n'est pas cohérente* **alors**
      Trouver $s, s' \in S$, $a \in \Sigma$ et $e \in E$ tels que $\texttt{ligne}(s) = \texttt{ligne}(s')$ et $f(sae) \neq f(s'ae)$.
      $E \leftarrow E \cup \{ae\}$
      Étendre f à $(S \cup S\Sigma)E$ en effectuant des requêtes d'appartenance.
  $M \leftarrow M(S, E, f)$
  Soumettre à l'enseignant la conjecture M.
  si *l'enseignant répond par un contre-exemple* w **alors**
    Ajouter w et ses préfixes à S.
    Étendre f à $(S \cup S\Sigma)E$ en effectuant des requêtes d'appartenance.
  sinon
    renvoyer M

Algorithme 4 Apprentissage $\mathcal{L}^\star$

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 ;
- $|(S, E, f)| \stackrel{df}{=} \left|\{\texttt{ligne}(s) \mid s \in S\}\right|$ 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 $\mathcal{L}^\star$ 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 $\mathcal{L}^\star$.

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

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

```ocaml
(* Le type d'une table d'obsevation *)
type t

(* Renvoie le nombre de lignes *distinctes* de la table. *)
(* Les lignes sont numérotées de 0 à nb_rows - 1. *)
val nb_rows : t -> int

(* Renvoie la taille de l'alphabet *)
val nb_letters : t -> int

(* Renvoie le numéro de la ligne correspondant à un mot de $S \cup S\Sigma$. *)
val get_row_number : t -> word -> int

(* Prend en entrée un mot $w \in S\cup S\Sigma$ et un mot $e \in E$ *)
(* et renvoie $f(w\cdot e)$. *)
val compute_f : t -> word -> word -> bool

(* Applique une fonction g successivement à tous les mots de l'ensemble $S$. *)
val iter_s : t -> (word -> unit) -> unit

(* Applique une fonction g successivement à tous les mots de l'ensemble $S\Sigma$. *)
```

```ocaml
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
exception Incomplete of word
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 $\texttt{ligne}(w) \neq \texttt{ligne}(u)$ pour tout $u \in S$, si la table n'est pas close.
On ajoute à la signature du module `Obs` la fonction suivante :

```ocaml
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 $\texttt{ligne}(u) = \texttt{ligne}(u')$;
- `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 $\texttt{ligne}(u) = \texttt{ligne}(u')$ et $f(uaw') \neq f(u'aw')$.

3) Algorithme $\mathcal{L}^\star$

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

```ocaml
val add_to_s : t -> word -> (word -> bool) -> unit
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 4.

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

```ocaml
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 4).

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 $\mathcal{L}^\star$.

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 $L^\star$ 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
(* Crée une file vide *)
Queue.create : unit -> 'a Queue.t

(* Test de vacuité *)
Queue.is_empty : 'a Queue.t -> bool

(* Ajout d'un élément *)
Queue.push : 'a -> 'a Queue.t -> unit

(* Extraction de l'élément le plus ancien *)
(* Lève l'exception Queue.Empty si la file est vide *)
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'ϵ-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 reconnait un unique mot $u = u_1 ... u_n$.

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 :

- Veillez à numéroter 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 indépendemment 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'u

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 $f : A \to \mathbb{R}$ est une fonction dite de *pondération*.

Partie 0 - Application du cours

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

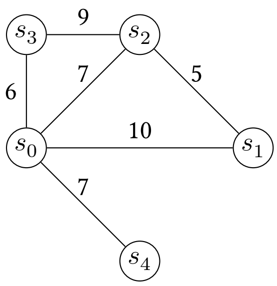


Figure 1 – Le graphe G_0.

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 $\overline{X} \neq \emptyset$ (où $\overline{X}$ 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 $E(X)$.

Partie 1 - Couplage maximum dans un graphe biparti

Cette partie est à traiter en OCaml.

