Exercice 1. 3-COLOR

Soit $G = (V, E)$ un graphe non orienté. On appelle **k-coloration** de G une fonction $c : V \longrightarrow \{1, 2, \ldots, k\}$ telle que pour tout arc $(u, v) \in E$, on a $c(u) \neq c(v)$.
On considère le problème suivant :

3-COLOR

Entrée : un graphe G non orienté
Sortie : un booléen indiquant si G est 3-colorable

1. Montrer que 3-COLOR appartient à la classe NP.
2. Écrire une fonction OCaml `check_3color` qui prend en entrée un graphe G (représenté par liste d'adjacence) et une coloration c (un tableau) et qui renvoie un booléen indiquant si c est une coloration de G.
3. Donner une réduction de 3-SAT à 3-COLOR.

Dans la suite, on veut trouver une réduction de 3-COLOR à 3-SAT.
On considère une formule φ de 3-SAT de variables $x_1, ..., x_n$. On veut construire un graphe G qui soit 3-colorable si et seulement si φ est satisfiable.
On commence par ajouter, dans G, n sommets (encore appelés x_1, ..., x_n par abus de notation) correspondant à x_1, ..., x_n, n sommets correspondant à $\neg x_1$, ..., $\neg x_n$ et 3 sommets T, F, B reliés 2 à 2.
Dans un 3-coloriage de G, V et F doivent être de couleurs différentes. Chaque variable x_i sera considérée comme fausse si le sommet correspondant est de la même couleur que F et vraie s'il est de la même couleur que T.

4. Expliquer comment ajouter des arêtes à G pour que chaque variable soit vraie ou fausse (c'est-à-dire coloriée avec la même couleur que F ou la même couleur que T).
5. Dessiner un graphe (un *gadget*) avec (au moins) 3 sommets l_1, l_2, s que l'on pourrait ajouter à G tels que :
   - Si e_1 et e_2 sont de la même couleur que F, alors s doit être de même couleur que F.
   - Si e_1 ou e_2 est de la même couleur que T, alors il existe un coloriage de G où s est de la même couleur que T.
6. Soit $l_1 \vee l_2 \vee l_3$ une clause de φ. Expliquer comment ajouter un gadget à G de façon à ce que $l_1 \vee l_2 \vee l_3$ soit vraie si et seulement si G est 3-coloriable.
7. Montrer que 3-COLOR est NP-complet.
8. Appliquer la réduction ci-dessus à la formule $\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$.

Cependant, 2-COLOR est dans la classe P :

9. Écrire une fonction OCaml `2color` qui prend en entrée un graphe G (représenté par liste d'adjacence) avec n sommets et p arêtes et qui renvoie un booléen indiquant si G est 2-colorable, en $O(n + p)$.

Exercice 2. Distance de Hamming
Soit Σ un alphabet. Si $u = u_1...u_n$ et $v = v_1...v_n$ sont deux mots de même longueur sur Σ, leur distance de Hamming est:

$$d(u, v) = |\{i \mid u_i \neq v_i\}|$$

1. Montrer que la distance de Hamming est une distance sur Σ^*.
2. Écrire une fonction `dist : 'a list -> 'a list -> int` calculant la distance de Hamming de deux mots de même longueur, sous forme de listes.

Étant donné un langage L sur Σ, on définit son voisinage de Hamming $\mathcal{H}(L) = \{u \in \Sigma^* \mid \exists v \in L,\ d(u, v) = 1\}$.

3. Donner une expression rationnelle du voisinage de Hamming de $L(0^*1^*)$.
4. Définir par récurrence une fonction H telle que, si e est une expression rationnelle d'un langage L sur $\Sigma = \{0, 1\}$, $H(e)$ est une expression rationnelle de $\mathcal{H}(L)$.
5. Écrire la fonction H précédente en Caml.

Exercice 3. Miroir
Si $m = m_1...m_n$ est un mot, on définit son miroir $\widetilde{m} = m_n...m_1$.
Si L est un langage, on définit son miroir $\widetilde{L} = \{\widetilde{m} \mid m \in L\}$.

