A1 – Évaluation accélérée d'automates

On suppose donnés un alphabet Σ, un automate fini déterministe complet A sur Σ, et un mot $w = a_0 \cdots a_{n-1}$ de Σ^*. Pour $0 \leq i < j \leq n$, on notera $w[i, j]$ le facteur $a_i a_{i+1} \cdots a_{j-1}$ de w.

L'objet du problème est de répondre efficacement à des *requêtes* de la forme suivante : étant donnés deux indices i et j avec $0 \leq i < j \leq n$, déterminer si $w[i, j]$ est accepté par A.

Question 0. Donner le pseudo-code d'un algorithme qui, étant donné une requête, y répond. Préciser sa complexité en temps et en espace.

Dans la suite de ce problème, on va chercher à concevoir, étant donnés A et w, une structure de données appelée *index* qui permette de répondre plus efficacement aux requêtes.

Question 1. Expliquer comment calculer à partir de A et w un index de taille $O(n^2)$ qui permette de répondre à toute requête en temps $O(1)$.

Question 2. Quelle est la complexité en temps de calculer l'index de la question 1 ?

Question 3. Notons Q l'ensemble d'états de l'automate A, et $\delta : Q \times \Sigma \to Q$ sa fonction de transition. On étend δ à une fonction $\delta^* : Q \times \Sigma^* \to Q$ par récurrence en posant $\delta^*(q, \epsilon) := q$ et $\delta^*(q, au) := \delta^*(\delta(q, a), u)$. On appelle *effet de transition* du mot $u \in \Sigma^*$ la fonction $f_u : Q \to Q$ définie par $f_u(q) := \delta^*(q, u)$ pour tout $q \in Q$.

Décrire l'effet de transition du mot vide ϵ. Pour tous mots $u, v \in \Sigma^*$, exprimer f_{uv} en fonction de f_u et de f_v.

Question 4. On supposera pour simplifier que la longueur $|w|$ de w est une puissance de 2. On dit que deux nombres $i < j$ sont des *multiples consécutifs d'une même puissance de 2* si $j - i$ est une puissance de 2 qui divise i et j.

En s'inspirant de la question 3, proposer un algorithme de type "diviser pour régner" qui calcule f_u pour tous les facteurs $u = w[i, j]$ de w où i et j sont des multiples consécutifs d'une même puissance de 2. Analyser la complexité en temps de ce calcul.

Question 5. Présenter un algorithme qui utilise l'index de la question 4 pour répondre à des requêtes, et préciser sa complexité en temps. Conclure.

Suite des questions

Question 6. On cherche à présent à supporter, en plus des requêtes présentées dans le sujet, des *mises à jour* dont l'effet est le suivant : étant donné $1 \leq i \leq n$ et $a \in \Sigma$, modifier la i-ème lettre de w pour la remplacer par a.

Expliquer comment la méthode de la question 5 peut être modifiée pour gérer des mises à jour. Quelle est la complexité en temps de l'opération de mise à jour ?

Question 7. Pourrait-on supporter des opérations d'insertion ou de suppression de lettres ? Quelles seraient les difficultés ?

Corrigé

Question 0. Sans index, on évalue simplement l'automate sur le facteur $w[i, j]$. Si on suppose que w est stocké dans un tableau, on accède en temps constant aux cases successives et on simule l'évaluation de l'automate. La complexité en temps est en $O(j - i)$, ou $O(n)$. L'espace nécessaire est celui pour stocker un état de l'automate, c'est-à-dire une complexité en espace constante si on peut stocker un état dans un mot mémoire.

Question 1. On peut calculer un index qui stocke, pour chaque paire $0 \leq i < j \leq n$, le résultat de la requête correspondante, i.e., un booléen indiquant si $w[i, j]$ est accepté ou non par l'automate. La taille mémoire de cet index est bien en $O(n^2)$. En stockant cet index dans un tableau à double entrée, on peut accéder à la case correspondant à (i, j) en temps constant, soit répondre aux requêtes en temps constant.

Question 2. Un algorithme naïf consiste à considérer toutes les paires $0 \leq i < j \leq n$ et à simuler, pour chacune d'entre elles, l'évaluation de l'automate. La complexité de cet algorithme est en $O(n^3)$.

On peut faire mieux en énumérant tous les choix de $0 \leq i \leq n$, et pour chaque i, simuler l'évolution de l'automate à partir de la i-ème lettre de w jusqu'à la dernière lettre, et calculer ainsi la valeur de $w[i, j]$ pour j de i à n. La complexité est en $O(n^2)$. Bien sûr, on ne peut pas espérer faire mieux que $O(n^2)$ parce que c'est la taille de la structure de données à calculer.

Question 3. Pour tout $q \in Q$, on a par définition $f_\epsilon(q) = \delta^*(q, \epsilon) = q$, donc f_ϵ est la fonction identité.

Pour tous $u, v \in \Sigma^*$, pour tout $q \in Q$, on a $f_{uv}(q) = \delta^*(q, uv) = \delta^*(\delta^*(q, u), v) = f_v(f_u(q))$, ainsi on a $f_{uv} = f_v \circ f_u$.

Question 4. Écrivons $n = 2^k$. On va présenter un algorithme qui calcule, pour e de 0 à k inclus, la fonction f_u pour tous les facteurs $u = w[i, j]$ de w où i et j sont des multiples consécutifs de 2^e, c'est-à-dire que i est un multiple de 2^e et $j = i + 2^e$.

Le cas de base est celui de $e = 0$, auquel cas i et j sont des entiers consécutifs. Le mot $w[i, j]$ est alors un singleton, et $f_{w[i,j]}$ est la fonction qui à q associe $\delta(q, a_i)$; on peut le calculer en temps $O(|Q|)$.

Pour $e > 0$, on considère les 2^{k-e} valeurs de i entre 0 inclus et n exclu, c'est-à-dire $i = p2^e$ pour $0 \leq p < 2^{k-e}$, et les valeurs de j correspondantes. On a alors $w[i, j] = w[i, i + 2^{e-1}]w[i + 2^{e-1}, j]$, et il est clair que $i, i + 2^{e-1}$, et $i + 2^{e-1}, j$ sont deux paires de multiples consécutifs de 2^{e-1}. En effet, pour chacune de ces paires, leur différence vaut bien 2^{e-1}, et les deux extrémités sont divisibles par 2^{e-1} car c'est le cas de i et j (qui sont divisibles par 2^e) et c'est donc le cas de $i + 2^{e-1}$. Ainsi, on peut calculer $f_{w[i,j]} = f_{w[i+2^{e-1},j]} \circ f_{w[i,i+2^{e-1}]}$ par la question 3, ces deux valeurs ayant été calculées lorsqu'on a considéré la valeur précédente de e. Chacun de ces calculs s'effectue en temps $O(|Q|)$: pour chaque $q \in Q$, on calcule son image par $f_{w[i,j]}$ en calculant son image par la deuxième fonction, puis par la première.

Il est clair que cet algorithme effectue un nombre linéaire d'opérations de complexité $O(|Q|)$, donc sa complexité est en $O(n \times |Q|)$.

Question 5. Étant donné une requête $0 \leq i < j \leq n$, on va chercher à calculer la fonction $f_{w[i,j]}$. Cette fonction permet en particulier de déterminer en temps constant si $w[i,j]$ est accepté par l'automate, puisqu'il suffit de vérifier si $f_{w[i,j]}(q_0) \in F$, pour q_0 l'état initial et F l'ensemble des états finals.

On dit qu'une paire $0 \leq i < j \leq n$ est *alignée à gauche* si i est un multiple de $2^{\lfloor \log_2(j-i) \rfloor}$, et qu'elle est *alignée à droite* si j satisfait cette condition. On explique d'abord comment traiter le cas d'une requête dont la paire est alignée à gauche. Si $j = i + 1$, on renvoie $f_{w[i,i+1]}$ qui a été calculé comme cas de base de la question 4. Sinon, soit $e := \lfloor \log_2(j-i) \rfloor > 0$. On sait que i est un multiple de 2^e, par définition de e on a $j > i + 2^{e-1}$. Mais il est clair que $i, i + 2^{e-1}$ sont des multiples consécutifs de 2^{e-1}, et il est clair également que $i + 2^{e-1}, j$ est une requête dont la paire est alignée à gauche, car $j - (i + 2^{e-1}) = j - i - 2^{e-1} \leq 2^e - 2^{e-1} \leq 2^{e-1}$ et $i + 2^{e-1}$ est un multiple de 2^{e-1}. Ainsi, on peut calculer $f_{w[i,j]}$ comme la composition de $f_{w[i,i+2^{e-1}]}$ (pré-calculée à la question précédente) et de $f_{w[i+2^{e-1},j]}$ (calculée par un appel récursif, cette récursion étant bien fondée car l'entier $\lfloor \log_2(j-i) \rfloor$ a décru strictement). L'algorithme récursif ainsi décrit permet ainsi de répondre à une requête dont la paire est alignée à gauche en k opérations, chacune de coût $|Q|$ (si l'on suppose que le calcul du $\log_2$ s'effectue en temps constant), donc son coût est $O(|Q| \times \log n)$.

Il est clair qu'on peut traiter de manière symétrique le cas d'une requête alignée à droite. Pour traiter les requêtes générales, il ne reste plus qu'à observer que l'intervalle $[i, j[$ décrit par une paire quelconque $0 \leq i < j \leq n$ peut s'écrire comme l'union de l'intervalle d'une paire alignée à droite, de celui d'une paire alignée à gauche, et de celui d'une paire de multiples consécutifs d'une même puissance de 2 (avec la possibilité que certains de ces intervalles soient en fait vides). En effet, fixons i et j et posons $e := 2^{\lfloor \log_2(j-i) \rfloor} > 0$. On sait donc que $2^{e-1} < j - i \leq 2^e$, donc l'intervalle $[i, j[$ de $\mathbb{N}$ contient un ou deux multiples de 2^{e-1}. S'il n'y en a qu'un, soit m ce multiple, on a $i \leq m \leq j$, et $m - i < 2^{e-1}$ et $j - m < 2^{e-1}$ sinon l'intervalle correspondant contiendrait un autre multiple de 2^{e-1}. Il est alors clair que la paire i, m est alignée à droite (ou bien $i = m$), et que la paire m, j est alignée à gauche (ou bien $j = m$), donc on peut résoudre la requête en calculant la composition du résultat pour ces paires (avec la fonction identité pour les paires vides). S'il y a deux multiples $i \leq m_1 < m_2 \leq j$ de 2^{e-1}, on observe à nouveau que i, m_1 est aligné à droite ou vide, m_2, j est aligné à gauche ou vide, et m_1, m_2 est une paire de multiples consécutifs de 2^{e-1}, donc on conclut comme précédemment en utilisant l'index pour la paire m_1, m_2. Là encore, on pourra supposer que les calculs arithmétiques du logarithme et des multiples s'effectuent en temps constant.

Ainsi, la complexité de traitement d'une requête arbitraire est en $O(|Q| \times \log n)$. On note qu'on peut éliminer le facteur $|Q|$ de la complexité si on remplace toutes les routines qui calculent explicitement la fonction $f_{w[i,j]}$ par une routine qui prend comme argument supplémentaire un état et qui renvoie l'image de cet état par $f_{w[i,j]}$; et on remplace la composition explicite de fonctions par la composition des appels récursifs. On aboutit ainsi à une complexité de $O(\log n)$.

En conclusion, la structure de données décrite aux questions 4–5 (pré-calcul en temps linéaire, traitement de chaque requête en temps logarithmique) réalise donc un bon compromis entre le fait de ne pas avoir d'index (pas de pré-calcul, traitement de chaque requête en temps linéaire) et l'index naïf des questions 1–2 (pré-calcul quadratique, traitement de chaque requête en temps constant).

Question 6. La manière naïve d'appliquer une mise à jour est de recalculer complètement la structure d'index, on paie alors à nouveau le coût linéaire du pré-calcul.

Une meilleure solution consiste à observer que, dans l'algorithme de pré-calcul de la question 4, pour chaque valeur de e, il y a une unique paire i, j de multiples consécutifs de 2^e pour laquelle le mot $w[i,j]$ a été modifié (et pour laquelle il faut donc recalculer $f_{w[i,j]}$ dans l'index). On peut ainsi relancer l'algorithme de pré-calcul mais pour chaque valeur de e on ne traite que cette paire. Le résultat est correct (de la même manière que l'algorithme de la question 4 est correct), et la complexité est en

$O(|Q| \times \log n)$.

L'algorithme de la question 5 est inchangé. On aboutit donc aux complexités suivantes en fonction du mot w : pré-calcul linéaire, réponse aux requêtes en temps logarithmique, support des mises à jour en temps logarithmique.

Question 7. Une première difficulté posée par l'insertion et la suppression de lettres est qu'on ne peut plus supposer que la longueur du mot est une puissance de deux, mais il ne s'agit pas d'une différence essentielle.

La principale difficulté est que les insertions et suppressions décalent les positions dans le mot, de sorte que les effets de transition pré calculés ne sont plus alignés avec des puissances de 2. On peut voir de manière plus générale la structure d'index de la question 4 comme un arbre binaire (qui, dans le cas de la question 4, est complet) ; et l'algorithme de la question 5 comme la décomposition d'un intervalle de feuilles de cet arbre comme l'union de l'ensemble des feuilles accessibles depuis au plus $2h + 1$ nœuds de l'arbre, où h est la hauteur de l'arbre. Autrement dit, la question 4 calcule un index qui est un arbre binaire à n feuilles où la i-ème feuille stocke $f_{w[i,i+1]}$ et où tout nœud interne stocke la composition des fonctions stockées par ses enfants : et la question 5 décompose un intervalle I de feuilles de l'arbre binaire en recherchant leur plus grand ancêtre commun et en déterminant un ensemble d'au plus $2h + 1$ descendants incomparables de cet ancêtre commun tel que I est exactement l'union, pour chaque nœud n de cet ensemble, des feuilles accessibles à partir de n.

Lors d'une insertion ou d'une suppression, on peut mettre à jour l'arbre d'index en insérant ou en supprimant la feuille concernée, et en mettant à jour ses ancêtres : la complexité en fonction du mot est en $O(h)$ où h est la hauteur courante de l'arbre d'index. De même, l'algorithme de la question 5 s'exécute en $O(h)$.

La véritable difficulté pour gérer les insertions et suppressions est donc de garantir que l'arbre d'index reste équilibré (de hauteur logarithmique), ce qui n'est pas le cas en général. On pourrait montrer que l'aspect équilibré de l'arbre peut être maintenu en effectuant des rotations lorsque l'arbre est modifié, comme pour les arbres AVL, ce qui permet ainsi après un pré-calcul linéaire de gérer les insertions, suppressions, substitutions en temps logarithmique et de répondre aux requêtes en temps logarithmique. Un résultat plus général est montré dans [Nie18].

Références

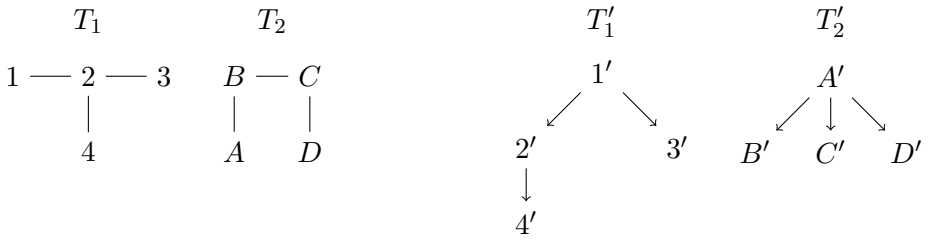
[Nie18] Matthias Niewerth. MSO queries on trees : Enumerating answers under updates using forest algebras. In *Proc. LICS*, 2018.

A2 – Homomorphismes d'arbres

On rappelle qu'un *arbre* est un graphe non-orienté connexe acyclique. Un *arbre enraciné* T est obtenu en prenant un arbre, en distinguant un sommet de l'arbre appelé *racine*, et en orientant les arêtes de sorte que la racine n'ait aucune arête entrante et chaque sommet ait au plus une arête entrante.

Un *homomorphisme d'arbres* d'un arbre T_1 vers un arbre T_2 est une application h de l'ensemble des nœuds de T_1 vers celui des nœuds de T_2 qui satisfait la condition suivante : pour toute arête $\{u, v\}$ de T_1, la paire $\{h(u), h(v)\}$ est une arête de T_2. Un *homomorphisme d'arbres enracinés* h d'un arbre enraciné T_1 vers un arbre enraciné T_2 est une application h de l'ensemble des nœuds de T_1 vers celui des nœuds de T_2 telle que l'image par h de la racine de T_1 est la racine de T_2, et telle que, pour toute arête (u, v) de T_1, le couple $(h(u), h(v))$ est une arête de T_2.

Question 0. Construire un homomorphisme de l'arbre T_1 vers l'arbre T_2. Y a-t-il un homomorphisme de l'arbre enraciné T_1' vers l'arbre enraciné T_2' ?



Question 1. Proposer un algorithme qui, étant donné un arbre T_1, construit un homomorphisme de T_1 vers l'arbre comportant deux sommets et une unique arête joignant ces deux sommets. En déduire un algorithme qui, étant donné deux arbres T_1 et T_2, détermine s'il existe un homomorphisme de T_1 vers T_2, et en calcule un le cas échéant.

Question 2. Proposer un algorithme qui, étant donné deux arbres enracinés T_1 et T_2, détermine s'il existe un homomorphisme de T_1 vers T_2, et en calcule un le cas échéant.

Question 3. On fixe un ensemble de couleurs C. Un *graphe C-colorié* est un graphe G et une application qui associe à chaque arête $\{u, v\}$ de G une couleur de C. Un *graphe C-colorié orienté* est défini de la même manière mais les arêtes sont des couples (u, v). On étend ces définitions pour parler d'arbres et arbres enracinés C-coloriés. On définit un homomorphisme h d'arbres C-coloriés d'un arbre T_1 à un arbre T_2 en imposant également que la couleur de toute arête $\{u, v\}$ de T_1 soit la même que celle de son image par h dans T_2, et on définit de même la notion d'homomorphisme d'arbres enracinés C-coloriés.

Proposer un algorithme qui, étant donné deux arbres enracinés C-coloriés T_1 et T_2, détermine s'il existe un homomorphisme de T_1 vers T_2, et en calcule un le cas échéant.

Question 4. Proposer un algorithme qui, étant donné deux arbres C-coloriés T_1 et T_2, détermine s'il existe un homomorphisme de T_1 vers T_2, et en calcule un le cas échéant.

Corrigé

Question 0. On peut définir un homomorphisme de T_1 dans T_2 par :

- $h(1) := A$
- $h(2) := B$
- $h(3) := C$
- $h(4) := C$

Il n'existe pas d'homomorphisme de T_1' dans T_2'. En effet, supposons par l'absurde qu'il existe un tel homomorphisme h, on sait que $h(1') = A'$, de sorte que $h(2')$ doit être l'un de $\{B', C', D'\}$, mais $h(3')$ doit alors être un enfant de l'un de ces sommets, ce qui n'est pas possible.

Question 1. Cette question revient à bicolorier l'arbre d'entrée T_1, ce que l'on peut faire en temps et occupation mémoire linéaires par un parcours en profondeur à partir d'un sommet arbitraire de T_1. Ainsi, étant donné deux arbres T_1 et T_2 :

- Si T_1 est vide alors l'homomorphisme trivial convient
- Sinon, si T_2 est vide alors il n'y a pas d'homomorphisme
- Sinon, si T_1 ne contient aucune arête alors on prend un homomorphisme envoyant chaque sommet de T_1 vers un sommet quelconque de T_2
- Sinon, si T_2 ne contient aucune arête alors il n'y a pas d'homomorphisme
- Sinon, on utilise l'algorithme mentionné précédemment pour calculer un homomorphisme vers une arête quelconque de T_2, ce qui conclut.

Question 2. On montre d'abord que T_1 a un homomorphisme vers T_2 si et seulement si la hauteur de T_1 est inférieure ou égale à la hauteur de T_2. En effet, si la hauteur de T_1 est strictement plus grande que celle de T_2, en considérant un chemin orienté dans T_1 de longueur maximale (c'est-à-dire un chemin de la racine à une feuille), on voit que son image dans T_2 devrait être un chemin orienté de la même longueur, ce qui n'est pas possible.

À l'inverse, si la hauteur de T_1 est plus petite ou égale à celle de T_2, on peut construire un homomorphisme en choisissant un chemin orienté de longueur maximale dans T_2 et en envoyant chaque sommet T_1 à une distance i de la racine vers le i-ème sommet du chemin orienté dans T_2 en partant de la racine.

Ainsi, on peut résoudre le problème en temps et en occupation mémoire linéaires en calculant récursivement la hauteur de chaque sommet de T_1 et de T_2, en comparant ces grandeurs, et si un homomorphisme existe on construit un chemin orienté quelconque de longueur maximale dans T_2 et en calculant l'image de chaque sommet de T_1 comme on l'a expliqué.

Question 3. On calcule, pour chaque paire de nœuds n_1 de T_1 et n_2 de T_2, l'information booléenne $f(n_1, n_2)$ indiquant s'il y a un homomorphisme du sous-arbre de T_1 enraciné en n_1 vers T_2 qui envoie n_1 sur n_2. On initialise d'abord $f(n_1, n_2)$ à VRAI pour toutes les feuilles n_1 de T_1 et tous les nœuds n_2 de T_2. Ensuite, on parcourt T_1 de bas en haut. Pour calculer $f(n_1, n_2)$ pour un nœud n_1 quelconque de T_1 et un nœud quelconque n_2 de T_2, on passe en revue chaque enfant u_1 de n_1 relié à n_1 par une arête de couleur $c \in C$. Si, pour l'une des valeurs de u_1, le nœud n_2 n'a pas d'arête sortante colorée par c vers un nœud u_2 tel que $f(u_1, u_2)$ soit VRAI, alors on stocke que $f(n_1, n_2)$ est FAUX. À l'inverse, si ces tests ont réussi pour chaque choix de u_1, on stocke que $f(n_1, n_2)$ est VRAI.

L'algorithme général renvoie VRAI si $f(r_1, r_2)$ est VRAI pour r_1, r_2 les racines respectives de T_1 et de T_2. La correction de cet algorithme est facilement établie par récurrence sur les sous-arbres de T_1. Sa complexité mémoire est en $O(|T_1| \times |T_2|)$. Sa complexité en temps de calcul est la même (chaque paire d'arêtes ne sera considérée qu'une seule fois).

Si l'algorithme renvoie VRAI, on peut facilement calculer l'homomorphisme en temps linéaire en procédant de haut en bas : on envoie la racine de T_1 vers celle de T_2, et pour chaque enfant u_1 de la racine de T_1, on trouve un enfant u_2 de la racine de T_2 tel que le sous-arbre $T_1^{u_1}$ de T_1 enraciné en u_1 ait un homomorphisme vers T_2 qui envoie u_1 vers u_2 (il existe nécessairement par construction) et on rappelle récursivement le calcul de haut en bas pour calculer un homomorphisme pour le sous-arbre $T_1^{u_1}$.

Question 4. **Indication :** Adapter la question précédente

On enracine l'arbre T_1 en un sommet arbitraire et on applique une variante de l'algorithme de la question 3. La différence principale est que, dans la définition inductive de $f(u_1, u_2)$, pour chaque enfant n_1 de u_1 par une arête coloriée c, on considère maintenant toutes les arêtes incidentes à u_2 coloriées c sans se préoccuper de leur orientation. Par ailleurs, comme il n'y a plus de racine, la condition de succès est que $f(r_1, n_2)$ soit VRAI pour r_1 la racine arbitrairement choisie pour T_1 et pour un nœud quelconque n_2 de T_2. Enfin, on prend soin de mémoïser la fonction récursive afin d'obtenir la bonne complexité.

Attention, l'algorithme consistant à enraciner T_2 n'est pas correct.

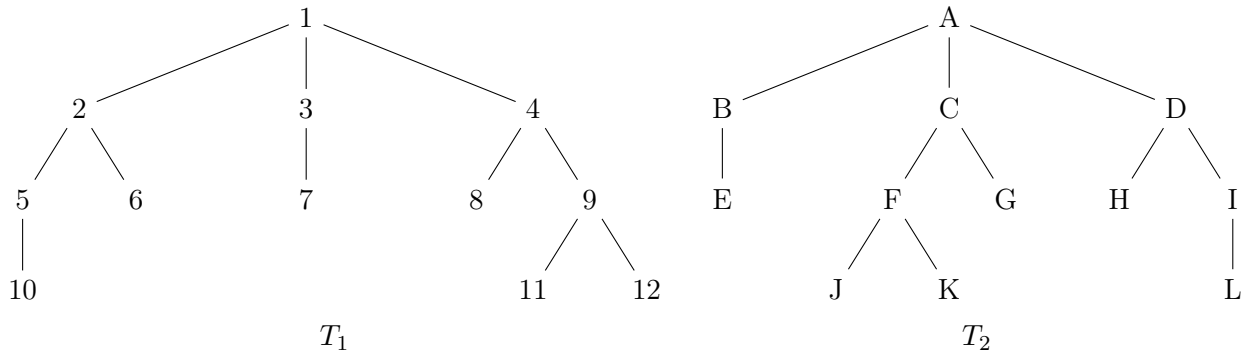
A3 – Isomorphisme d'arbres

Un *isomorphisme* d'un arbre T_1 vers un arbre T_2 est une bijection ι des nœuds de T_1 vers ceux de T_2 qui satisfait les conditions suivantes :

— On a $\iota(r_1) = r_2$, où r_1 et r_2 sont les racines respectives de T_1 et de T_2

— Pour tout nœud n_1 de T_1 qui n'est pas la racine, pour p_1 le parent de n_1, on a que $\iota(n_1)$ est un enfant de $\iota(p_1)$.

Question 0. Donner un isomorphisme de T_1 vers T_2 pour l'exemple suivant. Cet isomorphisme est-il unique ?



Question 1. On définit une relation sur les arbres en disant que T_1 et T_2 sont *isomorphes* s'il existe un isomorphisme de T_1 à T_2. Montrer qu'il s'agit d'une relation d'équivalence.

Question 2. On suppose que les arbres T_1 et T_2 sont *binaires*, c'est-à-dire que chaque nœud interne a exactement deux enfants. Proposer un algorithme qui détermine si T_1 et T_2 sont isomorphes, et calcule un isomorphisme le cas échéant. Préciser sa complexité en temps et en espace.

Question 3. Généraliser cet algorithme au cas où T_1 et T_2 ne sont plus binaires, et discuter de la complexité en temps et en espace.

Question 4. On suppose à nouveau que T_1 et T_2 sont binaires, et on cherche un algorithme plus efficace. Étant donné deux séquences d'entiers naturels s_1 et s_2, on note $\min(s_1, s_2)$ la plus petite de ces séquences dans l'ordre lexicographique, on note $\max(s_1, s_2)$ la plus grande, et on note $s_1 \odot s_2$ la séquence obtenue en concaténant s_1 et s_2. Pour $i \in \mathbb{N}$, on note (i) la séquence d'entiers singleton comportant un seul terme égal à i.

Le *code* $\chi(T)$ d'un arbre binaire T est la séquence d'entiers définie inductivement comme suit :

— Si T est une feuille, alors on a $\chi(T) := (1)$

— Si T n'est pas une feuille, alors soit n la racine de T, soit T' le sous-arbre enraciné au premier enfant de n, soit T'' le sous-arbre enraciné au second enfant de n, soit v le nombre de sommets de T, soit $\kappa' := \chi(T')$, et soit $\kappa'' := \chi(T'')$, alors on a $\chi(T) := (v) \odot \min(\kappa', \kappa'') \odot \max(\kappa', \kappa'')$.

Montrer qu'il existe un isomorphisme de T_1 vers T_2 si et seulement si $\chi(T_1) = \chi(T_2)$.

Suite des questions

Question 5. Toujours sous l'hypothèse que T_1 et T_2 sont binaires, utiliser la question précédente pour proposer un algorithme pour déterminer si T_1 et T_2 sont isomorphes et construire un isomorphisme le cas échéant. Discuter de sa complexité en temps.

Question 6. Généraliser les questions 4 et 5 au cas où les arbres T_1 et T_2 ne sont plus binaires, et discuter de la complexité obtenue.

Question 7. Proposer un algorithme en temps linéaire pour déterminer, étant donnés T_1 et T_2, s'ils sont isomorphes, et calculer le cas échéant un isomorphisme.

Corrigé

Question 0. Oui, on peut définir h comme suit :

- $h(1) := A$
- $h(2) := D$
- $h(3) := B$
- $h(4) := C$
- $h(5) := I$
- $h(6) := H$
- $h(7) := E$
- $h(8) := G$
- $h(9) := F$
- $h(10) := L$
- $h(11) := J$
- $h(12) := K$

Cet isomorphisme n'est pas unique, on peut obtenir un autre isomorphisme h' défini comme h sauf que $h(11) := K$ et $h(12) := L$. Ce sont là les deux seules possibilités.

Question 1. On vérifie les trois propriétés :

Réflexive. La fonction identité est toujours un isomorphisme d'un arbre vers lui-même.

Symétrique. S'il existe un isomorphisme ι d'un arbre T_1 vers un arbre T_2, alors la fonction inverse ι^{-1} est une fonction des nœuds de T_2 vers les nœuds de T_1 qui est toujours une bijection, on a bien $\iota^{-1}(r_2) = \iota^{-1}(\iota(r_1)) = r_1$, et pour tout nœud n_2 de T_2 qui n'est pas la racine, pour p_2 son parent, en notant $n_1 := \iota^{-1}(n_2)$, et p_1 le parent de n_1, on sait par définition que $\iota(n_1)$ est un enfant de $\iota(p_1)$, c'est-à-dire que $\iota(p_1)$ est le parent de $\iota(\iota^{-1}(n_2)) = n_2$. Ainsi, comme n_2 n'a qu'un seul parent, on sait que $\iota(p_1) = p_2$, c'est-à-dire $\iota^{-1}(p_2) = p_1$. Ainsi, on a bien que $\iota^{-1}(n_2)$ est un enfant de $\iota^{-1}(p_2)$, donc la condition d'un isomorphisme d'arbres est vérifiée.

Transitive. S'il existe un isomorphisme ι_{12} de T_1 vers T_2 et un isomorphisme ι_{23} de T_2 vers T_3, alors en notant $\iota_{13} = \iota_{23} \circ \iota_{12}$, on sait que ι_{13} est une bijection des nœuds de T_1 vers les nœuds de T_3. En notant r_1, r_2, r_3 les racines respectives de T_1, T_2, T_3, on sait que $\iota_{13}(r_1) = \iota_{23}(\iota_{12}(r_1)) = \iota_{23}(r_2) = r_3$, donc la condition sur les racines est vérifiée. Enfin, pour tout nœud n_1 de T_1 qui n'est pas la racine, pour p_1 le parent de n_1, on sait par la condition sur ι_{12} que $\iota_{12}(n_1)$ est un enfant de $\iota_{12}(p_1)$, et on sait par la condition sur ι_{23} que $\iota_{23}(\iota_{12}(n_1))$ est une enfant de $\iota_{23}(\iota_{12}(n_1))$, c'est-à-dire que $\iota_{13}(n_1)$ est un enfant de $\iota_{13}(p_1)$, et ainsi ι_{13} est bien un isomorphisme d'arbres.

Question 2. On calcule une fonction récursive $f(n_1, n_2)$ qui indique, étant donné un nœud n_1 de T_1 et un nœud n_2 de T_2, si le sous-arbre de T_1 enraciné en n_1 et le sous-arbre de T_2 enraciné en n_2 sont isomorphes :

- Si n_1 et n_2 sont tous deux des feuilles, alors $f(n_1, n_2)$ est VRAI
- Si l'un de $\{n_1, n_2\}$ est une feuille et l'autre est un nœud interne, alors $f(n_1, n_2)$ est FAUX
- Si n_1 est un nœud interne de T_1 ayant pour enfants n_1' et n_1'', et n_2 est un nœud interne de T_2 ayant pour enfants n_2' et n_2'', alors $f(n_1, n_2)$ est VRAI si et seulement si l'une des conditions suivantes est vraie :
  - $f(n_1', n_2')$ est VRAI et $f(n_1'', n_2'')$ est VRAI
  - $f(n_1', n_2'')$ est VRAI et $f(n_1'', n_2')$ est VRAI

On termine en renvoyant vrai si et seulement si $f(r_1, r_2)$ est VRAI, pour r_1 et r_2 les racines respectives de T_1 et de T_2. Il est clair par récurrence que cet algorithme effectue le bon calcul, en notant que si n_1 et n_2 sont des nœuds internes alors un isomorphisme du sous-arbre de T_1 enraciné en n_1 au sous-arbre de T_2 enraciné en n_2 doit envoyer n_1 vers n_2, et doit être une bijection entre les deux enfants de n_1 et les deux enfants de n_2 de sorte que les sous-arbres enracinés aux paires de sommets en correspondance bijective soient isomorphes.

La complexité en temps et en espace est de $O(|T_1| \times |T_2|)$. Une fois l'exécution terminée, on peut facilement construire un isomorphisme s'il en existe un, avec les mêmes complexités, en parcourant les deux arbres de haut en bas : on envoie r_1 vers r_2, quand n_1 et n_2 sont des feuilles alors on envoie n_1 vers n_2, et quand n_1 et n_2 sont tous deux des nœuds internes alors on définit l'isomorphisme sur les enfants de n_1 et de n_2 suivant la bijection qui justifie que $f(n_1, n_2)$ était VRAI, et on descend récursivement dans les paires de sous-arbres correspondantes.

Question 3. Si les arbres T_1 et T_2 ne sont pas binaires, alors le cas des nœuds internes est modifié. On connaît par hypothèse d'induction un graphe biparti entre les enfants de n_1 et ceux de n_2 avec une arête entre deux enfants s'il a été déterminé par un appel récursif que leurs sous-arbres sont isomorphes. Notre but est de déterminer si ce graphe biparti a un couplage parfait (et si oui le calculer), en temps linéaire en la taille de ce graphe (c'est-à-dire, nombre d'enfants de n_1 fois nombre de sommets de n_2).

On vérifie d'abord s'il n'y a aucun sommet isolé dans ce graphe biparti, c'est-à-dire que chaque sommet a un voisin. Si ce n'est pas le cas, alors il faut clairement répondre FAUX pour n_1 et n_2. Sinon, on procède de la façon suivante : pour chaque sommet n de gauche non encore visité, on le marque comme visité, on compte le nombre q de ses voisins à droite, on prend un voisin quelconque à droite (qui existe par hypothèse), on prend tous ses voisins à gauche (y compris n), on compte leur nombre p, on les marque tous comme visités, et on renvoie FAUX si $p \neq q$. L'idée est que tous les sommets visités au cours de cette phase sont isomorphes à n, grâce à la transitivité de la relation d'isomorphisme (question 1). Ainsi, s'il n'y en a pas le même nombre à gauche et à droite, il faut échouer ; sinon on peut définir un isomorphisme quelconque entre les p sommets de gauche et les q sommets de droite (et le stocker pour la reconstruction ultérieure). Si on peut traiter ainsi tous les sommets de gauche alors on renvoie VRAI, ce qui est correct.