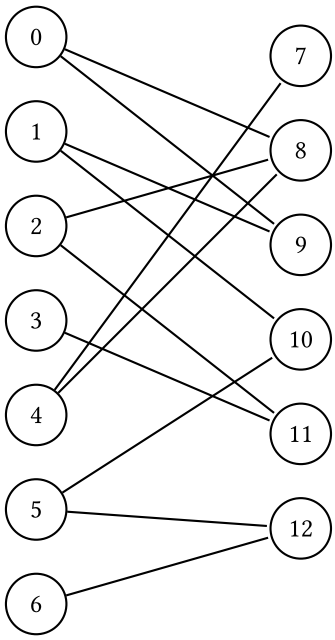
Dans cette partie, on cherche à implémenter en OCaml la recherche d'un couplage de cardinalité maximale dans un graphe biparti, selon la méthode vue en cours.

On travaillera dans cette partie avec les types suivants :

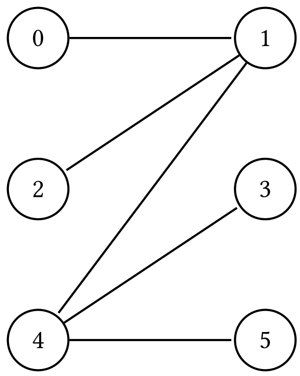
```ocaml
type sommet = int
type graphe = sommet list array

type arete = {x : sommet; y : sommet}

type couplage = arete list
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`Δ`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\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_parcours : graphe -> sommet -> sommet array` telle que `arbre_parcours 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.

$C \leftarrow \emptyset$
tant que *il existe un chemin c augmentant pour C* **faire**
  $C \leftarrow C \Delta c$
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 σ 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 (puisqu'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 $c^*_{G_1}(s_0, s_8)$.

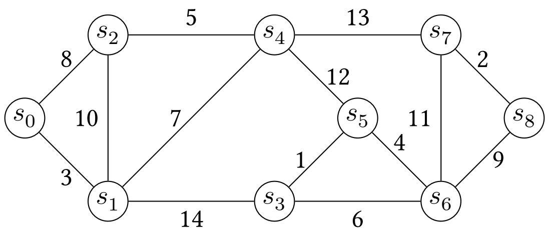


Figure 3 – Le graphe pondéré G_1.

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.

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 :

```c
struct edge {
    int x;
    int y;
    double weight;
};
typedef struct edge edge_t;

struct graph {
    int n;
    int* degrees;
    edge_t** adj;
};
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 $k \in [\![0; n-1]\!]$ 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 ?

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.

$F \leftarrow \emptyset$
$X \leftarrow \{x_0\}$ (un sommet quelconque de G)
tant que $X \neq S$ **faire**
  Trouver $xy \in A$ de poids minimal telle que $x \in X$ et $y \in S \backslash X$.
  $F \leftarrow F \cup \{xy\}$
  $X \leftarrow X \cup \{y\}$
renvoyer $T = (X, F)$

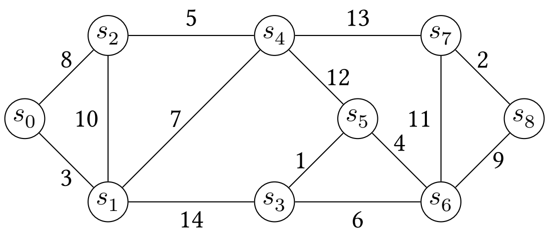


Figure 4 – Le graphe pondéré G_1.