1. Donner une expression rationnel du miroir de $L(a(a + b)^*b)$.
2. Soit e une expression rationnelle de langage L. Définir récursivement une expression rationnelle $\widetilde{e}$ de langage $\widetilde{L}$.
3. Écrire une fonction Caml `miroir : 'a regexp -> 'a regexp` renvoyant le miroir d'une expression rationnelle.
   On utilisera le type suivant d'expression régulière (`L(a)` désigne une lettre `a`):

```
type 'a regexp =
    | Vide | Epsilon | L of 'a
    | Somme of 'a regexp * 'a regexp
    | Concat of 'a regexp * 'a regexp
    | Etoile of 'a regexp
```

On suppose disposer d'une structure impérative de dictionnaire en Caml de type `('a,'b) dict` avec les primitives suivantes :

Primitive	Type	Description
`new`	`unit -> ('a,'b) dict`	Crée un nouveau dictionnaire vide
`add`	`('a,'b) dict -> 'a -> 'b -> unit`	`add d cl val` associe, dans le dictionnaire `d`, la clef `cl` à la valeur `val`
`find`	`('a,'b) dict -> 'a -> 'b`	`find d cl` lit, dans le dictionnaire `d`, la valeur associée à la clef `cl`
`keys`	`('a,'b) dict -> 'a list`	`keys d` renvoie la liste (dans un ordre arbitraire) des clefs du dictionnaire `d`

`'a` est le type des clefs, et `'b` le type des valeurs associées aux clefs.

On suppose disposer d'une structure persistante d'ensemble d'entiers de type `set` avec les primitives suivantes :

Primitive	Type	Description
`empty`	`set`	L'ensemble vide
`union`	`set -> set -> set`	L'union de 2 ensembles
`inter`	`set -> set -> set`	L'intersection de 2 ensembles
`card`	`set -> int`	Le cardinal d'un ensemble
`equal`	`set -> set -> bool`	Teste l'égalité de 2 ensembles
`mem`	`int -> set -> bool`	`mem i s` renvoie `true` si l'entier `i` appartient à l'ensemble `s` et `false` sinon
`singleton`	`int -> set`	`singleton i` renvoie l'ensemble à un élément ne contenant que `i`

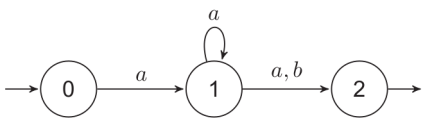
On représente un automate non-déterministe en Caml par le type suivant

```
type auto = {etats : set; init: set;
       trans: (int * char, set) dict ; final: set};;
```

Étant donné un automate `m`,

- `m.init` représente l'ensemble des états initiaux de `m` ;
- `m.final` l'ensemble de ses états finaux,
- `m.trans` sa fonction de transition qui à chaque couple `q,x` associe l'ensemble des états accessibles à partir de l'état `q` en lisant le caractère `x`.

Par exemple, considérons l'automate $\mathcal{M}_1$ suivant :



Si `m1` représente l'automate $\mathcal{M}_1$ en Caml alors `m1.final` est l'ensemble $\{2\}$ et `find m1.trans (1,`a`)` est l'ensemble $\{1;2\}$.

Un automate déterministe est un automate non-déterministe ayant un unique état initial et tel que pour tout état q et toute lettre a, en lisant a à partir de l'état q on peut aller dans au plus un état.

1. L'automate $\mathcal{M}_1$ est-il déterministe ?
2. Écrire une fonction `max_card : ('a, set) dict -> int` qui étant donné un dictionnaire d'ensembles, renvoie le cardinal maximal des ensembles stockés dans le dictionnaire.
3. Écrire une fonction `est_deterministe : auto -> bool` qui renvoie `true` si l'automate donné en argument est déterministe et `false` sinon.
4. Considérons le code incomplet suivant :

```
let etats_suivants m s x =                                    1
let rec parcours_clefs clefs acc = match clefs with           2
        | [] -> acc                                           3
        | (etat,y)::r -> if  .................                4
                 then  .................                      5
                 else  .................                      6
in parcours_clefs (keys m.trans) empty;;                      7
```

   Compléter le code de sorte que `etats_suivants m s x` renvoie l'ensemble des états accessibles à partir d'un état de l'ensemble `s` en lisant `x` dans l'automate `m`.