Question 4. On montre d'abord par induction que si deux arbres T_1 et T_2 sont isomorphes alors ils ont le même code. En effet, si T_1 et T_2 sont deux feuilles alors c'est vrai, sinon il s'agit de deux arbres non-singletons qui ont le même nombre de sommets (ainsi la première composante de leur code est la même), et dont les deux sous-arbres descendants sont isomorphes : ils ont donc même code, et l'utilisation de min et max assure ainsi que T_1 et T_2 ont bien le même code.

On montre à présent la réciproque, en montrant que, pour toute séquence d'entiers κ correspondant à un code, on peut construire un arbre binaire T_κ tel que pour tout arbre T avec $\chi(T) = \kappa$, les arbres T et T_κ soient isomorphes. Ceci implique clairement le résultat demandé, puisque on établira ainsi que si T_1 et T_2 ont le même code κ alors ils sont tous deux isomorphes à T_κ, ce qui conclut d'après la question 1.

Pour construire T_κ à partir de κ, on enlève le premier élément de la séquence pour construire la racine de T, et tant que κ n'est pas vide on répète l'opération suivante : on lit le premier élément i (qui est le premier élément du code d'un enfant de la racine de T), puis on lit i entiers (correspondant au reste du code de cet enfant), on construit inductivement l'arbre $T_{\kappa'}$ pour le code κ' ainsi obtenu, et on l'ajoute comme enfant de T. Il est inductivement clair que T est isomorphe à T_κ, puisque chaque enfant n de la racine de T est isomorphe par hypothèse d'induction à l'enfant de la racine de T_κ correspondant au code de n.

[Le résultat de cette question est démontré dans [Val02], Theorem 4.9.]

Question 5. On calcule récursivement le code de T_1 et celui de T_2, et on vérifie s'ils sont égaux. Cet algorithme est correct d'après la question précédente, il reste à analyser sa complexité et à expliquer comment reconstruire l'isomorphisme.

Pour la complexité, le calcul dans T_1 est en complexité en temps $O(|T_1|^2)$, parce qu'à chaque nœud interne on doit passer un temps linéaire dans le pire cas pour comparer les codes des deux enfants suivant l'ordre lexicographique. Ainsi la complexité totale en temps est-elle en $O(|T_1|^2 + |T_2|^2)$, c'est-à-dire la même que celle de la question 2 puisqu'il est manifeste que T_1 et T_2 ne sont pas isomorphes lorsque $|T_1| \neq |T_2|$.

Pour retrouver un isomorphisme, il suffit de stocker, à chaque nœud interne, quel enfant réalisait le max et quel enfant réalisait le min. Ainsi, on peut traiter les deux arbres de haut en bas en appariant les enfants suivant cette information, ce qui permet de retrouver un isomorphisme lorsqu'il existe avec la même complexité.

Question 6. On utilise la définition suivante du code, généralisée à des arbres non nécessairement binaires :

- Pour une forêt $F = T_1, \ldots, T_n$, le code $\chi(F)$ est la concaténation des $\chi(T_1), \ldots, \chi(T_n)$ triés par ordre lexicographique.
- Pour un arbre T ayant une forêt F d'enfants, le code $\chi(T)$ est la concaténation de la séquence singleton ($|T|$) et de $\chi(F)$.

La preuve de la question 3 s'adapte sans difficulté. Pour la question 4, lors du calcul du code dans un arbre T, il faut trier les enfants de chaque nœud interne par code dans l'ordre lexicographique. Il y a au plus $|T|$ enfants, et chaque comparaison de code prend au pire cas $|T|$ opérations (la longueur maximale du code), donc en utilisant un algorithme de tri en temps $N \log N$ on obtient une complexité de $|T| \times |T| \log |T|$ à chaque niveau, soit $|T|^3 \log |T|$ en tout. Pour retrouver l'isomorphisme, il faut mémoriser au cours du tri l'inverse de la permutation de tri, c'est-à-dire se souvenir, pour chaque numéro d'enfant dans le code trié, de l'enfant de l'arbre original auquel il correspondait : on peut utiliser cela pour retrouver l'isomorphisme en traitant l'arbre de haut en bas comme à la question précédente.

On peut faire mieux et conserver la borne de $|T|^2$ en utilisant un tri radix (cf [AHU74] Section 3.2) : on prépare une liste, pour chaque longueur, des chaînes ayant cette longueur, et des symboles qui apparaissent à cette position, puis on considère toutes les positions de poids faible par ordre croissant et on trie les chaînes où cette position est remplie en faisant une liste par valeur possible à cette position et en concaténant les listes. Ceci permet d'assurer que chaque opération de tri s'effectue en temps linéaire en la taille totale des listes, soit une borne totale en $O(|T|^2)$.

[L'algorithme de cette question est présenté dans [Val02] p160, ou [Bon10] Algorithm 1.]

Question 7. On suit l'algorithme de l'exemple 3.2 de [AHU74] p84 ; cf aussi [Bon10], Algorithm 3. L'idée est de traiter les deux arbres simultanément, de considérer les sommets par "niveau" (c'est-à-dire par profondeur), et, une fois que l'on a calculé un code pour les sommets de chaque niveau, de renuméroter les codes pour ne plus perdre de temps à en parcourir le contenu.

Plus formellement, à un niveau donné, on traite les feuilles en leur donnant le code 0. On traite les autres sommets en passant en revue la liste triée des codes des sommets du niveau inférieur (qui sont des

entiers) et on attribue un code temporaire à chaque sommet du niveau courant (une séquence d'entiers) de la façon suivante : à chaque fois qu'on voit un sommet u du niveau précédent, on ajoute son code à celui du parent. Ceci garantit que les codes temporaires du niveau courant dans T_1 et T_2 sont des séquences croissantes d'entiers. On trie les sommets du niveau courant par ces séquences comme à la question 5, on vérifie que ces listes sont les mêmes, et puis on attribue aux sommets du niveau courant leur code en donnant à chaque code temporaire distinct un nouvel entier (à partir de 1). Pour faire cela, on utilise un dictionnaire avec accès en temps constant (par exemple avec une table de hachage) pour vérifier, en lisant chaque étiquette, si cette étiquette a déjà été rencontrée.

Cet algorithme atteint la complexité désirée car, à chaque niveau, le travail total à accomplir est proportionnel au nombre de nœuds à ce niveau (donc le calcul intermédiaire est toujours en temps linéaire mais cette fois ce ne sont que les nœuds du niveau courant qui contribuent).

Pour retrouver l'isomorphisme quand il existe, on peut mémoriser à chaque nœud le code temporaire qui lui a été attribué, et la séquence des enfants dans l'ordre dans lequel ils ont été considérés lors de la construction du code temporaire ; ceci permet de conserver la même complexité en temps.

Références

[AHU74] A. V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Pearson, 1974.

[Bon10] Marthe Bonamy. A small report on graph and tree isomorphism. 2010. `http://perso.ens-lyon.fr/eric.thierry/Graphes2010/marthe-bonamy.pdf`.

[Val02] Gabriel Valiente. *Algorithms on trees and graphs*. Springer, 2002.

A4 – Automates et palindromes

On fixe un alphabet Σ avec $|\Sigma| > 1$. Un mot $w \in \Sigma^*$ est un *palindrome* s'il s'écrit $w = a_1 \cdots a_n$ et qu'on a $a_i = a_{n-i+1}$ pour tout $1 \leq i \leq n$. On note $\Pi \subseteq \Sigma^*$ le langage des palindromes. Pour un automate fini A sur Σ, on note $L(A)$ le langage reconnu par A.

Question 0. Soit $\Pi_n := \Pi \cap \Sigma^n$. Montrer que pour tout automate fini déterministe complet A, pour tout $n \in \mathbb{N}$, si $L(A) \cap \Sigma^{2n} = \Pi_{2n}$, alors A a au moins $|\Sigma|^n$ états.

Question 1. En déduire que le langage Π n'est pas régulier.

Question 2. Étant donné un automate fini A sur Σ, peut-on calculer un automate A_Π qui reconnaisse $L(A) \cap \Pi$?

Question 3. Pour tout mot $u = b_1 \cdots b_m$ de Σ^*, on note $\overline{u} := b_m \cdots b_1$ son miroir. Étant donné A, peut-on calculer un automate A'_Π qui reconnaisse $\{u \in \Sigma^* \mid u\overline{u} \in L(A)\}$?

Question 4. On appelle Π_{pair} l'ensemble des palindromes de longueur paire, i.e., $\Pi_{\text{pair}} := \bigcup_{n\in\mathbb{N}} \Pi_{2n}$. Proposer un algorithme qui, étant donné un automate fini A sur Σ, détermine si $L(A) \cap \Pi_{\text{pair}}$ est vide, fini, ou infini. Discuter de sa complexité en temps et en espace.

Question 5. Modifier l'algorithme de la question 4 pour calculer la cardinalité de $L(A) \cap \Pi_{\text{pair}}$ quand cet ensemble est fini, en faisant l'hypothèse que l'automate d'entrée A est déterministe. Comment la complexité est-elle affectée ?

Question 6. Modifier l'algorithme des questions 4 et 5 pour qu'il s'applique à $L(A) \cap \Pi$.

Corrigé

Question 0. Soit A un tel automate fini déterministe d'ensemble d'états Q, et fixons $n \in \mathbb{N}$. Considérons la fonction $f : \Sigma^n \to Q$ qui associe à $w \in \Sigma^n$ l'état q (unique) auquel A aboutit en lisant w. Montrons que f est injective. Procédons par l'absurde et supposons que $f(u) = f(v)$ pour $u \neq v$ de Σ^n. On sait que $u\overline{u}$ est un palindrome de longueur $2n$ donc il y a un chemin étiqueté par $\overline{u}$ de $f(u)$ à un état final de A. En combinant ce chemin avec le chemin de l'état initial à q étiqueté par v, on conclut que l'automate accepte $v\overline{u}$. Comme $u \neq v$, c'est pourtant un mot de longueur $2n$ qui n'est pas un palindrome, contradiction. Ainsi, f est injective, donc $|Q| \geq |\Sigma^n| \geq |\Sigma|^n$.

Question 1. Procédons par l'absurde et supposons que Π soit régulier. Soit A un automate fini déterministe complet qui reconnaisse Π, et soit n son nombre d'états. Comme $L(A) = \Pi$ par hypothèse, on sait que $L(A) \cap \Sigma^{2n} = \Pi_{2n}$, ainsi d'après la question précédente A a au moins $|\Sigma|^n \geq 2^n$ états, mais il en a n et on a $n < 2^n$, donc contradiction. Ainsi Π n'est pas régulier.

Question 2. C'est manifestement déraisonnable : pour A un automate reconnaissant le langage régulier Σ^*, la question nous demanderait de calculer un automate qui reconnaisse $\Sigma^* \cap \Pi = \Pi$, et on sait d'après la question précédente qu'il n'en existe pas.

Question 3. De façon un peu contre-intuitive, c'est possible. Posons $A = (Q, I, F, \delta)$ où Q est l'ensemble d'états de A, où I est l'ensemble d'états initiaux, où F est l'ensemble d'états finaux, et où $\delta \subseteq Q \times \Sigma \times Q$ est la relation de transition. On construit d'abord en temps linéaire l'automate $\overline{A} := (Q, F, I, \overline{\delta})$ où $\overline{\delta} := \{(q', a, q) \mid (q, a, q') \in \delta\}$. Il est clair que $\overline{A}$ reconnaît $\overline{L(A)} := \{\overline{u} \mid u \in L(A)\}$, puisqu'il y a une correspondance bijective entre les chemins acceptants dans A et dans $\overline{A}$, et l'effet de cette bijection sur l'étiquette des chemins est l'opération miroir.

On construit ensuite (en temps quadratique en A) l'automate produit $A \times \overline{A}$ défini comme suit : $A \times \overline{A} := (Q \times Q, I \times F, F', \delta \times \overline{\delta})$ où la relation de transition est définie comme $\overline{\delta} := \{((q, \overline{q'}), a, (q', \overline{q}) \mid (q, a, q') \in \delta \wedge (\overline{q'}, a, \overline{q}) \in \overline{\delta}$, et où les états finaux sont $\{(q, q) \mid q \in Q\}$. Il est clair que l'automate produit peut atteindre l'état $(q, \overline{q})$ en lisant un mot $w \in \Sigma^*$ si et seulement si l'automate A peut atteindre l'état q en lisant w (dans la première composante) et l'automate $\overline{A}$ peut atteindre l'état $\overline{q}$ en lisant w (dans la seconde composante), ce qui est le cas, par définition de $\overline{A}$, si et seulement s'il y a un chemin dans A étiqueté par $\overline{u}$ de $\overline{q}$ à un état final de F. En particulier, l'automate produit peut atteindre l'état (q, q) en lisant un mot $w \in \Sigma^*$ si et seulement si l'automate A peut atteindre q en lisant w et l'automate A a un chemin étiqueté par $\overline{u}$ de q à un état final. Ainsi, la définition de F' assure que l'automate produit accepte un mot $w \in \Sigma^*$ si et seulement s'il existe un état q tel que A a un chemin acceptant pour $w\overline{w}$ qui passe par un certain état q après la lecture de w, c'est-à-dire si et seulement si A accepte $w\overline{w}$. Ceci établit que la construction est correcte.

[Cette construction est donnée dans [ALR+09], Lemma 2.]

Question 4. La construction de l'automate produit A'_Π de la question précédente s'effectue en temps quadratique en A. Il est clair que $L(A) \cap \Pi_{\text{pair}}$ a la même cardinalité que $L(A'_\Pi)$, puisque la fonction $f : \Sigma^* \to \Pi_{\text{pair}}$ définie par $f(u) := u\overline{u}$ pour tout $u \in \Sigma^*$ définit une bijection entre $L(A'_\Pi)$ et $L(A) \cap \Pi_{\text{pair}}$. Ainsi, il suffit de déterminer si $L(A'_\Pi)$ est vide, fini, ou infini.

On détermine d'abord si $L(A'_\Pi)$ est non-vide en vérifiant qu'il existe un chemin d'un état initial de A'_Π à un état final de A'_Π, en temps linéaire, par exemple avec un DFS.

On détermine ensuite si A'_Π contient un cycle d'états accessibles et co-accessibles. Pour ce faire, on détermine les états accessibles et co-accessibles par un parcours de graphe, et ensuite on utilise un DFS pour déterminer si on rencontre un cycle dans ces sommets. Ceci est toujours en temps linéaire, et conclut.

Question 5. Dans le cas où l'automate produit est déterministe, on peut compter le nombre de mots qu'il accepte par de la programmation dynamique. Spécifiquement, le nombre de mots acceptés à partir d'un état q est 1 ou 0 selon que q est final ou non, plus le nombre de mots acceptés à partir des états vers lesquels q a des transitions sortantes. Noter que, comme l'automate est déterministe, les étiquettes de ces transitions sont différentes, ainsi les ensembles de mots concernés sont disjoints, et c'est effectivement correct de faire la somme comme expliqué.

On n'a en fait pas vraiment besoin que l'automate produit soit déterministe : il suffit qu'il soit *inambigu*, c'est-à-dire que tout mot accepté a un unique chemin acceptant. Dans ce cas, la construction présentée reste correcte, parce que si un état q a des transitions étiquetées par la même lettre vers deux états distincts q_1 et q_2, alors la définition de l'inambiguïté impose que l'ensemble des mots acceptés à partir de q_1 et celui des mots acceptés à partir de q_2 sont disjoints.

Il suffit alors d'observer que, si A est déterministe (ou, en fait, s'il est inambigu), alors $\overline{A}$ est toujours inambigu (même si pas forcément déterministe). Maintenant, le produit $A \times \overline{A}$ avec l'ensemble d'états finaux indiqué est également inambigu : si un mot $u \in \Sigma^*$ avait un chemin vers (q, q) et vers (q', q') dans l'automate produit avec $q \neq q'$, alors on saurait que $u\overline{u}$ a deux chemins acceptants distincts dans A (un où l'état intermédiaire est q, un autre où c'est q'), ce qui contredirait le fait que A soit inambigu. Ainsi, si A est inambigu on peut construire le produit avec la même complexité qu'avant, il est inambigu, et on compte la taille du langage qu'il accepte comme expliqué. Comme cet ensemble est en bijection avec $L(A) \cap \Pi_{\text{pair}}$ (comme prouvé à la question 4), on a établi le résultat.

[L'hypothèse que l'automate d'entrée est déterministe ou inambigu est probablement indispensable, parce que compter le nombre de mots acceptés par un automate quelconque est #P-difficile : voir [KSM95] Theorem 2.1.]

Question 6. L'automate produit A' de la question 4 ne gère intuitivement que les u qui se complètent en un palindrome de longueur paire. Ceci dit, pour chaque $a \in \Sigma$, on peut adapter l'ensemble d'états finaux de l'automate produit pour construire un automate A'_a qui reconnaisse exactement $\{u \in \Sigma^* \mid ua\overline{u} \in L(A)\}$, c'est-à-dire le langage des mots u qui se complètent en un palindrome de longueur impaire accepté par A en ajoutant a comme lettre centrale, puis le miroir de u. On définit A'_a pour chaque $a \in \Sigma$ exactement comme A' mais avec l'ensemble d'états finaux $F'_a := \{(q, q') \mid (q, a, q') \in \delta\}$, ce qui est clairement correct : un mot $u \in \Sigma^*$ a un chemin acceptant dans A'_a finissant en (q, q') si et seulement s'il y a un chemin d'un état initial de A à q étiqueté par u, une transition dans A étiquetée par a de q à q', et un chemin de q' à un état final de A étiqueté par $\overline{u}$ dans A. Du reste, si A est inambigu alors tous ces automates produits le sont, par le même raisonnement qu'à la question précédente. On peut ainsi appliquer la construction de la question précédente à A' et à A'_a pour chaque $a \in \Sigma$, ce qui ne fait que rajouter un facteur $|\Sigma|$ à la complexité : on peut obtenir la cardinalité du nombre de palindromes reconnus en sommant les quantités pour A' et pour chaque A'_a (en traitant ∞ de la manière attendue). Ceci est correct parce que les palindromes de $\Pi \cap L(A)$ se partitionnent entre ceux de longueur paire et ceux de longueur impaire dont la lettre centrale est $a \in \Sigma$ pour chaque a. (En revanche, noter que le langage de A' et celui des A'_a ne sont pas forcément disjoints, vu qu'il est tout à fait possible, par exemple, que A accepte $u\overline{u}$ et $ua\overline{u}$ pour diverses valeurs de $a \in \Sigma$. Autrement dit, il faut bien sommer la cardinalité de ces langages même si leur union n'est pas disjointe, pour la raison qu'on vient d'expliquer.)

Références

[ALR+09] Terry Anderson, John Loftus, Narad Rampersad, Nicolae Santean, and Jeffrey Shallit. Detecting palindromes, patterns and borders in regular languages. *Information and Computation*, 207(11), 2009. `https://www.sciencedirect.com/science/article/pii/S0890540109000650`.

[KSM95] Sampath Kannan, Z. Sweedyk, and Steve Mahaney. Counting and random generation of strings in regular languages. In *Proc. SODA*, 1995.

A5 – Réparation de mots pour un langage régulier

On fixe un langage régulier L sur un alphabet Σ. Étant donné un mot $w \in \Sigma^*$, une *réparation par insertion* du mot w pour le langage L est une décomposition $w = uv$ de w en deux mots, et un mot $z \in \Sigma^*$, de sorte que le mot uzv appartienne à L.

Question 0. On pose L_0 le langage régulier défini par l'expression rationnelle $(ab)^*$. Proposer une réparation par insertion du mot $w_0 = aab$ pour L_0. Proposer une réparation par insertion du mot $w_0' = abab$ pour L_0. Le mot $w_0'' = aa$ admet-il une réparation par insertion pour L_0 ?

Question 1. Une *réparation par insertion finale* d'un mot $w \in \Sigma^*$ pour un langage régulier L est un mot $z \in \Sigma^*$ tel que wz appartienne à L. Proposer un algorithme qui, étant donné w et L, détermine s'il existe une réparation par insertion finale de w pour L. Préciser sa complexité en temps et en espace.

Question 2. On souhaite modifier l'algorithme de la question 1 pour que, lorsqu'une réparation par insertion finale existe, l'algorithme calcule un z correspondant qui soit de longueur minimale. Expliquer comment procéder, et préciser la complexité en temps et en espace.

Question 3. Proposer un algorithme qui, étant donné un mot $w \in \Sigma^*$ et un langage régulier L, détermine s'il existe une réparation par insertion de w pour L. Préciser sa complexité en temps et en espace.

Question 4. Modifier l'algorithme de la question 3 pour que, lorsqu'une réparation par insertion existe, l'algorithme calcule une décomposition $w = uv$ et un z correspondant qui soit de longueur minimale. Préciser la complexité en temps et en espace.

Question 5. Une *réparation par suppression* d'un mot $w \in \Sigma^*$ pour un langage L est une décomposition $w = uzv$ de w de sorte que $uv \in \Sigma^*$. Proposer un algorithme naïf qui, étant donné w et L, détermine s'il existe une réparation par suppression de w pour L, et si oui, calcule une telle suppression telle que z soit de longueur minimale. Préciser sa complexité en temps et en espace.

Question 6. Proposer un algorithme plus efficace pour déterminer, étant donné w et L, s'il existe une réparation par suppression de w pour L. La complexité de l'algorithme doit être linéaire en w.

Question 7. Modifier l'algorithme de la question 6 pour que, quand une réparation par suppression existe, l'algorithme calcule une décomposition $w = uzv$ et un z correspondant qui soit de longueur minimale. On demande toujours une complexité linéaire en w.

Suite des questions

Question 8. Une *réparation par insertions* d'un mot $w \in \Sigma^*$ pour un langage L est un mot $w' \in \Sigma^*$ tel que w' appartienne à L et w soit un *sous-mot* de w', c'est-à-dire qu'il existe une fonction strictement croissante ϕ de $\{1, \ldots, |w|\}$ dans $\{1, \ldots, |w'|\}$ telle que $w_i = w'_{\phi(i)}$ pour tout $1 \leq i \leq |w|$, où $|w|$ dénote la longueur de w et w_i dénote la i-ème lettre de w.

Proposer un algorithme qui détermine, étant donné L et w, s'il existe une réparation par insertions de w pour L, et le cas échéant calcule une telle réparation qui réalise un nombre minimal d'insertions, c'est-à-dire un w' de longueur minimale. Préciser la complexité en temps et en espace.

Question 9. On souhaite à présent autoriser des réparations par insertion et par suppression de caractères du mot initial : partant de w, on supprime un certain nombre k_1 de caractères, puis on insère un nombre k_2 de caractères, de sorte à obtenir un mot de L. Le coût de cette réparation est $k_1 + k_2$. Étant donné L et w, calculer le coût minimal d'une réparation et un exemple de réparation de coût minimal.

Corrigé

Question 0. Pour $w_0 = aab$, on pose $u_0 := a$, $v_0 := ab$, $z_0 := b$, et on obtient $u_0 z_0 v_0 = abab \in L_0$.

Pour $w'_0 = abab$, on a $w'_0 \in L_0$ donc on peut par exemple poser $u'_0 := w'_0$, $v'_0 := \epsilon$, $z'_0 := \epsilon$, et on obtient $u'_0 z'_0 v'_0 = abab \in L_0$.

Pour $w''_0 = aa$, la seule décomposition de w''_0 qui évite de terminer par a (ce qui n'est le cas d'aucun mot de L_0) est $u''_0 := w''_0$, $v''_0 := \epsilon$, mais alors pour tout $z''_0 \in \Sigma^*$ on a que $u''_0 z''_0 v''_0$ commence par aa donc n'appartient pas à L_0. Ainsi, w''_0 n'admet aucune réparation par insertion pour L_0.

Question 1. *Discussion avec le candidat : comment le langage L est-il représenté en entrée ? On conviendra de le représenter par un automate fini non-déterministe.*

On demandera un pseudo-code de l'algorithme.

L'algorithme calcule d'abord l'ensemble des états Q_w défini comme suit : un état q est dans Q_w si et seulement s'il existe une exécution de l'automate qui lise w et aille d'un état initial à q. On peut le calculer itérativement sur w : l'ensemble Q_ϵ est celui des états initiaux, et une fois déterminé Q_u pour un préfixe u de w, étant donné la lettre suivante $a \in \Sigma$ de w, l'ensemble Q_{ua} est l'ensemble des états accessibles par une transition étiquetée a à partir d'un état de Q_u. Ce calcul s'effectue en temps $O(|w| \times |\delta|)$ où $|\delta|$ est le nombre de transitions de l'automate : à chaque lettre du mot, on considère l'ensemble des transitions, et le nouvel ensemble des états où l'on peut se trouver sont les états cibles des transitions dont la source se trouve dans l'ensemble précédent des états où l'on peut se trouver.

Ensuite, il suffit de déterminer si Q_w contient un état co-accessible. Il suffit pour cela de déterminer les états co-accessibles par un test d'accessibilité à partir des états finaux suivant le renversement des transitions : ce calcul s'effectue en temps $O(|\delta|)$. (À noter que, dans le cas idiot où A a plus d'états que de transitions, on peut utiliser δ pour se restreindre aux états finaux qui apparaissent dans δ pour l'exploration.) On vérifie ensuite si Q_w contient un tel état. Ainsi, la complexité en temps est de $O(|A| + |w| \times |\delta|)$, en prenant en compte la complexité de lire l'automate en entrée. La complexité en espace est de $O(|Q|)$ puisqu'il suffit au plus de mémoriser un ensemble d'états du calcul initial et une structure de données (par exemple une file d'états) pendant le test d'accessibilité.

Question 2. *On demandera un pseudo-code de l'algorithme obtenu à partir du pseudo-code de la question précédente.*

On modifie l'algorithme précédent : au lieu de simplement calculer les états co-accessibles, on calcule, pour chaque état co-accessible, la longueur du plus court chemin de cet état à un état final. En effet,

cette longueur est celle de la plus petite insertion finale permettant d'atteindre un état final à partir de cet état, et un exemple de mot à insérer est donné par l'étiquette d'un tel chemin de longueur minimale.

Le calcul mentionné peut se faire avec un BFS, avec les mêmes complexités en temps et en espace. Pour construire un exemple de chemin, il suffit de stocker pour chaque état atteint par le parcours un état prédécesseur, et on trouve un chemin allant d'un état donné à un état final en suivant les prédécesseurs.

On obtient ainsi une réparation de longueur minimale en prenant l'état co-accessible de Q_w où la distance à un état final est minimale. Les complexités sont inchangées.

Question 3. *Pas de pseudo-code demandé.*

On calcule les ensembles d'états Q_u itérativement pour tous les préfixes u de w. On calcule itérativement, de la même manière, les ensembles d'états Q'_v pour chaque suffixe v de w définis comme suit : un état q est dans Q'_v si et seulement s'il existe un chemin de q à un état final étiqueté par v.

On considère ensuite chaque position de w : en notant u, v le préfixe et le suffixe correspondant, on peut effectuer une réparation par insertion à cet endroit si et seulement s'il existe une paire $(q_u, q_v) \in Q_u \times Q'_v$ telle qu'il y ait un chemin de q_u à q_v dans l'automate. On peut déterminer cela par un test d'accessibilité en $O(|\delta|)$. La complexité en temps est toujours de $O(|A| + |w| \times |\delta|)$. La complexité en espace est à présent de $O(|w| \times |Q|)$ vu qu'il faut mémoriser tous les ensembles intermédiaires.

Remarque : si le mot d'entrée est très long, une optimisation judicieuse en pratique serait probablement de déterminiser l'automate (ceci prend un temps indépendant du mot d'entrée, et assure que les ensembles Q_u seront tous de cardinal au plus 1). Ainsi, les ensembles d'états pour les préfixes sont en fait des singletons, donc on peut les calculer (ce qui prend un espace $O(|w|)$), puis calculer les ensembles pour les suffixes et faire le test. Ainsi, la complexité en espace tombe à $O(|w| + |Q|)$.

Question 4. *Pas de pseudo-code demandé.*

Il suffit de calculer un plus court chemin comme en question 2. Les complexités sont inchangées.

Autre remarque : On pourrait par ailleurs envisager de pré-calculer, pour chaque paire d'états de l'automate, la distance d'un plus court chemin de l'un à l'autre, par exemple avec l'algorithme de Floyd-Warshall ($O(|Q|^3)$). Ainsi, lors de la deuxième phase, au lieu de faire un BFS à chaque position du mot, il suffit de considérer chaque état de Q'_v et prendre le min de la distance avec les états de Q_u (ou l'unique état de Q_u s'il existe, dans le cas d'un automate déterministe). On peut ensuite ne faire le calcul explicite du chemin que pour une décomposition de longueur minimale. Malgré tout, cette optimisation ne change pas la complexité asymptotique pour le temps d'exécution, qui reste $O(|A| + |w| \times \times |\delta|)$, i.e., dominée par le calcul des ensembles Q_u et Q'_v.

Question 5. *Pas de pseudo-code demandé.*

On teste simplement toutes les décompositions possibles de $w = uzv$, et on regarde pour chacune d'entre elles si $uv \in L$. On peut ensuite facilement en renvoyer une de longueur minimale (en testant les décompositions en prenant des z de longueur croissante). La complexité est en $O(|A| + |w|^3 \times |\delta|)$, puisqu'il y a $O(|w|^2)$ décompositions à tester, et que chaque test prend $O(|w| \times |\delta|)$.

Question 6. On calcule pour chaque préfixe u de w trois ensembles d'états Q_u, Q'_u, Q''_u :

- L'ensemble Q_u est l'ensemble des états où on peut aboutir après lecture de u depuis un état initial (sans suppression), exactement comme à la question 1.
- L'ensemble Q'_u est l'ensemble des états où on peut aboutir après lecture d'un préfixe de u (correspondant intuitivement à une suppression en cours) : on définit $Q'_\epsilon := Q_\epsilon$, et pour tout préfixe u et lettre suivante $a \in \Sigma$ on définit $Q'_{ua} := Q'_u \cup Q_{ua}$
- L'ensemble Q''_u est l'ensemble des états où on peut aboutir après lecture d'une réparation par suppression de u. On définit $Q''_\epsilon := Q_\epsilon$, et pour tout préfixe u et lettre suivante $a \in \Sigma$ on définit

$Q''_{ua} := Q'_{ua} \cup \delta(Q''_u, a)$, où $\delta(Q''_u, a)$ est l'ensemble des états accessibles par un état de Q''_u par une transition étiquetée a, comme dans le cas d'induction de la définition de Q_u à la question 1.

On peut montrer immédiatement la correction par récurrence : le seul point important est la définition inductive de Q''_{ua}, où on observe que les états accessibles par une suppression sur le préfixe ua sont exactement les états accessibles par une suppression qui se finit à la fin du préfixe ua, c'est-à-dire Q'_{ua}, et l'image après lecture de a des états accessibles par une suppression qui se finit strictement avant la fin du préfixe ua, c'est-à-dire $\delta(Q''_u, a)$.

À la fin, pour décider l'existence d'une réparation par suppression, il suffit de vérifier si Q''_w contient un état final.

Le calcul des ensembles a la même complexité qu'en question 1, c'est-à-dire $O(|w| \times |\delta|)$, et la complexité en espace est toujours $O(|Q|)$.

Question 7. On modifie l'algorithme de la question 6. Au lieu de représenter Q'_u et Q''_u comme des ensembles, on va en faire des tableaux ayant pour valeur des entiers. Plus précisément, pour chaque état q de l'automate et préfixe u du mot w, l'entier $Q'_u[q]$ sera la longueur minimale du suffixe de u à supprimer pour aboutir à l'état q (ou une longueur spéciale ∞ s'il n'est pas possible du tout d'aboutir à cet état), et $Q''_u[q]$ sera la longueur minimale d'une suppression dans u pour aboutir à l'état q. À la fin, on va renvoyer le minimum de $Q''_w[q]$ pour q parcourant l'ensemble des états finaux.

Pour le calcul de Q', on initialise $Q'_u[q] := 0$ pour les états q qui sont initiaux, et $Q'_u[q] := \infty$ sinon ; et pour un préfixe u et une lettre a, on définit $Q'_{ua}[q]$ pour chaque q comme 0 si $q \in Q'_{ua}$ (on peut y aboutir par la suppression vide) et sinon comme $1 + Q'_u$ (on peut y aboutir en poursuivant la suppression précédente).

Pour le calcul de Q'', on initialise Q'' de la même manière que Q', et pour un préfixe u et une lettre a, pour chaque état q, on définit $Q''_{ua}[q]$ comme le min de $Q''_u[q']$ sur les q' tels qu'il y ait une transition étiquetée a de q à q' (on peut y aboutir par une suppression qui se finit plus tôt) et de $Q'_{ua}[q]$ (on peut y aboutir par une suppression qui se finit à ce moment).

Il est clair par induction que les quantités des ensembles Q et des tableaux Q' et Q'' ont la sémantique prescrite, et ainsi l'algorithme est correct. La complexité est inchangée.

Si on veut retrouver une suppression témoin, on modifie l'algorithme pour conserver toutes les valeurs calculées pour Q, Q', et Q'', de sorte que l'espace mémoire utilisé est à présent en $O(|Q| \times |w|)$. Une fois le calcul terminé, on choisit un état q final quelconque tel que $Q''_w[q]$ soit minimal parmi les finaux. Si $c := Q''_w[q]$ est nul alors la suppression vide est une réparation correcte. Sinon, comme les valeurs de Q'' ont été initialisées soit à 0 soit à ∞, par définition, il doit exister un préfixe x de w et un état q' tel que $c = Q'_x[q']$ et il y a un chemin de q à q' étiqueté par le suffixe z de longueur $|w| - |x|$ de w. On considère un préfixe x de longueur maximale avec cette propriété. Par définition encore, pour y le préfixe de w de longueur $|x| - c$, on a $q \in Q_y$. Considérons à présent le mot xz. On sait que l'automate peut aboutir à l'état q en lisant x puisque $q \in Q_x$, et la définition de Q'' garantit que l'automate peut aller de q à q' en lisant z, ainsi on a bien obtenu une réparation.

Question 8. On explique d'abord comment calculer le coût minimal sans calculer un témoin. On va suivre un algorithme dynamique. On construit un tableau $Q_u[q]$ où u parcourt les préfixes de w et q les états de l'automate, stockant le nombre minimal d'insertions à réaliser dans u pour que l'automate puisse aboutir à l'état q.

Le cas de base est de définir $Q_\epsilon[q] := 0$ pour tous les états initiaux q, de définir $Q_\epsilon[q] := \infty$ pour les états inaccessibles, et pour les autres états q' on définit $Q_\epsilon[q']$ comme étant égal à la longueur du plus court chemin d'un état initial à q' dans l'automate. On peut calculer cela efficacement par un BFS.

L'induction est comme suit : pour tout préfixe u de w et lettre suivante a, pour tout état q, on assigne $Q_{ua}[q] := \min_{q'} Q_u[q'] + d_a(q', q') - 1$ où $d_a(q', q)$ dénote la longueur minimale d'un chemin de q à q' dans l'automate qui commence par a. Autrement dit, le nombre minimal d'insertions pour atteindre q en lisant le préfixe ua avec des insertions est le min, sur les états q' atteints avant a avec des insertions,

du nombre d'insertions nécessaires pour atteindre q' avant a plus le nombre d'insertions à effectuer, après avoir lu a depuis q', pour atteindre q.