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 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 :

```c
// Création d'une file vide.
// L'argument capacity indique le nombre maximal
// d'arêtes que la file pourra contenir.
// La complexité de cette fonction est en O(capacity).
heap_t *heap_create(int capacity);

// Libération des ressources associées à une file
// Complexité O(1)
void heap_free(heap_t *heap);

// Détermine si la file est vide
// Complexité O(1)
bool heap_is_empty(heap_t *heap);

// Ajoute une arête à la file.
void heap_push(heap_t *heap, edge_t pair);

// Renvoie l'arête de poids minimal présente dans la file,
// et la supprime de la file.
// Erreur si la file est vide.
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.

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 :

- $S' = S \backslash X \cup \{x\}$, où x est un nouveau sommet ;
- $A' = (P_2(S \backslash X) \cap A) \cup \{\{s, x\} \mid s \in S \backslash X, \exists t \in X, s, t \in A\}$;
- 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 4 : Algorithme BSP (*Bottleneck Shortest Path*)

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

tant que $|A| > 1$ **faire**
    $M \leftarrow$ médiane de $\{f(a) \mid a \in A\}$.
    Supprimer de A les arêtes a de poids $f(a) < M$.
    si *il n'existe pas de chemin de s à t* **alors**
        Poser $X_1, \dots, X_k$ les composantes connexes de G.
        Rajouter les arêtes supprimées avant le test.
        Fusionner $X_1, \dots, X_k$ dans G.
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.

Informatique (version étoilée)

Durée : 4 heures

Consignes :

- Veillez à numéroter 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 indépendemment 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'u

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 $f : A \to \mathbb{R}$ est une fonction dite de *pondération*.

Partie 0 - Application du cours

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

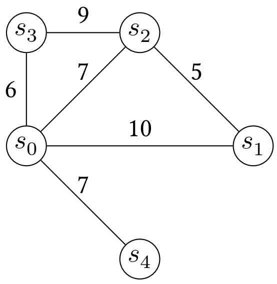


Figure 1 – Le graphe G_0.

Question 1 Soit $G = (S, A)$ un graphe non orienté. On fixe une numérotation $\{a_1, \ldots, a_p\}$ des arêtes, et l'on note $G_i = (S, \{a_1, \ldots, a_i\})$ 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 :

(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 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 $\overline{X} \neq \emptyset$ (où $\overline{X}$ 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 $E(X)$.

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.

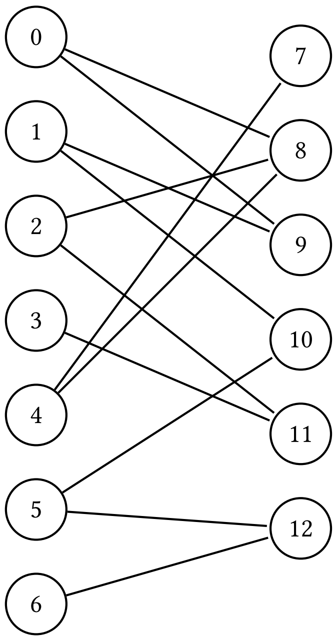
On travaillera dans cette partie avec les types suivants :

OCaml

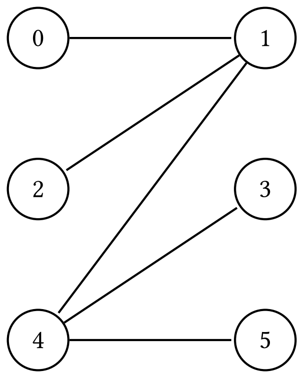
```ocaml
type sommet = int
type graphe = sommet list array

type arete = {x : sommet; y : sommet}

type couplage = arete list
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 $u \to v$ ou $v \to u$. 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`Δ`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_parcours : graphe -> sommet -> sommet array` telle que `arbre_parcours 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.

$C \leftarrow \emptyset$
tant que *il existe un chemin c augmentant pour C* **faire**
  $C \leftarrow C \Delta c$
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 σ 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 (puisqu'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 $c^*_{G_1}(s_0, s_8)$.

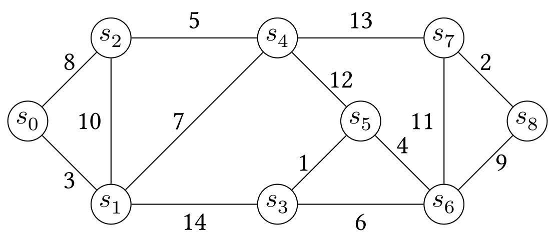


Figure 3 – Le graphe pondéré G_1.

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.

