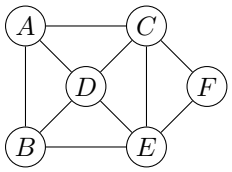
Exercice 1

1. Rappeler la définition de couplage dans un graphe non orienté. Quand peut-on qualifier un couplage de maximal ? Maximum ? Parfait ?
2. Donner un couplage maximum dans le graphe suivant. Donner un couplage maximal qui n'est pas maximum.



3. Qu'est-ce qu'un chemin augmentant alternant pour un graphe $G = (S, A)$ et un couplage $C \subseteq A$ sur ce graphe ?
4. Soit $G = (S, A)$ un graphe. Montrer qu'un couplage C est maximum si et seulement s'il n'existe pas de chemin augmentant alternant pour ce couplage dans le graphe G.

Corrigé

1. Soit $G = (S, A)$ un graphe non orienté. Un couplage C de G est une partie $C \subseteq A$ telle que pour $a_1, a_2 \in C$, $a_1 = a_2$ ou $a_1 \cap a_2 = \emptyset$, c'est-à-dire un ensemble d'arêtes disjointes. Le couplage est maximal si pour tout $a \in A \setminus C$, $C \cup a$ n'est pas un couplage. Il est maximum s'il est de cardinal maximum parmi tous les couplages. Il est parfait si $|C| = \frac{|S|}{2}$ (c'est-à-dire que tous les sommets sont dans une arête de C).
2. L'ensemble $\{\{A, B\}, \{C, D\}, \{E, F\}\}$ est un couplage maximum (il est même parfait). L'ensemble $\{\{A, B\}, \{C, E\}\}$ est maximal sans être maximum.
3. Un chemin augmentant est un chemin élémentaire $(s_0, s_1, \ldots, s_k)$ tel que s_0 et s_k sont libres (ils n'appartiennent à aucune arête de C), et les arêtes du chemin sont alternativement dans $A \setminus C$ et C.
4. – Sens direct : s'il existe un chemin σ augmentant pour C, alors $C \Delta \sigma$ est un couplage de cardinal strictement supérieur à celui de C.
   – Sens réciproque : supposons que C n'est pas un couplage maximum. Soit C' un couplage maximum. Alors le graphe $(S, C \Delta C')$ est un graphe dont les composantes connexes sont soit des chemins, soit des cycles, dont les arêtes alternent entre C et C' (chaque sommet est de degré au plus 2 car il ne peut pas être incident à 2 arêtes de C ou 2 arêtes de C'). L'une de ces composantes connexes contient strictement plus d'arêtes de C' que de C. Cette composante connexe forme donc un chemin augmentant pour C.

Exercice 2

Soit Σ un alphabet. Pour deux mots $u, v \in \Sigma^*$, on appelle **entrelacement** de u et v un mot w qui utilise exactement les lettres de u et v dans leur ordre dans chaque mot.

Par exemple, *babaa* est un entrelacement de *bb* et *aaa* ; *abcabc* est un entrelacement de *aba* et *cbc*. *bacb* n'est pas un entrelacement de *ab* et *cb* car l'ordre n'est pas respecté.

On peut voir le lien avec les entrelacements des tâches de plusieurs fils d'exécution.

On note $\mathcal{E}(u, v)$ l'ensemble des entrelacements de u et v.

1. Donner les entrelacements de *ab* et *cb*.
2. Soient $u, v \in \Sigma^*$.
   (a) Majorer grossièrement le cardinal de $\mathcal{E}(u, v)$.
   (b) Montrer que $\mathcal{E}(u, v)$ est un langage régulier.
3. Soient L_1, L_2 deux langages réguliers. Montrer que $\mathcal{E}(L_1, L_2) = \bigcup_{\substack{u \in L_1 \\ v \in L_2}} \mathcal{E}(u, v)$ est un langage régulier.
4. Soient $u, v, w \in \Sigma^*$. Proposer un algorithme de programmation dynamique permettant de savoir si $w \in \mathcal{E}(u, v)$. Préciser la complexité.

Corrigé

1. On a $\mathcal{E}(ab, cb) = \{abcb, acbb, cabb, cbab\}$.
2. (a) On a plusieurs majorations possibles. La plus simple est peut-être $|\Sigma|^{|u|+|v|}$.
   (b) C'est un langage fini, donc régulier.
3. Soient $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ et $A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ deux AFD complets reconnaissant L_1 et L_2 respectivement. L'idée est de créer un automate constitué de A_1 et A_2, où la lecture d'une lettre emprunte une transition dans A_1 OU dans A_2. On est dans un état final si on atteint un état final dans chacune des copies.

   Formellement, on pose $A = (Q, \Sigma, \Delta, I, F)$ où :
   - $Q = Q_1 \times Q_2$;
   - $I = \{(q_1, q_2)\}$;
   - $F = F_1 \times F_2$;
   - pour $(q, q') \in Q$, $a \in \Sigma$, $\Delta((q, q'), a) = \{(\delta(q, a), q'), (q, \delta(q', a))\}$.