À la fin, on renvoie le min de $Q_w[q]$ sur les états finaux q. Il est clair que cet algorithme est correct. La complexité en espace est en $O(|Q|)$ vu qu'on ne mémorise que la colonne précédente du tableau. La complexité en temps est $O(|A| + |w| \times |Q| \times |\delta|)$.

Pour calculer également un témoin, on conserve tous les tableaux (donc un espace mémoire en $O(|w| \times |Q|)$) et on stocke, dans chaque case $Q_u[q]$, un choix d'état q' qui réalise le min. On peut alors calculer un témoin en remontant depuis la fin comme à la question 7, en calculant un plus court chemin à chaque étape.

Question 9. Il s'agit d'une variante de la question 8 qui s'inspire de l'algorithme de Levenshtein. On modifie simplement la définition inductive de $Q_{ua}[q]$ pour prendre le min de la définition précédente et de $1 + Q_u[q]$: on peut soit aboutir à l'état q en aboutissant à un état q' avant a puis en lisant a et en effectuant un certain nombre d'insertions, ou bien en aboutissant à l'état q juste avant a et en supprimant a (inutile de considérer la possibilité d'effectuer des insertions après a dans ce cas vu qu'on peut tout aussi bien décider de les faire toutes avant de supprimer a).

A6 – Clôture par sur-mots et sous-mots

On fixe un alphabet Σ. Étant donné deux mots $w, w' \in \Sigma^*$, on dit que w' est un *sur-mot* de w, noté $w \preccurlyeq w'$, s'il existe une fonction strictement croissante ϕ de $\{1, \dots, |w|\}$ dans $\{1, \dots, |w'|\}$ telle que $w_i = w'_{\phi(i)}$ pour tout $1 \leq i \leq |w|$, où $|w|$ dénote la longueur de w et w_i dénote la i-ème lettre de w. Étant donné un langage L, on note $\overline{L}$ le langage des sur-mots de mots de L, c'est-à-dire $\overline{L} := \{w' \in \Sigma^* \mid \exists w \in L, w \preccurlyeq w'\}$.

Question 0. On pose L_0 le langage défini par l'expression rationnelle ab^*a, et L_1 le langage défini par l'expression rationnelle $(ab)^*$. Donner une expression rationnelle pour $\overline{L_0}$ et pour $\overline{L_1}$.

Question 1. Montrer que, pour tout langage L, on a $\overline{\overline{L}} = \overline{L}$.

Question 2. Existe-t-il des langages L' pour lesquels il n'existe aucun langage L tel que $\overline{L} = L'$?

Question 3. Montrer que, pour tout langage régulier L, le langage $\overline{L}$ est également régulier.

Question 4. On admettra pour cette question le résultat suivant : pour toute suite $(w_n)_{n\in\mathbb{N}}$ de mots de Σ^*, il existe $i < j$ tels que $w_i \preccurlyeq w_j$.

Montrer que, pour tout langage L (non nécessairement régulier), il existe un langage fini $F \subseteq L$ tel que $\overline{F} = \overline{L}$.

Question 5. Un langage L est *clos par sur-mots* si, pour tout $u \in L$ et $v \in \Sigma^*$ tel que $u \preccurlyeq v$, on a $v \in L$. Déduire de la question précédente que tout langage clos par sur-mots est régulier.

Suite des questions

Question 6. On considère un langage L arbitraire, non nécessairement régulier, et on souhaite construire effectivement un automate pour reconnaître $\overline{L}$. Comment peut-on procéder, et de quelles opérations sur L a-t-on besoin ? Discuter de l'efficacité de cette procédure.

Question 7. Un langage L est *clos par sous-mots* si, pour tout $u \in L$ et $v \in \Sigma^*$ tel que $v \preccurlyeq u$, on a $v \in L$. Montrer que tout langage clos par sous-mots est régulier.

Question 8. Démontrer le résultat admis à la question 4.

Corrigé

Question 0. Le langage $\overline{L_0}$ est le langage des mots qui contiennent deux a, c'est-à-dire $\Sigma^* a \Sigma^* a \Sigma^*$. En effet, tout sur-mot d'un mot de ab^*a doit clairement contenir deux a. Réciproquement, tout mot contenant deux a est un sur-mot de aa qui appartient à L_0.

Le langage $\overline{L_1}$ est Σ^*, puisque tout mot est un sur-mot de $\epsilon \in L_1$.

Question 1. On observe d'abord que la relation $\preccurlyeq$ est transitive. En effet, pour tous mots $w, w', w'' \in \Sigma^*$ tels que $w \preccurlyeq w'$ et $w' \preccurlyeq$, en notant ϕ et ϕ' les fonctions strictement croissantes qui en témoignent, leur composition $\phi' \circ \phi$ est une fonction strictement croissante de $\{1, \dots, |w|\}$ dans $\{1, \dots, |w''|\}$, et pour tout $1 \leq i \leq w$ on a $w''_{\phi'(\phi(i))} = w'_{\phi(i)} = w_i$.

On montre à présent l'égalité demandée. Il est clair que $\overline{L} \subseteq \overline{\overline{L}}$, donc on montre l'inclusion inverse. Soit $u'' \in \overline{\overline{L}}$, il existe un mot $u' \in \overline{L}$ tel que $u' \preccurlyeq u$. Par définition de $\overline{L}$, il existe un mot $u \in L$ tel que $u \preccurlyeq u'$. Par transitivité, on a $u \preccurlyeq u''$. Ainsi, on a bien $u'' \in \overline{L}$, ce qui conclut.

Question 2. Pour tout langage non-vide L, le langage $\overline{L}$ est nécessairement infini : en effet, pour $u \in L$ quelconque, on a $u\Sigma^* \subseteq \overline{L}$. Par ailleurs, on a clairement $\overline{\emptyset} = \emptyset$. Ainsi, si l'on prend L' fini non-vide, on sait qu'il n'existe aucun langage L tel que $\overline{L} = L'$.

Autre preuve possible : on considère le langage L_0. Supposons par l'absurde qu'il existe un langage L tel que $\overline{L} = L_0$. Dans ce cas, on a $\overline{\overline{L}} = \overline{L_0}$, donc d'après la question 1, on a $\overline{L} = \overline{L_0}$. C'est absurde car L_0 et $\overline{L_0}$ sont manifestement différents. Ainsi, $L' := L_0$ convient.

Question 3. Soit A un automate fini non-déterministe qui reconnaisse le langage régulier L. Construisons un automate A' en ajoutant à chaque état de A une boucle pour toutes les lettres de l'alphabet : formellement, on initialise $A' := A$ et pour chaque $a \in \Sigma$ et chaque état q de A, on ajoute à A' une transition de q à q étiquetée par a.

Il est clair que, pour tout mot u accepté par A et pour tout mot u' tel que $u \preccurlyeq u'$, le mot u' est accepté par A' : pour ϕ une fonction strictement croissante qui témoigne du fait que $u \preccurlyeq u'$, il suffit de suivre le chemin pour u dans A' pour les positions de u' appartenant à l'image de ϕ, et de suivre les nouvelles transitions pour les positions de u' qui n'appartiennent pas à l'image de ϕ. Réciproquement, si l'on considère un mot u' accepté par A' et un chemin qui en témoigne, on peut construire un mot u accepté par A tel que $u \preccurlyeq u'$ en considérant la restriction de ce chemin aux transitions de A.

On peut aussi démontrer cette question par induction structurelle sur les expressions rationnelles à l'aide des identités suivantes :

— $\overline{\emptyset} = \emptyset$

— $\overline{\epsilon} = \Sigma^*$

— $\overline{a} = \Sigma^* a \Sigma^*$ pour tout $a \in \Sigma$

— $\overline{L_1 L_2} = \overline{L_1}\ \overline{L_2}$
— $\overline{L_1 \cup L_2} = \overline{L_1} \cup \overline{L_2}$
— $\overline{L^*} = \Sigma^*$.

La dernière égalité est due au fait que L^* contient toujours le mot vide ; en revanche il n'est pas vrai que $\overline{L^*} = \overline{L}^*$, prendre par exemple $L = a$.

Question 4. Soit L un langage quelconque. Si L est vide, on peut prendre $F = \emptyset$ et conclure. Sinon, posons $(w_n)_{n\in\mathbb{N}}$ une suite infinie énumérant les mots du langage L (éventuellement avec des doublons). Une position $i \in \mathbb{N}$ est dite *innovante* s'il n'existe aucun $j < i$ tel que $w_j \preccurlyeq w_i$. On choisit pour F le sous-ensemble de L formé des mots aux positions innovantes, c'est-à-dire $\{w_i \mid i \text{ est innovante}\}$.

On observe à présent qu'il y a un nombre fini de positions innovantes. En effet, dans le cas contraire, la suite extraite obtenue à partir de $(w_n)_{n\in\mathbb{N}}$ en conservant les lettres aux positions innovantes serait un contre-exemple à la question 4. Ainsi, F est-il bien fini.

Montrons à présent que $\overline{F} = \overline{L}$. En effet, comme $F \subseteq L$, on a $\overline{F} \subseteq \overline{L}$ par monotonie de la clôture par sur-mots. Pour la réciproque, il suffit de montrer que $L \subseteq \overline{F}$, car cela implique (à nouveau par monotonie de la clôture par sur-mots) que $\overline{L} \subseteq \overline{\overline{F}}$, ce qui implique par la question 1 que $\overline{L} \subseteq \overline{F}$. Montrons par induction sur $i \in \mathbb{N}$ que $w_j \in \overline{F}$ pour tout $j < i$. Le cas de base est tautologique. Pour le cas de récurrence, choisissons $i \in \mathbb{N}$. Soit i est innovante, soit i n'est pas innovante. Dans le premier cas, on a $w_i \in F$ donc $w_i \in \overline{F}$. Dans le second cas, il existe $j < i$ tel que $w_j \preccurlyeq w_i$, et par hypothèse de récurrence on a $w_j \in \overline{F}$, ainsi on a $w_i \in \overline{F}$. Ainsi, dans les deux cas on a $w_i \in \overline{F}$. On a donc établi notre résultat par récurrence, et on a donc bien l'inclusion réciproque $L \subseteq \overline{F}$.

Question 5. Soit L un langage clos par sur-mots. On sait par la question 5 qu'il existe un langage fini $F \subseteq L$ tel que $\overline{F} = \overline{L}$. Or on a $\overline{L} = L$. En effet, il est clair que $L \subseteq \overline{L}$, et réciproquement, pour tout $v \in \overline{L}$, il existe par définition de $\overline{L}$ un mot $u \in L$ tel que $u \preccurlyeq v$, et ainsi $v \in L$ car L est clos par sur-mots. On sait donc que $L = \overline{F}$, et on sait que F est régulier (car fini), donc $\overline{F}$ est régulier par la question 3, ainsi L est-il régulier.

Question 6. On a besoin de pouvoir énumérer efficacement des mots de L qui ne sont pas des sur-mots de mots précédemment énumérés. En d'autres termes, il nous faudrait un oracle qui, étant donné une liste de mots $W = (w_0, \ldots, w_n)$ de L, produit un mot w_{n+1} de L tel que $w_i \not\preccurlyeq w_{n+1}$ pour tout $0 \leq i \leq n$, ou bien conclut qu'aucun tel mot n'existe, de sorte que $\overline{L} = \overline{W}$. On pourrait aussi se contenter d'un oracle qui, étant donné un langage régulier L' (ici, $L' := \overline{W}$), produit un mot de $L \setminus L'$, ou conclut que $L \subseteq L'$.

Étant donné un tel oracle, on construirait petit à petit un automate de $\overline{W}$ jusqu'à avoir couvert tout $\overline{L}$.

On ne peut pas espérer que cette procédure soit efficace, car le nombre d'invocations de l'oracle (et la taille de l'automate construit) serait très grande même pour des langages simples ; formellement, même quand L est rationnel, elle peut être exponentielle en la longueur d'une expression rationnelle pour L. Si l'on prend par exemple L_k l'ensemble rationnel des mots de longueur k pour un certain $k \in \mathbb{N}$, que l'on peut décrire avec une expression régulière ou un automate de taille $O(k \times |\Sigma|)$, les mots de L_k sont tous incomparables pour la relation $\preccurlyeq$ vu qu'ils sont de même longueur, et il y a un nombre exponentiel de tels mots, à savoir $|\Sigma|^k$.

Question 7. Soit L un langage clos par sous-mots, et soit $L' := \Sigma^* \setminus L$ son complémentaire. Montrons que L' est clos par sur-mots. En effet, soit $u \in L'$ et $v \in \Sigma^*$ tels que $u \preccurlyeq v$. Procédons par l'absurde et supposons que $v \notin L'$. On a alors $v \in L$. Comme $u \preccurlyeq v$ et que L est clos par sous-mots, on sait que $u \in L$, et ainsi $u \notin L'$, contredisant notre hypothèse. Ainsi, $v \in L'$, ce qui établit que L' est clos par

sur-mots. On sait donc que L' est régulier. Comme les langages réguliers sont clos par complémentation, le complémentaire L de L' est lui aussi régulier.

Question 8. *[Il s'agit du lemme de Higman dans le cas particulier des alphabets finis [Wik18].]*

Procédons par l'absurde et supposons qu'il existe une *mauvaise suite*, c'est-à-dire une suite $(w_n)_{n\in\mathbb{N}}$ telle qu'il n'existe pas de $i < j$ telle que $w_i \preccurlyeq w_j$. Construisons une nouvelle suite $(w'_n)_{n\in\mathbb{N}}$ de la façon suivante : le mot w'_0 est un mot de longueur minimale telle qu'il existe une mauvaise suite commençant par w'_0 (un tel w'_0 existe par notre hypothèse), le mot w'_1 est un mot de longueur minimale telle qu'il existe une mauvaise suite commençant par w'_0, w'_1 (un tel w'_1 existe par définition de w'_0), et ainsi de suite. La suite $(w'_n)_{n\in\mathbb{N}}$ ainsi définie est clairement mauvaise : pour tout $i < j$, la définition de w'_j assure qu'on ne peut avoir $w'_i \preccurlyeq w'_j$. Par ailleurs, la définition de $(w'_n)_{n\in\mathbb{N}}$ assure qu'elle est *minimale*, c'est-à-dire que pour toute mauvaise suite $(w''_n)_{n\in\mathbb{N}}$, si on pose $i \in \mathbb{N}$ le premier indice tel que $w'_i \neq w''_i$, on a nécessairement $|w'_i| \leq |w''_i|$.

On va aboutir à notre contradiction en construisant à partir de $(w'_n)_{n\in\mathbb{N}}$ une nouvelle mauvaise suite qui contredise sa minimalité. Soit $a \in \Sigma$ une lettre quelconque telle qu'il existe un nombre infini de mots de $(w'_n)_{n\in\mathbb{N}}$ ayant a comme première lettre : comme Σ est fini, un tel a existe nécessairement. Soit $p \in \mathbb{N}$ le plus petit entier tel que w'_p commence par a. On construit la suite $(w''_n)_{n\in\mathbb{N}}$ comme la concaténation de $w'_0, \ldots, w'_{p-1}$ et de la suite extraite de $(w'_n)_{n\in\mathbb{N}}$ des mots commençant par a à qui on a retiré leur première lettre.

La suite $(w''_n)_{n\in\mathbb{N}}$ est une mauvaise suite. En effet, soit $p < q$. Si $q < i$, alors $w''_p = w'_p$ et $w''_q = w'_q$, donc $w''_p \not\preccurlyeq w''_q$ car $(w'_n)_{n\in\mathbb{N}}$ est mauvaise. Si $i \leq p$, alors $aw''_p = w'_p$ et $aw''_q = w'_q$, donc on conclut encore car $(w'_n)_{n\in\mathbb{N}}$ est mauvaise. Si $p < i \leq q$, alors $w''_p = w'_p$ et $aw''_q = w'_q$, et on conclut de même.

Par ailleurs, la suite $(w''_n)_{n\in\mathbb{N}}$ contredit la minimalité de $(w'_n)_{n\in\mathbb{N}}$. En effet, le premier indice où ces deux suites diffèrent est p, et on a $\left|w''_p\right| = \left|w'_p\right| - 1$, ce qui contredit bien la minimalité. C'est absurde, et ainsi notre hypothèse initiale affirmant l'existence d'une mauvaise suite est-elle fausse.

[Certaines idées de ce sujet sont inspirées de [Mad16].]

Références

[Mad16] David Madore. Le lemme de Higman expliqué aux enfants, 2016. `http://www.madore.org/~david/weblog/d.2016-04-26.2368.html#d.2016-04-26.2368`.

[Wik18] Wikipedia. Higman's lemma, 2018. `https://en.wikipedia.org/wiki/Higman%27s_lemma`.

A7 – Langages continuables et mots primitifs

On fixe un alphabet fini Σ et on suppose $|\Sigma| > 1$. Dans ce sujet, on considérera des automates sur l'alphabet Σ qui seront toujours supposés déterministes complets.

Un mot non-vide $w \in \Sigma^*$ est dit *primitif* s'il n'existe pas de mot $u \in \Sigma^*$ et d'entier $p > 1$ tel que $w = u^p$.

Question 0. Le mot *abaaabaa* est-il primitif ? Le mot *ababbaabbbababbab* (de longueur 17) est-il primitif ?

Question 1. Proposer un algorithme naïf qui, étant donné un mot, détermine s'il est primitif, et discuter de sa complexité en temps et en espace.

Question 2. Donner un exemple d'un langage régulier infini qui ne contienne aucun mot primitif.

Question 3. Donner un exemple d'un langage régulier infini qui ne contient que des mots primitifs.

Question 4. Un langage régulier L est dit *continuable* s'il a la propriété suivante : pour tout $u \in \Sigma^*$, il existe $v \in \Sigma^*$ tel que $uv \in L$. Donner un exemple de langage régulier infini non continuable. Existe-t-il des langages réguliers continuables dont le complémentaire soit infini ?

Question 5. Étant donné un automate A, proposer un algorithme pour déterminer si le langage $L(A)$ qu'il reconnaît est continuable. Justifier sa correction et discuter de sa complexité en temps et en espace.

Question 6.

(a) Montrer que tout langage régulier continuable contient une infinité de mots primitifs.

(b) Étant donné un langage régulier continuable L, donner une borne supérieure sur la taille du plus petit mot primitif de L.

(c) La réciproque de la question (a) est-elle vraie ?

Corrigé

Question 0. Le mot $abaaabaa$ n'est pas primitif puisqu'on peut l'écrire $(abaa)^2$.

Le second mot proposé est primitif : comme sa longueur est un nombre premier, la seule factorisation possible serait comme l'exponentiation d'un mot de longueur 1, ce qui n'est pas possible car tous ses caractères ne sont pas identiques.

Question 1. On suppose sans perte de généralité que le mot d'entrée est non-vide.

Étant donné un mot $w \in \Sigma^*$, il suffit de tester, pour chaque $1 \leq p \leq |w|/2$, si p divise $|w|$, et si oui on fait le test suivant : pour chaque $1 \leq i \leq |w| - p$, vérifier si $w_i = w_{i+p}$. Il est clair que ces tests réussissent pour un p si et seulement si on peut écrire w comme la puissance d'un mot de longueur $|w|/p$.

La complexité en espace est constante (en plus du mot w). La complexité en temps est en $O(|w|^2)$: le seul point possiblement litigieux serait le test de divisibilité, mais son temps d'exécution est clairement dominé par $O(|w|)$.

Question 2. On peut prendre par exemple $(aa)^*$, ou $(aa)(aa)^*$ si on veut éviter de parler du mot vide.

Question 3. On peut prendre par exemple ab^+ : il est clair que tout mot w de ce langage est primitif parce qu'il contient un seul a, et si on avait une écriture $w = u^p$ avec $p > 1$ le mot u devrait contenir $1/p$ occurrences de a, or ce nombre doit être entier, contradiction.

Question 4. Le langage régulier ab^* est infini mais n'est pas continuable (prendre $u := b$).

Il existe des langages réguliers continuables dont le complémentaire est infini, par exemple le langage des mots de longueur paire.

Question 5. Il suffit d'observer la caractérisation suivante : un automate déterministe complet représente un langage continuable si et seulement si tous ses états accessibles sont co-accessibles, c'est-à-dire que tout état accessible par un chemin depuis l'état initial a un chemin vers un état final. Cette propriété peut être testée en temps et mémoire linéaire en la taille de l'automate : on effectue d'abord un parcours en largeur depuis l'état initial pour identifier les états accessibles, on effectue un second parcours en largeur depuis les états finaux en suivant l'inverse des transitions pour identifier les états co-accessibles, et on vérifie que tous les états accessibles sont bien co-accessibles. (En particulier, si l'automate n'était pas complet initialement, alors il n'est pas continuable, sauf si le puits que l'on a ajouté n'est pas accessible, i.e., aucun des états auquel il manquait des transitions sortantes était accessible.)

Démontrons que cette caractérisation est correcte. Supposons d'abord que A ait un état q accessible mais non-co-accessible. Soit $u \in \Sigma^*$ l'étiquette d'un chemin de l'état initial de A à q. On sait alors qu'il n'existe pas de $v \in \Sigma^*$ tel que $uv \in L$, puisqu'un tel v témoignerait de l'existence d'un chemin de q à un état final de A. Ainsi, u est un contre-exemple au caractère continuable de $L(A)$.

Réciproquement, supposons que tous les états accessibles de A soient co-accessibles. Soit $u \in \Sigma^*$ quelconque, et soit q l'état auquel A aboutit après lecture de u depuis l'état final (cet état existe parce que l'automate est complet) : l'existence de u montre que q est accessible. Ainsi, q est co-accessible. Soit v l'étiquette d'un chemin de q à un état final de A. On voit alors que uv est l'étiquette d'un chemin de l'état initial de A à un état final de A, c'est-à-dire que $uv \in L(A)$. Ainsi, on a bien montré que $L(A)$ est continuable.

Question 6.

(a) Soient a, b deux lettres distinctes de Σ. Soit L un langage régulier continuable. Posons A un automate déterministe complet tel que $L(A) = L$, et supposons quitte à éliminer les états inaccessibles que tous les états de A sont accessibles. Comme $L(A)$ est un langage continuable, on sait que A satisfait le critère de la question 5 : plus précisément, comme tous les états de A sont accessibles, A doit être complet et tous ses états doivent être co-accessibles.

Soit n le nombre d'états de A. Pour chaque $i > 0$, écrivons $w'_i := ba^i$, écrivons q_i l'état auquel aboutit l'automate en lisant w'_i depuis son état initial (cet état existe car A est complet), et soit $x_i \in \Sigma^*$ un mot qui étiquette un chemin allant de l'état q_i à un état final de A : un tel chemin existe car tous les états de A sont co-accessibles, et on peut clairement toujours assurer que $|x_i| < n$. Pour tout $i > 0$ on définit $w_i := w'_i x_i$. Par définition, on a $w_i \in L(A)$ pour tout $i > 0$. Montrons à présent que, pour tout $i \geq n - 1$, le mot w_i est primitif, ce qui suffit à conclure la première partie de l'énoncé. En effet, procédons par l'absurde et supposons qu'on puisse écrire $w_i = u^p$ avec $p > 1$. On sait alors que la première lettre de w_i, qui est un b, est la même que la lettre $1 + |w_i|/p$: cette quantité est $\leq 1 + |w_i|/2$. Or on sait que toutes les lettres de w_i aux positions $\{2, \ldots, i + 1\}$ sont des lettres de w'_i qui sont des a, donc il faut que $1 + |w_i|/2$ soit $\geq i + 2$, i.e., que $|w_i| \geq 2(i + 1)$. Mais on sait que $|w_i| = 1 + i + |x_i|$, donc il faut que $|x_i| \geq i + 1$. Mais on a $|x_i| < n \leq i + 1$, donc on a une contradiction. Ainsi, le mot w_i est bien primitif.

(b) La construction de (a) montre en particulier que le mot w_{n-1} est primitif, et sa longueur est de $1 + (n - 1) + (n - 1)$ au plus, i.e., $2n - 1$. Autrement dit, tout langage régulier continuable L contient un mot primitif de longueur $\leq 2n - 1$, où n est le nombre d'états d'un automate déterministe reconnaissant L.

(c) La réciproque est fausse, car le langage ba^+ contient une infinité de mots primitifs mais n'est pas continuable.

[Les questions (a) et (b) sont une adaptation de la preuve de la Proposition 5 de [IKSY88].]

Références

[IKSY88] M Ito, M Katsura, HJ Shyr, and SS Yu. Automata accepting primitive words. In *Semigroup Forum*, volume 37, pages 45–52. Springer, 1988. `https://link.springer.com/article/10.1007/BF02573122`.

A8 – Graphes avec un nombre prescrit de chemins

Un *graphe orienté* est une paire $G = (V, E)$ où $E \subseteq V \times V$ est l'ensemble des arêtes. Un *chemin* de $u \in V$ à $v \in V$ est une séquence $u_1, \ldots, u_n$ avec $n \geq 1$ telle que $u_1 = u$, $u_n = v$, et pour tout $1 \leq i < n$, le couple (u_i, u_{i+1}) est dans E. Un *graphe pointé* est un triplet (G, s, t) où s et t sont deux sommets de G. On dit qu'un graphe pointé *réalise* un entier $n \in \mathbb{N}$ s'il existe exactement n chemins de s à t dans G.

L'objet de ce sujet est de construire des petits graphes pointés réalisant n'importe quel entier.

Question 0. Construire un graphe pointé qui réalise 3.

Question 1. Pour tout $n \in \mathbb{N}$, proposer une construction naïve d'un graphe pointé qui réalise n et expliciter son nombre de sommets.

Question 2. Quelles hypothèses simplificatrices peut-on faire sur un graphe pointé qui réalise un entier fini et non-nul ?

Question 3. Pour tout $k \in \mathbb{N}$, construire un graphe pointé à $k + 2$ sommets qui réalise 2^k.

Question 4. Pour tout $n > 0$, construire un graphe pointé à $\lceil \log_2 n \rceil + 2$ sommets qui réalise n.

Question 5. Cette construction est-elle optimale ?

Suite des questions

Question 6. Par automate, on entend un automate déterministe *avec un unique état final* sur l'alphabet $\Sigma = \{a, b\}$. On dit qu'un tel automate A *réalise* un entier $n \in \mathbb{N}$ si le langage reconnu par A contient exactement n mots.

Montrer que, dans un automate A réalisant un entier $n \in \mathbb{N}$ avec un nombre minimal d'états, on peut toujours supposer que l'état final n'a pas de transitions sortantes, et que tout autre état a exactement 2 transitions sortantes.

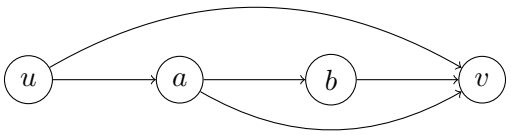
Question 7. Montrer que, pour tout $n > 0$, il existe un automate qui réalise n, dont le nombre d'états soit $\lfloor \log_2 n \rfloor$ plus un état par bit du nombre qui est à 1.

Question 8. On considère un groupe G. Étant donné un élément $g \in G$ et un entier $n \in \mathbb{N}$, on souhaite calculer g^n. Comment faire ceci efficacement en utilisant ce qui précède ?

Question 9. La construction de la question 7 est-elle optimale ?

Corrigé

Question 0. Par exemple :



Question 1. On considère simplement le graphe $G = (V, E)$ où $V = \{u, v\} \cup \{a_i \mid 1 \leq i \leq n\}$ et $E = \bigcup_{1 \leq i \leq n} \{(u, a_i), (a_i, v)\}$, qui a $n + 2$ sommets.

Question 2. Si un graphe pointé réalise un entier non nul, alors il existe un chemin de u à v, et on peut également supprimer tous les sommets du graphe qui ne sont pas à la fois accessibles à partir de u et co-accessibles à partir de v.

Par ailleurs, si le graphe pointé réalise un entier fini, alors il ne peut pas contenir de cycle : s'il existe un cycle, alors en prenant un chemin quelconque de u à un sommet z du cycle, un chemin quelconque de z à v, et l'ensemble infini des chemins de z à z dans le cycle, on obtient un nombre infini de chemins de u à v. Ainsi, on peut supposer que le graphe est un graphe acyclique orienté (DAG).

Question 3. Considérons d'abord la famille de DAGs (G_i) où G_1 est le graphe sur les sommets $\{u, v\}$ avec l'arête (u, v), et G_{i+1} est le graphe avec un sommet u ayant une arête vers chacun des sommets de G_i. Il est clair que chaque G_i a exactement $i+2$ sommets. On démontre par récurrence que le nombre de chemins de u à v dans G_i est de 2^i : c'est vrai pour G_1 et pour G_i le nombre de chemins est un plus la somme du nombre de chemins dans chaque G_j pour $j < i$, c'est-à-dire $1 + \sum_{j<i} 2^j$ par hypothèse de récurrence avec prédécesseurs, soit 2^i. Cette famille remplit donc les conditions demandées.

Question 4. Si n est une puissance de 2, on utilise directement la construction de la question 3, et on a un graphe avec $2 + \log_2 n$ sommets qui réalise n.

Sinon, si n n'est pas une puissance de 2, on pose $k := \lfloor \log_2 n \rfloor$, et on construit G_k, qui a $2 + k$ sommets. Comme n n'est pas une puissance de 2, on sait alors que n a exactement $k + 1$ chiffres en écriture binaire. On ajoute un sommet u ayant des arêtes vers les sommets différents de v pour lesquelles

il y a un 1 dans l'écriture binaire de n. Le nombre de chemins de u à v est alors la somme des 2^j telle que le j-ème bit de n soit à 1, c'est-à-dire n. Le nombre de sommets est $3 + \lfloor \log_2 n \rfloor$, c'est-à-dire $2 + \lceil \log_2 n \rceil$ vu que n n'est pas une puissance de 2.

Question 5. Cette construction est presque optimale, à part pour $n = 1$ où on peut construire un graphe pointé avec un seul sommet ($u = v$) là où la borne en prévoit 2.

Pour montrer l'optimalité pour $n > 1$, montrons d'abord le résultat que, pour tout $i \in \mathbb{N}$, un graphe avec $i + 2$ sommets a au plus 2^i chemins de u à v. En effet, si l'on considère un graphe arbitraire à $i + 3$ sommets, si l'on considère u et les sommets $u_1, \ldots, u_k$ qu'il permet d'atteindre en une étape, et que l'on trie les u_i suivant un tri topologique (de sorte qu'il n'y ait jamais de chemin de u_q à u_p pour $p < q$), alors u_1 permet d'atteindre au plus i sommets (tout sauf u et u_1), u_2 permet d'atteindre au plus $i - 1$ sommets (tout sauf u, u_1, et u_2), etc., et on obtient la borne souhaitée. Une fois ce résultat montré, on voit que, pour tout $n > 0$, un graphe qui satisfasse les conditions demandées doit avoir au moins $\lceil \log_2 n \rceil + 2$, la borne que nous avons montrée.

Question 6. Comme à la question 2, on peut supposer sans perte de généralité que tous les états sont accessibles et co-accessibles et que l'automate n'a pas de cycle. Il est alors clair que l'état final n'a pas de transitions sortantes, car de telles transitions seraient vers un état qui ne pourrait être co-accessible à moins d'introduire un cycle. De la même manière, l'état initial n'a aucune transition entrante.

Pour montrer que chaque état non-final a deux transitions sortantes, notons d'abord que clairement un état non-final doit avoir au moins une transition sortante, sinon il n'est pas co-accessible. Ainsi, procédons par l'absurde et supposons qu'un état q a une unique transition sortante. Si c'est l'état initial, on le supprime et on prend pour nouvel état initial la cible de la transition. Le résultat est toujours un automate déterministe avec un unique état final, et le langage reconnu a la même cardinalité (on a retiré la lettre étiquetant la transition comme lettre initiale de tous les mots du langage). Ainsi, on a un automate qui réalise le même entier avec un état de moins, ce qui contredit la minimalité du nombre d'états.

Si l'état q n'est pas l'état initial, soit q' la cible de la transition. Considérons toutes les transitions vers q et remplaçons-les par des transitions vers q' (avec la même étiquette). Il est clair que le résultat est toujours un automate déterministe avec un seul état final, et les chemins acceptants correspondent : la seule chose qui pourrait changer la cardinalité du langage accepté est si le même mot se mettait à avoir plusieurs chemins acceptants, mais c'est impossible parce que l'automate est déterministe.

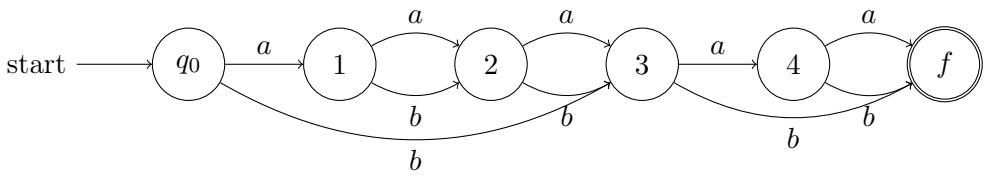
Question 7. On procède par induction :

- Si $n = 1$ alors on le réalise comme un automate avec 1 état.
- Si n est pair, alors en considérant l'automate A' pour $n/2$, on réalise n avec un nouvel état initial dont les deux transitions pointent vers l'état initial de A'
- Si n est impair, alors en considérant l'automate A' pour $n/2$, on réalise n avec un nouvel état initial avec une transition vers l'état final et une transition vers un nouvel état intermédiaire dont les deux transitions pointent vers l'état initial de A'.

Il est clair par induction que la construction est correcte. En termes de nombre d'états, pour $n = 1$ on a 1 état, et pour $n > 1$ le nombre d'états est celui pour $\lfloor n/2 \rfloor$ plus 1 ou 2 selon que le dernier bit de 1 est 1 ou 0. Ainsi, au total, on utilise $\lfloor \log_2 n \rfloor$ états plus un état par bit du nombre qui est à 1.

Question 8. C'est l'exponentiation rapide : on écrit g^n pour $n > 1$ impair comme $g \times g^{\lfloor n/2 \rfloor} \times g^{\lfloor n/2 \rfloor}$, et g^n pour n pair comme $g^{n/2} \times g^{n/2}$. Ainsi, un automate qui réalise l'entier n avec k états nous fournit un moyen de calculer g^n en k multiplications. Évidemment la construction de cette question fonctionne pour n'importe quel semi-groupe (elle ne nécessite pas d'inversibilité ou de neutre).

Question 9. La construction de la question 7 n'est pas optimale puisque l'automate pour $n = 15$ a 7 états alors qu'un automate à 6 états existe :



La question générale de trouver un automate réalisant un entier donné avec un nombre minimal d'états est encore largement ouverte. La formulation sous laquelle ce problème est habituellement étudiée est celle des *chaînes d'addition* [Wik18], voir notamment [Knu98] Section 4.6.3.

Références

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 2 : Seminumerical Algorithms.* Addison Wesley, 1998.