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 :

```c
struct edge {
    int x;
    int y;
    double weight;
};
typedef struct edge edge_t;

struct graph {
    int n;
    int* degrees;
    edge_t** adj;
};
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 $m^k_{i,j}$ 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 $m^k_{i,j}$ cherche-t-on à calculer pour répondre au problème initial ?

Question 4 Montrer que pour tout i, j sommets, $m^0_{i,j} = \begin{cases} +\infty & \text{si } i = j \\ -\infty & \text{si } i \neq j \text{ et il n'y a pas d'arêtes entre } i \text{ et } j \\ f(ij) & \text{sinon} \end{cases}$

Question 5 Trouver une formule de récurrence : exprimer $m^{k+1}_{i,j}$ en fonction des $m^{k'}_{i',j'}$ 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 ?

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 9 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.

$F \leftarrow \emptyset$
$X \leftarrow \{x_0\}$ (un sommet quelconque de G)
tant que $X \neq S$ **faire**
  Trouver $xy \in A$ de poids minimal telle que $x \in X$ et $y \in S \backslash X$.
  $F \leftarrow F \cup \{xy\}$
  $X \leftarrow X \cup \{y\}$
renvoyer $T = (X, F)$

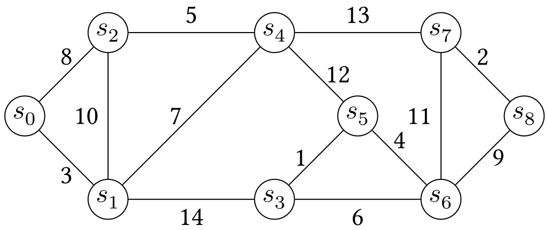


Figure 4 – Le graphe pondéré G_1.

Question 10 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 :

```c
// Création d'une file vide.
// L'argument capacity indique le nombre maximal
// d'arêtes que la file pourra contenir.
// La complexité de cette fonction est en O(capacity).
heap_t *heap_create(int capacity);

// Libération des ressources associées à une file
// Complexité O(1)
void heap_free(heap_t *heap);

// Détermine si la file est vide
// Complexité O(1)
bool heap_is_empty(heap_t *heap);

// Ajoute une arête à la file.
void heap_push(heap_t *heap, edge_t pair);

// Renvoie l'arête de poids minimal présente dans la file,
// et la supprime de la file.
// Erreur si la file est vide.
edge_t heap_extract_min(heap_t *heap);
```

Question 11 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 12 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 13
Quelle est la complexité de l'algorithme de Prim que vous avez implémenté ? Comparer avec celle de l'algorithme de Kruskal.

Question 14 Comment trouver un arbre couvrant maximal à partir de l'algorithme de Prim ?

Question 15 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 16 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.

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 :

- $S' = S \backslash X \cup \{x\}$, où x est un nouveau sommet ;
- $A' = (P_2(S \backslash X) \cap A) \cup \{\{s, x\} \mid s \in S \backslash X, \exists t \in X, s, t \in A\}$;
- 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 4 : Algorithme BSP (*Bottleneck Shortest Path*)

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

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

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

Question 18 Montrer que l'algorithme renvoie la capacité d'un goulot maximal de s à t dans G.

DEVOIR SURVEILLÉ XV

Informatique

Durée : 4 heures

Consignes :

- Veillez à numéroter 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.
- 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, 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 Σ^* 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 :

- $\Sigma = \{1, +, \times, (,)\}$;
- $V_0 = \{S\}$;
- $P_0 = \{S \to S + S \mid S \times S \mid (S) \mid 1\}$.

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 $1 \times (+1) \notin L(G_0)$.

Programmation

Pour une grammaire hors-contexte $G = (\Sigma, V, P, S)$, on représente Σ 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
```ocaml
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
```ocaml
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 α tels que $X_i \to \alpha$ soit une production de la grammaire. On définit donc le type `grammaire` :

OCaml
```ocaml
type grammaire = mot list array
```

Exemple On considère la grammaire G_1 définie sur $\Sigma = \{a, b, c\}$ par :

- $S \to SA \mid A \mid B$;
- $A \to AC \mid CC \mid a$;
- $B \to b$;
- $C \to c \mid \varepsilon$.

En supposant que `a`, `b` et `c` sont des objets de type `token`, cette grammaire peut être représentée par :

OCaml
```ocaml
let g1 = [| [[V 0; V 1]; [V 1]; [V 2]];
            [[V 1; V 3]; [V 3; V 3]; [T a]];
            [[T b]];
            [[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 Σ

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 <= 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
```
 fusion [0; 3; 4] [2; 3; 5; 6];;
- : 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
```
 tri_unique [4; 1; 2; 4; 0; 5; 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 Σ, l'alphabet terminal de la grammaire G. On supposera que Σ 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
```ocaml
val calcul_sigma : unit -> token list
```