5. Écrire une fonction `reconnu : auto -> string -> bool` qui, étant donné un automate et une chaîne de caractères, renvoie `true` si l'automate reconnaît la chaîne et `false` sinon.
6. Si au lieu d'une structure impérative de dictionnaire nous avions une structure persistante de dictionnaire, quel serait le type de la primitive `add` ? Si nous avions une structure impérative d'ensemble d'entiers et non pas une structure persistante, pourquoi le type de `empty` devrait-il être changé ?

A3 – Maintenance incrémentale de langages réguliers

On fixe un alphabet fini Σ. Un *mot* $w \in \Sigma^*$ est une suite finie d'éléments de Σ, et un langage $L \subseteq \Sigma^*$ est un ensemble de mots. Un langage est *régulier* s'il est reconnu par un automate fini, ou dénoté par une expression rationnelle (on admet que ces caractérisations sont équivalentes).

Question 0. Si l'on fixe un langage régulier $L \subseteq \Sigma^*$, le problème d'*appartenance à* L est de déterminer, étant donné en entrée un mot $w \in \Sigma^*$, si $w \in L$. Quelle est la complexité du problème d'appartenance à L en fonction de la longueur du mot d'entrée ?

Ce sujet s'intéresse à la *complexité incrémentale* du problème d'appartenance à un langage régulier L. Dans ce problème, on reçoit en entrée un mot $w \in \Sigma^*$ de longueur n. On effectue d'abord un *pré-traitement* pour déterminer si $w \in L$ et pour construire si on le souhaite une structure de données auxiliaire : cette phase de pré-traitement doit s'exécuter en $O(n)$. Ensuite, on reçoit des *mises à jour*, c'est-à-dire des paires (i, a) pour $1 \leq i \leq n$ et $a \in \Sigma$, données l'une après l'autre. À chaque mise à jour, on modifie le mot w pour que sa i-ème lettre devienne a, et on doit déterminer si $w \in L$ après cette modification. La longueur n du mot ne change jamais. La *complexité incrémentale* d'un langage est la complexité dans le pire cas pour prendre en compte une mise à jour, exprimée en fonction de n.

Question 1. Montrer que tout langage régulier a une complexité incrémentale en $O(n)$.

Question 2. Montrer que le langage régulier a^* sur l'alphabet $\Sigma = \{a, b\}$ a une complexité incrémentale en $O(1)$.

Question 3. Soit L_3 le langage des mots sur l'alphabet $\Sigma = \{a, b\}$ comportant au moins deux a, un nombre pair de a, et un nombre de b qui n'est pas divisible par 3. Ce langage est-il régulier ? Quelle est sa complexité incrémentale ?

Question 4. On dénote par $w_1, \ldots, w_n$ les lettres d'un mot $w \in \Sigma^*$ de longueur n. Pour toute permutation $\sigma \colon \{1, \ldots, n\} \to \{1, \ldots, n\}$, on écrit par abus de notation $\sigma(w)$ pour désigner le mot $w_{\sigma(1)} \cdots w_{\sigma(n)}$. Un langage L est *commutatif* si pour tout $w \in \Sigma^*$, pour n la longueur de w, pour toute permutation $\sigma \colon \{1, \ldots, n\} \to \{1, \ldots, n\}$, on a $w \in L$ si et seulement si $\sigma(w) \in L$.

Montrer que tout langage régulier commutatif a une complexité incrémentale en $O(1)$.

Question 5. Proposer une structure de données pour stocker un ensemble d'entiers S qui supporte les opérations suivantes :

- Ajouter un entier dans S, en $O(1)$;
- Retirer un entier de S, en $O(1)$;
- Parcourir les entiers actuellement stockés dans S, en $O(|S|)$.

Écrire le pseudocode pour ces opérations.

Question 6. On considère le langage L_6 sur l'alphabet $\Sigma = \{a, b, c\}$ dénoté par l'expression rationnelle $c^*ac^*bc^*$. Montrer que sa complexité incrémentale est $O(1)$ en utilisant la structure de données de la question précédente.