[Wik18] Wikipedia. Addition chain, 2018. `https://en.wikipedia.org/wiki/Addition_chain`.

A9 – Notions d'acyclicité pour les hypergraphes

Dans ce sujet, on représentera des *graphes* non-orientés comme un ensemble G d'éléments appelés *arêtes* qui sont soit des paires, soit des singletons, et on impose que si $\{x, y\}$ est une paire de G alors G contient aussi les singletons $\{x\}$ et $\{y\}$. Intuitivement, une paire $\{x, y\}$ représente une arête non-orientée entre x et y au sens usuel, et chaque singleton représente un sommet.

Un *hypergraphe* H est un ensemble d'ensembles non-vides appelés *hyperarêtes*. L'*ensemble de sommets* de H est $\mathrm{V}(H) := \bigcup_{E \in H} E$. Étant donné un hypergraphe H et un sous-ensemble $X \subseteq \mathrm{V}(H)$ de ses sommets, le *sous-hypergraphe induit par* X est l'hypergraphe $H[X] := \{E \cap X \mid E \in H\}$.

Question 0. On considère le graphe G_0 (avec 11 arêtes dont 6 singletons) et l'hypergraphe H_0 (avec 3 hyperarêtes dont 1 singleton) suivants. Décrire $G_0[\{2, 3, 4, 5\}]$ et $H_0[\{1, 5\}]$.

Question 1. Pour tout hypergraphe H, on note $\mathrm{M}(H)$ le sous-ensemble de H obtenu en retirant les arêtes qui ne sont pas maximales au sens de l'inclusion ; par exemple, $\mathrm{M}(H_0)$ est identique à H_0 sauf que l'on retire l'hyperarête $\{5\}$.

Un *cycle induit* dans un hypergraphe H est un n-uplet $(x_1, \ldots, x_n)$ d'éléments distincts de $\mathrm{V}(H)$ avec $n \geq 3$ tel que, pour $X := \{x_1, \ldots, x_n\}$, l'hypergraphe $\mathrm{M}(H[X])$ est exactement $\{\{x_i, x_{i+1}\} \mid 1 \leq i < n\} \cup \{\{x_n, x_1\}\}$.

G_0 contient-il un cycle induit ? H_0 contient-il un cycle induit ?

Question 2. Montrer qu'un graphe a un cycle induit si et seulement s'il a un cycle au sens usuel, c'est-à-dire une séquence de sommets $(v_1, \ldots, v_n)$ pour $n \geq 3$ tels que $\{\{v_i, v_{i+1}\} \mid 1 \leq i < n\}$ et $\{v_n, v_1\}$ sont des arêtes du graphe qui ne sont pas des singletons.

Question 3. Donner un exemple d'un hypergraphe H avec une hyperarête $E \in H$ tels que H ne contienne pas de cycle induit mais $H \setminus \{E\}$ en contienne un. Commenter.

Question 4. Le *graphe d'incidence* $\mathrm{I}(H)$ d'un hypergraphe H est le graphe qui a comme sommets les sommets de $\mathrm{V}(H)$ et les arêtes de H, et qui a comme arêtes non-singletons l'ensemble suivant : $\bigcup_{E \in H} \{\{x, E\} \mid x \in E\}$. On dit que H est *Berge-cyclique* si le graphe $\mathrm{I}(H)$ est cyclique (au sens usuel).

Montrer qu'un graphe est Berge-cyclique si et seulement s'il est cyclique au sens usuel.

Est-il possible qu'un hypergraphe ait un cycle induit mais ne soit pas Berge-cyclique ? Est-il possible qu'un hypergraphe soit Berge-cyclique mais n'ait pas de cycle induit ?

Question 5. Étant donné un hypergraphe H, on dit que $x \in \mathrm{V}(H)$ est une *feuille* si, en notant H_x l'hypergraphe $\{E \in H \mid x \in E\}$, on a $|\mathrm{M}(H_x)| = 1$. On note $H[-x] := H[\mathrm{V}(H) \setminus \{x\}]$.

Montrer que, pour tout hypergraphe H et feuille x de H, l'hypergraphe H a un cycle induit si et seulement si $H[-x]$ en a un.

Question 6. Montrer qu'il existe un hypergraphe H et une feuille x de H de sorte que H soit Berge-cyclique mais $H[-x]$ ne le soit pas. Commenter.

Suite des questions

Question 7. Montrer qu'un hypergraphe qui n'est pas Berge-cyclique et qui n'est pas vide contient toujours une feuille. Que dire de l'énoncé analogue pour un hypergraphe sans cycle induit ?

Corrigé

Question 0. On a $G_0[\{2,3,4,5\}] = \{\{2\},\{3\},\{4\},\{5\},\{2,5\},\{4,5\}\}$.
On a $H_0[\{1,5\}] = \{\{1\},\{5\}\}$.

Question 1. L'hypergraphe G_0 contient un cycle induit, par exemple $(1,2,5,4)$.

L'hypergraphe H_0 ne contient pas de cycle induit : si on suppose par l'absurde qu'il en contienne un, et qu'on pose le X correspondant, alors $H_0[X]$ a au plus 2 hyperarêtes non-singleton, mais la définition d'un cycle induit exigerait qu'il en contienne 3.

Question 2. Une implication est claire, car un cycle induit est un cycle au sens usuel. Montrons la réciproque. Soit G un graphe et $(v_1,\dots,v_n)$ un cycle de ce graphe au sens usuel. On prend un cycle de longueur minimale, et on va démontrer qu'il s'agit d'un cycle induit. Procédons par l'absurde et supposons que $(v_1,\dots,v_n)$ n'est pas un cycle induit. Si les v_i ne sont pas deux à deux distincts, alors on peut directement construire un cycle plus court à partir de $v_1,\dots,v_n$, ce qui contredit la minimalité ; donc il suffit de considérer le cas où les v_i sont deux à deux distincts. Si le cycle n'est pas induit, cela signifie alors qu'il existe une autre arête entre les v_i que celles du cycle, c'est-à-dire une arête de la forme $\{v_p, v_q\}$ où $|p-q| > 1$, et où $\{p,q\} \neq \{1,n\}$: on suppose sans perte de généralité que $p < q$. Dans ce cas, il est clair que $v_1,\dots,v_p,v_q,\dots,v_n$ est toujours un cycle : les sommets consécutifs sont bien adjacents, et la longueur est d'au moins 3 car on $n \geq 3$ et on ne peut avoir à la fois $p = 1$ et $q = n$. Or, ce cycle est plus court que le cycle initial, ce qui contredit l'hypothèse de minimalité. On conclut donc bien que tout cycle de longueur minimale est induit, ce qui démontre l'autre implication et conclut la preuve.

Question 3. L'observation clé est qu'un hypergraphe H contenant $\mathrm{V}(H)$ comme hyperarête ne contient jamais de cycle induit, puisque pour tout n-uplet $(x_1,\dots,x_n)$ avec $n \geq 3$, pour $X := \{x_1,\dots,x_n\}$, l'hypergraphe $H[X]$ contient X comme hyperarête, or $|X| > 2$.

Ainsi, on peut prendre n'importe quel hypergraphe avec un cycle induit, par exemple G_0, et poser $G_0' := G_0 \cup \{\mathrm{V}(G_0)\}$: c'est un hypergraphe sans cycle induit mais $G_0' \setminus \{\mathrm{V}(G_0)\}$ contient un cycle induit.

C'est un résultat contre-intuitif parce que ce phénomène ne peut pas se produire avec des graphes : ajouter une arête ne peut jamais faire disparaître un cycle.

Question 4. Pour tout graphe G, on sait que deux sommets $x, y \in \mathrm{V}(G)$ sont adjacents dans G si et seulement s'il existe un chemin de longueur 2 entre x et y dans $\mathrm{I}(G)$ (qui passe par le sommet de $\mathrm{I}(G)$ décrivant l'arête $\{x,y\}$ qui les relie). Ainsi, tout chemin dans G correspond à un chemin dans $\mathrm{I}(G)$, donc si G contient un cycle alors $\mathrm{I}(G)$ aussi. La réciproque est claire vu que tout cycle dans $\mathrm{I}(G)$ alterne nécessairement entre des sommets et des arêtes, et qu'il décrit donc un cycle dans G.

La réponse à la première question est non : si H a un cycle induit alors il est Berge-cyclique. En effet, soit $(x_1,\dots,x_n)$ un cycle induit de H, et posons $X := \{x_1,\dots,x_n\}$. Il existe donc des hyperarêtes $E_1,\dots,E_n$ de H, nécessairement deux à deux distinctes, telles que $\{x_i, x_{i+1}\} \subseteq E_i$ pour tout $1 \leq i < n$, et $\{x_n, x_1\} \subseteq E_n$. Ainsi, $\mathrm{I}[H]$ contient le cycle $x_1, E_1, x_2, E_2, \dots, x_n, E_n$.

La réponse à la seconde question est oui : H_0 est Berge-cyclique, puisque $\mathrm{I}(H_0)$ contient le cycle $2, \{2,4,5\}, 4, \{1,2,3,4\}$, pourtant on a vu à la question 1 qu'il ne contient pas de cycle induit.

Question 5. Une implication est triviale : si $H[-x]$ contient un cycle induit alors il s'agit aussi d'un cycle induit de H. Montrons donc l'autre implication. Observons d'abord que, si H contient un cycle induit $(x_1, \ldots, x_n)$ dans lequel la feuille x n'apparaît pas, alors il s'agit toujours d'un cycle induit de $H[-x]$. En effet, en notant $X = \{x_1, \ldots, x_n\}$, il est clair que $H[-x][X] = H[X]$. Ainsi, procédons par l'absurde et supposons que $x_i = x$ pour un certain $1 \leq i \leq n$. Mais l'existence du cycle induit témoigne alors de l'existence de deux hyperarêtes contenant x (une contenant x et x_{i-1}, ou x_n si $i = 1$; et une contenant x et x_{i+1}, ou x_1 si $i = n$) ; ces deux hyperarêtes sont nécessairement incomparables pour l'inclusion puisque, si l'une contenait l'autre, alors ce serait une hyperarête de H contenant trois sommets de X, ce qui est interdit par la définition d'un cycle induit. Ainsi, x apparaît dans deux hyperarêtes incomparables, ce qui contredit la définition d'une feuille. On a donc établi le résultat.

Question 6. Prenons $H = \{\{1, 2\}, \{1, 2, 3\}\}$. Cet hypergraphe est Berge-cyclique, comme en atteste le cycle $(1, \{1, 2\}, 2, \{1, 2, 3\})$. À présent, considérons $H[-3] = \{\{1, 2\}\}$: cet hypergraphe contient une seule arête donc ne peut pas contenir de Berge-cycle.

C'est un résultat contre-intuitif parce qu'on a montré à la question 5 que ce résultat ne peut pas se produire avec des graphes (où la notion de cycle induit coïncide avec la notion usuelle de cycle d'après la question 2) : retirer une feuille ne peut pas rendre un graphe acyclique.

Question 7. Indication : Commencer à montrer le résultat sur les graphes.

Soit H un hypergraphe non-vide sans feuille. On sait ainsi que, pour chaque sommet $x \in \mathrm{V}(H)$, il existe deux arêtes différentes qui contiennent x et sont incomparables pour l'inclusion. Soit $H' \subseteq H$ l'hypergraphe obtenu en ne conservant que ces arêtes. On a clairement $\mathrm{V}(H') = \mathrm{V}(H)$, et H' ne contient pas d'arêtes singleton. Montrons que H' a un Berge-cycle : ce sera alors aussi un Berge-cycle de H, ce qui conclut (car la notion d'hypergraphe Berge-cyclique est clairement stable par ajout d'arête, contrairement à la notion de cycle induit comme on l'a vu en question 3).

On sait que $\mathrm{I}(H')$ est un graphe où chaque sommet a degré au moins 2, puisque chaque sommet appartient à deux arêtes distinctes et chaque arête contient au moins deux sommets. Ainsi, il suffit de montrer l'affirmation pour les graphes : un graphe sans feuille (au sens classique de sommet de degré 1) et non-vide contient nécessairement un cycle (au sens classique, qui coïncide avec la notion de Berge-cycle).

On peut donc montrer le résultat sur les graphes, comme proposé dans l'indication. Construisons un chemin dans le graphe en partant d'un sommet quelconque x_1, un voisin x_2 de x_1, et à chaque étape on prend un voisin x_{i+1} du précédent sommet x_i qui est différent du voisin antérieur x_{i-1}. Dès qu'on arrive à un sommet x_{i+1} visité auparavant, c'est-à-dire $x_{i+1} = x_j$ pour $j < i - 1$, on arrête le processus, car il est clair qu'on a trouvé un cycle $x_j, \ldots, x_i$. Il faut simplement justifier qu'on parvient toujours à trouver un voisin x_{i+1} de x_i différent du sommet antérieur x_{i-1}. Mais si tel n'est pas le cas, cela voudrait dire que x_i est un sommet qui a un unique voisin, et donc qu'il apparaît dans une unique arête (non-singleton) : ce serait donc une feuille, ce qui contredit l'hypothèse.

L'énoncé analogue pour un hypergraphe sans cycle induit est faux : si on prend par exemple l'hypergraphe "tétraèdre" $H_7 = \{\{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}\}$, il est non-vide et n'a clairement pas de feuille (vu que chaque élément apparaît dans 3 arêtes incomparables pour l'inclusion) mais il n'a pas non plus de cycle induit car pour tout sous-ensemble X d'au moins trois sommets on sait que $H[X]$ contient une hyperarête de cardinalité 3.

[Pour davantage de détails sur les notions d'acyclicité pour les hypergraphes, on pourra par exemple consulter [BB14].]

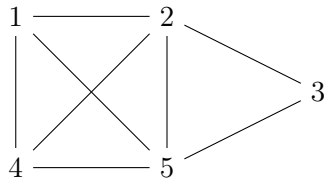
Références

[BB14] Johann Brault-Baron. Hypergraph acyclicity revisited, 2014. `https://arxiv.org/abs/1403.7076`.

A10 – Calcul de triangles

On considère un graphe non-orienté $G = (V, E)$ où $V = \{1, \ldots, n\}$ est l'ensemble des sommets et E est un ensemble d'arêtes qui sont des paires de sommets de V. Un *triangle* dans G est un sous-ensemble $\{x, y, z\} \in V$ tel que $\{x, y\}$, $\{y, z\}$, et $\{x, z\}$ appartiennent à E. Dans ce problème, on veut concevoir des algorithmes qui prennent en entrée un graphe et calculent (sans doublons) l'ensemble des triangles du graphe.

Question 0. Déterminer les triangles du graphe suivant :



Question 1. Étant donné un graphe G représenté comme une matrice d'adjacence, proposer un algorithme naïf en $O(|V|^3)$ pour déterminer l'ensemble des triangles de G.

Dans la suite du sujet, on supposera toujours que le graphe d'entrée est fourni sous forme de listes d'adjacence, et on supposera toujours que chacune de ces listes est triée.

Question 2. Étant donné un graphe G, proposer un algorithme en $O(|E| \times |V|)$ pour déterminer l'ensemble des triangles de G.

Question 3. Si l'on compare l'algorithme de la question 1 et celui de la question 2, lequel parmi ces algorithmes a la meilleure complexité ? Le choix de la représentation du graphe d'entrée était-il important ?

Question 4. Étant donné un graphe G, si l'on note Δ le degré maximal d'un sommet de G, proposer un algorithme en temps $O(|E| \times \Delta)$ pour déterminer l'ensemble des triangles de G. Commenter la performance de cet algorithme.

Question 5. Étant donné un graphe G, proposer un algorithme astucieux en $O\big(|E|^{3/2} + |V|\big)$ pour déterminer l'ensemble des triangles de G.

Suite des questions

Question 6. On reprend l'algorithme de la question 2 et on s'intéresse à la phase où l'on considère toutes les arêtes incidentes à un sommet i. Écrire précisément la complexité de cette phase en fonction du degré $d(i)$ de i et du degré de ses voisins. Proposer une amélioration pour énumérer la liste de tous les triangles contenant un sommet i en temps $O\left(\sum_{\{i,j\}\in E} d(j)\right)$.

Corrigé

Question 0. Les triangles sont :

— $\{1, 2, 4\}$
— $\{1, 2, 5\}$
— $\{1, 4, 5\}$
— $\{2, 3, 5\}$
— $\{2, 4, 5\}$

Question 1. *Demander un pseudo-code.* Soit $G = (V, E)$ le graphe d'entrée et $V = \{1, \ldots, n\}$ son ensemble de sommets. On énumère les triplets $1 \leq i < j < k \leq n$ avec trois boucles **for** imbriquées, et pour chaque triplet on teste en temps constant si $\{i, j\}$, $\{j, k\}$, et $\{i, k\}$ sont des arêtes, par trois accès à la matrice d'adjacence.

(Le seul piège est qu'il ne faut pas calculer de doublons, donc imposer $i < j < k$.)

En pseudo-code :

```
Pour i de 1 à n:
  Pour j de i+1 à n:
    Pour k de j+1 à n:
      Si M[i][j] et M[j][k] et M[i][k]:
        Produire {i, j, k}
      Fin Si
    Fin Pour
  Fin Pour
Fin Pour
```

Question 2. *Demander un pseudo-code.*

Indication : Montrer comment, étant donné deux listes triées L_1 et L_2, on peut calculer leur intersection en temps linéaire.

On passe en revue toutes les arêtes $\{i, j\}$ du graphe une unique fois : pour chaque sommet i, pour chaque sommet adjacent j, si $i < j$ alors on considère l'arête. Lorsque l'on considère une arête $\{i, j\}$, on détermine les k tels que $i < j < k$ et tels que (i, j, k) soit un triangle (on prend garde à éviter les doublons en prenant $j < k$) : ce sont les éléments plus grands que j qui sont dans l'intersection de la liste d'adjacence de i et de celle de j. La taille totale de ces listes est de $2\,|V|$ au plus, et on peut la déterminer en temps $|V|$ par parcours simultané des deux listes car elles sont triées par notre hypothèse initiale. Ainsi la complexité globale de l'algorithme est en $O(|E| \times |V|)$.

En code :

```
Pour u de 1 à n:
  Pour j dans L[i]:
    Si i < j:
      // Intersecter L[i] et L[j]
```

```
      pi = 0
      pj = 0
      Tant que pi < taille(L[i]) et pj < taille(L[j]):
        Si L[i][pi] < L[j][pj]:
          pi++
        Sinon Si L[i][pi] > L[j][pj]:
          pj++
        Sinon Si // L[i][pi] == L[j][pj]:
          k = L[i][pi]
          Si j < k:
            Produire {i, j, k}
          Fin Si
          pi++
          pj++
        Fin Si
      Fin Tant que
    Fin Si
  Fin Pour
Fin Pour
```

Question 3. La borne de $O(|V| \times |E|)$ est toujours plus favorable que celle de $O(|V|^3)$, puisque $|E| = O(|V|^2)$.

La représentation du graphe d'entrée importe peu, vu que l'on peut passer de listes d'adjacence à une matrice d'adjacence, ou vice-versa, en temps $O(|V|^2)$, ce qui est moins que la complexité des algorithmes que l'on a présenté. (Pour cela, noter qu'on peut supposer sans perte de généralité que $|E| \geq |V|$ quitte à supprimer les sommets qui n'apparaissent dans aucune arête, vu qu'ils ne peuvent pas faire partie d'un triangle.)

Question 4. L'algorithme de la question 2 fonctionne en temps $O(\Delta \times |E|)$: pour chaque arête, on considère deux listes de taille Δ et on les fusionne.

Autre façon de le voir, qui revient essentiellement au même : on énumère les sommets de G : pour chaque sommet i on considère chaque paire $i < j < k$ de voisins de i en itérant sur les paires d'éléments de la liste d'adjacence, et on teste en temps constant si j et k sont voisins. Il est clair que cet algorithme est correct. Son temps d'exécution est borné par $\sum_i d(i)^2$, ce qui est $\leq \Delta \sum_i d(i)$, ce qui est $\leq 2 \times \Delta \times |E|$ (la somme des degrés est deux fois le nombre d'arêtes, par double comptage). On a donc bien la complexité attendue.

En général la complexité $O(\Delta \times |E|)$ se ramène à celle de la question 2, puisque Δ peut être égal à V. Cependant, si le degré maximal Δ est petit, par exemple s'il est constant, on obtient un algorithme linéaire en G.

Question 5. **Indication :** Traiter différemment les sommets de fort degré et les sommets de faible degré en utilisant les questions 2 et 4.

Posons $H \subseteq V$ le sous-ensemble des sommets de V de degré $\geq \sqrt{|E|}$, appelés sommets *lourds*, et $L := V \setminus H$ l'ensemble des sommets *légers*. On peut calculer ces ensembles (représentés comme un vecteur de booléens) en temps linéaire en G. Noter que, si l'on considère juste les arêtes incidentes à H (chaque arête étant comptée au plus pour deux sommets de H), on a $|E| \geq \frac{|H| \times \sqrt{|E|}}{2}$, et ainsi $|H| \leq 2\sqrt{|E|}$.

On passe en revue les listes d'adjacence en temps linéaire en $|E|$, pour produire une nouvelle version des listes d'adjacence pour chaque sommet qui ne contient que ses voisins lourds : ces nouvelles listes sont toujours triées.

On énumère d'abord les triangles contenant au moins un sommet lourd. Pour ce faire, on considère chaque arête $\{i, j\}$ du graphe avec $i < j$, et ensuite on passe en revue la liste des sommets lourds adjacents à i et à j et on les intersecte comme en question 2. Pour éviter les doublons, on ne produit un triangle $\{i, j, k\}$ que pour un sommet lourd k de l'intersection où k est le sommet lourd de plus fort indice parmi $\{i, j, k\}$, c'est-à-dire si $i < k$ si $i \in H$ et $j < k$ si $j \in H$.

En pseudo-code, en notant $LH[i]$ la liste triée des sommets lourds adjacents à i :

```
Pour u de 1 à n:
  Pour j dans L[i]:
    Si i < j:
      Pour k dans LH[i] inter LH[j]: // intersection comme en question 2
        // on sait que k est lourd
        Si ((i est léger ou i < k) ET (j est léger ou j < k)):
          Produire {i, j, k}
    Fin Si
  Fin Pour
Fin Pour
```

Il est clair que tout triangle énuméré ici est bien un triangle et contient un sommet lourd ; à l'inverse pour tout triangle contenant un sommet lourd, en prenant le sommet lourd d'indice maximal du triangle, il va être produit quand on considère l'arête qui ne le contient pas. Par ailleurs, il est clair qu'on ne produit pas deux fois le même triangle : chaque triangle avec un unique sommet lourd est produit une seule fois, et pour les autres il n'est produit que quand on choisit pour k le sommet lourd de plus fort indice. La complexité de cette étape est en $O(|E|^{3/2})$: il y a $|E|$ arêtes considérées, et pour chaque arête il y a $\leq 2\sqrt{|E|}$ sommets lourds considérés.

On énumère à présent les triangles ne contenant que des sommets légers. Pour ce faire, on calcule les listes d'adjacence pour les sommets légers en $O(|E|)$, et on utilise directement la question 2 sur le sous-graphe formé des sommets légers (c'est-à-dire qu'on utilise seulement la liste d'adjacence des sommets légers : on ne fait pas d'intersections du tout pour des arêtes qui contiennent un sommet lourd). On obtient pour cette étape une complexité de $O(\sqrt{|E|} \times |E|)$, soit $O(|E|^{3/2})$, d'après l'analyse de la question 4, puisque $\Delta \leq \sqrt{|E|}$ sur le sous-graphe des sommets légers. Ainsi la complexité globale est $O(|E|^{3/2})$.

[C'est l'algorithme heavy-light, cf [IR78] et [AYZ97].]

Question 6. Quand on traite le sommet i et qu'on cherche les triangles contenant i, on considère chaque arête $\{i, j\}$ incidente à i (où $i < j$), et ensuite on intersecte les listes d'adjacence de i et de j. La complexité est donc en $O\left(\sum_{\{i,j\}\in E} d(i) + d(j)\right)$, soit en $O\left(d(i)^2 + \sum_{\{i,j\}\in E} d(j)\right)$.

Pour obtenir la complexité demandée, on va changer l'algorithme de la question 2 comme suit : quand on traite i on marque tous les voisins j de i (en temps $d(i)$), puis pour chaque voisin marqué j on passe en revue sa liste d'adjacence (en temps $d(j)$) et si on y trouve un sommet marqué k on produit le triangle $\{i, j, k\}$, puis finalement on retire la marque de j. Autrement dit, au lieu d'intersecter les listes de chaque j avec celle de i, on passe en revue la liste de j et on teste l'appartenance à la liste de i en l'ayant représentée comme un vecteur de booléens. Noter qu'on n'a même plus besoin que les listes d'adjacence soient triées quand on procède comme cela.

Il est clair que les triplets produits par cet algorithme sont effectivement des triangles contenant i, et inversement, pour tout triangle i, j, k, quand on considère le premier sommet parmi j, k à être traité, l'autre est encore marqué donc le triangle est produit : ainsi l'algorithme produit-il bien la liste des triangles contenant i. Il est clair que chaque triangle contenant i est produit une unique fois : quand on produit un triangle $\{i, j, k\}$, les deux sommets j, k étaient marqués, et d'ici à ce qu'on considère le deuxième, le premier aura perdu sa marque.

On produit ainsi les triangles contenant i en complexité $O(d(i)+\sum_{\{i,j\}\in E} d(j))$, soit $O(\sum_{\{i,j\}\in E} d(j))$ parce que $d(j) \geq 1$ pour tout voisin j de i.

(Noter qu'on produit tous les triangles contenant i, là où en question 2 on imposait $i < j$ pour éviter les doublons. Ceci est délibéré, parce qu'en question 7 on voudra effectivement produire tous les triangles contenant i sans imposer $i < j$.)

Références

[AYZ97] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3) :209–223, 1997. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.4120&rep=rep1&type=pdf`.

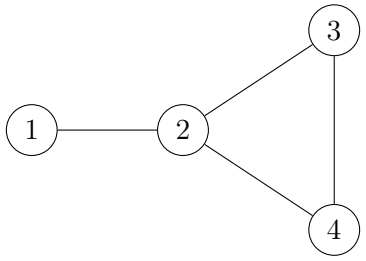
[IR78] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4) :413–423, 1978. `https://www.researchgate.net/profile/Michael_Rodeh/publication/220618165_Finding_a_Minimum_Circuit_in_a_Graph/links/0912f50947e332288e000000.pdf`.

A11 – Dégénérescence de graphes

On considère un graphe non-orienté $G = (V, E)$ où V est l'ensemble des sommets et E est un ensemble d'arêtes qui sont des paires de sommets de V.

Un *sous-graphe* H de G est un graphe (V', E') où $V' \subseteq V$ et $E' \subseteq E$. Pour $k \in \mathbb{N}$, on dit qu'un graphe non-vide G est *k-dégénéré* si tout sous-graphe non-vide de G contient un sommet de degré au plus k dans H. La *dégénérescence* de G est le plus petit $k \in \mathbb{N}$ tel que G est k-dégénéré.

Question 0. Montrer que le graphe G_0 suivant est 2-dégénéré. Quelle est sa dégénérescence ?



Question 1. Pour tout graphe G, on note Δ_G le degré maximal de G. Montrer que tout graphe G est Δ_G-dégénéré. Est il vrai que tout graphe G est de dégénérescence Δ_G ?

Question 2. Un sous-graphe $H = (V', E')$ de G est appelé *sous-graphe induit* si E' est la restriction de E à V', i.e., $E' = \{\{u, v\} \in E \mid u, v \in V'\}$. Montrer que, dans la définition d'un graphe k-dégénéré et de la dégénérescence, il suffit de considérer les sous-graphes induits.

Question 3. Donner le pseudo-code d'un algorithme naïf pour calculer la dégénérescence d'un graphe, et en préciser la complexité en temps.

Question 4. Caractériser les graphes de dégénérescence 1.

Question 5. Un graphe G est dit *régulier* si tous ses sommets ont le même degré. Montrer que tout graphe régulier G est de dégénérescence Δ_G. La réciproque est-elle vraie ?

Question 6. Pour $k \in \mathbb{N}$, un *k-arrangement* d'un graphe G est un ordre total sur ses sommets $v_1 < \cdots < v_n$ tel que, pour tout $1 \leq i \leq n$, on ait $|\{v_j \in V \mid j < i \wedge \{v_i, v_j\} \in E\}| \leq k$. Montrer qu'un graphe a un k-arrangement si et seulement s'il est k-dégénéré.

Question 7. Proposer un algorithme en temps linéaire pour calculer la dégénérescence.

Corrigé

Question 0. Les sous-graphes qui contiennent le sommet 1 contiennent un sommet de degré 1 donc ne posent pas problème. Pour les sous-graphes qui ne contiennent pas 1, le degré maximal est forcément 2 car 2 a perdu son arête incidente avec 1, ainsi ils contiennent toujours un sommet de degré au plus 2.

La dégénérescence du graphe est donc 2 ou moins, mais ce n'est clairement pas 1 ou moins parce que le sous-graphe $\{2, 3, 4\}$ ne contient aucun sommet de degré 1. Ainsi, la dégénérescence de G_0 est 2.

Question 1. Pour tout sous-graphe $H = (V', E')$ de G, il est clair que le degré de n'importe quel sommet de V' dans H n'est pas plus grand que son degré dans G, donc n'importe quel sommet de H a degré $\leq \Delta_G$. On conclut car H est non-vide.

La réciproque est fausse et G_0 est un contre-exemple puisqu'il est de dégénérescence 2 alors que son degré maximal est 3.

Question 2. Il est clair que si un graphe est k-dégénéré pour la définition originale alors il l'est toujours pour la définition modifiée. Pour la réciproque, il suffit de montrer que pour tout graphe G et tout $k \in \mathbb{N}$, si tout sous-graphe induit non-vide $H' = (V', E')$ de G contient un sommet de degré $\leq k$, alors c'est le cas de tout un sous-graphe non-vide H'' de G. Prenons $H'' = (V', E'')$ un tel sous-graphe, et prenons $H' = (V', E')$ le sous-graphe induit de G pour l'ensemble V' de sommets de H''. On sait par hypothèse que H' contient un sommet de degré $\leq k$. Ainsi, c'est également le cas de H'', parce que le degré de tout sommet de V' dans H'' est toujours inférieur ou égal à son degré dans H'.

Comme la notion de graphe k-dégénéré ne change pas quand on ne considère que les sous-graphes induits, il en va de même pour la dégénérescence.

Question 3. Il est clair que la dégénérescence de k est le maximum, sur tous les sous-graphes non-vides de G, du degré minimal du sous-graphe. L'algorithme naïf est d'appliquer bêtement cette définition, en considérant chaque sous-graphe non-vide en force brute. Voici un pseudo-code avec des listes d'adjacence :

```
Fonction degeneracy():
  Conserve := tableau de n booléens


  Fonction degre_min_sousgraphe():
    // Calcule le degré minimal du sous-graphe Conserve
    // S'il est vide, on renvoie n
    degre_min := n
    Pour i de 1 à n:
      Si Conserve[i]:
        degre = 0
        Pour j adjacent à i:
          Si Conserve[j]:
            degre++
          Fin Si
        Fin Pour
        degre_min := min(degre, degre_min)
      Fin Si
    Fin Pour
    Renvoie degre_min
  Fin Fonction
```

```
Fonction explore(i):
  Si i > n:
    // On a fini de choisir le sous-graphe dans Conserve
    Renvoie degeneracy()
  Fin Si
  degeneracy_max = -1
  // J'essaie en jetant i...
  Conserve[i] := 0
  degeneracy_max = max(degeneracy_max, explore(i+1))
  // ... et en gardant i...
  Conserve[i] := 1
  degeneracy_max = max(degeneracy_max, explore(i+1))
  Renvoie degeneracy_max
Fin Fonction

Renvoie explore(1)
```

La complexité en temps est en $O\left(2^{|V|}(|E|+|V|)\right)$. C'est très mauvais évidemment, et on va voir en question 7 qu'on peut faire bien mieux.

Question 4. Montrons qu'un graphe est de dégénérescence 1 si et seulement s'il a des arêtes et n'a pas de cycle, c'est-à-dire si c'est une forêt. Pour prouver la première direction, un graphe sans arêtes est manifestement de dégénérescence 0 ; et si un graphe a un cycle, alors en se restreignant au cycle, on a un sous-graphe où chaque sommet a degré au moins 2, comme en témoignent les arêtes du cycle, donc la dégénérescence est ≥ 2.

Pour prouver l'autre implication, considérons un graphe de dégénérescence $\neq 1$. Si la dégénérescence est 0, alors le graphe ne peut pas avoir d'arête (sinon on prend un sous-graphe formé de deux sommets adjacents, où le degré minimal est 1). Si la dégénérescence est ≥ 2, alors comme elle n'est pas 1, il doit y avoir un sous-graphe G' de G qui n'a aucun sommet de degré ≤ 1. Il suffit de montrer qu'un tel graphe a nécessairement un cycle. Pour ce faire, on part d'un sommet quelconque, et à chaque étape on se déplace vers un voisin différent du sommet par lequel on est arrivé. On n'est jamais coincé¸ parce que chaque sommet a degré ≥ 2 donc a au moins deux voisins. Par le principe des tiroirs, on finit forcément par repasser par un sommet qu'on a déjà vu, et ainsi on obtient un cycle.

Question 5. Si G est régulier, alors tous les sommets ont le degré maximal Δ_G, donc en prenant G comme sous-graphe on a un sous-graphe où le degré minimal est de Δ_G, ainsi la dégénérescence est-elle $\geq \Delta_G$. Comme G est Δ_G-dégénéré d'après la question 1, on sait donc que la dégénérescence est égale à Δ_G. En fait, il suffit que G ait une composante connexe qui soit Δ_G-régulière : en prenant cette composante comme sous-graphe, on peut conclure de la même manière.

La réciproque du résultat énoncé en question 5 est donc fausse : si l'on considère par exemple un graphe Δ_G-régulier auquel on ajoute un sommet isolé, la dégénérescence sera de Δ_G comme on vient de l'expliquer, alors que le graphe tout entier n'est pas Δ_G-régulier.

On peut en revanche montrer que, si la dégénérescence est de Δ_G, alors il y a une composante connexe du graphe qui est Δ_G-régulière. Prenons un sous-graphe induit G' où le degré minimal est de Δ_G, qui soit minimal au sens de l'inclusion. Tous les sommets de ce graphe G' sont de degré Δ_G (puisque c'est également le degré maximal possible dans G, donc dans un sous-graphe de G), ainsi G' est Δ_G-régulier : et ce graphe est connexe sinon on pourrait prendre une de ses composantes connexes comme choix de sous-graphe pour G', ce qui contredirait la minimalité de G'. Ainsi, vu que Δ_G est le degré maximal, on sait que G' est une composante connexe de G : il n'y a aucune arête dans G qui puisse contenir un

sommet de G' à part les arêtes de G' (qui réalisent déjà le degré maximal), et G' est connexe comme on vient de le dire. Ainsi, G contient une composante connexe qui est Δ_G-régulière.