   Alors A est un automate non déterministe reconnaissant $\mathcal{E}(L_1, L_2)$.
4. On note $u = a_1 \ldots a_n$, $v = b_1 \ldots b_m$ et $w = c_1 \ldots c_p$. Si $p \neq n + m$, alors $w \notin \mathcal{E}(u, v)$. Pour la suite, supposons $p = n + m$. On pose $B_{i,j} = \begin{cases} \top & \text{si } c_1 \ldots c_{i+j} \in \mathcal{E}(a_1 \ldots a_i, b_1 \ldots b_j) \\ \bot & \text{sinon} \end{cases}$

   On remarque que $B_{0,0} = \top$ et que dans le cas général : $B_{i,j} = ((a_i = c_{i+j}) \wedge B_{i-1,j}) \vee ((b_j = c_{i+j}) \wedge B_{i,j-1})$.

   Avec cette formule de récurrence, on peut calculer tous les $B_{i,j}$. La réponse à la question posée est donnée par $B_{n,m}$. La complexité est en $\mathcal{O}(n \times m)$, car il y a cet ordre de grandeur de $B_{i,j}$ à calculer, et chacun peut se calculer en temps constant si on mémoïse les résultats.

Exercice 3: L'exercice suivant est à traiter dans le langage OCaml.

On considère le problème `k-partition` :

* **Instance** : un multi-ensemble S d'entiers de cardinal $n = k \times m$.
* **Question** : existe-t-il une partition de S en m sous-ensembles de même cardinal k et de même somme ?

Par exemple, le multi-ensemble $S = \{4, 5, 5, 5, 5, 6\}$ est une instance positive de `3-partition`, car il existe une partition de S en deux triplets $S_1 = \{4, 5, 6\}$ et $S_2 = \{5, 5, 5\}$, de même somme 15.

On s'intéresse dans un premier temps au problème `2-partition`.

1. Écrire une fonction `deux_partition : int array -> bool` qui prend en argument un tableau d'entiers, qu'on supposera de taille paire $n = 2m$, représentant un multi-ensemble S et renvoie un booléen qui vaut `true` si et seulement si S est une instance positive de `2-partition`. La fonction devra avoir une complexité en $\mathcal{O}(n \log n)$ et pourra modifier le tableau si nécessaire. On rappelle qu'on peut trier un tableau `tab` par la commande `Array.sort compare tab`.
2. Décrire en français comment obtenir une complexité linéaire en moyenne en utilisant une table de hachage. On ne demande pas de coder cette solution.

On souhaite écrire une fonction pour résoudre le problème `3-partition`. On représente une partition en triplets d'un tableau de taille $n = 3m$ par une permutation σ de $[\![0, n]\!]$ telle que le premier triplet est celui constitué des éléments d'indices $(\sigma(0), \sigma(1), \sigma(2))$, le deuxième triplet est constitué des éléments d'indices $(\sigma(3), \sigma(4), \sigma(5))$, etc.

On note que cette représentation n'est pas bijective, car deux permutations peuvent correspondre à la même partition.

3. Écrire une fonction `valide : int array -> int array -> bool` qui prend en argument un tableau `tab` de taille n et un tableau `sigma` de taille n représentant une permutation σ de $[\![0, n]\!]$ et renvoie un booléen qui vaut `true` si cette permutation représente une partition de `tab` en triplets qui sont tous de même somme, et `false` sinon.

Pour résoudre le problème `3-partition`, on souhaite énumérer les permutations de $[\![0, n-1]\!]$ dans l'ordre lexicographique. Par exemple, les permutations de $[\![0, 2]\!]$ dans l'ordre lexicographique sont $(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1)$ et $(2, 1, 0)$. On se donne une fonction

`suivante : int array -> bool` qui prend en argument un tableau `sigma` de taille n représentant une permutation σ de $[\![0, n]\!]$ et :

- si σ n'est pas la dernière permutation selon l'ordre lexicographique, alors `suivante sigma` modifie `sigma` pour qu'il devienne la permutation suivante, et renvoie le booléen `true` ;
- si σ est la dernière permutation, alors `suivante sigma` renvoie `false` sans modifier le tableau.

4. Écrire une fonction `trois_partition : int array -> bool` qui prend en argument un tableau d'entiers, qu'on supposera de taille paire $n = 2m$, représentant un multi-ensemble S et renvoie un booléen qui vaut `true` si et seulement si S est une instance positive de `3-partition`.
5. Quelle est la complexité temporelle de la fonction précédente (en admettant que `suivante` a une complexité amortie constante) ?

On considère le problème `ABC-partition` :

* **Instance** : trois multi-ensembles A, B et C d'entiers, de même cardinal m.
* **Question** : existe-t-il une partition de $A \sqcup B \sqcup C$ en m sous-ensembles de même somme, chaque sous-ensemble contenant exactement un élément de A, un élément de B et un élément de C ?

On admet que ce problème est NP-complet.

6. En considérant un multi-ensemble $S = \{1000a+100 \mid a \in A\} \sqcup \{1000b+10 \mid b \in B\} \sqcup \{1000c+1 \mid c \in C\}$, montrer que `3-partition` est NP-complet par une réduction depuis `ABC-partition`.