Suite des questions

Question 7. À quelle classe de langages peut-on généraliser cette technique ?

Question 8. Soient deux langages L_1 et L_2 de complexité incrémentale en $O(1)$. Quelle est la complexité incrémentale des langages $L_1 \cap L_2$ et $L_1 \cup L_2$?

Question 9. On s'intéresse aux langages réguliers admettant au moins une "lettre neutre" (notion que le candidat ou la candidate aura dû identifier à la question 7). Proposer une classe aussi générale que possible de langages de complexité incrémentale en $O(1)$.

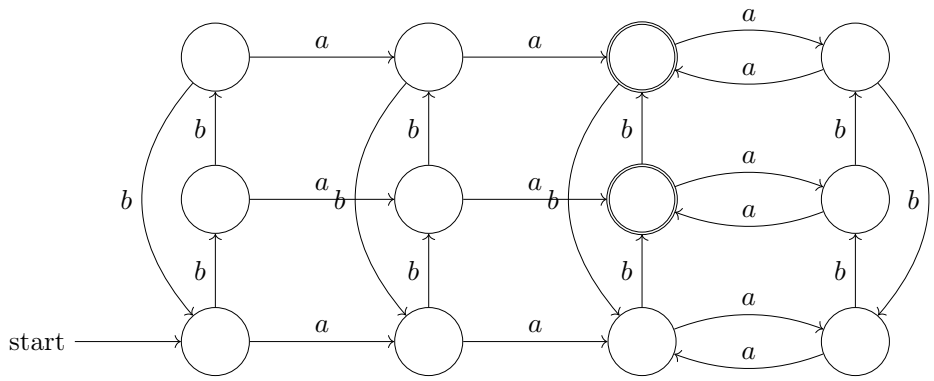
Corrigé

Question 0. On se donne un automate déterministe pour le langage L (on n'a pas besoin de se demander de la complexité de le calculer). Étant donné le mot w, on simule son exécution dans l'automate. La complexité est donc en $O(n)$. *[C'est surtout une question de cours, on attend juste $O(n)$ avec une phrase d'explication. On ne demande pas la complexité en fonction de l'automate ou autre représentation du langage.]*

Question 1. On stocke le mot w dans un tableau (ce qu'on fera toujours par la suite). Il suffit, à chaque mise à jour, de refléter la modification sur le tableau, puis de réévaluer si le mot courant appartient ou non au langage avec la question précédente.

Question 2. On maintient un compteur du nombre courant de a et du nombre courant de b, qu'on initialise au prétraitement. On maintient à jour ces compteurs à chaque mise à jour (ceci nécessite de connaître le nouveau caractère, et l'ancien caractère qui vient d'être changé, qu'on retrouve avec le tableau). Le mot courant est dans le langage si et seulement si le nombre de b est de 0.

Question 3. Le langage est régulier. On peut exhiber un automate :



On peut aussi remarquer plus intelligemment que c'est l'intersection de trois langages dont on montre facilement qu'ils sont réguliers (en exhibant un automate), or on sait d'après le cours que les langages réguliers sont clos par intersection.

Sa complexité incrémentale est en $O(1)$: on maintient un compteur du nombre de a et de b et on teste la condition.

Question 4. *Indication : considérer un automate déterministe et précalculer des chemins.*

On considère un automate déterministe pour le langage concerné. Dans le cadre du précalcul en $O(n)$, on calcule pour chaque état q de l'automate, pour chaque lettre $a \in \Sigma^*$, pour chaque entier

$0 \leq i \leq n$, la fonction $\delta^i(q, a)$ qui indique l'état auquel on aboutit après lecture de a^i depuis q. Ce calcul se fait de proche en proche : $\delta^0(q, a) := q$ et $\delta^{i+1}(q, a) := \delta(\delta^i(q, a), a)$ pour $\delta = \delta^i$ la fonction de transition de l'automate. Noter que le nombre d'états de l'automate et de lettres est une constante indépendante de la longueur n du mot, donc ce précalcul est bien en $O(n)$.