Question 6. Étant donné un k-arrangement $v_1 < \cdots < v_n$ de G, montrons que G est k-dégénéré. Soit G' un sous-graphe non-vide de G, et considérons le dernier sommet de G' dans l'ordre $<$. On sait que tous ses voisins dans G' sont plus petits que lui dans $<$, donc il y en a au plus k, et ainsi G' contient un sommet de degré $\leq k$. Ceci est vrai pour tout G', donc G est bien k-dégénéré.

Pour la réciproque, montrons par induction sur le nombre de sommets de G que si G est k-dégénéré alors il a un k-arrangement. Pour un graphe à 1 sommet, le résultat est trivial : G est k-dégénéré et a un k-arrangement pour tout $k \in \mathbb{N}$. Pour la récurrence, soit G un graphe à $n+1$ sommets et supposons qu'il soit k-dégénéré. On sait donc, en prenant G comme sous-graphe, que G contient un sommet v_{n+1} de degré $\leq k$. Considérons le graphe G' obtenu en retirant v_{n+1} de G et toutes ses arêtes incidentes. Ce graphe G' est toujours k-dégénéré, car tout sous-graphe non-vide de G' est un sous-graphe non-vide de G ; et G' a n sommets. Par hypothèse de récurrence, il existe un k-arrangement $v_1 < \cdots < v_n$ des sommets de G'. Considérons l'ordre total $v_1 < \cdots < v_n < v_{n+1}$ sur les sommets de G. C'est un k-arrangement : la propriété est préservée pour les sommets de 1 à n, et pour v_{n+1} elle est vraie car son degré dans G est $\leq k$ par définition. On a donc démontré le sens réciproque par récurrence, ce qui conclut la preuve de l'équivalence.

Question 7. En utilisant la question 5, on va calculer un ordre total sur les sommets de G qui est un k-arrangement pour la plus grande valeur possible de k, avec l'algorithme suivant :

```
D = tableau d'entiers à n cases initialisé à 0
P = tableau de booléens à n cases initialisé à faux
L = liste vide

// Initialiser les degrés
Pour chaque arête u - v:
  D[u]++
  D[v]++
Fin Pour

k = 0

Pour tour de 1 à n:
  // Trouver le sommet de plus faible degre
  degre_min = n
  sommet_degre_min = -1
  Pour i de 1 à n:
    Si P[i] = faux:
      Si D[i] < degre_min:
        degre_min = D[i]
        sommet_degre_min = i
      Fin Si
    Fin Si
  Fin Pour

  // Ajouter sommet_degre_min en tête de liste
  // (donc on la construit en partant de la fin comme à la question 6)
  L := sommet_degre_min::L
  P[i] := true
```

```
    // Mettre à jour k
    k = max(k, degre_min)

    // Mettre à jour les degrés
    Pour chaque voisin v de sommet_degre_min:
      Si P[v] = faux:
        D[v]--
      Fin Si
    Fin Pour
  Fin Pour
```

Il est clair que cet algorithme calcule une liste L qui réalise un ordre total des sommets du graphe, et que c'est effectivement un k-arrangement, vu que k est le max, sur chaque position i, du nombre de voisins de v_i qui n'ont pas encore été pris (donc qui vont apparaître avant lui dans la liste).

Il faut justifier que cet algorithme est optimal, c'est-à-dire que la valeur de k qu'il réalise est bien la plus petite possible. Pour justifier de cela, prenons un ordre total quelconque $v_1 < \cdots < v_n$ qui réalise la meilleure valeur de k possible, et justifions qu'on peut le modifier pour être de la forme trouvée par l'algorithme. Noter qu'on peut éliminer le suffixe où cette solution coïncide avec la liste que renvoie l'algorithme, donc considérons le plus grand préfixe $v_1 < \cdots < v_p$ tel que v_p n'est pas le sommet que l'algorithme a retenu. Soit $v_q < v_p$ le sommet que l'algorithme a retenu à la place. Si l'on modifie $v_1 < \cdots < v_p$ pour prendre le sommet v_q et le placer à la fin, on sait que la valeur de k réalisée à la fin n'est pas pire, puisque v_q a un degré dans $\{v_1, \ldots, v_p\}$ qui n'est pas plus grand que v_p, par minimalité. Pour ce qui est des autres positions, pour chaque sommet de la nouvelle liste, il y a moins de voisins avant lui dans la nouvelle liste que dans la liste originale (vu qu'on a juste retiré v_q). Ainsi, la valeur de k réalisée par la nouvelle liste n'est pas pire qu'avant. En itérant cet argument, on justifie que la liste optimale peut être récrite en la liste choisie par l'algorithme sans que la valeur de k n'augmente, donc l'algorithme calcule bien la valeur optimale de k.

Il reste à analyser la complexité. L'algorithme tel qu'écrit a complexité $O(|V|^2)$: on fait $|V|$ tours de la boucle Pour extérieure, et pour chacun on fait $|V|$ tours de la première boucle Pour ; la boucle Pour de mise à jour des degrés a comme complexité globale $O(|E|)$ sur la totalité des tours de la boucle Pour extérieure, donc elle n'intervient pas dans l'analyse.

Pour que l'algorithme s'exécute en temps linéaire, il faut l'optimiser avec une *bucket queue* : on a un tableau Q contenant $|V|$ listes doublement chaînées, de sorte que $Q[i]$ est la liste des sommets restants de degré i. Une fois les degrés initialisés, on remplit ces queues en temps $O(|V|)$. On conserve aussi, pour chaque sommet v, un pointeur $O[v]$ vers l'unique occurrence de v dans une liste chaînée du tableau Q, pour pouvoir la supprimer efficacement.

On n'utilise plus le tableau P, et on remplace la première boucle Pour intérieure dans la boucle Pour extérieure par le code qui suit :

```
...
degre_min = n
sommet_degre_min = -1
Pour d de 1 à n:
  Si Q[d] est non vide:
    degre_min = d
    h = Tête de la liste Q[d]
    Retirer le premier élément de Q[d]
    sommet_degre_min = h
    Sortir de la boucle Pour
  Fin Si
```

```
Fin Pour
...
```

On remplace l'affectation à P par :

```
...
O[p] := nil
...
```

Pour la mise à jour des degrés, on fait :

```
...
Pour chaque voisin v de sommet_degre_min:
  Si O[v] != nil:
    D[v]--
    Ajouter v en tête de Q[v-1]
    Supprimer l'élément O[v] de Q[v]
  Fin Si
Fin Pour
...
```

La complexité de cette dernière boucle est toujours $O(|E|)$ au total sur toutes les itérations de la boucle Pour extérieure. En ce qui concerne la nouvelle version de la boucle Pour interne ci-dessus, la complexité totale est également en $O(|E|)$: à chaque fois qu'on va trouver un sommet v, on paie le degré actuel de v à cette étape, et donc le coût total payé est de l'ordre de la somme des degrés dans le graphe original, soit $O(|E|)$. Ainsi, le coût total de cet algorithme est à présent en $O(|E|)$, soit linéaire en le graphe.

Si on ne veut pas utiliser de listes doublement chaînées et le pointeur O, on peut alternativement utiliser des listes simplement chaînées, choisir de laisser les vieilles occurrences de sommets dans Q, et continuer à utiliser le tableau P pour les marquer ces sommets comme visités (et ne pas utiliser O). Ainsi, quand on sort une tête d'une liste de Q dans la première boucle Pour interne, si le sommet a en fait déjà été pris d'après P, on le jette et on continue à itérer sur la liste de Q (puis sur d'autres listes de Q si nécessaire). Ces occurrences parasites n'ont pas d'impact sur la complexité, parce que le nombre total d'occurrences de sommets qu'on va considérer et puis jeter est encore de l'ordre de la somme des degrés (chaque sommet v apparaît autant de fois au plus que son degré), donc leur contribution totale au temps d'exécution est bornée aussi par $2\,|E|$.

[Pour plus d'information sur la dégénérescence, on peut consulter Wikipedia [Wik18] et les références qui y sont citées.]

Références

[Wik18] Wikipedia. Degeneracy (graph theory), 2018. `https://en.wikipedia.org/wiki/Degeneracy_(graph_theory)`.

A12 – Plus grands facteurs

On fixe l'alphabet $\Sigma := \{a, b\}$. Pour un langage L sur Σ, étant donné un mot $w \in \Sigma^*$, on veut calculer la longueur du plus grand facteur de w dans le langage L, notée $\mathrm{lpgf}(L, w)$; on convient que cette longueur est de -1 si aucun facteur de w n'est dans L. On a donc $-1 \leq \mathrm{lpgf}(L, w) \leq |w|$ où $|w|$ dénote la longueur du mot w.

Question 0. On considère le langage L_0 défini par l'expression rationnelle $(ab)^*$, et le mot $w_0 := baabababa$. Calculer $\mathrm{lpgf}(L_0, w_0)$.

Question 1. Étant donné un langage L, représenté comme un automate fini déterministe complet, et un mot $w \in \Sigma^*$, proposer un algorithme naïf pour calculer $\mathrm{lpgf}(L, w)$. Donner un pseudo-code pour cet algorithme, et déterminer sa complexité en temps et en espace.

Question 2. Proposer un algorithme plus efficace pour ce problème qui s'exécute en temps et en espace $O(|Q| \times |w|)$. Est-il important que l'automate soit déterministe ?

Question 3. On dit que $w' \in \Sigma^*$ est un *sous-mot* de w s'il existe une fonction strictement croissante ϕ de $\{1, \ldots, |w'|\}$ dans $\{1, \ldots, |w|\}$ telle que $w'_i = w_{\phi(i)}$ pour tout $1 \leq i \leq |w'|$, où w'_i dénote la i-ème lettre de w' et de même pour w.

Étant donné un langage L représenté comme un automate et un mot w, comment calculer la longueur du plus grand *sous-mot* de w dans le langage L ?

Question 4. On pose le langage $L_4 := \{a^n b^n \mid n \in \mathbb{N}\}$. Proposer un algorithme efficace qui calcule $\mathrm{lpgf}(L_4, w)$ pour un mot d'entrée w.

Question 5. On pose le langage $L_5 := \{uu \mid u \in \Sigma^*\}$. Proposer un algorithme qui calcule $\mathrm{lpgf}(L_5, w)$ pour un mot d'entrée w en temps $O(|w|^2)$.

Question 6. On pose le langage L_6 des mots bien parenthésés sur Σ défini inductivement comme suit : le mot vide ϵ est bien parenthésé, la concaténation de deux mots bien parenthésés est bien parenthésée, et si $w \in \Sigma^*$ est bien parenthésé alors awb l'est aussi. Proposer un algorithme efficace qui calcule $\mathrm{lpgf}(L_6, w)$ pour un mot d'entrée w.

Corrigé

Question 0. Il est clair que $(ab)^3$ est un facteur de w_0 mais que $(ab)^4$ ne l'est pas, donc $\mathrm{lpgf}(L_0, w_0)$ est 6.

Question 1. On a d'abord l'algorithme naïf de complexité en temps cubique (et de complexité en espace constante) :

```
T := tableau de taille n contenant les lettres du mot (de 0 à n-1)
lpgf := -1

// i inclus, j exclu, pour également tester le mot vide
Pour i de 0 à n-1:
  Pour j de i à n:
    q := q_0 // état initial de l'automate
    Pour k de i à j-1:
      q := delta(T[k], q) // fonction de transition de l'automate
    Fin Pour
    Si F[q]: // états finaux de l'automate
      lpgf := max(lpgf, j-i)
    Fin Si
  Fin Pour
Fin Pour

Renvoie lpgf
```

On peut rendre l'algorithme quadratique (et conserver la complexité constante en espace) en changeant la boucle comme suit :

```
...
Pour i de 0 à n-1:
  q := q_0 // état initial de l'automate
  Si F[q]:
    lpgf := max(lpgf, 0)
  Fin Si
  Pour j de i à n-1:
    q =: delta(T[k], q) // fonction de transition de l'automate
    Si F[q]: // états finaux de l'automate
      lpgf := max(lpgf, j-i+1)
    Fin Si
  Fin Pour
Fin Pour
...
```

Question 2. On considère un automate non-déterministe $A = (Q, I, F, \delta)$, et on construit son graphe produit avec le mot w, c'est-à-dire le graphe ayant pour sommets $\{(q, i) \mid q \in Q, i \in \{0, \ldots, |w|\}\}$ et ayant, pour chaque $0 \leq i < |w|$ et chaque transition (q, a, q') où a est la lettre à la position $i + 1$ de w, une arête de (i, q) à $(i + 1, q')$. Un sommet est dit *initial* si sa seconde composante est dans I, et *final* si sa seconde composante est dans F. Il est clair que, pour tous $q, q' \in Q$ et $0 \leq i \leq j \leq |w|$, il existe un chemin de (q, i) à (q', j) dans le graphe si et seulement s'il existe un chemin dans A de q à q' étiqueté par $w[i + 1 \cdots j]$. Ainsi, il y a un facteur de longueur k accepté par l'automate ss'il y a un chemin de

longueur k d'un sommet initial à un sommet final. Il suffit donc de calculer le plus long chemin dans ce graphe acyclique, ce que l'on peut faire facilement par programmation dynamique, et on en déduit la longueur du plus long facteur accepté par l'automate.

```
T := tableau de taille n contenant les lettres du mot
D := tableau d'entiers de taille (n+1) fois |Q|
lpgf := -1

Pour i de 0 à n:
  Pour q de 0 à |Q|-1:
    Si F[q]:
      D[i][q] := 0
    Sinon:
      D[i][q] := -1
    Fin Si
  Fin Pour
Fin Pour

Pour i de (n-1) à 0:
  Pour q de 0 à |Q|-1:
    Pour chaque transition (q, T[i], q') partant de q:
      Si D[i+1][q'] >= 0:
        D[i][q] := max(D[i][q], 1+D[i+1][q'])
      Fin Si
    Fin Pour
    Si I[q]: // q est initial
      lpgf := max(lpgf, D[i][q])
  Fin Pour
Fin Pour

Renvoie lpgf
```

Pour un automate déterministe, la complexité en temps et en mémoire est de $O(|Q| \times |w|)$ comme annoncé. Pour un automate non-déterministe, la complexité en temps passe à $O(|\delta| \times |w|)$ (en supposant que chaque état de Q apparaît dans δ, puisque si tel n'est pas le cas on peut supprimer les autres états sans perte de généralité). On peut aboutir à une meilleure complexité en espace, i.e., $O(|Q|)$, en ne conservant le tableau $D[i]$ que pour la valeur courante et la valeur précédente de i.

Question 3. On procède comme en question 2 mais on s'autorise à sauter des lettres. Concrètement, quand on calcule `L[i][q]`, on ajoute l'instruction :

```
...
L[i][q] := max(L[i][q], L[i+1][q])
...
```

Question 4. On peut écrire w en regroupant les a contigus et les b contigus, i.e., $w = a^{p_1}b^{q_1} \cdots a^{p_k}b^{q_k}$, où p_1 et q_k sont éventuellement nuls. Il est clair qu'un facteur de w de la forme $a^n b^n$ pour un certain $n \in \mathbb{N}$ doit être formé d'un suffixe (non nécessairement strict) d'un groupe de a dans cette décomposition, et d'un préfixe (non nécessairement strict) du groupe suivant de b dans cette décomposition : et ainsi $\mathrm{lpgf}(L_4, w) = \max_{1 \leq i \leq k} \min(p_i, q_i)$.

En pseudo-code :

```
T := tableau de taille n contenant les lettres du mot
p := 0
lpgf := -1

Tant que p < n:
  n_a := 0
  n_b := 0
  Tant que p < n et T[p] == 'a':
    n_a++
    p++
  Fin Tant que
  Tant que p < n et T[p] == 'b':
    n_b++
    p++
  Fin Tant que
  lpgf := max(lpgf, min(n_a, n_b))
Fin Tant que

Renvoie lpgf
```

Question 5. L'algorithme naïf en temps cubique est de tester chaque facteur (de taille paire), et vérifier si c'est un carré : on vérifie si le facteur entre i inclus et j exclu est un carré en lisant simultanément les $T[i+k]$ et les $T[i+(j-i)/2+k]$ pour k de 0 inclus à $(j-i)/2$ exclu et en vérifiant que c'est le même mot.

Pour le faire en temps quadratique, on peut plus astucieusement tester chaque décalage possible (la moitié de la longueur du facteur), et lire avec ce décalage pour voir si un facteur convient. En pseudo-code :

```
T := tableau de taille n contenant les lettres du mot
lpgf := 0 // le mot vide est toujours un facteur

Pour d de 1 à \lfloor n/2 \rfloor:
  n_ok := 0
  Pour i de 0 à n-d-1:
    Si T[i] == T[i+d]:
      n_ok++
    Sinon:
      n_ok := 0
    Fin Si
    Si n_ok == d:
      // T[i-d+1..i] == T[i+1..i+d] (bornes incluses)
      // donc on a trouvé un facteur carré de longueur 2d
      lpgf := 2d
      Sortir de la boucle Pour interne // optimisation
    Fin Si
  Fin Pour
Fin Pour

Renvoie lpgf
```

[Il existe un algorithme plus sophistiqué pour résoudre ce problème en temps linéaire [GS04].]

Question 6. On remarque d'abord qu'on peut tester en temps linéaire si un mot est bien parenthésé en vérifiant que chaque préfixe contient au moins autant de a que de b, et que le mot au total contient autant de a que de b. En pseudo-code :

```
T := tableau de taille n contenant les lettres du mot
d := 0

Pour i de 0 à n-1:
  Si T[i] == 'a':
    d++
  Sinon // T[i] == 'b'
    d--
  Fin Si
  Si d < 0:
    Renvoie faux
  Fin Si
Fin Pour

Renvoie (d == 0)
```

L'algorithme très naïf en $O(n^3)$ est donc de tester tous les facteurs, et de tester en temps linéaire pour chaque facteur s'il est bien parenthésé.

L'algorithme un peu moins naïf en temps $O(n^2)$, comme en question 1, est de tester toutes les positions de départ :

```
T := tableau de taille n contenant les lettres du mot
lpgf := 0 // le mot vide est toujours un facteur

Pour i de 0 à n-1:
  d := 0
  Pour j de i à n-1:
    Si T[j] == 'a':
      d++
    Sinon: // T[j] == 'b'
      d--
    Fin Si
    Si d == 0:
      lpgf := max(lpgf, j-i+1)
    Fin Si
    Si d < 0:
      Sortir de la boucle Pour interne
    Fin Si
  Fin Pour
Fin Pour

Renvoie lpgf
```

Pour un algorithme en temps linéaire, l'idée générale est de parcourir le mot une seule fois avec une pile où on conserve les parenthèses ouvrantes (a) et leur position : ainsi, quand on lit une parenthèse fermante (b), on peut mettre à jour lpgf en considérant le plus long facteur bien parenthésé qui se finit à cet endroit. Le piège est que, sur un mot comme $abab$, à la lecture du dernier b, le plus long facteur

n'est pas celui qui commence à la parenthèse ouvrante correspondante (i.e., *ab*), mais c'est *abab*. Ainsi, plutôt que de mettre les parenthèses fermantes en correspondante avec leur parenthèse ouvrante, on va les mettre en correspondance avec la position de la parenthèse ouvrante *du niveau précédent* (si elle existe, auquel cas le facteur le plus long commence une position après), ou, s'il n y en a pas, avec une parenthèse fermante qui n'a pas de correspondance dans ce qui précède.

Formellement, on va dire qu'une parenthèse fermante dans le mot w est *abyssale* si elle n'est pas en correspondance avec une parenthèse ouvrante dans les positions précédentes. On va considérer qu'à la position -1 il y a une parenthèse fermante abyssale qui va servir de sentinelle ; ceci ne change clairement pas la valeur du lpgf. Maintenant, l'observation clé est que, pour toute parenthèse fermante qui n'est pas abyssale, le facteur bien parenthésé le plus long qui lui correspond commence juste après la parenthèse ouvrante du niveau inférieur qui est dans la pile à ce moment-là, ou (s'il n'y en a pas) commence juste après la dernière parenthèse fermante abyssale que l'on a vu. La raison pour cela est que, si on revient en arrière à partir de cette parenthèse fermante, on va trouver sa parenthèse ouvrante correspondante, et puis il y a trois cas : (i) la position d'avant est une parenthèse ouvrante auquel cas le plus long facteur commence juste après ; (ii) la position d'avant est une parenthèse fermante abyssale (y compris la parenthèse fermante abyssale sentinelle à la position -1) auquel cas le plus long facteur commence aussi juste après ; (iii) la position d'avant est une parenthèse fermante non-abyssale, auquel cas elle a une parenthèse ouvrante correspondante jusqu'à laquelle on peut remonter pour répéter l'argument.

On en déduit donc l'invariant à garantir : à tout moment de la lecture de w, quand on a fini de lire un préfixe, la pile doit contenir les parenthèses ouvrantes actuellement ouvertes et leur position (dans l'ordre), et au fond de la pile doit se trouver la dernière parenthèse fermante abyssale que l'on a vue et sa position (initialement il s'agit de la sentinelle à la position -1). Quand on lit une parenthèse ouvrante, on l'empile simplement et l'invariant est préservé. Quand on lit une parenthèse fermante, soit la pile ne contient que le fond, et auquel cas on le remplace par la parenthèse fermante courante (car elle est abyssale) ; soit elle contient autre chose, auquel cas il suffit de pop la parenthèse ouvrante au sommet car elle vient de se finir. La valeur du lpgf est mise à jour quand on lit une parenthèse fermante non abyssale, et elle vaut la distance entre la position courante et la position du sommet de pile (c'est soit la dernière parenthèse abyssale vue, soit la position de la dernière parenthèse ouvrante qui n'a pas encore été fermée, et on a expliqué au paragraphe précédent pourquoi le facteur bien parenthésé maximal qui se termine à la parenthèse fermante courante doit commencer juste après cette position).

On parvient donc au code suivant :

```
T := tableau de taille n contenant les lettres du mot
P := pile vide
lpgf := 0 // le mot vide est toujours un facteur
P.push(-1) // sentinelle

Pour i de 1 à n:
  Si T[i] == 'a':
    P.push(i)
  Sinon: // T[i] == 'b'
    P.pop()
    Si P n'est pas vide:
      lpgf := max(lpgf, i - P.top())
    Sinon:
      P.push(i)
    Fin Si
  Fin Si
Fin Pour

Renvoie lpgf
```

[Cet algorithme est adapté de [Gee].]

Références

[Gee] GeeksforGeeks. Length of the longest valid substring. `https://www.geeksforgeeks.org/length-of-the-longest-valid-substring/`.

[GS04] Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *JCSS*, 69(4), 2004. `https://www.sciencedirect.com/science/article/pii/S0022000004000364`.

A13 – Entrelacements et langages réguliers

On fixe un alphabet Σ. Pour tout mot $z \in \Sigma^*$, on note $|z|$ la longueur de z, et on note z_i la i-ème lettre de z pour $1 \leq i \leq |z|$. Un *entrelacement* de deux mots $u, v \in \Sigma^*$ est un mot $w \in \Sigma^*$ tel qu'il existe deux fonctions $f_u : \{1, \ldots, |u|\} \to \{1, \ldots, |w|\}$ et $f_v : \{1, \ldots, |v|\} \to \{1, \ldots, |w|\}$ satisfaisant les propriétés suivantes :

— f_u et f_v sont strictement croissantes ;

— L'image de f_u et l'image de f_v sont des ensembles disjoints et leur union vaut $\{1, \ldots, |w|\}$;

— Pour tout $1 \leq i \leq u$, on a $u_i = w_{f_u(i)}$;

— Pour tout $1 \leq j \leq v$, on a $v_j = w_{f_v(j)}$.

Pour deux mots $u, v \in \Sigma^*$, on note $u \sqcup\!\sqcup v$ le langage des mots qui sont un entrelacement de u et de v.

Question 0. Le mot *abacab* est-il un entrelacement de *baa* et *acb* ? Le mot *bacb* est-il un entrelacement de *ab* et de *cb* ?

Question 1. Donner le pseudo-code d'un algorithme qui, étant donné $u, v, w \in \Sigma^*$, détermine si $w \in u \sqcup\!\sqcup v$. Analyser sa complexité en temps et en espace.

Question 2. Proposer un algorithme qui, étant donné $u, v \in \Sigma^*$ et un automate fini A qui reconnaît un langage L, détermine si $(u \sqcup\!\sqcup v) \cap L$ est non-vide. En donner un pseudo-code, et analyser sa complexité en temps et en espace.

Question 3. Pour deux langages $L, L' \subseteq \Sigma^*$, on note $L \sqcup\!\sqcup L' := \bigcup_{(u,v) \in L \times L'} u \sqcup\!\sqcup v$.
Montrer que si L_1 et L_2 sont deux langages réguliers alors L l'est aussi.

Question 4. On pose $(\Sigma^*)_2 := \bigcup_{i \in \mathbb{N}} \Sigma^i \times \Sigma^i$, où Σ^i dénote les mots de longueur i sur l'alphabet Σ. Étant donné un langage L, on définit $\psi(L) := \{(u, v) \in (\Sigma^*)_2 \mid (u \sqcup\!\sqcup v) \cap L \neq \emptyset\}$. Décrire $\psi(a^*b^*)$ en français et le caractériser.

Question 5. On considère l'alphabet $\Sigma_2 := \Sigma \times \Sigma$. On définit une fonction $\zeta : (\Sigma^*)_2 \to (\Sigma_2)^*$ inductivement par $\zeta((\epsilon, \epsilon)) := (\epsilon, \epsilon)$ et, pour tout $(x, y) \in \Sigma_2$ et $(u, v) \in \Sigma^*$, on a $\zeta((xu, yv)) := (x, y)\zeta(u, v)$. Décrire le langage $\zeta(\psi(a^*b^*))$.

Question 6. Dans cette question, on fixe $\Sigma_2 := \{a, b\}$, et on considère le langage $L_6 := \{a^n b^n \mid n \in \mathbb{N}\}$. Soit A un automate fini déterministe complet sur Σ_2, soit Q son ensemble d'états, et soit $\delta^* : \Sigma_2^* \to Q$ la fonction telle que, pour tout $w \in \Sigma_2^*$, l'état $\delta^*(w)$ soit l'état auquel on aboutit dans A après avoir lu w. Montrer que, s'il existe $i \neq j$ tels que $\delta^*(a^i) = \delta^*(a^j)$, alors le langage reconnu par A est différent de L_6. Conclure que L_6 n'est pas régulier.

Question 7. Montrer qu'il existe un langage régulier L_7 tel que $\zeta(\psi(L_7))$ ne soit pas régulier.

Corrigé

Question 0. Le mot *abacab* est un entrelacement de *baa* et *acb* : on peut l'écrire $AbaCaB$. Le mot *bacb* n'est pas un entrelacement de *ab* et de *cb*, bien que *ab* et *cb* en soient tous deux un sous-mot ; en effet le premier *b* ne peut pas avoir de préimage.

Question 1. On fait un algorithme dynamique, qui utilise un tableau de taille $(|u|+1) \times (|v|+1)$: pour $0 \leq i \leq |u|$ et $0 \leq j \leq |v|$, la case (i, j) stocke s'il existe un entrelacement du préfixe de u de longueur i et du préfixe de v de longueur j qui réalise le préfixe de longueur $i+j$ de w. En pseudo-code :

```
Entrée: tableaux u, v, w
T := tableau de (|u|+1) par (|v|+1) cases initialisé à FAUX

Si |u| + |v| != |w|:
  Renvoie FAUX
Fin Si

T[0][0] := VRAI

Pour i de 0 à |u|:
  Pour j de 0 à |v|:
    Si i > 0 et u[i-1] == w[i+j-1] et T[i-1][j] == VRAI:
      T[i][j] := VRAI
    Fin Si
    Si j > 0 et v[j-1] == w[i+j-1] et T[i][j-1] == VRAI:
      T[i][j] := VRAI
    Fin Si
  Fin Pour
Fin Pour

Renvoie T[|u|][|v|]
```

La complexité en espace est clairement de $O(|u| \times |v|)$: on peut la ramener à $O(\min(|u|, |v|))$ en échangeant u et v si nécessaire pour que $|v| \leq |u|$, et en ne conservant que la dernière ligne du tableau dans chaque tour de la boucle Pour extérieure. La complexité en temps est clairement en $O(|u| \times |v|)$.

Question 2. On modifie l'algorithme dynamique précédent : dans chaque case du tableau, on stocke pour chaque état de l'automate s'il peut être atteint par un entrelacement des préfixes. On obtient :

```
Entrée: tableaux u, v, w, automate A d'ensemble d'états Q
T := tableau de (|u|+1) par (|v|+1) cases par |Q| cases initialisé à FAUX

Pour chaque état initial q:
  T[0][0][q] := VRAI
Fin Pour

Pour i de 0 à |u|:
  Pour j de 0 à |v|:
    Pour chaque transition (q, a, q') de A:
      Si i > 0 et u[i-1] == a et T[i-1][j][q] == VRAI:
        T[i][j][q'] := VRAI
```

```
      Fin Si
      Si j > 0 et v[j-1] == a et T[i][j-1][q] == VRAI:
        T[i][j][q'] := VRAI
      Fin Si
    Fin Pour
  Fin Pour
Fin Pour

Pour chaque état final q:
  Si T[|u|][|v|][q] == VRAI:
    Renvoie VRAI
  Fin Si
Fin Pour

Renvoie FAUX
```

La complexité en temps est clairement de $O(|u| \times |v| \times |\delta|)$: on suppose, quitte à retirer les états inutiles, que tous les états sauf éventuellement un nombre constant apparaissent dans δ. La complexité en espace est de $O(|u| \times |v| \times |Q|)$: on peut la ramener à $O(\min(|u|, |v|) \times |Q|)$ comme précédemment.

Question 3. Soient $A_1 = (Q_1, I_1, F_1, \delta_1)$ et $A_2 = (Q_2, I_2, F_2, \delta_2)$ deux automates non-déterministes qui acceptent L_1 et L_2 respectivement, où la relation de transition δ_1 est un sous-ensemble de $Q_1 \times \Sigma \times Q_1$, et δ_2 est défini de façon analogue.

On construit une variante de l'automate produit, à savoir l'automate $A' = (Q_1 \times Q_2, I_1 \times I_2, F_1 \times F_2, \delta')$ où $\delta' \subseteq (Q_1 \times Q_2) \times \Sigma \times (Q_1 \times Q_2)$ est définie comme l'union de $\{((q_1, q_2), a, (q_1, q_2')) \mid q_1 \in Q_1, (q_2, a, q_2') \in \delta_2\}$ et de $\{((q_1, q_2), a, (q_1, q_2')) \mid q_2 \in Q_2, (q_1, a, q_1') \in \delta_1\}$.

Il est clair que, si A' accepte un mot $w \in \Sigma^*$, alors on peut partitionner les positions de w entre celles où on a suivi une transition de la première forme et celles où on a suivi une transition de la seconde forme, et si on note w_1 et w_2 les sous-mots correspondants, on extrait du run acceptant un run acceptant de A_1 sur w_1 et un run acceptant de A_2 sur w_2. À l'inverse, pour tous mots w_1 et w_2 respectivement acceptés par A_1 et A_2, pour tout mot $w \in w_1 \sqcup\!\sqcup w_2$, on peut construire un run acceptant de A' sur w à partir d'un run acceptant de A_1 sur w_1 et d'un run acceptant de A_2 sur w_2. Ainsi, A' accepte bien $L_1 \sqcup\!\sqcup L_2$.

(On peut d'ailleurs voir l'algorithme de la question 2 comme un test d'accessibilité dans l'automate intersection (produit) de l'automate d'entrée A, et de l'automate A' pour les singletons $\{u\}$ et $\{v\}$.)

Question 4. (Il faut faire attention à ne pas confondre l'exponentiation, qui dénote la concaténation des langages, et le produit cartésien.) L'ensemble $\psi(a^*b^*)$ est l'ensemble des paires de mots $u, v \in \Sigma^*$ de même longueur tel qu'il y ait un mot de a^*b^* dans $u \sqcup\!\sqcup v$. Démontrons qu'il s'agit en fait de $(\Sigma^*)_2 \cap ((a^*b^*) \times (a^*b^*))$. Il est clair que, pour toute paire de mots $u, v \in a^*b^*$ (en particulier s'ils sont de même longueur), en écrivant $u = a^p b^q$ et $v = a^r b^s$, on a $a^{p+r}b^{q+s} \in u \sqcup\!\sqcup v$. À l'inverse, si $u \sqcup\!\sqcup v$ contient un mot w de a^*b^*, en considérant la préimage par les deux fonctions f_u et f_v du préfixe de a^* de w et de son suffixe de b^*, on déduit que u et v sont également dans a^*b^*.

Question 5. En fait, ζ est simplement la fonction "zip" qui transforme une paire de mots de même longueur en un mot formé de paires de lettres. On sait que $\psi(a^*b^*)$ est l'ensemble des paires de deux mots de a^*b^* de même longueur. Ainsi, $\zeta(\psi(a^*b^*))$ est exactement $[aa]^*([ab]^* + [ba]^*)[bb]^*$, où on note entre crochets les paires de Σ_2.

Question 6. Procédons par l'absurde et supposons que A reconnaît L_6. Soient $i \neq j$ deux valeurs telles que $\delta^*(a^i) = \delta^*(a^j)$. Par hypothèse, A accepte $a^i b^i$, donc l'état $\delta^*(a^i b^i)$ est final. Ainsi, comme $\delta^*(a^j b^i)$ est le même état, il est également final, donc A accepte $a^j b^i$, ce qui est une contradiction puisque $i \neq j$.

On en déduit que L_6 n'est pas régulier, parce que, pour un automate putatif A qui reconnaîtrait L_6, il y aurait une injection ι de $\mathbb{N}$ dans son ensemble d'états défini par $\iota(n) := \delta(a^n)$ pour tout $n \in \mathbb{N}$, ce qui contredirait la finitude de cet ensemble.

Question 7. **Indication :** Prendre $L_7 := (ab)^*$.

Procédons par l'absurde et supposons que $\zeta(\psi(L_7))$ est régulier. Ainsi, comme les langages réguliers sont clos par intersection, l'intersection de ce langage avec $L'_7 := ([aa][ba])^*([ba][bb])^*$ est également régulier. Mais cette intersection est exactement $L''_7 := \{([aa][ba])^n([ba][bb])^n \in \mathbb{N}\}$. En effet, pour une direction, il est clair que tout mot $([aa][ba])^n([ba][bb])^n$ de L'_7 est dans L'_7, et si on considère sa préimage par ζ (qui est en fait une bijection), il s'agit de $((ab)^n(bb)^n, (aa)^n(ab)^n)$: ces deux mots ont clairement un entrelacement dans $(ab)^*$ obtenu en prenant les n (ab) du premier mot, puis alternativement les $2n$ a du second mot et les $2n$ b du premier mot, et enfin les n (ab) du second mot. Réciproquement, considérons un mot de $\zeta(\psi(L_7)) \cap L'_7$, écrivons-le $([aa][ba])^p([ba][bb])^q$, et montrons que $p = q$. Il est clair que les paires de mots de $\psi(L_7)$ doivent contenir autant de a que de b pour avoir un entrelacement dans $(ab)^*$, et ainsi on a forcément $p = q$.