Corrigé

1. L'idée est de trier le tableau. Dès lors, s'il existe une 2-partition, il faut forcément regrouper le plus petit élément avec le plus grand, le deuxième plus petit avec le deuxième plus grand, etc. Il suffit donc de tester l'égalité entre ces sommes.

```
let deux_partition tab =
    Array.sort compare tab;
    let n = Array.length tab in
    try for i = 0 to n / 2 - 1 do
        if tab.(i) + tab.(n - 1 - i) <> tab.(0) + tab.(n - 1) then
           failwith "Faux"
        done;
        true
    with Failure _ -> false
```

2. On peut commencer par déterminer quelle doit être la somme cible c de chaque paire de la 2-partition. Il s'agit de la somme totale des éléments du tableau, divisée par m. Ensuite, on crée une table de hachage vide et on parcourt chaque élément x du tableau :
   - si $c - x$ est dans la table de hachage, on supprime $c - x$ de la table de hachage ;
   - sinon, on ajoute x dans la table de hachage.

   S'il reste des éléments dans la table de hachage à la fin du parcours, on renvoie Faux, sinon on renvoie Vrai.
3. Il faut déterminer la somme cible à atteindre. Ensuite, on peut vérifier que chaque triplet a bien une somme égale à la cible.

```
let valide tab sigma =
    let n = Array.length tab in
    let somme = Array.fold_left (+) 0 tab in
    try
        if somme mod (n / 3) <> 0 then failwith "Faux";
        for i = 0 to n / 3 - 1 do
            let i0 = sigma.(3*i) and i1 = sigma.(3*i+1) and i2 = sigma.(3*i+2) in
            if tab.(i0) + tab.(i1) + tab.(i2) <> somme / (n / 3) then
                failwith "Faux"
        done;
        true
    with Failure _ -> false;;
```

4. Il suffit alors de parcourir toutes les permutations :

```
let trois_partition tab =
    let n = Array.length tab in
    let sigma = Array.init n Fun.id in
    let b = ref (valide tab sigma) in
    while not !b && suivante sigma do
        b := valide tab sigma
    done;
    !b
```

5. Il y a de l'ordre de $n!$ permutations. L'appel à la fonction `valide` se fait en temps linéaire en n. Dans le pire des cas (celui d'une instance négative), la complexité totale sera en $\Theta((n+1)!)$ (ce qui est très mauvais).
6. Soit (A, B, C) une instance de `ABC-partition`. On pose $S = \{1000a + 100 \mid a \in A\} \sqcup \{1000b + 10 \mid b \in B\} \sqcup \{1000c + 1 \mid c \in C\}$ comme défini dans l'énoncé. Montrons que (A, B, C) est une instance positive de `ABC-partition` si et seulement si S est une instance positive de `3-partition` :
   - supposons qu'il existe une partition de $A \sqcup B \sqcup C$ en triplets de même somme s. Alors la partition correspondante de S ne contiendra que des triplets (a, b, c) de même somme $1000a + 100 + 1000b + 10 + 1000c + 1 = 1000s + 111$;
   - réciproquement, supposons qu'il existe une tri-partition de S. Alors chaque triplet de la partition a une somme avec la même unité, la même dizaine et la même centaine. On en déduit que chaque triplet contient exactement un élément de A, un élément de B et un élément de C. La partition correspondante de $A \sqcup B \sqcup C$ est bien une partition valide.

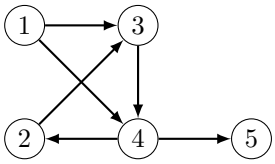
   La construction de S se faisant en temps polynomial, on en déduit que `ABC-partition` $\leqslant_{\mathsf{m}}^{\mathsf{p}}$ `3-partition`. Comme `ABC-partition` est NP-complet, on en déduit que `3-partition` est NP-difficile. Comme ce problème est dans NP (un certificat est une partition valide, cela se vérifie bien en temps polynomial, comme nous l'avons fait précédemment), il est NP-complet.

Exercice 4: Familles closes de graphes

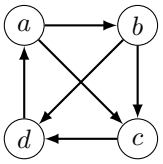
Définition

Un **graphe** $G = (V, E)$ est la donnée d'un ensemble fini non vide de sommets V et d'un ensemble d'arêtes $E \subseteq V \times V$. On imposera toujours que $V \subseteq \mathbb{N}$ et on autorise les **boucles**, c'est-à-dire les arêtes de la forme (v, v) allant d'un sommet $v \in V$ vers lui-même. Un **homomorphisme** d'un graphe $G = (V, E)$ vers un graphe $G' = (V', E')$ est une fonction $\phi : V \to V'$ telle que, pour tout $(x, y) \in E$, $(\phi(x), \phi(y)) \in E'$.