On suppose dans la suite que l'on a défini une variable globale `sigma` par :

OCaml
```ocaml
let sigma = calcul_sigma ()
```

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 $X \to \alpha Y \beta$ 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
```ocaml
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
```ocaml
val calcul_accessibles : unit -> bool array
```

Dans toute la suite, on supposera que toutes les variables de la grammaire sont accessibles.

IV - Symboles nuls

Pour $X \in V$, on dit que X est *nul* si $\epsilon \in L(X)$. On définit $\textsc{Nul}(X)$ comme le prédicat associé (autrement dit, $\textsc{Nul}(X) = $ vrai si $X \Rightarrow^* \epsilon$, $\textsc{Nul}(X) = $ faux sinon).

Question 12 Déterminer, en justifiant brièvement, $\textsc{Nul}(X)$ pour chaque variable X de la grammaire G_1.

On considère l'algorithme suivant :

Entrées : Une grammaire $G = (\Sigma, V, P, S)$
$\mathcal{N} \leftarrow \emptyset$
`fini` $\leftarrow$ `faux`
tant que $\neg$`fini` **faire**
  `fini` $\leftarrow$ `vrai`
  pour chaque $X \to X_1 \dots X_n \in P$ **faire** //$X \to \epsilon$ donne $n = 0$
    si $X \notin \mathcal{N}$ *et* $X_i \in \mathcal{N}$ *pour tout* $i \in [1 \dots n]$ **alors**
      $\mathcal{N} \leftarrow \mathcal{N} \cup \{X\}$
      `fini` $\leftarrow$ `faux`
renvoyer $\mathcal{N}$

Algorithme 5 Calcul des symboles nuls

Question 13 Montrer que l'algorithme 5 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 $\textsc{Nul}(X_i)$ est vrai.
On rappelle l'existence de la fonction `List.for_all` de type `('a -> bool) -> 'a list -> 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
```ocaml
val calcul_nul : unit -> bool array
```

On suppose à présent avoir créé une variable globale `nul` en exécutant :

OCaml
```ocaml
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 α tel qu'il existe une règle $X \to \alpha \in P$.

On étend la définition de $\textsc{Nul}$ à l'ensemble des mots $\alpha \in (\Sigma \cup V)^*$: un mot α est dit *nul* si $\alpha \Rightarrow^* \epsilon$. On admet que pour $\alpha = x_1 \dots x_n \in (\Sigma \cup V)^*$:

$$\textsc{Nul}(\alpha) = \bigwedge_{i=1}^{n} (x_i \in V \wedge \textsc{Nul}(x_i))$$

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
```ocaml
val mot_nul : mot -> bool
```

V - Ensembles Premier(α)

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 α. Formellement :

$$\textsc{Premier}(\alpha) = \{a \in \Sigma \mid \exists \beta \in (\Sigma \cup V)^*, \alpha \Rightarrow^* a\beta\}.$$

Question 18 Déterminer, sans justifier, les ensembles Premier(X) pour $X \in V$ et Premier(α) pour chaque α apparaissant du côté droit d'une règle de production de la grammaire G_1.

Question 19 Pour une grammaire quelconque, que vaut Premier(ε) ? 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(α) et Nul(x).

Question 21 En s'inspirant de l'algorithme 5, 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
```ocaml
val empty : TokenSet.t
val mem : token -> TokenSet.t -> bool
val add : token -> TokenSet.t -> TokenSet.t
val union : TokenSet.t -> TokenSet.t -> TokenSet.t
val inter : TokenSet.t -> TokenSet.t -> TokenSet.t
val cardinal : TokenSet.t -> int
val subset : TokenSet.t -> TokenSet.t -> bool
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
```ocaml
val calcul_premier : unit -> TokenSet.t array
```

Pour la suite, on supposera créée une variable globale `premier` définie par :

OCaml
```ocaml
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
```ocaml
val premier_mot : mot -> TokenSet.t
```

VI - Ensembles Suivant(X)

Pour $X \in V$, on définit Suivant(X) comme l'ensemble des symboles terminaux qui peuvent suivre X dans un mot obtenu par une dérivation depuis S. Formellement :

$$\textsc{Suivant}(X) = \{a \in \Sigma \mid \exists \alpha, \beta \in (\Sigma \cup V)^*,\ S \Rightarrow^* \alpha X a \beta\}$$

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 Suivant(X) contient `EOF`. En particulier, Suivant(S) contient toujours `EOF`.