Il reste simplement à observer que L''_7 n'est pas régulier, ce qui se prouve exactement comme à la question 6 (ou en utilisant le fait que les langages réguliers sont clos par morphisme inverse).

J1 – Arbre de suffixes

On fixe Σ un alphabet fini et $\$ \notin \Sigma$ un caractère spécial. Si u et u' sont deux mots, on dit que u est un *facteur* de u' s'il existe deux mots v et v' tels que $u' = vuv'$. Si v est le mot vide ε, on dit que u est un *préfixe* de u', et si $v' = \varepsilon$, on dit que u est un *suffixe* de u'. On note $\mathrm{Pref}(u')$ l'ensemble des préfixes de u', et $\mathrm{Suff}(u')$ l'ensemble des suffixes de u'.

Soit $D \subseteq \Sigma^*\$$ un ensemble fini de mots sur Σ terminés par \$, c'est-à-dire que les mots de D sont de la forme $u\$$ où $u \in \Sigma^*$ est un mot sur Σ. On note $\mathrm{Pref}(D) := \bigcup_{u \in D} \mathrm{Pref}(u)$, et $\mathrm{Suff}(D) := \bigcup_{u \in D} \mathrm{Suff}(u)$. L'*arbre lexicographique de* D est un arbre aux arêtes étiquetées par des lettres de $\Sigma \cup \{\$\}$, muni d'une bijection ϕ de l'ensemble de ses nœuds vers $\mathrm{Pref}(D)$, qui satisfait les conditions suivantes :

— la racine de l'arbre est envoyée vers le mot vide par ϕ ;

— pour $u \in \Sigma^*$ et $\alpha \in \Sigma \cup \{\$\}$, si on a $\phi(x) = u\alpha$ pour un nœud x, alors x n'est pas la racine, on a $\phi(x') = u$ pour le parent x' de x, et l'arête de x' à x est étiquetée par le caractère α.

On note en particulier que, pour chaque nœud x de l'arbre, le préfixe $\phi(x)$ est égal à la concaténation des étiquettes des arêtes le long du chemin qui va de la racine à x.

Question 1

(a) Donner l'arbre lexicographique de $D = \{aab\$; abb\$; ac\$; acbaa\$; b\$\}$.

(b) Donner un algorithme permettant de construire l'arbre lexicographique d'un ensemble de mots $D \subseteq \Sigma^*\$$. Quelle est sa complexité en temps ?

(c) Donner un algorithme qui, étant donné l'arbre lexicographique d'un ensemble de mots $D \subseteq \Sigma^*\$$ et un mot $u \in \Sigma^*$, détermine si $u\$ \in D$ et si u est un préfixe d'un mot de D. Quelle est sa complexité en temps ?

Question 2 Soit $t \in \Sigma^*\$$ un mot terminé par \$. On appelle *arbre inefficace des suffixes* de t l'arbre lexicographique de $\mathrm{Suff}(t) \setminus \{\epsilon\}$.

(a) Donner l'arbre inefficace des suffixes du mot *abbabab*\$.

(b) Asymptotiquement, quelle est la taille de l'arbre inefficace des suffixes de t, dans le pire cas ? Donner un exemple de famille de mots pour lequel le pire cas est atteint.

(c) En gardant une structure arborescente, proposer une amélioration de l'efficacité de la représentation en mémoire de cet arbre. Montrer que la structure de données ainsi obtenue nécessite un espace $O(|t|)$.

La structure de données de la question (c) est appelée *l'arbre des suffixes* de t. Dans les questions suivantes, **on admettra qu'il existe un algorithme pour calculer l'arbre des suffixes de t en temps $O(|t|)$.**

Question 3

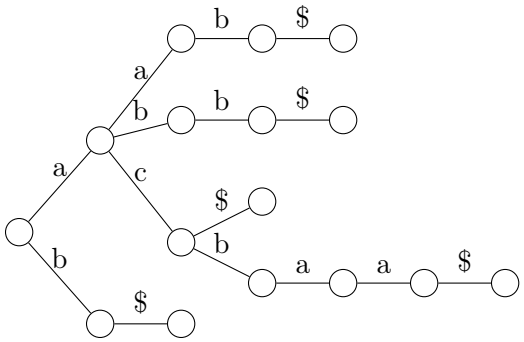
(a) Soit $M \subseteq \Sigma^*$ un ensemble fini de mots. Donner un algorithme efficace pour déterminer pour chaque mot de $m \in M$ s'il est un facteur de t. Quelle est sa complexité en temps ?

(b) Donner un algorithme efficace pour déterminer un plus long facteur commun de deux mots t et t'. Quelle est sa complexité en temps et en espace ?

Corrigé

Question 1

(a)



(b) On peut utiliser l'algorithme dont le pseudo-code est le suivant :

```
Fonction insere_mot(u, a)
  Si u est le mot vide
    Renvoyer a
  Si u = c.u'
    Si de t part une arête étiquetée par c vers un fils f
      Soit f' = insere_mot(u', f)
      Renvoyer a où l'on a remplacé le fils f par f'
    Sinon
      Soit f' = insere_mot(u', arbre vide)
      Renvoyer a où l'on a inséré le fils f' en étiquetant l'arête par c

a = arbre vide
Pour chaque mot u de D
  a = insere_mot(u, a)
```

La complexité en temps est $O(\sum_{u \in D} |u|)$: l'insertion d'un mot u parcourt exactement $|u| + 1$ nœuds dans l'arbre. La complexité en mémoire (non demandée) est $O(\sum_{u \in D} |u|)$ dans le pire cas où les mots ont des préfixes communs courts.

(c) On peut utiliser l'algorithme dont le pseudo-code est le suivant :

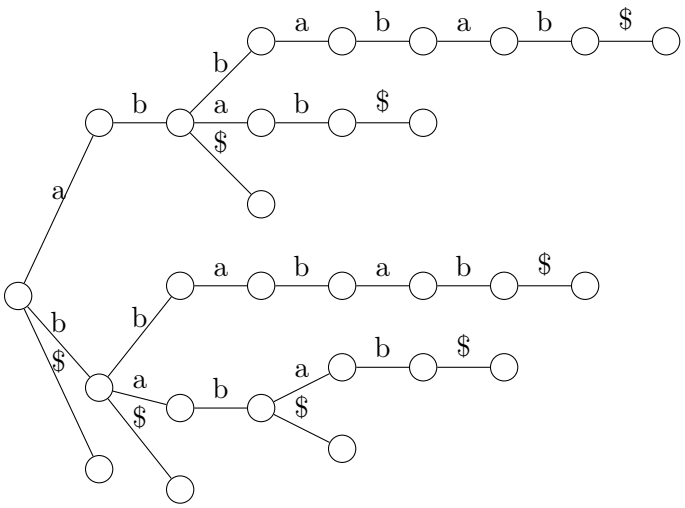
```
Fonction cherche_mot(u, a)
  Si u est vide
    Renvoyer vrai
  Si u = c.u'
    Si de t part une arête étiquetée par c vers un fils f
      Renvoyer cherche_mot(u', f)
    Sinon
      Renvoyer faux

contient_prefixe = cherche_mot(u, a)
contient_mot = cherche_mot(u.$, a)
```

La complexité en temps est $O(|u|)$, pour la même raison qu'à la question précédente.

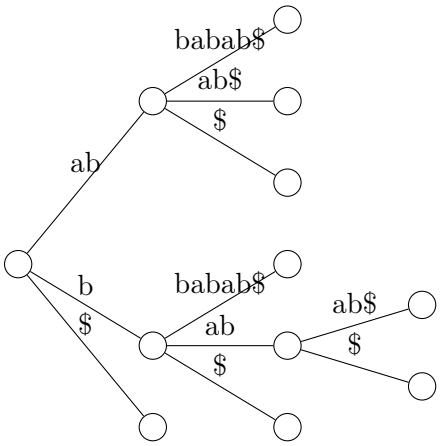
Question 2

(a)



(b) La taille de l'arbre est quadratique dans le pire cas. En effet, le mot $a^n b^n$ produit un peigne, avec n branches étiquetées par b^n, qui ont donc une longueur n.

(c) Pour améliorer l'efficacité de la représentation de l'arbre des suffixes, on peut se rendre compte que les nœuds ayant exactement un fils n'ont pas besoin d'être représentés. En effet, les chemins constitués uniquement de nœuds n'ayant qu'un fils peuvent être compressés en une seule arête, étiquetée par la concaténation des étiquettes des arêtes initiales. Dans l'exemple précédent, l'arbre devient :



Telle quelle, cette représentation reste de taille quadratique, puisqu'il faut stocker les étiquettes de toutes les arêtes. Cependant, ces étiquettes sont toutes des facteurs du mot initial. Au lieu de stocker les étiquettes telles quelles, on peut donc stocker les indices de début et de fin du facteur dans t, et obtenir donc une représentation linéaire en le nombre de nœuds de l'arbre.

Par ailleurs, on observe que cet arbre a $|t|$ feuilles (une par suffixe) exactement, et que chaque nœud interne (sauf éventuellement la racine) a au moins 2 fils. Le cas où la racine n'a qu'un fils conduit immédiatement à $u = \$$ et ne pose donc pas de difficulté. Pour montrer que la taille de l'arbre est en $O(|t|)$, il suffit donc de prouver qu'un arbre avec n feuilles et dont chaque nœud interne a au moins 2 fils a au plus $2n - 2$ arêtes et $2n - 1$ nœuds.

Montrons ce lemme par récurrence forte sur le nombre de feuilles. Si l'arbre contient exactement une feuille, alors c'est l'arbre singleton et le résultat est trivial. Sinon, l'arbre a une racine, et j sous-arbres. Soient $n_1, \ldots, n_j$ le nombre de feuilles de chacun des sous-arbres. Le nombre total de feuilles est $n = \sum_{i=1}^{j} n_i$. En utilisant l'hypothèse d'induction sur les sous-arbres, on en déduit que l'arbre a, au plus, $j + \sum_{i=1}^{j}(2n_i - 2) = 2n - j$ arêtes et $1 + \sum_{i=1}^{j}(2n_i - 1) = 1 + 2n - j$ nœuds. On peut conclure en utilisant $2 \leq j$.

Question 3

(a) On commence par construire l'arbre des suffixes de t. Ensuite, pour chaque mot m de M, on peut facilement parcourir l'arbre des suffixes comme à la question 1c, pour déterminer si m est un préfixe d'un élément de $\mathrm{Suff}(t)$, ce qui est le cas si et seulement s'il est facteur d'un élément de t. (Noter qu'on peut facilement adapter la construction de la question 1c pour fonctionner sur la version optimisée de l'arbre des suffixes de la question 2c.) Pour chaque mot m, le parcours s'effectue en $O(|m|)$. La complexité totale est donc $O(|t| + \sum_{m \in M} |m|)$.

(b) Sans perte de généralité, on peut supposer que t et t' se terminent par deux symboles réservés différents, \$ et \$′.

Commençons d'abord par résoudre le problème à partir l'arbre des suffixes inefficaces de tt' : sur cet arbre, les facteurs de t et de t' correspondent tous à des nœuds. Il suffit donc de caractériser les nœuds correspondant à des facteurs de t et ceux correspondant à facteurs de t'. Un nœud correspond à un facteur de t lorsque \$ apparaît dans une étiquette de sa descendance, et un nœud correspond à un facteur de t' si on peut arriver à une feuille sans passer par une étiquette contenant \$. Il est facile de faire remonter ces informations, en faisant un parcours en profondeur par exemple. En itérant sur tous les nœuds de l'arbre inefficace des suffixes, on peut donc calculer le plus long facteur commun.

Cependant, cet algorithme est trop lent, puisqu'il nécessite l'arbre inefficace des suffixes. On peut faire le même calcul dans l'arbre des suffixes : les critères pour qu'un nœud corresponde à un facteur de t et de t' restent les mêmes, et peuvent aussi être facilement calculés sur l'arbre des suffixes. (Notons qu'il est facile de déterminer si l'étiquette d'une arête contient \$ en connaissant uniquement l'indice de début et de fin de l'étiquette, puisque \$ n'apparaît qu'une fois dans tt', à une position connue.) Il reste donc à montrer que le facteur cherché correspond bien à un nœud de l'arbre des suffixes. Soit F un facteur commun maximal. Il ne peut pas être un suffixe de t ou de t', puisque ces mots se terminent par un symbole différent. Soit donc x et x' des symboles tels que Fx et Fx' soient facteurs de t et t', respectivement. Par maximalité, $x \neq x'$. Donc $Fx \neq Fx'$, et ces deux facteurs correspondent donc à deux nœuds différents dans l'arbre inefficace des suffixes. Le nœud associé à F dans l'arbre inefficace des suffixes a donc au moins deux fils, et, par conséquent, F correspond à un nœud dans l'arbre des suffixes de tt' (autrement dit, F ne se termine pas à l'intérieur de l'étiquette d'une arête).

Le parcours de l'arbre se fait en un temps linéaire, et on renvoie le nœud qui satisfait les deux conditions et qui est à une distance maximale de la racine (en pondérant les arêtes par la longueur du mot qui les étiquette). La complexité totale en temps de l'algorithme est donc $O(|t| + |t'|)$. Puisqu'il faut construire l'arbre des suffixes, la complexité en espace est aussi $O(|t| + |t'|)$.

J2 – Domaines abstraits linéaires

Le but de ce problème est d'étudier la structure de données d'*octogones*, qui permet d'approximer des parties de $\mathbb{R}^n$, et d'effectuer certaines opérations sur celles-ci.

Pour toutes les questions de ce problème, on suppose que l'on peut effectuer constant les opérations arithmétiques usuelles sur les nombres réels en temps constant. (C'est bien sûr faux dans la réalité, mais les nombres à virgule flottante donnent une bonne approximation.)

Soit $n \in \mathbb{N}^*$. Pour $A \in M_n(\mathbb{R} \cup \{+\infty\})$ une matrice carrée de taille n dont chaque case contient une valeur de $\mathbb{R} \cup \{+\infty\}$, on définit $\gamma_Z(A) \subseteq \mathbb{R}^n$ par :

$$\gamma_Z(A) := \{x \in \mathbb{R}^n \mid \forall\, 1 \leq i, j \leq n,\ x_i - x_j \leq A_{ij}\}$$

Intuitivement, on pensera à A comme un ensemble de contraintes d'inégalité de la forme $x_i - x_j \leq K$ sur $\mathbb{R}^n$.

Question 0. Représenter graphiquement $\gamma_Z\left(\begin{bmatrix} 0 & 2 \\ -1 & 0 \end{bmatrix}\right)$.

Si $U \subseteq \mathbb{R}^n$ est une partie non vide de $\mathbb{R}^n$, on définit $\alpha_Z(U) \in M_n(\mathbb{R} \cup \{+\infty\})$ comme suit pour tous $1 \leq i, j \leq n$:

$$\alpha_Z(U)_{ij} := \sup_{x \in U} x_i - x_j$$

Les contraintes d'inégalité imposées par A sont donc les plus fortes qui sont respectées par U.

On dit que A est *close* si $\gamma_Z(A) \neq \emptyset$ et $\alpha_Z(\gamma_Z(A)) = A$.

On dit qu'une matrice $A \in M_n(\mathbb{R} \cup \{+\infty\})$ vérifie *l'inégalité triangulaire* lorsque :

$$\forall ijk,\ A_{ij} \leq A_{ik} + A_{kj}$$

Question 1.

(a) Soit U une partie non vide de $\mathbb{R}^n$. Montrer que $\alpha_Z(U)$ vérifie l'inégalité triangulaire et est à diagonale nulle. En déduire qu'une matrice close vérifie l'inégalité triangulaire et est à diagonale nulle.

(b) Soit $A \in M_n(\mathbb{R} \cup \{+\infty\})$. On suppose que $\gamma_Z(A) \neq \emptyset$. Prouver que pour tout i et j, on a $\alpha_Z(\gamma_Z(A))_{ij} \leq A_{ij}$.

(c) Prouver qu'une matrice A est close si et seulement si elle vérifie l'inégalité triangulaire et est à diagonale nulle. On se limitera au cas où A est à coefficients finis, et on admettra le résultat général pour les questions suivantes.

(d) Pour $U \neq \emptyset$, prouver que $\alpha_Z(U)$ est close.

(e) Donner un algorithme qui, étant donné A, détecte si $\gamma_Z(A) \neq \emptyset$, et qui, le cas échéant, calcule $\alpha_Z(\gamma_Z(A))$ en temps $O(n^3)$.

Question 2. Soient A et B deux matrices closes.

(a) Donner un algorithme pour déterminer si $\gamma_Z(A) \subseteq \gamma_Z(B)$. Fonctionnerait-il toujours si A et B n'étaient pas supposées closes ?

(b) Donner un algorithme qui calcule une matrice C telle que $\gamma_Z(C) = \gamma_Z(A) \cap \gamma_Z(B)$. En général, C est-elle close ?

(c) Si on reprend la question (b) en supposant que tous les coefficients de B sont $+\infty$ sauf un, donner un algorithme qui calcule une telle matrice **close**, si elle existe, en $O(n^2)$.

(d) Donner un algorithme qui calcule la matrice $\alpha_Z(\gamma_Z(A) \cup \gamma_Z(B))$. En général, est-elle close ?

Suite des questions

Dans la suite du problème, on cherche à généraliser les contraintes que l'on peut représenter par une matrice. Plus précisément, on cherche à intégrer toutes les contraintes de la forme $\pm x_i \pm x_j \leq K$. Pour cela, on considère maintenant des matrices $A \in M_{2n}(\mathbb{R} \cup \{+\infty\})$, et, pour $x \in \mathbb{R}^n$ et $0 < i \leq n$, on pose, par convention, $x_{i+n} = -x_i$.

On peut alors définir :

$$\gamma_O(A) = \{x \in \mathbb{R}^n \mid \forall\, 1 \leq i, j \leq 2n,\ x_i - x_j \leq A_{ij}\}$$

De manière similaire à $\alpha_Z(U)$, pour U une partie non vide de $\mathbb{R}^n$, on définit $\alpha_O(U)$ pour tous $1 \leq i, j \leq 2n$:

$$\alpha_O(U)_{ij} := \sup_{x \in U} x_i - x_j$$

On dit alors qu'une matrice est *fortement close* si $\gamma_O(A) \neq \emptyset$ et $\alpha_O(\gamma_O(A)) = A$.

Pour simplifier les notations, on note $\bar{i} = i + n$ lorsque $1 \leq i \leq n$ et $\bar{i} = i - n$ lorsque $n + 1 \leq i \leq 2n$, de façon à ce qu'on ait toujours $x_{\bar{i}} = -x_i$.

Question 3

(a) Représenter $\gamma_O\left(\begin{bmatrix} 0 & +\infty & +\infty & +\infty \\ +\infty & 0 & +\infty & 6 \\ 2 & -1 & 0 & +\infty \\ +\infty & +\infty & +\infty & 0 \end{bmatrix}\right)$.

(b) Prouver qui si une matrice est fortement close, alors elle est close.

(c) Prouver que si A est fortement close, alors $\forall\, 1 \leq i, j \leq 2n,\ A_{ij} = A_{\bar{j}\bar{i}}$.

(d) Expliquer comment cette structure de données permet aussi de représenter des contraintes de la forme $K \leq x_i$ et $x_i \leq K$.

(e) En déduire que si A est fortement close, alors $\forall\, 1 \leq i, j \leq 2n,\ A_{ij} \leq \frac{A_{i\bar{i}} + A_{\bar{j}j}}{2}$.

On admet le résultat suivant : une matrice est fortement close si et seulement si elle vérifie les trois conditions données dans les questions 3(b), 3(c) et 3(e). C'est-à-dire, une matrice A est fortement close si et seulement si :

(i) A est close

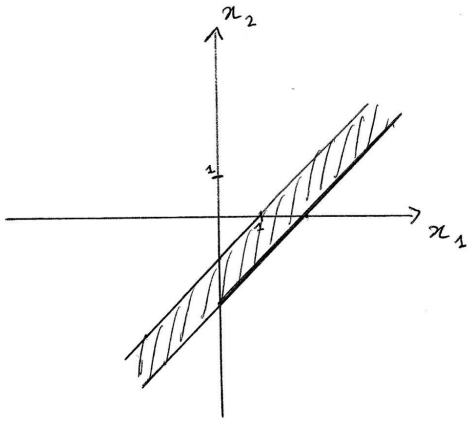
(ii) $\forall\, 1 \leq i, j \leq 2n,\ A_{ij} = A_{\bar{j}\bar{i}}$

(iii) $\forall\, 1 \leq i, j \leq 2n,\ A_{ij} \leq \frac{A_{i\bar{i}} + A_{\bar{j}j}}{2}$

Question 4 Donner un algorithme qui, étant donné A, détecte si $\gamma_O(A) \neq \emptyset$, et qui, le cas échéant, calcule $\alpha_O(\gamma_O(A))$ en temps $O(n^3)$.

Corrigé

Question 0



Question 1

(a) On a $\alpha_Z(U)_{ii} = \sup_{x \in U} 0 = 0$, donc $\alpha_Z(U)$ est à diagonale nulle. Par ailleurs,

$$\begin{aligned}\alpha_Z(U)_{ij} &= \sup_{x \in U} x_i - x_k + x_k - x_j \\ &\leq \sup_{(x,x') \in U^2} x_i - x_k + x'_k - x'_j \\ &= \alpha_Z(U)_{ik} + \alpha_Z(U)_{kj}\end{aligned}$$

Lorsque A est close, $\gamma_Z(A)$ est non vide. En appliquant le lemme précédent, on déduit que $\alpha_Z(\gamma_Z(A))$ est à diagonale nulle et vérifie l'inégalité triangulaire. On conclut en utilisant $A = \alpha_Z(\gamma_Z(A))$.

(b) Si $x \in \gamma_Z(A)$, alors $x_i - x_j \leq A_{ij}$. Donc :

$$\alpha_Z(\gamma_Z(A))_{ij} = \sup_{x \in \gamma_Z(A)} x_i - x_j \leq A_{ij}$$

(c) En combinant les résultats précédents, il reste à montrer que pour tout i_0 et j_0, si A est à diagonale nulle et vérifie l'inégalité triangulaire, alors $\gamma_Z(A) \neq \emptyset$ et $A_{i_0 j_0} \leq \alpha_Z(\gamma_Z(A))_{i_0 j_0}$. Par définition de α_Z, il suffit d'exhiber un $x \in \gamma_Z(A)$ tel que $A_{i_0 j_0} \leq x_{i_0} - x_{j_0}$. On choisit x défini par $x_i := A_{i j_0}$ pour tout i.

La matrice A est à diagonale nulle, donc $x_{j_0} = A_{j_0 j_0} = 0$ et donc $A_{i_0 j_0} = x_{i_0} - x_{j_0}$. On montre $x \in \gamma_Z(A)$ avec l'inégalité triangulaire :

$$x_i - x_j = A_{i j_0} - A_{j j_0} \leq A_{ij} + A_{j j_0} - A_{j j_0} = A_{ij}$$

(d) Selon les questions précédentes, il s'agit d'une matrice vérifiant l'égalité triangulaire à diagonale nulle. Elle est donc close.

(e) On peut interpréter A comme une matrice de poids dans un graphe dont les nœuds correspondent aux indices dans la matrice. Comme nous allons le démontrer, avec cette interprétation, $\gamma_Z(A) = \emptyset$ si et seulement si le graphe a un cycle de poids négatif, et calculer la clôture d'une matrice A revient à calculer tous les plus courts chemins du graphe dont la matrice de distances est donnée par A. On peut donc utiliser l'algorithme de Floyd-Warshall pour résoudre le problème. Notons

qu'il est tout de même nécessaire de s'assurer que la diagonale est nulle : cela correspond à calculer les plus courts chemins de chaque nœud à lui-même, ce que ne font pas certaines présentations de l'algorithme de Floyd-Warshall. Plus précisément, on vérifie les valeurs de la diagonale de la matrice et on échoue si une valeur est strictement négative ; sinon, on remplace toutes les valeurs de la diagonale de la matrice par zéro, on applique l'algorithme de Floyd-Warshall, et on échoue si et seulement si Floyd-Warshall signale l'existence d'un cycle de poids négatif.

Démontrons donc le résultat avancé ci-dessus. Tout d'abord, il est facile de voir que si le graphe a un cycle de poids négatif constitué des nœuds $i_1, \ldots, i_k$, alors pour tout $x \in \gamma_Z(A)$, on a :

$$0 = x_{i_1} - x_{i_2} + \cdots + x_{i_k} - x_{i_1} \leq A_{i_1 i_2} + \cdots + A_{i_k i_1} < 0.$$

Et donc $\gamma_Z(A) = \emptyset$.

On suppose maintenant qu'il n'y a pas de cycle négatif. Soit $\tilde{A}$ la matrice des plus courts chemins dans le graphe dont A est la matrice des poids. Manifestement, $\forall ij$, $\tilde{A}_{ij} \leq A_{ij}$, donc $\gamma_Z(\tilde{A}) \subseteq \gamma_Z(A)$. Mais, si $x \in \gamma_Z(A)$ et si $i_1, \ldots, i_k$ est le plus court chemin entre i_1 et i_k, alors :

$$x_{i_1} - x_{i_k} = x_{i_1} - x_{i_2} + \cdots + x_{i_{k-1}} - x_{i_k} \leq A_{i_1 i_2} + \cdots + A_{i_{k-1} i_k} = \tilde{A}_{i_1 i_k}$$

Donc $x \in \gamma_Z(\tilde{A})$, et on en déduit donc $\gamma_Z(\tilde{A}) = \gamma_Z(A)$

Mais $\tilde{A}$ est manifestement close, puisqu'elle est à diagonale nulle et vérifie l'inégalité triangulaire. Donc $\gamma_Z(A) \neq \emptyset$, et $\tilde{A} = \alpha_Z(\gamma_Z(\tilde{A})) = \alpha_Z(\gamma_Z(A))$

Question 2

(a) Il est naturel de tester si $\forall ij, A_{ij} \leq B_{ij}$, puisque si cette condition est vérifiée, alors $\gamma_Z(A) \subseteq \gamma_Z(B)$. Montrons que cette condition est nécessaire. En effet, si $\gamma_Z(A) \subseteq \gamma_Z(B)$, alors on déduit immédiatement $\forall ij$, $\alpha_Z(\gamma_Z(A))_{ij} \leq \alpha_Z(\gamma_Z(B))_{ij}$, et donc, par clôture de A et B, on a bien $\forall ij, A_{ij} \leq B_{ij}$.

En réalité, la clôture de B n'est jamais utilisée dans la preuve précédente (on sait que $\alpha_Z(\gamma_Z(B))_{ij} \leq B_{ij}$), et est donc une hypothèse inutile. Mais la clôture de A, elle, est importante. En effet, posons :

$$A = \begin{bmatrix} 0 & 1 & +\infty \\ +\infty & 0 & 1 \\ +\infty & +\infty & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & 1 & 2 \\ +\infty & 0 & 1 \\ +\infty & +\infty & 0 \end{bmatrix} = \alpha_Z(\gamma_Z(A))$$

On a $\gamma_Z(A) = \gamma_Z(B)$, mais pas $\forall ij$, $A_{ij} \leq B_{ij}$.

(b) On peut poser $C_{ij} = \min\{A_{ij}, B_{ij}\}$. On a alors :

$$\begin{aligned} x \in \gamma_Z(C) &\Leftrightarrow \forall ij,\ x_i - x_j \leq \min\{A_{ij}, B_{ij}\} \\ &\Leftrightarrow \forall ij,\ x_i - x_j \leq A_{ij} \wedge x_i - x_j \leq B_{ij} \\ &\Leftrightarrow x \in \gamma_Z(A) \wedge x \in \gamma_Z(B) \end{aligned}$$

Mais C n'est alors pas nécessairement close. Par exemple :

$$A = \begin{bmatrix} 0 & 1 & +\infty \\ +\infty & 0 & +\infty \\ +\infty & +\infty & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & +\infty & +\infty \\ +\infty & 0 & 1 \\ +\infty & +\infty & 0 \end{bmatrix} \qquad C = \begin{bmatrix} 0 & 1 & +\infty \\ +\infty & 0 & 1 \\ +\infty & +\infty & 0 \end{bmatrix}$$

(c) Sur les graphes, cela revient à mettre à jour une matrice de plus courts chemins lorsque l'on rajoute une arête, et de détecter un éventuel nouveau cycle de poids négatif.

Supposons donc qu'on veuille rajouter une arête du nœud i au nœud j, avec le poids p. Si un cycle de poids négatif existe, alors il passe nécessairement par la nouvelle arête, et on peut supposer sans perte de généralité qu'il emprunte le plus court chemin entre j et i dans l'ancien graphe. Ce cycle est de poids négatif si et seulement si $A_{ji} + p < 0$. L'intersection est donc vide si et seulement si cette condition est vérifiée.

Dans le cas contraire, il faut à nouveau calculer tous les plus courts chemins dans le graphe. Pour chacun de ces plus courts chemins, deux cas peuvent se présenter : il est soit identique au chemin déjà calculé, ou il bien il passe par la nouvelle arête. Le nouveau plus court chemin entre deux nœuds i' et j' a donc comme longueur $\min\{A_{i'j'}, A_{i'i} + p + A_{ij'}\}$, ce qui peut se calculer en temps constant. Le calcul de la nouvelle matrice s'effectue donc en temps $O(n^2)$.

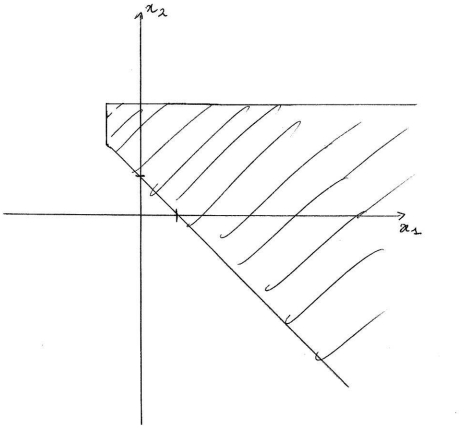
(d) De manière générale, on peut prouver facilement, pour U et V des sous-ensembles de $\mathbb{R}$, que $\sup(U \cup V) = \max\{\sup U, \sup V\}$. On en déduit facilement :

$$\alpha_Z(\gamma_Z(A) \cup \gamma_Z(B))_{ij} = \max\{\alpha_Z(\gamma_Z(A))_{ij}, \alpha_Z(\gamma_Z(B))_{ij}\} = \max\{A_{ij}, B_{ij}\}$$

Cette matrice est close, cela découle immédiatement de la question 1d.

Question 3

(a)



(b) La convention décrite dans l'énoncé pour définir x_i quand $n < i \leq 2n$ induit en fait une fonction $\phi : \mathbb{R}^n \to \mathbb{R}^{2n}$, définie par $\phi(x)_i = x_i$ si $i \leq n$ et $\phi(x)_i = -x_{i-n}$ sinon. Avec cette définition, on remarque que $\alpha_O(U) = \alpha_Z(\phi(U))$. Donc, si A est fortement close, alors $A = \alpha_O(\gamma_O(A)) = \alpha_Z(\phi(\gamma_O(A)))$. Donc A est close grâce à 1d.

(c) De manière générale, $x_i - x_j = x_{\bar{j}} - x_{\bar{i}}$. Par conséquent, pour $U \neq \emptyset$, on a $\alpha_O(U)_{ij} = \alpha_O(U)_{\bar{j}\bar{i}}$, ce qui est vrai en particulier pour $\gamma_O(A)$. On conclut grâce à la clôture forte de A.

(d) Les coefficients $A_{i\bar{i}}$ et $A_{\bar{i}i}$ imposent les contraintes $2x_i \leq A_{i\bar{i}}$ et $-2x_i \leq A_{\bar{i}i}$, respectivement. Pour représenter la contrainte $K \leq x_i$ et $x_i \leq K$, il suffit donc de choisir $A_{i\bar{i}} = 2K$ et $A_{\bar{i}i} = 2K$, respectivement.

(e) Si $x \in \gamma_O(A)$, alors $x_i - x_j \leq \frac{A_{i\bar{i}} + A_{\bar{j}j}}{2}$. Donc $\alpha_O(\gamma_O(A))_{ij} \leq \frac{A_{i\bar{i}} + A_{\bar{j}j}}{2}$. On peut alors conclure en utilisant le fait que A est fortement close.

Question 4 Étant donné une matrice A, on peut effectuer des réductions qui conservent $\gamma_O(A)$ mais permettent de valider les conditions de la caractérisation de la clôture forte. Lorsque, après avoir effectué ces réductions, on obtiendra une matrice A' close telle que $\gamma_O(A') = \gamma_O(A)$, on aura calculé la clôture forte de A, puisque $A' = \alpha_O(\gamma_O(A')) = \alpha_O(\gamma_O(A))$. Chacune des conditions de clôture forte données dans l'énoncé suggère une de ces réductions. Plus précisément :

(i) La condition $A_{ij} = A_{\bar{j}\bar{i}}$ suggère la réduction $A_{ij} \leftarrow \min\{A_{ij}, A_{\bar{j}\bar{i}}\}$.

(ii) La condition de clôture (simple) suggère le calcul de tous les plus courts chemins, via l'algorithme de Floyd-Warshall.

(iii) La condition $A_{ij} \leq \frac{A_{i\bar{i}}+A_{\bar{j}j}}{2}$ suggère la réduction $A_{ij} \leftarrow \min\{A_{ij}, \frac{A_{i\bar{i}}+A_{\bar{j}j}}{2}\}$.

Il n'est pas difficile de se convaincre que chacune de ces réductions permet, indépendamment des autres, de garantir l'une des conditions de clôture forte. La difficulté de cette question consiste en la combinaison de ces trois réductions afin de satisfaire les trois conditions de clôture.

Premièrement, la symétrie imposée par $A_{ij} = A_{\bar{j}\bar{i}}$ est manifestement conservée par les deux autres réductions. On ne perd donc rien à effectuer la réduction $A_{ij} \leftarrow \min\{A_{ij}, A_{\bar{j}\bar{i}}\}$ en premier.

On va prouver que si on a calculé tous les plus courts chemins sur un matrice vérifiant $A_{ij} = A_{\bar{j}\bar{i}}$, alors la réduction $A_{ij} \leftarrow \min\{A_{ij}, \frac{A_{i\bar{i}}+A_{\bar{j}j}}{2}\}$ conserve la clôture simple. Ainsi, si on effectue les trois réductions dans l'ordre dans lesquelles elles sont données plus haut, on obtiendra la matrice fortement close cherchée. Le cas $\gamma_O(A) = \emptyset$ est détecté dans le cas où cet algorithme échoue, c'est-à-dire si l'algorithme de Floyd-Warshall trouve un cycle de poids négatif.

Pour prouver ce résultat, pour A une matrice close et vérifiant $A_{ij} = A_{\bar{j}\bar{i}}$, on note A' la matrice définie par :

$$A'_{ij} = \min\{A_{ij}, \frac{A_{i\bar{i}} + A_{\bar{j}j}}{2}\}$$

On commence par prouver que A' est à diagonale nulle. En effet, par l'inégalité triangulaire, $0 = A_{ii} \leq A_{i\bar{i}} + A_{\bar{i}i}$. Donc $A'_{ii} = 0$.