1. Pour G et G' représentés ci-dessous, décrire un homomorphisme de G vers G'. Y a-t-il un homomorphisme de G' vers G ?



G



G'

2. Écrire le pseudocode d'un algorithme naïf qui détermine, étant donnée la matrice d'adjacence de deux graphes G et G', s'il existe un homomorphisme de G vers G' et le calcule le cas échéant. Déterminer sa complexité en temps et en espace.
3. Soient G et G' deux graphes orientés et soient $G_1, \dots, G_n$ et $G'_1, \dots, G'_m$ les composantes connexes (c'est-à-dire dans les représentations désorientées des graphes) de G et G' respectivement. Supposons que l'ont ait déterminé, pour chaque $(i,j) \in [\![1,n]\!] \times [\![1,m]\!]$ s'il existe un homomorphisme de G_i vers G'_j. Peut-on déterminer s'il existe un homomorphisme de G vers G' ?

Définition

Une **famille** de graphes est un ensemble de graphes (possiblement infini). Une famille $\mathcal{F}$ est dite **close** si pour tout graphe $G \in \mathcal{F}$, si G a un homomorphisme vers un graphe G', alors $G' \in \mathcal{F}$.

4. On pose $\mathcal{F}_C$ la famille des graphes possédant un cycle. Montrer que $\mathcal{F}_C$ est close.
5. Déterminer un graphe $G_\perp$ tel que la famille de tous les graphes est l'unique famille close contenant $G_\perp$.
6. Justifier qu'une famille close non vide est nécessairement infinie.

Définition

Un graphe $G = (V, E)$ est dit **minimal** pour une famille close $\mathcal{F}$ si tout graphe obtenu à partir de G en supprimant une arête n'est pas dans $\mathcal{F}$.

7. Montrer que si $\mathcal{F} \neq \emptyset$ alors $\mathcal{F}$ admet un graphe minimal.

Définition

Une famille close $\mathcal{F}$ est dite **finiment engendrée** s'il existe une famille finie $\mathcal{F}'$ telle que $\mathcal{F}$ est exactement l'ensemble des graphes G tel qu'il existe un homomorphisme d'un graphe de $\mathcal{F}'$ vers G.

8. Donner un exemple de famille close finiment engendrée et de famille close qui ne l'est pas.
9. Montrer que si $\mathcal{F}$ est une famille close qui n'est pas finiment engendrée, alors $\forall n \in \mathbb{N}$, il existe un graphe minimal pour $\mathcal{F}$ ayant au moins n arêtes.

Corrigé

1. Plusieurs solutions sont possibles, comme par exemple $\phi(1,2,3,4,5) = (b,a,c,d,a)$. Réciproquement, il n'existe pas d'homomorphisme : tous les sommets de G' ont une arête entrante et une arête sortante, donc ne pourraient être envoyés que vers $2,3,4$ (qui sont dans la même situation). Si $\phi(a) = 2$, alors $\phi(b) = 3$ et $\phi(c) = 4$ mais alors l'arête (a,c) ne peut pas être représentée (et de même si $\phi(a) = 3$ ou 4).
2. On peut agir par force brute en générant toutes les fonctions de V dans V' et en vérifiant si l'une d'entre elle est un homomorphisme. La vérification se fait de la manière suivante :

```
Algorithme : Vérification
Entrée : matrices carrées M et M' de tailles n et m, tableau Phi de taille n
Pour i = 1 à n faire
   Pour j = 1 à n faire
      Si M[i][j] = 1 et M'[Phi[i]][Phi[j]] = 0
          Renvoyer Faux
Renvoyer Vrai
```

Pour générer toutes les fonctions de V dans V', on peut penser à un algorithme d'incrémentation :

```
Algorithme : Incrémentation
Entrée : tableau Phi de taille n, entier m
i <- 1
Tant que i < n + 1 et Phi[i] = m faire
   Phi[i] = 1
   i += 1
Si i < n + 1
   Phi[i] += 1
```

Finalement, la vérification d'existence d'homomorphisme se fait de la manière suivante :

```
Algorithme : Homomorphisme
Entrée : matrices carrées M et M' de tailles n et m
Phi <- tableau de taille n contenant des 1
Tant que Vrai
   Si Vérification(M, M', Phi)
      Renvoyer Phi
   Si Phi ne contient que des m
      Renvoyer Faux
   Incrémentation(Phi)
```

Il est clair que la vérification se fait en temps $\mathcal{O}(n^2)$, et on peut montrer que l'incrémentation est en $\mathcal{O}(1)$ amortie (mais est trivialement en $\mathcal{O}(n)$). Comme il existe m^n fonctions de $[\![1,n]\!]$ dans $[\![1,m]\!]$, la complexité temporelle totale est $\mathcal{O}(n^2m^n)$. La complexité spatiale ne concerne que la création de `Phi`, donc est en $\mathcal{O}(n)$.

3. On commence par montrer que l'image d'un graphe connexe par un homomorphisme est connexe. Soit G un graphe connexe et G' son image par ϕ un homomorphisme. Soient $x, y \in \phi(V)$ et $u, v \in V$ tels que $\phi(u) = x$ et $\phi(v) = y$. G étant connexe, il existe un chemin non orienté entre u et v. Il est clair que l'image de ce chemin est un chemin non orienté entre x et y dans G'.