Question 24 Déterminer, sans justifier, les ensembles 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 $X_i \to \alpha_i X \beta_i$ avec $\alpha_i, \beta_i \in (\Sigma \cup V)^*$. Notons qu'une règle comme $Y \to aXbXc$ 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 Suivant(X) en fonction des X_i, α_i et β_i.

Question 26 Écrire une fonction `calcul_suivant` qui renvoie un tableau `suivant` de longueur $|V|$ tel que `suivant.(i)` soit l'ensemble Suivant(X_i).

OCaml
```ocaml
val calcul_suivant : unit -> TokenSet.t array
```

On suppose dans la suite avoir créé une variable globale `suivant` par :

OCaml
```ocaml
let suivant = calcul_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 $X \to \alpha \mid \beta$:

- Premier$(\alpha) \cap$ Premier$(\beta) = \emptyset$;
- si Nul(β), alors $\neg$Nul(α) et Premier$(\alpha) \cap$ 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 :
  - $S \to BA$;
  - $A \to +BA \mid \varepsilon$;
  - $B \to DC$;
  - $C \to \times DC \mid \varepsilon$;
  - $D \to 1 \mid (S)$.

Question 28 Déterminer les valeurs de Nul(X), Premier(X) et 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
```ocaml
val est_LL1 : unit -> bool
```

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 $a \in \Sigma$:

$$X \to \alpha \in TAS(X, a) \iff a \in \textsc{Premier}(\alpha) \vee (\textsc{Nul}(\alpha) \wedge a \in \textsc{Suivant}(X))$$

Par exemple, voici la table d'analyse syntaxique de la grammaire G_1 :

	a	b	c	EOF
S	$S \to SA \mid A$	$S \to B$	$S \to SA \mid A$	$S \to SA \mid A$
A	$A \to AC \mid a$		$A \to CC$	$A \to AC \mid CC$
B		$B \to b$		
C			$C \to c$	$C \to \varepsilon$

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 $X \to \alpha_i$ 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) Hasthtbl.t` est le type des tables de hachage ayant des clés de type `'a` et des valeurs de type `'b`;
- `Hashtbl.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);
- `Hasthtbl.add` permet d'ajouter une association à une table;
- `Hashtbl.mem` permet de tester la présence d'une clé;
- `Hashtbl.find` et `Hashtbl.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
```ocaml
val Hashtbl.create : int -> ('a, 'b) Hashtbl.t
val Hashtbl.add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit
val Hashtbl.mem : ('a, 'b) Hashtbl.t -> 'a -> bool
val Hashtbl.find : ('a, 'b) Hashtbl.t -> 'a -> 'b
val Hashtbl.find_opt : ('a, 'b) Hashtbl.t -> 'a -> 'b
```

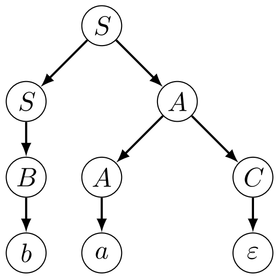
Question 33 Écrire une fonction `table_analyse` qui renvoie la table d'analyse syntaxique de g.
Remarque : on pensera à rajouter le token EOF à l'alphabet.

OCaml
```ocaml
val table_analyse : unit -> (int * token, symbole list list) Hashtbl.t
```

Pour la suite, on supposera avoir défini une variable globale `tas` par :

OCaml
```ocaml
let tas = table_analyse ()
```

On représente un arbre de dérivation par le type suivant :




```
N(0, [N(0, [N(2, [F b])]);
          N(1, [N(1, [F a]);
                N(3, [Epsilon])])])
```

```
type arbre_syntaxe =
  | Epsilon
  | F of token
  | 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).

```
val analyse_syntaxique : mot -> arbre_syntaxe
```

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.*

Observations sur le DS Analyse syntaxique LL(1)

I - Détermination de Σ

(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)