Maintenant, prouvons que A' vérifie l'inégalité triangulaire. On a :

$$A'_{ij} + A'_{jk} = \min\left\{A_{ij} + A_{jk}, \frac{A_{i\bar{i}} + A_{\bar{j}j}}{2} + A_{jk}, A_{ij} + \frac{A_{j\bar{j}} + A_{\bar{k}k}}{2}, \frac{A_{i\bar{i}} + A_{\bar{j}j}}{2} + \frac{A_{j\bar{j}} + A_{\bar{k}k}}{2}\right\}$$

On va montrer que chacun des termes utilisés dans le minimum est supérieur ou égal soit à A_{ik}, soit à $\frac{A_{i\bar{i}}+A_{\bar{k}k}}{2}$. Chacun de ces termes sera donc supérieur ou égal à A'_{ik}, et on pourra conclure à l'inégalité triangulaire. Considérons donc chacun de ces termes :

- Par l'inégalité triangulaire, $A_{ij} + A_{jk} \geq A_{ik}$.
- $\frac{A_{i\bar{i}}+A_{\bar{j}j}}{2} + A_{jk} = \frac{A_{i\bar{i}}+A_{\bar{k}\bar{j}}+A_{\bar{j}j}+A_{jk}}{2} \geq \frac{A_{i\bar{i}}+A_{\bar{k}k}}{2}$, en utilisant l'inégalité triangulaire et la relation $A_{\bar{k}\bar{j}} = A_{jk}$.
- Le terme $A_{ij} + \frac{A_{j\bar{j}}+A_{\bar{k}k}}{2}$ est traité de la même façon que le précédent.
- Par l'inégalité triangulaire, $\frac{A_{i\bar{i}}+A_{\bar{j}j}}{2} + \frac{A_{j\bar{j}}+A_{\bar{k}k}}{2} \geq \frac{A_{i\bar{i}}+A_{\bar{j}\bar{j}}+A_{\bar{k}k}}{2} = \frac{A_{i\bar{i}}+A_{\bar{k}k}}{2}$

J3 – Arbres de Braun

On prend la définition suivante pour la notion d'arbre binaire : il s'agit d'une structure de données finie, consistant soit en un arbre vide $\langle\rangle$ (*i.e.,* une feuille), soit en un nœud. Un nœud $\langle g, d, c\rangle$ contient un sous-arbre binaire gauche g, un sous-arbre binaire droit d et une *charge utile* c. On ne décrit pas précisément ce que contient la charge utile : cela dépend de l'utilisation qui est faite de l'arbre. La *taille* d'un arbre binaire a, notée $T(a)$, est le nombre de nœuds qu'il contient. La *hauteur* d'un arbre binaire est 0 si l'arbre est vide, et sinon c'est le nombre maximal de nœuds qu'il faut traverser pour atteindre une feuille depuis la racine (feuille exclue et racine incluse).

Un *arbre de Braun* est un arbre binaire dont tous les nœuds $\langle g, d, c\rangle$ vérifient :

$$0 \;\leq\; T(g) \;-\; T(d) \;\leq\; 1$$

Question 1

(a) Montrer que, si on ignore les charges utiles, alors pour chaque $n \in \mathbb{N}$ il existe un unique arbre de Braun de taille n. Dessiner les 6 premiers arbres de Braun non vides.

(b) Proposer un algorithme naïf pour calculer la taille d'un arbre de Braun.

(c) Donner le comportement asymptotique de la hauteur d'un arbre de Braun en fonction de sa taille.

Question 2 Expliquer comment on peut utiliser la structure d'arbre de Braun pour implémenter une file de priorité persistante. On demande de donner un algorithme pour les opérations d'insertion (en $O(\log n)$), de consultation du minimum (en $O(1)$) et de suppression du minimum (en $O(\log n)$).

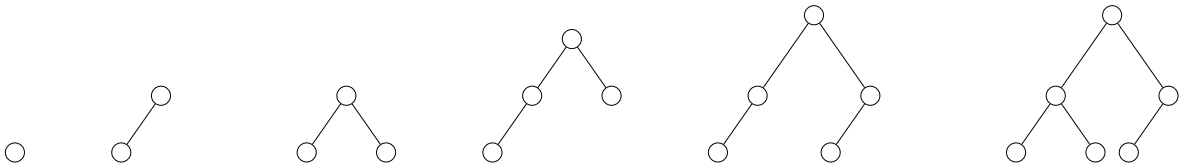
Question 3 Donner un algorithme pour calculer la taille d'un arbre de Braun ayant une complexité sous-linéaire. Quelle est sa complexité ?

Corrigé

Question 1

(a) On peut procéder par récurrence sur la taille de l'arbre. Il n'existe qu'un arbre de Braun de taille 0, l'arbre vide. Si $n \geq 1$, alors le sous-arbre gauche sera de taille $\lceil \frac{n-1}{2} \rceil$ et le sous-arbre gauche de taille $\lfloor \frac{n-1}{2} \rfloor$. Par hypothèse de récurrence, il n'existe qu'une seule possibilité pour les deux fils. L'arbre de taille n est donc défini de manière unique.

Les 6 premiers arbres de Braun non vides sont :



(b) On peut utiliser la fonction suivante :

```
Fonction taille(a)
  Si a = <>
    Renvoyer 0
  Sinon
    Renvoyer 1 + taille(a.gauche) + taille(a.droite)
```

(c) Il est facile de démontrer par induction que la hauteur d'un arbre de Braun est une fonction croissante de sa taille. En ce qui concerne le comportement asymptotique, puisque le sous-arbre gauche est toujours plus gros que le sous-arbre droit, la hauteur maximale sera toujours réalisée par une feuille du sous-arbre gauche. Par conséquent, la hauteur d'un arbre de Braun vérifie la récurrence :

$$H(n) = H\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + 1$$

Ainsi, on a $H(n) = O(\log n)$.

Question 2 On considère les arbres de Braun dont les charges utiles sont les éléments stockés dans la file de priorité, et on maintient la propriété de tas sur ces arbres. Plus précisément, si $\langle\langle t1, t2, x\rangle, t3, y\rangle$ est un nœud de l'arbre, alors $x \geq y$; et si $\langle t1, \langle t2, t3, x\rangle, y\rangle$ est un nœud de l'arbre, alors $x \geq y$.

La consultation du minimum est l'opération la plus simple, elle consiste à renvoyer la charge utile à la racine :

```
Fonction minimum(a, x)
  Posons <g, d, y> = a
  Renvoyer y
```

La complexité est $O(1)$.

L'ajout d'un nouvel élément peut être effectué comme suit :

```
Fonction insertion(a, x)
  Si a = <>
    Renvoyer <<>, <>, x>
  Sinon
    Posons <g, d, y> = a
    Si x < y
      échange de x et y
    Renvoyer <insertion(d, x), g, y>
```

La structure d'arbre de Braun est maintenue, car on insère toujours dans le sous-arbre droit, en échangeant les deux sous-arbres : ainsi, si leur différence de taille est 1, elle devient 0, et si leur différence de taille est 0, elle devient 1. La conservation de la structure de tas est assurée par la comparaison utilisée pour choisir quel élément sera inséré récursivement et quel élément sera utilisé à la racine. Il y a au plus $H(n)$ appels récursifs. La complexité est donc $O(\log n)$.

L'opération de suppression du minimum est plus complexe. En effet, une fois la racine enlevée, il est nécessaire de fusionner les deux sous-arbres pour former le nouveau tas. On va donc expliciter, dans la suite, une fonction `fusion`, qui fait cette opération (en supposant que les deux arbres passés en paramètres vérifient la propriété des arbres de Braun). La fonction de suppression du minimum est alors :

```
Fonction supprime_min(a)
  Posons <g, d, y> = a
  Renvoyer fusion(g, d)
```

Pour fusionner les deux arbres, il faut bien faire attention à conserver la structure d'arbre de Braun *et* la structure de tas. Si l'arbre de gauche (qui a un élément de plus, si les tailles ne sont pas égales) a une racine plus petite, alors on peut utiliser cette racine comme nouvelle racine, et fusionner récursivement ses sous-arbres. Attention, dans le nouvel arbre, il faudra échanger les deux fils pour conserver la structure d'arbre de Braun.

Le cas où c'est l'arbre de droite qui a une racine plus petite est plus embêtant : en effet, c'est sa racine qu'il faudrait prendre pour constituer la racine du nouvel arbre, mais cela casserait la structure d'arbre de Braun, car l'arbre de droite est potentiellement déjà plus petit que l'arbre de gauche. Il faut donc extraire un élément (quelconque) de l'arbre de gauche et l'insérer à droite pour maintenir la taille de l'arbre de droite après l'extraction de sa racine. Le pseudo-code de cet algorithme est le suivant :

```
Fonction extraire(a)
  Posons <g, d, x> = a
  Si g = <> et d = <>
    Renvoyer (<>, x)
  Sinon
    Posons (g', y) = extraire(g)
    Renvoyer (<d, g', x>, y)

Fonction fusion_log2(g, d)
  Si d = <>
    Renvoyer g
  Posons <gg, gd, xg> = g
  Posons <_, _, xd> = d
  Si xg <= xd
    Renvoyer <d, fusion_log2(gg, gd), xg>
  Sinon
    Posons (g', x) = extraire(g)
    Posons <dg, dd, xd> = insertion(d, x)
    Renvoyer <fusion_log2 dg dd, g', xd>
```

Malheureusement, cet algorithme est en $O(\log^2 n)$. En effet, dans le pire cas, on va appeler `extraire` et `insertion` pour chaque niveau de l'arbre. Pour faire mieux, il faut une autre fonction auxiliaire, qui réalise en même temps l'extraction de la racine dans l'arbre droit et l'insertion du nouvel élément :

```
Fonction remplace_min(a, x)
  Posons <g, d, y> = a
  Si (g = <> Ou x <= minimum(g)) Et (d = <> Ou x <= minimum(d))
```

```
    Renvoyer <g, d, x>
  Sinon
    // Noter qu'on sait que g != <> puisque g et d ne sont pas tous deux vides
    Si d = <> Ou minimum(g) <= minimum(d)
      Renvoyer <remplace_min(g, x), d, minimum(g)>
    Sinon
      Renvoyer <g, remplace_min(d, x), minimum(d)>

Fonction fusion(g, d)
  Si d = <>
    Renvoyer g
  Posons <gg, gd, xg> = g
  Posons <_, _, xd> = d
  Si xg <= xd
    Renvoyer <d, fusion(gg, gd), xg>
  Sinon
    Posons (g', x) = extraire(g)
    Renvoyer <remplace_min(d, x), g', xd>
```

Cette nouvelle version est de complexité $O(\log n)$: pour chaque niveau de l'arbre chacune des trois fonctions `fusion`, `remplace_min` et `extraire` ne sera appelée au plus qu'une fois.

Question 3 L'algorithme proposé en 1b a une complexité linéaire : il parcourt tous les nœuds de l'arbre. La raison pour laquelle cet algorithme peut être amélioré est qu'après l'appel récursif sur le sous-arbre de gauche, on connaît déjà presque totalement la taille de l'arbre droit : c'est soit la même, soit un de moins.

On va donc faire une fonction qui calcule la taille de l'arbre droit, à partir la taille de l'arbre gauche. Plus précisément, la fonction `taille_proche` prend en paramètre un arbre a et un entier n, avec l'hypothèse que $T(a) = n$ ou $T(a) = n - 1$, et renvoie $T(a)$. L'idée de l'implémentation de cette fonction est que, pour un arbre non vide, si on connaît n, alors on connaît toujours la taille de l'un des sous-arbres. En particulier, si n est pair, alors la taille du sous-arbre gauche est $\lceil \frac{n}{2} \rceil$, et si n est impair, alors la taille du sous-arbre droit est $\lfloor \frac{n}{2} \rfloor$.

Finalement, on obtient l'algorithme suivant :

```
Fonction taille_proche(a, n)
  Si t = <>
    Renvoyer 0
  Si n est pair
    Renvoyer ceil(n/2) + taille_proche(a.droite, floor(n/2))
  Sinon
    Renvoyer taille_proche(a.gauche, ceil(n/2)) + floor(n/2)

Fonction taille_rapide(a)
  Si a = <>
    Renvoyer 0
  Posons n = taille_rapide(a.gauche)
  Renvoyer n + taille_proche(a.droite, n)
```

Pour chaque niveau de l'arbre, on effectue un appel de `taille_proche`, qui lui-même effectue un appel récursif par niveau de l'arbre restant. La complexité est donc $O(\log^2 n)$.

Références

Okasaki, C. 1997. Three algorithms on Braun trees. *Journal of Functional Programming* Functional Pearls. https://www.eecs.northwestern.edu/~robby/courses/395-495-2013-fall/three-algorithms-on-braun-trees.pdf

J4 – Problèmes d'ordonnancement

On s'intéresse à des problèmes d'*ordonnancement* : on a n *tâches* à réaliser (des programmes à exécuter sur un ordinateur, par exemple), et on souhaite déterminer quand les réaliser en respectant certaines contraintes pour optimiser un objectif. La définition exacte des contraintes et de l'objectif varieront selon la question.

Chaque tâche aura toujours une durée d'exécution $p_i \in \mathbb{N}^*$, et on supposera qu'une tâche ne peut pas être interrompue une fois commencée. On cherche donc à associer à chaque tâche i un instant de début $t_i \in \mathbb{N}$, en s'assurant qu'une seule tâche est exécutée à la fois à chaque instant, c'est-à-dire que :

$$\forall 1 \leq i, j \leq n,\ t_i - t_j \geq p_j \text{ ou } t_j - t_i \geq p_i$$

Question 1 Dans cette question, on ne considère aucune contrainte supplémentaire.

(a) Proposer un algorithme pour calculer un ordonnancement qui minimise la date de fin de la dernière tâche. Quelle est sa complexité en temps ?

(b) Proposer un algorithme pour calculer un ordonnancement qui minimise la moyenne des dates de fin. Quelle est sa complexité en temps ?

(c) À chaque tâche i on associe un poids $w_i \in \mathbb{N}^*$. Proposer un algorithme pour calculer un ordonnancement qui minimise la moyenne des dates de fin, pondérée par les poids. Quelle est sa complexité en temps ?

Question 2 On suppose, **dans cette question uniquement**, qu'en plus d'une durée d'exécution p_i, chaque tâche a une *date de disponibilité* r_i. Une tâche ne peut jamais commencer avant cette date. Un ordonnancement valide $(t_i)_{1 \leq i \leq n}$ vérifiera donc toujours :

$$\forall 1 \leq i \leq n,\ r_i \leq t_i$$

(a) Adapter naïvement l'algorithme proposé en 1a. L'algorithme obtenu est-il correct ? S'il l'est, quelle est sa complexité en temps ?

(b) Adapter naïvement l'algorithme proposé en 1b. L'algorithme obtenu est-il correct ? S'il l'est, quelle est sa complexité en temps ?

Suite des questions

À partir de maintenant, on associe à chaque tâche une date d'échéance d_i en plus de la durée d'exécution p_i. Une tâche sera alors considérée *en retard* si elle se termine après la date d'échéance. Les dates d'échéances ne sont pas considérées comme des contraintes dures : un ordonnancement qui ne les respecte pas est toujours considéré comme valide. Cependant, un bon ordonnanceur essaiera toujours de minimiser les retards. C'est le sujet des questions suivantes :

Question 3

(a) Proposer un algorithme qui calcule un ordonnancement qui n'a pas de retard (si possible), ou qui minimise le retard de la tâche qui est le plus en retard. Quelle est sa complexité en temps ?

(b) Proposer un algorithme qui calcule un ordonnancement qui minimise le nombre de tâches en retard. Quelle est sa complexité en temps ?

Corrigé

Question 1

(a) Un ordonnancement qui n'a pas de "temps mort" aura toujours la même date de fin : il s'agira de $\sum_{i=0}^{n-1} p_i$. Ainsi, il n'est pas nécessaire de choisir un ordre particulier pour les tâches, et on peut utiliser l'algorithme suivant :

```
t_cur <- 0
Pour i de 0 à n-1
  t[i] <- t_cur
  t_cur <- t_cur + p[i]
```

La complexité est $O(n)$ en temps.

(b) **Indication** (pour la preuve) : on pourra montrer qu'on peut toujours trier un tableau en effectuant des échanges d'éléments successifs.

Si elles sont exécutées tôt, les tâches les plus longues vont retarder l'ensemble des tâches. Par conséquent, il est naturel de réaliser les tâches par ordre de durée : on réutilise le même algorithme qu'à la question précédente, en ayant préalablement trié les tâches par durée croissante. La complexité en temps est $O(n \log n)$, puisque le tri nécessite un temps $O(n \log n)$.

Prouvons que cet algorithme permet effectivement de minimiser la moyenne des dates de fin. Pour cela, on peut commencer par remarquer, comme à la question précédente, que seul l'ordre d'exécution importe : en effet, un "temps mort" peut toujours être supprimé en réduisant la moyenne des dates de fins d'exécution.

Soit donc $\sigma \in \mathfrak{S}(\llbracket 0, n-1 \rrbracket)$ une permutation représentant un ordre d'exécution, c'est-à-dire que la tâche $\sigma(i)$ est exécutée à la i-ème position. La date de fin d'exécution de la tâche $\sigma(i)$ est :

$$\sum_{j=0}^{i} p_{\sigma(j)}$$

et la somme de ces dates de fin est :

$$S(\sigma) = \sum_{i=0}^{n-1} \sum_{j=0}^{i} p_{\sigma(j)}$$

Minimiser la moyenne des dates de fin revient à minimiser $S(\sigma)$.

On va prouver que trier les tâches par durées croissantes ne peut que diminuer la date de fin moyenne. Pour cela, on utilise le fait qu'un tri peut toujours se décomposer en une série d'échanges

de positions successives i et $i+1$ telles que $p_{\sigma(i)} \geq p_{\sigma(i+1)}$. Si de tels échanges ne font que diminuer $S(\sigma)$, alors un tri ne peut que diminuer la date de fin moyenne, quelle que soit la permutation initiale. Cela permet de conclure finalement qu'un ordonnancement par $p_{\sigma(i)}$ croissants est optimal. Il suffit donc de prouver que si σ est une permutation et i est un indice tel que $p_{\sigma(i)} \geq p_{\sigma(i+1)}$, alors $S(\sigma \circ (i, i+1)) \leq S(\sigma)$. Pour cela, on peut facilement calculer :

$$S(\sigma \circ (i, i+1)) = S(\sigma) + p_{\sigma(i+1)} - p_{\sigma(i)}$$

Ce qui permet bien de conclure $S(\sigma \circ (i, i+1)) \leq S(\sigma)$.

(c) Dans la preuve de la question précédente, si on ajoute les poids, alors la définition de $S(\sigma)$ devient :

$$S(\sigma) = \sum_{i=0}^{n-1} \sum_{j=0}^{i} p_{\sigma(j)} w_{\sigma(i)}$$

Et on a :

$$S(\sigma \circ (i, i+1)) = S(\sigma) + w_{\sigma(i)} p_{\sigma(i+1)} - w_{\sigma(i+1)} p_{\sigma(i)}$$

Un échange entre les positions i et $i+1$ diminue donc $S(\sigma)$ si et seulement si $w_{\sigma(i)} p_{\sigma(i+1)} - w_{\sigma(i+1)} p_{\sigma(i)} \leq 0$, c'est-à-dire si et seulement si $\frac{p_{\sigma(i+1)}}{w_{\sigma(i+1)}} \leq \frac{p_{\sigma(i)}}{w_{\sigma(i)}}$. Pour calculer un ordonnancement optimal, on peut donc trier les tâches par $\frac{p_{\sigma(i)}}{w_{\sigma(i)}}$ croissant. Cela peut se faire en temps $O(n \log n)$.

Question 2

(a) De manière similaire à 1a, il existe toujours un ordonnancement optimal qui évite les "temps morts". Plus précisément, on peut supposer qu'à tout instant, soit une tâche est en cours d'exécution, soit toutes les tâches pouvant être exécutées l'ont été. Par ailleurs, tant que cette contrainte reste vérifiée, l'ordre dans lequel les tâches sont exécutées n'a aucune importance, comme à la question 1a.

On aboutit donc au pseudo-code suivant :

```
Tant qu'il reste des tâches à exécuter
  Si au moins une tâche peut être exécutée maintenant
    En exécuter une quelconque
  Sinon
    Attendre jusqu'à ce qu'une tâche soit disponible
```

Pour l'implémenter de manière efficace, on trie les tâches dans l'ordre de date de disponibilité croissante. Il sera alors facile de déterminer si l'on peut exécuter une tâche à la date actuelle, et, dans le cas contraire, quelle est la prochaine tâche disponible :

```
Trier les tâches par r[i] croissant
t_cur <- 0
Pour chaque tâche i
  t_cur <- max(r[i], t_cur)
  t[i] <- t_cur
```

Cet algorithme est donc en $O(n \log n)$.

(b) L'algorithme "glouton" qui serait une adaptation naïve de l'algorithme donné en 1b consisterait à toujours exécuter la tâche disponible minimisant p_i. Cependant, cela ne produit pas toujours un résultat optimal. En effet, considérons le cas où $(p_0, r_0) = (1000, 0)$ et $(p_1, r_1) = (1, 1)$. À l'instant 0, la seule tâche disponible est la tâche 0. L'ordonnancement sera donc $t_0 = 0; t_1 = 1000$, et la moyenne des temps de fin sera 1000,5. Cependant, l'ordonnancement (optimal) $t_0 = 2; t_1 = 1$ est aussi valide et a une moyenne des temps de fin égale à 502.

La caractéristique nouvelle de ce problème est qu'il peut être profitable de ne pas exécuter une tâche alors qu'une tâche est disponible à l'exécution.

Question 3

(a) De manière surprenante, et comme nous allons le prouver, les durées d'exécution des tâches n'influent pas sur leur ordre d'exécution optimal. Il faut exécuter les tâches dans l'ordre croissant des dates d'échéance, sans temps mort. On peut donc calculer l'ordonnancement en $O(n \log n)$.

Cette idée peut se trouver en résolvant à la main la question dans le cas $n = 2$.

Pour prouver ce résultat, on utilise une méthode de preuve proche de celle de la question 1b. Premièrement, il est clair qu'il faut éviter les temps morts, et donc qu'il suffit de trouver une permutation σ qui minimise le retard maximal.

Pour une permutation σ, le retard de la tâche à la i-ème position est $\max\{L_i(\sigma); 0\}$ où l'on a posé :

$$L_i(\sigma) = \left(\sum_{j=0}^{i} p_{\sigma(j)}\right) - d_{\sigma(i)}$$

et donc le retard maximal est $\max\{0; \max_{0\leq i<n} L_i(\sigma)\}$.

Examinons l'effet de l'échange de deux tâches successives $\sigma(k)$ et $\sigma(k+1)$: pour tout i tel que $i < k$ ou $i > k+1$, cet échange n'a aucun effet sur la date de fin de la tâche $\sigma(i)$, et donc sur son retard. On a donc $L_i(\sigma \circ (k, k+1)) = L_i(\sigma)$.

Concernant les tâches aux positions k et $k+1$, on pose $S = \sum_{j=1}^{k+1} p_{\sigma(j)}$, et on a :

$$\begin{aligned} L_k(\sigma) &= S - p_{\sigma(k+1)} - d_{\sigma(k)} & \qquad L_{k+1}(\sigma \circ (k, k+1)) &= S - d_{\sigma(k)} \\ L_{k+1}(\sigma) &= S - d_{\sigma(k+1)} & \qquad L_k(\sigma \circ (k, k+1)) &= S - p_{\sigma(k)} - d_{\sigma(k+1)} \end{aligned}$$

Donc, si $d_{\sigma(k+1)} < d_{\sigma(k)}$, on a $L_{k+1}(\sigma) \geq L_k(\sigma \circ (k, k+1))$ et $L_{k+1}(\sigma) \geq L_{k+1}(\sigma \circ (k, k+1))$, et donc, on en conclut :

$$\max\left\{0; \max_{0\leq i<n} L_i(\sigma \circ (k, k+1))\right\} \leq \max\left\{0; \max_{0\leq i<n} L_i(\sigma)\right\}$$

En effectuant des échanges successifs, on peut donc trier les tâches par date d'échéance croissante, tout en diminuant le retard maximal. Donc le retard maximal associé à l'ordre choisi par l'algorithme est toujours meilleur que n'importe quel autre ordre.

(b) S'il n'est pas possible d'exécuter toutes les tâches à l'heure, il va falloir en sacrifier certaines pour gagner du temps pour les autres. Mais dès qu'une tâche est sacrifiée, la date de son exécution n'importe plus : quitte à l'exécuter à la toute fin, on l'ignorer complètement dans le calcul du retard des autres tâches. La solution de cette question consistera donc à trouver un sous-ensemble maximal de tâches pouvant être toutes exécutées à l'heure. Une fois ce sous-ensemble trouvé, il suffira de trier les tâches par date d'échéance croissante, comme à la question précédente.

Pour trouver ce sous-ensemble de tâches exécutables sans retard, on essaie de les exécuter par date d'échéance croissante, jusqu'à ce que l'une de ces dates d'échéance soit dépassée. Il faut alors revenir en arrière et sacrifier l'une des tâches déjà exécutées. Pour la choisir, on remarque qu'il est toujours préférable de sacrifier la tâche la plus longue, parmi celles qui sont déjà exécutées. En effet, on gagnera alors plus de temps, et on pourra alors sacrifier moins de tâches à l'avenir. Ce choix peut donc se faire de manière "gloutonne", sans consulter les tâches ultérieures. Pour déterminer quelle est la tâche la plus longue parmi celles qui sont déjà exécutées, on peut utiliser un file de priorité.

On arrive donc à l'algorithme suivant, avec une complexité de $O(n \log n)$ en temps :

`trier les tâches par date d'échéance croissante.`

```
# tâchesChoisies est une file de priorité de tâches, permettant d'extraire
# la tâche de durée maximale
tâchesChoisies <- {}
t_cur <- 0
Pour chaque tâche i
  tâchesChoisies.push(i)
  t_cur <- t_cur + p[i]
  Si t_cur > d[i]
    j <- tâchesChoisies.pop()
    t_cur <- t_cur - p[j]

t_cur <- 0
Pour chaque tâche i dans tâchesChoisies, par date d'échéance croissante
  t[i] <- t_cur
  t_cur <- t_cur + p[i]
Pour chaque autre tâche i
  t[i] <- t_cur
  t_cur <- t_cur + p[i]
```

J5 – Constitution d'échantillons aléatoires

On se propose d'étudier plusieurs algorithmes permettant de tirer aléatoirement un élément dans un ensemble, selon une distribution donnée. Comme source de hasard, on suppose que l'on dispose de deux primitives dans notre langage de programmation :

— `unif(x)` prend un entier $x \in \mathbb{N}^*$, et renvoie un entier naturel tiré selon la distribution uniforme sur $\{0, \dots, x-1\}$;

— `bern(p)` prend un réel $p \in [0, 1]$, et renvoie un booléen tiré selon une distribution de Bernoulli de paramètre p.

On suppose que ces primitives s'exécutent en temps $O(1)$. Par ailleurs, on suppose que les opérations arithmétiques usuelles sur les réels peuvent être calculées de manière exacte en temps constant. (Ces hypothèses sont fausses en pratique, mais de bonnes approximations sont possibles.)

Question 1 Proposer un algorithme permettant de mélanger uniformément un tableau. Plus précisément, l'algorithme doit choisir uniformément aléatoirement une permutation des indices du tableau, et l'appliquer au tableau. Quelle est sa complexité en temps ?

Question 2

(a) Proposer un algorithme permettant de choisir uniformément k éléments distincts dans un ensemble fini E. Quelle est sa complexité en temps et en mémoire ?

On se place dans le cas où l'ensemble E est trop gros pour tenir dans la mémoire : le seul moyen de lire ses éléments est d'appeler une fonction `suivant_E()`, qui renverra successivement tous ses éléments. Par ailleurs, on ne connaît pas à l'avance le nombre d'éléments de E : une autre fonction `terminé_E()` renvoie un booléen indiquant si tous les éléments ont été renvoyés par `suivant_E()`. (On garantit cependant bien entendu que E contient au moins k éléments.)

(b) Proposer un algorithme permettant de choisir uniformément k éléments distincts dans un ensemble fini E. Quelle est sa complexité en temps et en mémoire ?

Question 3. On s'intéresse maintenant à tirer aléatoirement dans $\{0, \dots, n-1\}$, selon une distribution fixée. Plus précisément, on prend en entrée un tableau de poids entiers W de somme S et on veut que la probabilité de tirer l'entier i soit $\frac{W[i]}{S}$.

(a) Proposer un algorithme pour résoudre ce problème, avec une complexité $O(n)$ en temps pour chaque tirage.

On se place maintenant dans le cas où de nombreux tirages aléatoires seront effectués successivement sur cette même distribution. On veut donc que le tirage aléatoire soit rapide, quitte à passer un peu de temps à faire des *pré-calculs* sur P : ils ne seront effectués qu'une fois, et pourront être réutilisés pour faire des tirages plus rapides.

(b) Proposer un algorithme pour résoudre ce problème, qui nécessite un temps de pré-calcul $O(n)$, et qui a une complexité en temps de $O(\log n)$ pour chaque tirage. Donner un pseudo-code **détaillé** de cet algorithme.

Corrigé

Question 1 Le premier élément du tableau mélangé a une distribution uniforme sur tout le tableau. Une fois choisi ce premier élément, on peut le mettre à la première position en l'échangeant avec l'élément qui est actuellement le premier (en particulier, si l'élément choisi est déjà en première place, on ne fait rien). La distribution du mélange des éléments restants est alors uniforme sur toutes les permutations. On peut donc recommencer le processus :

```
Pour i = 0 à n-2
  j <- i + unif(n-i)
  Échange(t[i], t[j])
```

La complexité de cet algorithme est $O(n)$.

[Cet algorithme s'appelle "mélange de Fischer-Yates", voir [Wik18b].]

Autre méthode possible, moins efficace : générer explicitement le tableau de permutation en mémorisant les images déjà assignées et en faisant un nouveau tirage à chaque fois qu'on a tiré une image déjà assignée.

Question 2

(a) On peut utiliser le même algorithme qu'à la question 1, mais en s'arrêtant après avoir fait k itérations de la boucle. On prend ensuite les k premiers éléments du tableau. La complexité en temps et en mémoire est $O(n)$ (en supposant qu'on lise toute l'entrée d'abord).

(b) Tout en itérant sur les éléments de E, on va maintenir un "réservoir" de k éléments distincts sélectionnés de manière uniforme parmi les éléments déjà parcourus dans E. Lorsque l'on voit le n-ème élément, la probabilité qu'il soit dans le nouveau réservoir est $\frac{k}{n}$. Si on le choisit, alors il faut sélectionner uniformément $k-1$ éléments parmi les éléments déjà parcourus : cela peut se faire simplement en retirant un élément aléatoire dans le précédent réservoir.

On obtient donc l'algorithme suivant :

```
Pour i = 0 à k-1
  R[i] <- suivant_E()

n <- k
Tant que (Non terminé_E())
  n <- n + 1
  x <- unif(n)
  Si x < k
    R[x] <- suivant_E()
  Sinon
    ignorer(suivant_E())
```

[Il s'agit de l'algorithme R, étudié en reservoir sampling *: voir [Wik18a].]*

Question 3

(a) Soit X une variable aléatoire uniforme sur $[\![0; S-1]\!]$, et soit I la variable aléatoire définie par :

$$\sum_{k=0}^{I-1} W[k] \leq X < \sum_{k=0}^{I} W[k]$$

Les valeurs de X correspondant à une valeur i de la variable aléatoire I sont exactement $[\![\sum_{k=0}^{i-1} W[k]; \sum_{k=0}^{i} W[k] - 1]\!]$. Or, cet ensemble a pour cardinal $W[i]$. Donc $P(I = i) = \frac{W[i]}{S}$.

On peut donc utiliser l'algorithme suivant :

```
Fonction tirage(W)
  S <- 0
  Pour i = 0 à n-1
    S <- S + W[i]

  x <- unif(S)
  s <- W[0]
  i <- 0
  Tant que s <= x
    i <- i + 1
    s <- s + W[i]
  Renvoyer i
```

(b) On peut améliorer l'algorithme présenté à la question précédente en pré-calculant les sommes partielles $\sum_{k=0}^{i} W[i]$ en $O(n)$. Il suffira alors de faire une dichotomie pour calculer la valeur de I en fonction de la valeur de X. L'algorithme est donc :

```
SW[0] <- W[0]
Pour i = 1 à n-1
  SW[i] <- SW[i-1] + W[i]

Fonction tirage()
  x <- unif(SW[n-1])

  # Dichotomie pour trouver le plus grand i tel que x < SW[i]
  i_min <- 0
  i_max <- n
  Tant que i_max - i_min > 1
    i_mid <- (i_min + i_max) / 2
    Si x < SW[i_mid]
      i_min <- i_mid
    Sinon
      i_max <- i_mid
  Renvoyer i_min
```

Références

[Wik18a] Wikipedia. Reservoir sampling, 2018. https://en.wikipedia.org/wiki/Reservoir_sampling.

[Wik18b] Wikipédia. Mélange de Fisher-Yates, 2018. https://fr.wikipedia.org/wiki/M%C3%A9lange_de_Fisher-Yates.

J6 – Arithmétique de Presburger

Soit $\mathcal{X} = \{x_1, \ldots, x_n\}$ un ensemble fini de variables. Une *valuation entière* (respectivement *valuation booléenne*) sur $\mathcal{X}$ est une fonction de $\mathcal{X}$ dans $\mathbb{N}$ (respectivement dans $\{0,1\}$). On note $\mathbb{N}^{\mathcal{X}}$ (respectivement $2^{\mathcal{X}}$) l'ensemble des valuations entières (respectivement booléennes).

On considère l'alphabet $\Sigma = 2^{\mathcal{X}}$, et on associe à chaque mot $a = a_0 \cdots a_{m-1}$ de Σ^* la valuation entière ϕ_a définie par :

$$\phi_a(x_i) := \sum_{j=0}^{m} 2^j a_j(x_i)$$

On dit alors qu'un ensemble E de valuations entières est *régulier* si le langage L_E défini par :

$$L_E := \{a \in \Sigma^* \mid \phi_a \in E\}$$

est régulier (c'est à dire s'il correspond exactement aux mots reconnus par un automate fini).

Question 1

(a) Soit $x_i \in \mathcal{X}$ une variable et $v \in \mathbb{N}$ un entier. Montrer que l'ensemble de valuations entières suivant est régulier :

$$\{\phi \in \mathbb{N}^{\mathcal{X}} \mid \phi(x_i) = v\}$$

(b) Montrer que si E et F sont deux ensembles réguliers de valuations entières, alors $E \cup F$, $E \cap F$ et $\mathbb{N}^{\mathcal{X}} \setminus E$ sont réguliers.