   Cela suggère, dans le cas général, qu'il existe un homomorphisme de G vers G' si et seulement si pour chaque C.C. de G, il existe un homomorphisme vers une C.C. de G'. Le sens direct découle de la propriété précédente (si V_i est l'ensemble de sommets d'une C.C. de G, alors $\phi(V_i)$ est inclus dans une C.C. de G'). Réciproquement, il suffit de définir $\phi(v)$ comme étant égal à $\phi_{i,j}(v)$, $\phi_{i,j}(v)$ étant un homomorphisme de la C.C. V_i contenant v vers une C.C. V'_j de G'.
4. Soit $v_1, \dots, v_n = v_1$ un cycle de G et ϕ un homomorphisme de G vers G'. Alors $\phi(v_1), \dots, \phi(v_n)$ est un cycle de G', donc $G' \in \mathcal{F}_C$.
5. On définit $G_\perp = (\{0\}, \emptyset)$. Soit $G = (V, E)$ un graphe et $v \in V$. On définit une fonction ϕ de $G_\perp$ vers G par $\phi(0) = v$. Il est clair que ϕ est un homomorphisme, donc si $G_\perp \in \mathcal{F}$ une famille close, alors $G \in \mathcal{F}$. De la même manière, tout graphe sans arête convient.
6. Soit $G \in \mathcal{F}$ une famille close. En rajoutant un nombre arbitraire de sommets à G pour former G', il existe clairement un homomorphisme de G vers G'. On en déduit que $\mathcal{F}$ est infinie.
7. Soit $G \in \mathcal{F}$. On peut supprimer des arêtes de G tant qu'il en existe une qui le laisse dans $\mathcal{F}$. Le graphe ainsi obtenu est bien minimal (éventuellement sans arête auquel cas $\mathcal{F}$ contient tous les graphes).
8. La famille contenant tous les graphes est finiment engendrée (par un unique graphe sans arête comme précédemment). Montrons que $\mathcal{F}_C$ n'est pas finiment engendrée. Par l'absurde, supposons $\mathcal{F}$ une famille finie de cardinal qui engendre $\mathcal{F}_C$. Soit n l'ordre maximal d'un graphe de $\mathcal{F}$ et soit C_{n+1} le cycle à $n+1$ sommets. Supposons que $G = (V, E) \in \mathcal{F}$ est tel qu'il existe un homomorphisme ϕ de G vers C_{n+1}. Par hypothèse, $|\phi(V)| \leqslant |V| \leqslant n$. Considérons G' le sous-graphe de G induit par $\phi(V)$. Comme $\phi(V) < n+1$, il est clair que G' n'est pas cyclique. De plus, il est clair que ϕ est un homomorphisme de G vers G'. Enfin, comme $\mathcal{F}$ engendre $\mathcal{F}_C$, cela implique que $G' \in \mathcal{F}_C$. On conclut par l'absurde.
9. On montre la contraposée. Soit $\mathcal{F}$ une famille close telle qu'il existe $n \in \mathbb{N}$ tel que tout graphe minimal de $\mathcal{F}$ a au plus n arêtes. Soit $\mathcal{F}_{\min}$ l'ensemble des graphes minimaux pour $\mathcal{F}$. On peut restreindre $\mathcal{F}_{\min}$

en supprimant les sommets isolés (sauf éventuellement le dernier) et en renommant les sommets de 1 à $2n$ (un graphe à n arêtes sans sommet isolé a au plus $2n$ sommets), de telle sorte qu'on considère que $\mathcal{F}_{\min}$ est fini. Montrons que cette famille engendre $\mathcal{F}$. Soit G tel qu'il existe un homomorphisme de $\mathcal{F}_{\min}$ vers G. $\mathcal{F}$ étant close et $\mathcal{F}_{\min} \subseteq \mathcal{F}$, cela implique que $G \in \mathcal{F}$. Réciproquement, soit $G \in \mathcal{F}$. Montrons qu'il existe un homomorphisme d'un graphe de $\mathcal{F}_{\min}$ vers G. Comme à une question précédente, on peut retirer des arêtes de G jusqu'à obtenir un graphe minimal (donc dans $\mathcal{F}_{\min}$, à homomorphisme près), qui aura bien un homomorphisme vers G.

Exercice 5: Langages continuables et mots primitifs

On fixe un alphabet Σ de taille au moins 2. Dans le sujet, on considèrera des automates sur Σ qui seront toujours considérés finis, déterministes et complets.

Définition

Soit $u \in \Sigma^* \setminus \{\varepsilon\}$. u est dit **primitif** s'il n'existe pas de mot $v \in \Sigma^*$ et d'entier $p > 1$ tel que $u = v^p$.

Un langage rationnel L est dit **continuable** si et seulement si

$$\forall u \in \Sigma^*, \exists v \in \Sigma^*, uv \in L$$

1. Le mot *abaaabaa* est-il primitif ? Le mot *ababbaabbbababbab* (de longueur 17) est-il primitif ?
2. Proposer un algorithme naïf qui, étant donné un mot, détermine s'il est primitif. Préciser la complexité en temps et en espace.
3. Donner un exemple de langage rationnel infini qui ne contient aucun mot primitif.
4. Donner un exemple de langage rationnel infini qui ne contient que des mots primitifs.
5. Donner un exemple de langage rationnel infini non continuable. Existe-t-il des langages rationnels continuables dont le complémentaire est infini ?
6. Étant donné un automate A, proposer un algorithme qui détermine si $L(A)$ est continuable. Justifier sa correction et préciser la complexité en temps et en espace.
7. Montrer qu'un langage rationnel continuable contient une infinité de mots primitifs.
8. Étant donné un langage rationnel continuable L reconnu par un automate A, donner une borne supérieure de la taille du plus petit mot primitif de L.
9. La réciproque à la question 7 est-elle vraie ?