On maintient avec complexité incrémentale $O(1)$ un compteur du nombre d'occurrences de chaque lettre dans le mot, comme à la question précédente.

Pour savoir si le mot courant appartient à L, on utilise le fait que L est commutatif pour savoir que c'est le cas si et seulement si le mot $a_1^{n(a_1)} a_2^{n(a_2)} \cdots a_k^{n(a_k)}$ y appartient, où $a_1, \ldots, a_k$ désigne les lettres de Σ et $n(a)$ désigne le nombre d'occurrences dans le mot courant de la lettre a, qui est l'information maintenue par les compteurs. En effet, par définition des compteurs, le mot en question est une permutation du mot courant.

Or l'acceptation de ce mot par l'automate se détermine en $O(1)$ à l'aide des fonctions précalculées : on calcule $q_1 = \delta^{n(a_1)}(q_0, a_1)$, puis $q_2 = \delta^{n(a_2)}(q_1, a_2)$, et ainsi de suite, en k étapes où k est la taille de l'alphabet qui est une constante indépendante de n. On teste enfin si le dernier état obtenu est final ou non.

Question 5. *Indication : utiliser un tableau, mais s'inspirer de la structure de liste.*

Le problème est qu'une simple liste ne permet pas de retirer un élément identifié par sa valeur (il faut retrouver le maillon de liste qui le stocke), et qu'un tableau ne permet pas de parcourir efficacement les cases occupées. On pourrait naturellement concilier les deux, c'est-à-dire une structure de liste avec un tableau stockant des pointeurs vers les maillons. Pour que la suppression soit plus simple, on utiliserait spontanément des listes doublement chaînées.

Ceci dit, les pointeurs et les listes chaînées ne sont pas au programme, donc on attend plutôt une solution élémentaire à base de tableaux uniquement.

On stocke un tableau "suivant", un tableau "précédent", et une variable "début", avec l'invariant que les éléments actuellement stockés peuvent être parcourus comme début, suivant[début], suivant[suivant[début]], etc., jusqu'à atteindre la valeur -1.

```
début := -1
suivant := tableau d'entiers de taille n initialisé à -1
précédent := tableau d'entiers de taille n initialisé à -1

Fonction Énumérer():
  courant := début
  Tant que courant != -1:
    Produire courant
    courant := suivant[courant]
  Fin Tant que
Fin Fonction

Fonction Ajouter(x):
  // On ajoute au début
  Assertion(suivant[x] == précédent[x] == -1)
  suivant[x] := début
  // précédent[x] reste à -1
  Si début != -1:
    précédent[début] := x
  Fin Si
  début := x
Fin Fonction
```

```
Fonction Supprimer(x):
  // Si on supprime le premier élément, on change début
  Si début == x:
    début := suivant[x]
  Fin Si
  // On saute par dessus l'élément supprimé
  Si suivant[x] != -1:
    précédent[suivant[x]] := précédent[x]
  Fin Si
  Si précédent[x] != -1:
    suivant[précédent[x]] := suivant[x]
  Fin Si
  // Maintenant, on supprime
  suivant[x] := -1
  précédent[x] := -1
Fin Fonction
```

Noter que bien sûr les éléments ne sont pas parcourus dans l'ordre.

Question 6. On maintient une structure de données de la question 5 pour l'ensemble de positions contenant la lettre a, une autre pour l'ensemble des positions contenant la lettre b, en $O(1)$ par la question précédente. On maintient le nombre de a et de b comme aux questions 2–4, en $O(1)$.

Pour savoir si le mot courant est dans L, on vérifie d'abord s'il y a exactement un a et un b. Si non, alors la réponse est non. Si oui, alors on énumère le contenu des deux structures, ce qui est en $O(1)$ car elles contiennent chacune un élément. On trouve aussi la position de l'unique a et de l'unique b, et on les compare, pour savoir si on est ou non dans L.

Question 7. Intuitivement, si on peut partitionner l'alphabet Σ entre des lettres qui n'ont pas d'effet et des lettres sur lesquelles on doit réaliser un mot fini (ou un langage fini), alors la même technique s'appliquera.