Corrigé

1. Le premier est non primitif car égal à $(abaa)^2$. Le deuxième est primitif car sa taille est un nombre premier.
2. Pour u de taille n, il s'agit de vérifier, pour chaque préfixe v de taille $\leqslant \frac{n}{2}$, si $u = v^{|u|/|v|}$. Vérifier si u est de cette forme se fait en complexité $\mathcal{O}(n)$, d'où un algorithme de complexité $\mathcal{O}(n^2)$ en temps et $\mathcal{O}(1)$ en espace.
3. Le langage aa^+ convient.
4. Le langage a^*b convient.
5. Le langage a^* convient : le mot b est un contre-exemple. Pour la deuxième partie de la question, Σ^*a convient (car l'alphabet est de taille au moins 2).
6. Dans un premier temps, on supprime tous les états non accessibles, puis on complète l'automate (en rajoutant éventuellement un état puits). Dès lors, il suffit de vérifier que tous les états sont co-accessibles. Ces vérifications se font en temps et espace linéaires (il s'agit d'un parcours de graphe, en partant de l'état initial pour trouver les états accessibles, en partant des états finaux pour trouver les états co-accessibles).

   Pour la preuve, s'il existe un état accessible et non co-accessible q, alors il existe $u \in \Sigma^*$ tel que $\delta^*(q_0, u) = q$ et $\forall v \in \Sigma^*$, $\delta^*(q, v) \notin F$. Cela signifie bien que $L(A)$ est non co-accessible. La réciproque se fait de la même manière.
7. Soit A un automate fini à n états dont $L(A)$ est continuable. On peut supposer que tous les états de A sont accessibles et co-accessibles par la question précédente. Dès lors, $\forall u \in \Sigma^*, \exists v \in \Sigma^*$ tel que $|v| < n$

et $uv \in L(A)$. On pose alors, pour $i \in \mathbb{N}$, $i \geqslant n-1$, $u_i = ab^i$. On pose de plus v_i le plus petit mot tel que $w_i = u_i v_i \in L(A)$. On a $|v_i| < n$, donc w_i est un mot primitif. En effet, si $w_i = x^p$ avec $p \geqslant 2$, alors x (la première occurrence) commence par un a (car u_i commence par un a), mais comme $|u_i| \geqslant n > |v_i|$, on a $|x| < |u_i|$, et donc x (la deuxième occurrence) commence par un b. C'est absurde, donc w_i est un mot primitif. Il existe bien une infinité de tels mots (car $1 + i + n > |w_i| > i$).

8. Avec la construction de la preuve précédente, $2n-1$ semble être une borne supérieur, où n est le nombre d'états de l'automate.
9. La réciproque est fausse, par exemple avec a^*b (tous les mots sont primitifs, mais il n'est pas continuable).

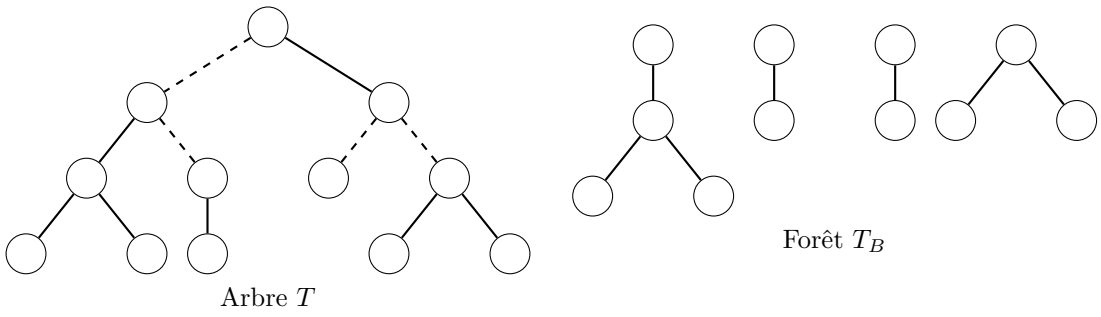
Exercice 6: Comptage de sous-arbres

Définition

On appelle **forêt** un ensemble d'arbres binaires. Si T est un arbre binaire dont A sont les arêtes, pour $B \subseteq A$, on note T_B la forêt obtenue en conservant les arêtes de B, et en ne gardant que les sous-arbres de taille $\geqslant 2$.

Exemple

La figure suivante représente à gauche un arbre T et un ensemble d'arêtes B en traits pleins et à droite la forêt T_B.



On s'intéresse dans ce problème, étant donné un arbre T d'arêtes A et une certaine condition Φ, à compter le nombre de sous-ensembles d'arêtes $B \subseteq A$ tels que la forêt T_B satisfasse la condition Φ.

1. Pour l'arbre T et la forêt T_B donnés en exemple, combien de sous-ensembles d'arêtes B' existe-t-il tels que $T_{B'} = T_B$ (à réétiquetage près) ? Combien de sous-ensembles d'arêtes B' existe-t-il tels que T_B consiste exactement en deux arêtes isolées ?
2. Proposer un algorithme naïf pour déterminer, étant donné un arbre T et un entier $h \in \mathbb{N}^*$, le nombre de sous-ensemble d'arêtes B tels que T_B contienne un arbre de hauteur au moins h. Donner sa complexité en fonction de T et de h.
3. Proposer un algorithme plus efficace en utilisant de la programmation dynamique. Discuter de la complexité en temps et en espace.
4. Comment compter le nombre de sous-ensembles B tels que T_B contienne un arbre de hauteur **exactement** h ?

Définition

On définit le **diamètre** d'une forêt F comme le plus grand entier $d \in \mathbb{N}^*$ tel qu'il existe un chemin simple non orienté de longueur d dans F.

5. Proposer un algorithme qui, étant donné T et d, calcule le nombre de sous-ensembles B tels que T_B soit de diamètre au moins d.

Définition

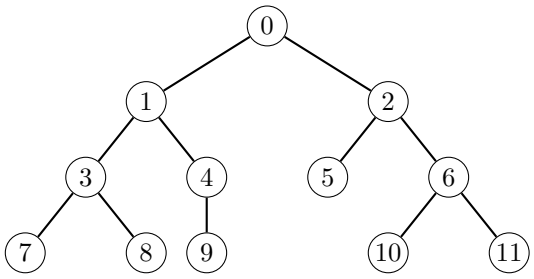
Pour un alphabet Σ, on appelle **Σ-arbre** un arbre dont chaque arête est étiquetée par un élément de Σ. Étant donné un Σ-arbre T et un sous-ensemble d'arêtes B, on définit T_B comme précédemment ; il s'agit d'une forêt de Σ-arbres.

Étant donné un mot $u \in \Sigma^*$, on dit qu'une forêt F de Σ-arbres **contient** u s'il existe un chemin simple non-orienté dans F dont la séquence d'étiquette est égale à u.

6. Proposer un algorithme qui, étant donné un Σ-arbre T et un mot $u \in \Sigma^*$, calcule le nombre de sous-ensembles B tels que T_B contienne u.
7. Proposer un algorithme qui, étant donné un Σ-arbre T et un langage rationnel L (donné sous la forme d'un automate), calcule le nombre de sous-ensembles B tels que T_B contienne un mot de L.
8. On ne fait plus à présent l'hypothèse que les arbres sont binaires. Comment adapter les algorithmes précédents dans ce cas ? Comment la complexité évolue-t-elle en fonction du degré maximal des arbres ?

Corrigé

1. Donnons des noms aux nœuds pour l'explication :



Le premier sous-arbre à obtenir dans T_B apparaît quatre fois dans T : $(1,3,7,8)$, $(0,1,3,4)$, $(0,2,5,6)$ et $(2,6,10,11)$. Distinguons :

- si on garde $(1,3,7,8)$, alors le dernier sous-arbre ne peut être que $(2,5,6)$ ou $(6,10,11)$. Dans le premier cas, on ne peut pas former les deux arbres de taille 2 ; dans le deuxième cas, on obtient la bonne forêt soit en conservant $(0,2)$ et $(4,9)$ (c'est l'ensemble B), soit en conservant $(2,5)$ et $(4,9)$;
- si on garde $(0,1,3,4)$, alors le dernier sous-arbre ne peut être que $(2,5,6)$ ou $(6,10,11)$. Dans les deux cas, on ne peut pas former les deux arbres de taille 2. Ce cas est donc à éliminer ;
- si on garde $(0,2,5,6)$, alors le dernier sous-arbre ne peut être que $(1,3,4)$ ou $(3,7,8)$. Dans les deux cas, on ne peut pas former les deux arbres de taille 2 ;
- si on garde $(2,6,10,11)$, alors le dernier sous-arbre ne peut être que $(1,3,4)$ ou $(3,7,8)$. Dans le premier cas, on ne peut pas former les deux arbres de taille 2. Dans le deuxième cas, on obtient la bonne forêt en conservant $(0,1)$ et $(4,9)$.

Il y a donc trois sous-ensembles B' (dont B) tels que $T_{B'} = T_B$.

Pour la deuxième partie de la question, il s'agit de dénombrer le nombre de façons de choisir deux arêtes qui n'ont pas de sommet en commun. Il y a 11 arêtes au total, et 14 paires d'arêtes adjacentes, soit $\binom{11}{2} - 14 = 41$ possibilités.