Pour être précis (on attend une telle formalisation du candidat ou de la candidate) : on définit une lettre a qui est *neutre* pour L comme à la question 13. On définit le *réduit* de L pour une lettre neutre a comme le langage $L^{-a} := \{w^{-a} \mid w \in L\}$. On étend cette définition à un ensemble non-vide de lettres neutres : $L^{-\Sigma'}$ pour $\Sigma' \subseteq \Sigma$ non-vide est l'ensemble des $w^{-\Sigma'}$ pour $w \in L$ où $w^{-\Sigma'}$ s'obtient en retirant de w toutes les lettres de Σ'. Pour un langage L avec un ensemble non-vide Σ' de lettres neutres, si $L^{-\Sigma'}$ est fini, alors la complexité incrémentale est en $O(1)$ comme à la question précédente.

Pour le montrer, on crée une structure de données de la question 5 pour chaque lettre de $\Sigma \setminus \Sigma'$. On maintient ces structures, ainsi que le nombre d'occurrences de chaque lettre, en $O(1)$. Soit N la longueur maximale d'un mot de $L^{-\Sigma'}$; c'est une constante indépendante de n. Pour savoir si le mot courant appartient ou non à L, tant que le nombre d'occurrences total des lettres de $\Sigma \setminus \Sigma'$ est $> N$, on peut répondre non directement, en $O(1)$. Sinon, on utilise les structures de données de la question 5 pour identifier le sous-ensemble (de taille au plus N) de positions contenant des lettres de Σ'. On le trie en temps $O(N \log N)$, ce qui est toujours constant. On lit le mot formé et on teste (en temps constant car il est de taille $\leq N$) s'il appartient à $L^{-\Sigma'}$, ce qui conclut.

Question 8. Si on peut maintenir une structure avec complexité incrémentale en $O(1)$ pour l'appartenance à L_1, et pour l'appartenance à L_2, alors ces structures nous permettent de savoir si on appartient à L_1 et à L_2, ou à L_1 ou à L_2, toujours en $O(1)$.

Question 9. La réponse naïve (mais nécessitant un peu de recul) est : la clôture par intersections et par unions de la classe de la question 7 et des langages commutatifs couverts en question 4. Mais on s'attend à ce que le candidat ou la candidate sache se représenter la puissance d'expression de cette classe :

— L'intersection d'un langage de la question 7 avec un langage commutatif permet de poser des conditions commutatives sur les lettres neutres (qui ne sont du coup plus neutres), mais elle n'est pas intéressante sur les autres lettres (puisque le langage est fini)

— L'union de tels langages est intéressante parce qu'elle permet de changer la partition entre lettres neutres et non-neutres. En d'autres termes, on a une complexité incrémentale de $O(1)$ pour le langage "j'ai un nombre pair de a, une lettre neutre b, et exactement un c avant exactement un d, OU j'ai un nombre pair de c, une lettre neutre d, et exactement un a avant exactement un b".

— L'union permet aussi, pour la même partition, de poser des conditions commutatives différentes suivant ce qui est attendu pour les lettres non-neutres, par exemple couvrir des langages comme "j'ai une lettre neutre d, et j'ai exactement un a avant exactement un b et un nombre pair de c OU exactement un b avant exactement un a et un nombre impair de c.

Pour résumer, une forme normale serait : une union finie de langages qui sont l'entrelacement (shuffle) d'un langage commutatif sur un alphabet Σ', et d'un singleton (un seul mot) sur l'alphabet $\Sigma \setminus \Sigma'$. *[Cette caractérisation est celle des langages ZG, voir [AJP21, AP21]. On peut conjecturer qu'il s'agit des seuls langages réguliers de complexité incrémentale $O(1)$, ou en tout cas c'est le cas sous une certaine hypothèse de complexité.]*

Références

[AJP21] Antoine Amarilli, Louis Jachiet, and Charles Paperman. Dynamic Membership for Regular Languages. In *ICALP*, 2021.

[AP21] Antoine Amarilli and Charles Paperman. Locality and Centrality: The Variety ZG. Preprint : `http://arxiv.org/abs/2102.07724`, 2021.