2. Vu qu'on parle d'algorithme naïf, on pense ici à parcourir tous les sous-ensembles d'arêtes et à calculer la hauteur maximale d'un sous-arbre pour chacun, et ne garder que ceux pour lesquels la hauteur dépasse h. On obtiendrait une complexité en $\mathcal{O}(n2^n)$, où $n = |T|$.
3. Pour chaque nœud x, notons $T(x)$ le sous-arbre de T enraciné en x. Pour chaque entier $k \in [\![0, h-1]\!]$, définissons :
   - $\varphi(x)$ comme le nombre de sous-ensembles d'arêtes de $T(x)$ ayant un sous-arbre de hauteur supérieure ou égale à h ;
   - $\psi(x,k)$ comme le nombre de sous-ensembles d'arêtes de $T(x)$ ayant un sous-arbre enraciné en x de hauteur exactement k, et aucun sous-arbre de hauteur $\geqslant h$;
   - $\sigma(x,k)$ comme le nombre de sous-ensembles d'arêtes de $T(x)$ ayant un sous-arbre enraciné en x de hauteur inférieure ou égale à k, et aucun sous-arbre de hauteur $\geqslant h$.

On remarque que $\sigma(x,k) = \sum_{i=0}^{k} \psi(x,k)$, et que $\varphi(x) + \sigma(x,h-1) = 2^{|T(x)|-1}$ (car $T(x)$ possède $T(x)-1$ arêtes).

Remarquons de plus que pour un nœud $x = N(g,d)$, on a :

- $\varphi(x)$ est la somme de :
  - $4(\varphi(g)\times\varphi(d) + \varphi(g)\times\sigma(d,h-1) + \sigma(g,h-1)\times\varphi(d))$ (tous les cas où l'un des deux enfants contient un sous-arbre de hauteur $\geqslant h$, fois 4, ce qui correspond au nombre de choix possibles parmi les arêtes (x,g) et (x,d)) ;
  - $\psi(g,h-1)\times\sigma(d,h-1) + \sigma(g,h-1)\times\psi(d,h-1)$ (on ne garde qu'une seule arête parmi (x,g) et (x,d), qui forme un arbre enraciné en x de hauteur exactement h) ;
  - $\psi(g,h-1)\times\sigma(d,h-2) + \sigma(g,h-2)\times\psi(d,h-1) + \psi(g,h-1)\times\psi(d,h-1)$ (on garde les deux arêtes (x,g) et (x,d), qui forment un arbre enraciné en x de hauteur exactement h) ;
- $\psi(x,0) = \sigma(g,h-1)\times\sigma(d,h-1)$ (on ne garde aucune arête parmi (x,g) et (x,d), les deux enfants n'ont pas de sous-arbre de hauteur $\geqslant h$) ;
- pour $k>0$, $\psi(x,k)$ est la somme de :
  - $\psi(g,k-1)\times\sigma(d,k-1) + \sigma(g,k-1)\times\psi(d,k-1)$ (on ne garde qu'une seule arête parmi (x,g) et (x,d), qui forme un arbre enraciné en x de hauteur exactement k) ;
  - $\psi(g,k-1)\times\sigma(d,k-2) + \sigma(g,k-2)\times\psi(d,k-1) + \psi(g,k-1)\times\psi(d,k-1)$ (on garde les deux arêtes (x,g) et (x,d), qui forment un arbre enraciné en x de hauteur exactement k) ;

Cela donne alors l'idée d'un algorithme de programmation dynamique. Il y a de l'ordre de $\Theta(|T|\times h)$ valeurs à calculer au maximum, et chaque valeur prend un temps $\mathcal{O}(h)$ à calculer. On obtient une complexité temporelle en $\mathcal{O}(|T|h^2)$ et spatiale en $\mathcal{O}(|T|h)$. À noter, on n'a pas pris en compte la complexité des opérations arithmétiques, qui n'est pas négligeable dans le cas général, sachant que les valeurs à calculer sont de l'ordre de $2^{|T|}$ au maximum. Cela rajouterait un facteur $|T|$ dans la complexité si on considérait des entiers de taille non bornée (si $|T|>64$, par exemple).

4. Il suffit de faire deux appels à l'algorithme précédent, avec h et $h+1$, et de faire la différence entre les deux résultats obtenus. La complexité est la même.
5. On peut appliquer le même type d'algorithme que précédemment, en remarquant que :
   - si on ne garde ni (x,g), ni (x,d), alors le diamètre maximal est celui dans l'un des enfants ;
   - si on ne garde que (x,g) (resp. (x,d)), alors le diamètre maximal est le max entre le diamètre maximal dans g, dans d et 1 + la hauteur de l'arbre enraciné en g (resp. d) ;
   - si on garde (x,g) et (x,d), le diamètre maximal est le max entre le diamètre maximal dans g, dans d et 2 + la somme des hauteurs des arbres enracinés en g et d.

   On obtiendrait la même complexité.
6. Même idée que précédemment, en gardant en mémoire l'existence de suffixes et préfixes de u qui apparaissent dans un sous arbre en terminant/commençant à la racine.
7. Même idée que précédemment, en gardant en mémoire les états accessibles et co-accessibles en lisant un mot de bas en haut ou de haut en bas (selon le type d'état).
8. On peut adapter les algorithmes précédents, mais les formules nécessite d'envisager, pour un degré d, les 2^d cas possibles d'arêtes conservées entre x et ses d enfants.