Question 2 Soit $x_i \in \mathcal{X}$ une variable. Si $\phi \in \mathbb{N}^{\mathcal{X}\setminus\{x_i\}}$, et $u \in \mathbb{N}$, on note $\phi[x_i \leftarrow u]$ la valuation dans $\mathbb{N}^{\mathcal{X}}$ définie par :

$$\phi[x_i \leftarrow u](x_j) := \begin{cases} u & \text{si } x_i = x_j \\ \phi(x_i) & \text{sinon} \end{cases}$$

Soit $E \subseteq \mathbb{N}^{\mathcal{X}}$ un ensemble régulier de valuations entières. Montrer que les ensembles de valuations entières suivants sont réguliers :

$$E_{\exists x_i} := \{\phi \in \mathbb{N}^{\mathcal{X}\setminus\{x_i\}} \mid \exists u \in \mathbb{N},\ \phi[x_i \leftarrow u] \in E\}$$
$$E_{\forall x_i} := \{\phi \in \mathbb{N}^{\mathcal{X}\setminus\{x_i\}} \mid \forall u \in \mathbb{N},\ \phi[x_i \leftarrow u] \in E\}$$

Question 3 On dit qu'une fonction $f : \mathbb{N}^{\mathcal{X}} \to \mathbb{N}$ est *affine* s'il existe des entiers naturels $u_0, u_1, \ldots, u_n$ tels que :

$$\forall \phi \in \mathbb{N}^{\mathcal{X}},\ f(\phi) = u_0 + u_1\phi(x_1) + \cdots + u_n\phi(x_n)$$

Soient f et g deux fonctions affines de $\mathbb{N}^{\mathcal{X}}$ dans $\mathbb{N}$.

(a) Montrer que l'ensemble $\{\phi \in \mathbb{N}^{\mathcal{X}} \mid f(\phi) = g(\phi)\}$ est régulier.

(b) Montrer que l'ensemble $\{\phi \in \mathbb{N}^{\mathcal{X}} \mid f(\phi) \leq g(\phi)\}$ est régulier.

Question 4 Déduire des questions précédentes que l'ensemble suivant est régulier :

$$\{\phi \in \mathcal{X} \mid \phi(x_1) = \max(\phi(x_2), 3\phi(x_3)) \wedge (\exists u \in \mathbb{N},\ (u \leq 18 \wedge \phi(x_2) \bmod 42 = u) \vee \phi(x_3) = \phi(x_1)+1)\}$$

Suite des questions

Question 5 Proposer une généralisation des résultats précédents aux valuations sur $\mathbb{Z}$.

Question 6

(a) Les constructions d'ensembles réguliers de valuations entières vues dans les questions 1 à 4 peuvent-ils se généraliser à d'autres bases que 2 ?

(b) À l'aide des constructions d'ensembles réguliers de valuations entières vues dans les questions 1 à 4, peut-on construire tous les ensembles réguliers (au sens de la base 2) de valuations entières ?

Corrigé

Question 1

(a) On construit un automate avec $\lfloor \log_2 v \rfloor + 2$ états (1 seul état si $v = 0$), numérotés de 0 à $\lfloor \log_2 v \rfloor + 1$. Si le chiffre de poids 2^j dans la représentation en base 2 de v est v_j, alors on ajoute *toutes* les transitions $j \xrightarrow{u} j+1$, avec u une valuation booléenne telle que $u(x_i) = v_j$ pour l'indice i correspondant à la variable de l'énoncé. On ajoute aussi *toutes* les transitions $\lfloor \log_2 v \rfloor + 1 \xrightarrow{u} \lfloor \log_2 v \rfloor + 1$, avec u une valuation booléenne telle que $u(x_i) = 0$. L'état $\lfloor \log_2 v \rfloor + 1$ est le seul état final.

On voit alors facilement que cet automate reconnaît exactement le langage voulu.

(b) On prouve facilement que $L_{E \cup F} = L_E \cup L_F$, $L_{E \cap F} = L_E \cap L_F$ et $L_{\mathbb{N}^{\mathcal{X}} \setminus E} = \Sigma^* \setminus L_E$. Le résultat découle alors immédiatement de la clôture des langages rationnels par intersection, union et complémentaire.

Question 2 Commençons par $E_{\exists x_i}$: à partir d'un automate $\mathcal{A}$ reconnaissant L_E, construisons un automate $\mathcal{A}'$ reconnaissant $L_{E_{\exists x_i}}$. Le nouvel automate $\mathcal{A}'$ a les mêmes états que $\mathcal{A}$, et les mêmes états initiaux. Pour chaque transition $s \xrightarrow{u} s'$ de $\mathcal{A}$, on a une transition $s \xrightarrow{u'} s'$ dans $\mathcal{A}'$. Son étiquette $u' \in \mathbb{N}^{\mathcal{X} \setminus \{x_i\}}$ est la restriction de la valuation u à $\mathcal{X} \setminus \{x_i\}$, c'est-à-dire que pour toute variable x_j différente de x_i, on a $u'(x_j) = u(x_j)$. Les états finaux de $\mathcal{A}'$ sont tous ceux qui permettent d'atteindre un état qui est final dans $\mathcal{A}$ en ne parcourant que des transitions étiquetées dans $\mathcal{A}'$ par la valuation booléenne nulle de $2^{\mathcal{X} \setminus \{x_i\}}$ (cela est nécessaire pour prendre en compte le cas où la représentation en base 2 de $\phi(x_i)$ est plus longue que celles des autres $\phi(x_j)$).

Un mot $a_0 \dots a_m$ est reconnu par $\mathcal{A}'$ si et seulement si il existe $m' \in \mathbb{N}$ et des booléens $b_0, \dots, b_m$, $b_{m+1}, \dots, b_{m+m'}$ tels que $a_0[x_i \leftarrow b_0] \dots a_m[x_i \leftarrow b_m] 0[x_i \leftarrow b_{m+1}] \dots 0[x_i \leftarrow b_{m+m'}]$ est reconnu par $\mathcal{A}$. En effet, les booléens $b_0, \dots, b_m$ sont donnés par les étiquettes des transitions du chemin dans $\mathcal{A}$ correspondant au chemin acceptant dans $\mathcal{A}'$, et les booléens $b_{m+1}, \dots, b_{m+m'}$ sont donnés par les étiquettes des transitions d'un chemin de l'état final de $\mathcal{A}'$ vers un état final de $\mathcal{A}$.

Soient $a_0 \dots a_m$ un mot de valuations booléennes sur $\mathcal{X} \setminus \{x_i\}$. On va prouver que $\phi_{a_0 \dots a_m} \in E_{\exists x_i}$ si et seulement si $a_0 \dots a_m$ est reconnu par $\mathcal{A}'$:

— Si $a_0 \dots a_m$ est reconnu par $\mathcal{A}'$, alors il existe $m' \in \mathbb{N}$ et des booléens $b_0, \dots, b_m, b_{m+1}, \dots, b_{m+m'}$ comme ci-dessus. Mais alors :

$$\phi_{a_0 \dots a_m} \left[x_i \leftarrow \sum_{j=0}^{m+m'} 2^j b_j \right] = \phi_{a_0[x_i \leftarrow b_0] \dots a_m[x_i \leftarrow b_m] 0[x_i \leftarrow b_{m+1}] \dots 0[x_i \leftarrow b_{m+m'}]} \in E$$

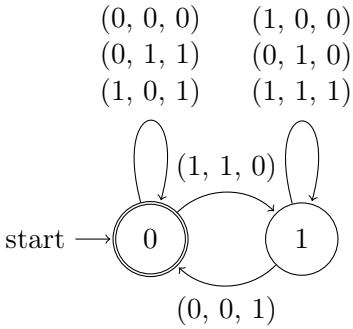
Donc $\phi_{a_0 \dots a_m} \in E_{\exists x_i}$.

— Si $\phi_{a_0 \dots a_m} \in E_{\exists x_i}$, alors il existe $u \in \mathbb{N}$ tel que $\phi_{a_0 \dots a_m}[x_i \leftarrow u] \in E$. En considérant la représentation en base 2 de u, et quitte à la compléter par des 0 dans les chiffres de poids fort pour atteindre un état final de $\mathcal{A}$, il existe un entier m' et des booléens $b_0, \dots, b_m, b_{m+1}, \dots, b_{m+m'}$ tels que $u = \sum_{j=0}^{m+m'} 2^j b_j$. Alors, le mot $a_0[x_i \leftarrow b_0] \dots a_m[x_i \leftarrow b_m]0[x_i \leftarrow b_{m+1}] \dots 0[x_i \leftarrow b_{m+m'}]$ est reconnu par $\mathcal{A}$. Et donc, le mot $a_0 \dots a_m$ est reconnu par $\mathcal{A}'$.

Pour traiter $E_{\forall x_i}$, il suffit de passer au complémentaire : $E_{\forall x_i} = \mathbb{N}^{\mathcal{X} \setminus \{x_i\}} \setminus (\mathbb{N}^{\mathcal{X}} \setminus E)_{\exists x_i}$. On conclut alors à l'aide du résultat de la question 1b.

Question 3

(a) On traite d'abord le cas où $f(\phi) = \phi(x_i) + \phi(x_j)$ et $g(\phi) = \phi(x_k)$. On cherche donc à prouver que l'ensemble $\{\phi \in \mathbb{N}^{\mathcal{X}} \mid \phi(x_i) + \phi(x_j) = \phi(x_k)\}$ est régulier. Pour cela, il suffit d'un automate à deux états. Le premier état est parcouru lorsqu'il n'y a pas de retenue à la position correspondante lors de l'addition, et l'autre correspond à la présence d'une retenue :



Une transition étiquetée (b_i, b_j, b_k) sur ce schéma correspond à toutes les transitions étiquetées par une valuation booléenne b telle que $b(x_i) = b_i$, $b(x_j) = b_j$ et $b(x_k) = b_k$.

Soit f une fonction affine avec $f(\phi) = u_0 + u_1\phi(x_1) + \dots + u_n\phi(x_n)$. Soit x_{n+1} une variable supplémentaire. On va maintenant montrer que l'ensemble $\{\phi \mid f(\phi) = \phi(x_{n+1})\}$ est régulier, par récurrence sur $\sum_{k=1}^{n} u_k$. Si cette somme est nulle, alors c'est la question 1a. Sinon, soient f' une autre fonction affine et x_i une variable telles que $f(\phi) = \phi(x_i) + f'(\phi)$. On a :

$$f(\phi) = \phi(x_{n+1}) \Leftrightarrow \exists y,\, y = f'(\phi) \wedge \phi(x_{n+1}) = \phi(x_i) + y$$

Donc :

$$\{\phi \mid f(\phi) = \phi(x_{n+1})\} = \{\phi \mid \exists y,\, y = f'(\phi) \wedge \phi(x_{n+1}) = \phi(x_i) + y\}$$

Mais, en combinant le résultat de la question 2, le résultat de la question 1 sur l'intersection, le résultat de la question 1a pour la première condition de la conjonction et le résultat ci-dessus sur l'addition pour le second membre de la conjonction, on obtient que le membre droit de cette égalité est régulier. Donc $\{\phi \mid f(\phi) = \phi(x_{n+1})\}$ est régulier.

Finalement, si f et g sont deux fonctions affines, alors :

$$f(\phi) = g(\phi) \Leftrightarrow \exists y, y = f(\phi) \wedge y = g(\phi)$$

Donc, encore une fois, en combinant les résultats précédents, on obtient que l'ensemble $\{\phi \mid f(\phi) = g(\phi)\}$ est régulier.

(b) On a :

$$f(\phi) \leq g(\phi) \Leftrightarrow \exists y \in \mathbb{N},\, f(\phi) + y = g(\phi)$$

En combinant les résultats précédents, on obtient donc bien que $\{\phi \mid f(\phi) \leq g(\phi)\}$ est régulier.

Question 4 On remarque que :

$$a \bmod 42 = b \Leftrightarrow b < 42 \wedge \exists c \in \mathbb{N}, a = 42c + b$$
$$a = \max(b, c) \Leftrightarrow b \leq a \wedge b \leq a \wedge (a = b \vee a = c)$$

La condition proposée est donc équivalente à une condition ne faisant intervenir que des constructions que nous avons déjà vues dans les questions précédentes. Il s'agit donc bien d'un ensemble régulier.

Question 5 On ne détaillera pas les constructions ici. Plusieurs approches sont possibles :

- On peut encoder chaque entier de $u \in \mathbb{Z}$ par tout couple d'entiers $u^{(+)}, u^{(-)} \in \mathbb{N}$ tel que $u = u^{(+)} - u^{(-)}$. Un entier relatif est alors représenté par une infinité de couples d'entiers naturels, et les constructions d'ensembles réguliers vues dans les questions précédentes se transposent facilement à cet encodage. La construction la plus délicate est celle de la question 3 : on peut traduire une égalité (ou inégalité) de fonctions affines sur $\mathbb{Z}$ en une égalité (ou inégalité) de fonctions affines sur $\mathbb{N}$ dépendant des couples d'entiers vus plus haut.
- Dans l'encodage précédent, on peut rajouter la contrainte $u^{(+)} = 0$ ou $u^{(-)} = 0$. Cela a pour avantage d'avoir une représentation unique pour chaque entier relatif, mais ne simplifie en fait pas vraiment les preuves.
- On peut construire directement des automates travaillant sur une représentation en base 2 des entiers relatifs (par exemple, en complément à 2 ou en complément à 1). Le bit de signe peut soit être le bit de poids le plus fort, soit être stocké dans une variable séparée.

Question 6

(a) Toutes les constructions peuvent se transposer à d'autres bases (entières, supérieures à 2). La seule construction nécessitant une adaptation est l'automate pour l'addition présenté à la question 3a : dans la généralisation à la base b, il faut b états, un pour chaque valeur de la retenue.

(b) On va exhiber un ensemble régulier au sens de la base 2, mais pas au sens de la base 3. Cet ensemble ne sera donc pas constructible avec les constructions des questions 1 à 4.
L'ensemble $\{\phi \mid \exists k \in \mathbb{N},\ \phi(x_1) = 2^k\}$ est clairement régulier au sens de la base 2. Cependant, il ne l'est pas au sens de la base 3. En effet : supposons qu'il l'est. Il existerait alors un k tel que la représentation en base 3 de 2^k passe deux fois par le même état dans l'automate. En répétant la boucle, on obtient une suite arithmético-géométrique de la forme :

$$u_{k+1} = 3^\alpha u_k + \beta \qquad \alpha \in \mathbb{N}, \quad \beta \in \mathbb{Z}$$

dont les représentations en base 3 des éléments sont reconnus par l'automate. Donc tous les u_k sont des puissances de 2.
Pour obtenir une contradiction, on étudie cette suite. On peut résoudre la récurrence linéaire, et obtenir :

$$u_k = \gamma 3^{\alpha k} + \delta \qquad \gamma, \delta \in \mathbb{R}$$

Et donc, pour tout k, $\log_2(\gamma 3^{\alpha k} + \delta) \in \mathbb{N}$. Or :

$$\log_2(\gamma 3^{\alpha k} + \delta) = \log_2(\gamma) + \log_2(1 + \frac{\delta}{\gamma 3^{\alpha k}}) + k\alpha \log_2(3)$$

Où $\log_2(\gamma) + \log_2(1 + \frac{\delta}{\gamma 3^{\alpha k}})$ converge quand k tend vers $+\infty$. Mais, puisque $\log_2(3)$ est irrationnel (facile !), les parties fractionnelles de $k\alpha \log_2(3)$ sont denses dans $[0, 1]$, ce qui est contradictoire avec le fait que les $\log_2(\gamma 3^{\alpha k} + \delta)$ sont tous des entiers.

Référence

Carton, O. (2008). *Langages formels, calculabilité et complexité.* Section 3.8.2 *Arithmétique de Presburger.* Vuibert.

J7 – Détection de cycle

Dans ce problème, on admettra qu'il est possible d'implémenter la structure de donnée de dictionnaire à clés entières de façon à ce que les accès en lecture et en écriture se fassent en temps constant, et en n'occupant qu'un espace mémoire linéaire en le nombre d'entrées dans le dictionnaire.

Soit $N > 2$ un entier, f une fonction de $[\![0, N-1]\!]$ dans lui-même, et $x_0 \in [\![0, N-1]\!]$. On définit la suite $(x_i)_{i\in\mathbb{N}}$ par :

$$x_{i+1} = f(x_i)$$

Question 0 Montrer qu'il existe i et j deux entiers naturels tels que $x_i = x_j$ et $i < j$.

On pose :

$$\mu = \min\{i \in \mathbb{N} \mid \exists j \in \mathbb{N},\, j > i \wedge x_i = x_j\}$$
$$\lambda = \min\{i \in \mathbb{N}^* \mid x_\mu = x_{\mu+i}\}$$

Le but de ce problème est de trouver des algorithmes *efficaces* pour calculer μ et λ, étant donnée une implémentation de f.

Question 1

(a) Donner un algorithme simple pour calculer λ et μ, qui est efficace en temps. Quel est le nombre (exact) d'appels à f effectués ? Quelle est la complexité en mémoire, en fonction de λ, μ et N ? Le nombre d'appels à f est-il optimal ?

(b) Donner un algorithme simple pour calculer λ et μ, qui est efficace en mémoire. Quelle est sa complexité en temps et en mémoire, en fonction de λ, μ et N ?

On cherche maintenant un algorithme efficace en temps *et* en mémoire.

Question 2

(a) Quels sont les éléments de l'ensemble $\{n \in \mathbb{N}^* \mid x_n = x_{2n}\}$, en fonction de λ et μ ?

(b) En déduire un algorithme pour calculer λ et μ en temps $O(\lambda + \mu)$ et en espace mémoire $O(1)$.

(c) Combien d'appels à f sont effectués au total par cet algorithme ?

Corrigé

Question 0 Si tel n'était pas le cas, alors $i \mapsto x_i$ serait une injection de $\mathbb{N}$ vers $[\![0, N-1]\!]$. Mais $\mathbb{N}$ est infini alors que $[\![0, N-1]\!]$ est fini. Absurde.

Question 1

(a) On peut utiliser l'algorithme suivant :

```
dict <- {}
i <- 0
x <- x0
Tant que (Non dict.contient(x))
  dict.Insere(x, i)
  x <- f(x)
  i <- i + 1
mu <- dict.lire(x)
lambda <- i - mu
```

Il effectue exactement $\lambda + \mu$ appels à f et a une complexité en mémoire de $O(\lambda + \mu)$ (toutes les valeurs calculées des itérées de f sont dans le dictionnaire). Le nombre d'appels à f est optimal, si on ne connaît rien sur f. En effet, tant que l'on a pas calculé $x_{\lambda+\mu}$, les valeurs connues de f laissent possibles plusieurs valeurs de λ et μ, selon la valeur choisie pour la prochaine itération de f.

(b) On peut essayer toutes les valeurs possibles de i et j. On obtient alors un algorithme $O(1)$ en mémoire, mais qui effectue $O((\lambda + \mu)^2)$ appels à f :

```
xj <- f(x0)
j <- 1
Répéter
  xi <- x0
  Pour i = 0 à j-1
    Si xi = xj
      mu <- i
      lambda <- j - i
      Stop
    xi <- f(xi)
  xj <- f(xj)
  j <- j+1
```

Question 2

(a) À partir de l'indice μ, la suite des x_i est périodique, de période λ. Donc, pour $n' > n \geq 0$, on a $x_n = x_{n'}$ si et seulement si $n' - n$ est un multiple de λ et $n \geq \mu$. Donc, pour $n \geq 0$, on a $x_n = x_{2n}$ si et seulement si n est multiple de λ et $n \geq \mu$. On en déduit :

$$\{n \in \mathbb{N}^* \mid x_n = x_{2n}\} = \{n_0, n_0 + \lambda, n_0 + 2\lambda, \ldots\} \quad \text{où } n_0 = \lambda \left\lceil \frac{\mu}{\lambda} \right\rceil$$

(b) On calcule simultanément tous les x_n et x_{2n}, en comparant ces deux nombres jusqu'à trouver n_0. Ensuite, en partant de x_0 et de x_{n_0}, on calcule tous les x_i et x_{n_0+i}, jusqu'à trouver un i tel que $x_i = x_{n_0+i}$. Puisque n_0 est multiple de λ, on en déduit que la dernière valeur de i est égale à μ. Par ailleurs, si l'un des x_{n_0+i} calculés est égal à x_{n_0} avec $i > 0$, alors le plus petit tel i est λ. Sinon, c'est que $\lambda < \mu$, et donc $n_0 = \lambda$.

```
xn <- f(x0)
x2n <- f(f(x0))
n <- 1
Tant que xn <> x2n
  xn <- f(xn)
  x2n <- f(f(x2n))
  n <- n + 1

xni <- xn
xi <- x0
i <- 0
lambda <- -1
Tant que xi <> xni
  xi <- f(xi)
  xni <- f(xni)
  i <- i + 1
  Si lambda = -1 Et xni = xn
    lambda <- i
mu <- i
Si lambda = -1
  lambda <- n
```

(c) Lors de la première phase, on appelle $3n_0$ fois f. Lors de la deuxième phase, on appelle 2μ fois f. Au total, on appelle donc $3\lambda \left\lceil \frac{\mu}{\lambda} \right\rceil + 2\mu$ fois la fonction f.

Référence

Knuth, D. E. (1998). *The art of computer programming, 2 : seminumerical algorithms*. Addision Wesley. Exercice 3.1-6.

J8 – Sommes d'intervalles dans un tableau

Soit T un tableau d'entiers relatifs, de taille $N > 0$.

Question 1 Donner un algorithme pour calculer des indices i et j tels que $0 \leq i \leq j < N$ et tels que la somme $\sum_{k=i}^{j} T[k]$ soit maximale. Quelle est sa complexité en temps ?

Question 2 Si, pour une valeur de j donnée, on connaît la valeur de $\max_{i \in [\![0,j]\!]} \sum_{k=i}^{j} T[k]$, comment calculer efficacement $\max_{i \in [\![0,j+1]\!]} \sum_{k=i}^{j+1} T[k]$? En déduire un algorithme en temps linéaire pour répondre à la question 1.

Question 3 Soit $m \in [\![1, N]\!]$. Donner un algorithme pour calculer en temps linéaire un indice $i \in [\![0, N-m]\!]$ tel que la somme $\sum_{k=i}^{i+m-1} T[k]$ soit maximale.

Question 4 Donner un algorithme efficace pour calculer des indices i et j optimaux comme aux questions 1 et 2, mais avec la contrainte supplémentaire $j - i < m$. Quelle est sa complexité en temps et en espace ?

Suite des questions

Question 5 Soit $U \in [\![1, N]\!]$. On veut maintenant trouver plusieurs indices $0 \leq i_1 \leq j_1 < i_2 \leq j_2 < \cdots < i_u \leq j_u < N$, avec $0 \leq u \leq U$ tels que la somme $\sum_{l=1}^{u} \sum_{k=i_l}^{j_l} T[k]$ soit maximale. Proposer un algorithme qui calcule la somme maximale. Quelle est sa complexité en temps et en espace ?

Corrigé

Question 1 **On attend ici un algorithme en $O(N^2)$.** Par exemple, pour chaque valeur de i, on peut calculer la somme pour chaque valeur de j en se servant de la somme pour $j-1$:

```
maxs <- T[0]
maxi <- 0
maxj <- 0
Pour i = 0 à N-1
  s <- 0
  Pour j = i à N-1
    s <- s + T[j]
    Si s > maxs
      maxs <- s
      maxi <- i
      maxj <- j
Renvoyer (maxi, maxj)
```

Question 2 On a :

$$\max_{i \in [\![0, j+1]\!]} \sum_{k=i}^{j+1} T[k] = T[j+1] + \max \left\{ 0, \max_{i \in [\![0,j]\!]} \sum_{k=i}^{j} T[k] \right\}$$

On en déduit l'algorithme suivant (algorithme de Kadane) :

```
maxs <- T[0]
maxi <- 0
maxj <- 0
i <- 0
s <- T[0]
Pour j = 1 à N-1
  Si s < 0
    i <- j
    s <- 0
  s <- s + T[j]
  Si s > maxs
    maxs <- s
    maxi <- i
    maxj <- j
Renvoyer (maxi, maxj)
```

Question 3 On essaie toutes les valeurs de i possibles, en utilisant une fenêtre glissante : on calcule la valeur de la somme grâce à la valeur de la somme à l'itération précédente. On obtient l'algorithme suivant :

```
s <- 0
Pour i = 0 à m-1
  s <- s + T[i]
maxs <- s
maxi <- 0
Pour i = 1 à N-m
  s <- s + T[i+m-1] - T[i-1]
  Si s > maxs
    maxs <- s
    maxi <- i
Renvoyer maxi
```

Question 4 La tentation serait d'adapter les algorithmes des questions 2 et 3 et d'obtenir l'algorithme INCORRECT suivant :

```
maxs <- T[0]
maxi <- 0
maxj <- 0
i <- 0
s <- T[0]
Pour j = 1 à N-1
  Si s < 0
    i <- j
    s <- 0
  s <- s + T[j]
  Si j-i >= m
    i <- i + 1
  Si s > maxs
    maxs <- s
    maxi <- i
    maxj <- j
Renvoyer (maxi, maxj)
```

En effet, sur la séquence $(3, -1, 1, 10)$ avec $m = 3$, cet algorithme renverrait $(1, 3)$, pour une somme de 10, alors que l'optimum est 11.

Un algorithme toujours correct est l'adaptation de l'algorithme de la question 1. On obtient alors une complexité de $O(mN)$. À cette question, on attend un algorithme plus efficace.

Indication : Proposer une structure de données permettant, après un pré-calcul en temps linéaire, de calculer en temps constant les sommes $\sum_{k=i}^{j} T[k]$ à partir des indices i et j. Réponse à l'indication : calculer en temps linéaire les sommes $S[i] = \sum_{k=0}^{i-1} T[k]$, pour $0 \leq i \leq n$, puis on a $\sum_{k=i}^{j} T[k] = S[j+1] - S[i]$.

Pour obtenir un algorithme en $O(N \log N)$, on va itérer sur tous les j possibles, et on stocke dans une file de priorité les valeurs de $S[i]$ pour $j - m < i \leq j$ et $0 \leq i$. Pour une valeur de j possible, il sera alors facile de trouver la valeur de i minimisant $S[i]$, et donc maximisant $S[j+1] - S[i]$. Lorsque la valeur de j change, il faut mettre à jour la file de priorité : il faut rajouter une nouvelle valeur (ce qu'on sait faire facilement), et il faudrait en retirer une autre (ce qui est plus difficile, car non supporté par les files de priorités habituelles). Au lieu d'effectuer ce retrait au moment de la mise à jour de j, on le fait de manière paresseuse, en ignorant les éléments non pertinents au moment où ils sortent de la file de priorité.

On obtient l'algorithme suivant :

```
maxs <- T[0]
```

```
maxi <- 0
maxj <- 0
file <- {(0, 0)}
s <- T[0]
Pour j = 1 à N-1
  s <- s + T[j]
  Tant que file.top().snd <= j - m
    file.pop()
  Si s - file.top().fst > maxs
    maxs <- s - file.top().fst
    maxi <- file.top().snd
    maxj <- j
  file.push((s, j))
Renvoyer (maxi, maxj)
```

Chaque $S[i]$ est inséré une fois dans la file de priorité. Sa taille est donc bornée par $N+1$ et les opérations d'insertion et d'extraction sont en $O(\log N)$. Par ailleurs, le corps de la boucle `Tant que` n'est exécuté au total que $O(N)$ fois, puisque chaque élément ne peut être retiré qu'une fois. L'algorithme est donc en temps $O(N \log N)$. En espace, la seule structure de données et la file de priorité : la complexité est $O(N)$.

Avec une implémentation de file a priorité permettant de retirer des éléments quelconques, on peut obtenir une complexité en temps $O(N \log m)$ et en espace $O(m)$.

Question 5 Bien qu'un algorithme en $O(N)$ existe [BC07], on attend ici un algorithme de programmation dynamique en $O(NU)$. Cet algorithme va résoudre le problème pour des plus petites valeurs de U et des préfixes de T. Plus précisément, on note $f(n,u)$ la somme maximale de au plus u segments disjoints dans le préfixe $T[0 \dots n-1]$. On a les relations :

$$
\begin{aligned}
f(0,u) &= 0 && u \geq 0 \\
f(n,0) &= 0 && n \geq 0 \\
f(n,u) &= \max\left\{ f(n-1,u),\quad \max_{0 \leq i < n} f(i,u-1) + \sum_{k=i}^{n-1} T[k] \right\} && n > 0,\ u > 0
\end{aligned}
$$

Posons $g(n,u) = \max_{0 \leq i < n} f(i,u-1) + \sum_{k=i}^{n-1} T[k]$, de sorte à pouvoir récrire :

$$
f(n,u) = \max\{f(n-1,u),\ g(n,u)\} \qquad n > 0,\ u > 0
$$

On a alors :

$$
\begin{aligned}
g(0,u) &= -\infty && u > 0 \\
g(n+1,u) &= T[n] + \max\{g(n,u),\ f(n,u-1)\} && n \geq 0,\ u > 0
\end{aligned}
$$

La dernière relation est à rapprocher de celle vue à la question 2. En calculant les valeurs de f et de g (dans un tableau, par exemple), on obtient un algorithme en temps $O(NU)$ (la valeur recherchée est $f(N,U)$). En ne stockant en mémoire que les valeurs de f et de g correspondant à une "colonne" fixée (i.e., une valeur de u fixée), cet algorithme est en $O(U)$ en mémoire.

Références

[BC07] Fredrik Bengtsson and Jingsen Chen. Computing maximum-scoring segments optimally, 2007. https://www.diva-portal.org/smash/get/diva2:995901/FULLTEXT01.pdf.

J9 – Tas et compagnie

Dans ce sujet, les éléments de tout tableau T de taille n sont numérotés de 0 à $n-1$.

Question 1.

(a) Qu'est-ce qu'un tas ? Comment peut-on utiliser un tableau pour implémenter un tas ?

(b) Proposer un algorithme pour transformer, en temps linéaire, un tableau quelconque d'entiers en un tas tel que décrit dans la question précédente, en conservant ses éléments.

Question 2. On dit qu'un tableau T d'entiers de taille $2n$ est un *tas d'intervalles* s'il vérifie les propriétés suivantes :

$$T\left[2\left\lfloor\tfrac{i-1}{2}\right\rfloor\right] \leq T[2i] \qquad i \in [\![1, n-1]\!] \tag{1}$$

$$T\left[2\left\lfloor\tfrac{i-1}{2}\right\rfloor + 1\right] \geq T[2i+1] \qquad i \in [\![1, n-1]\!] \tag{2}$$

$$T[2i] \leq T[2i+1] \qquad i \in [\![0, n-1]\!] \tag{3}$$

(a) Dans un tas d'intervalles, où sont les entiers le plus petit et le plus grand ?

(b) Proposer un algorithme pour extraire simultanément le minimum et le maximum d'un tas d'intervalles. Quelle est sa complexité ?

(c) Prouver que, pour un tableau d'entiers T quelconque, si T vérifie (1) et (2), alors (3) est vraie pour tout $i \in [\![0, n-1]\!]$ si et seulement si elle est vraie pour tout $i \in \left[\!\left[\left\lfloor\tfrac{n}{2}\right\rfloor, n-1\right]\!\right]$.

(d) Proposer un algorithme pour insérer deux entiers dans un tas d'intervalles. Quelle est sa complexité ?

(e) Montrer qu'un tas d'intervalles permet d'implémenter une structure de données représentant un ensemble d'entiers distincts et répondant efficacement aux requêtes suivantes :

- lecture du plus petit entier ;
- lecture du plus grand entier ;
- extraction du plus petit entier ;
- extraction du plus grand entier ;
- insertion d'un entier.

Quelle est la complexité de chacune de ces opérations ?

Corrigé

Question 1.

(a) **On ne demande pas, dans cette question, de donner l'implémentation de chaque opération de file de priorité sur un tas.** Un tas permet d'implémenter une structure de file de priorité supportant l'opération d'extraction du minimum et d'insertion d'un élément. Ces opérations peuvent toutes deux être effectuées en temps logarithmique en fonction de la taille du tas. Un tas est un arbre binaire tel que chaque nœud contient une valeur plus faible que chacun de ses deux fils (s'ils existent).

On peut stocker un tas dans un tableau : dans ce cas, on maintient l'invariant :

$$\forall i \in [\![1, T.\text{taille} - 1]\!],\ T\left[\left\lfloor \frac{i-1}{2} \right\rfloor\right] \leq T[i]$$

(b) On construit le tas en partant du bas, en faisant descendre les éléments si ils ne respectent pas la condition de tas :

```
Pour i de n-1 à 0
  j <- i
  Tant que (2*j+1 < n Et T[j] > T[2*j+1]) Ou (2*j+2 < n Et T[j] > T[2*j+2])
    Si 2*j+2 < n Et T[2*j+2] < T[2*j+1]
      Échanger(T[j], T[2*j+2])
      j <- 2*j+2
    Sinon
      Échanger(T[j], T[2*j+1])
      j <- 2*j+1
```

Les nœuds situés au niveau k (c'est-à-dire dans une cellule i telle que $2^k - 1 \leq i < 2^{k+1} - 1$) descendront au plus de $K - 1 - k$ niveaux, où $K = O(\log_2 n)$ est le nombre total de niveaux. Puisqu'il y a au plus 2^k nœuds au niveau k, le nombre total de niveaux descendus est :

$$\sum_{k=0}^{K-1} 2^k (K - 1 - k) = 2^{K-1} \sum_{k'=0}^{K-1} \frac{k'}{2^{k'}}$$

Or, la série $\sum_{k'=0}^{+\infty} \frac{k'}{2^{k'}}$ est convergente, on a $2^{K-1} = O(n)$, et le nombre total de niveaux descendus est le nombre total d'itérations de la boucle intérieure. Donc cet algorithme est bien en $O(n)$.

Question 2.

(a) Chaque couple de cellules $(T[2i], T[2i + i])$ dans le tableau correspond à un nœud dans lequel on stocke un intervalle (d'où la contrainte $T[2i] \leq T[2i + 1]$). La propriété de tas d'intervalles revient à dire que l'intervalle d'un nœud est toujours inclus dans l'intervalle de son parent. Ainsi, l'intervalle du nœud racine inclut tous les entiers du tas d'intervalles. Par conséquent, le minimum est $T[0]$ et le maximum $T[1]$.

(b) On vient de montrer que le minimum et le maximum se trouvent dans les deux premières cases du tableau. Pour les retirer, il faut donc trouver une nouvelle valeur pour ces cases du tableau : on diminue de 2 la taille du tableau, et on prend le contenu des deux dernières cellules pour les mettre dans les premières cases. Il faut alors reconstituer la structure de tas de manière analogue à l'algorithme habituel pour les tas : on fait descendre dans le tas les deux entiers nouvellement positionnés dans les premières cases. En plus de maintenir les deux premières propriétés, qui correspondent à des propriétés "standard" de tas, il faut veiller à bien conserver la propriété $\forall i \in [\![0, n - 1]\!],\ T[2i] \geq T[2i + 1]$: si elle n'est plus vérifiée, alors on échange $T[2i]$ et $T[2i + 1]$, puis on continue. On obtient l'algorithme suivant :

```
Fonction extraitMinMax(T)
  T[0] = T[T.taille - 2]
  T[1] = T[T.taille - 1]
  T.taille -= 2

  i <- 0
  Tant que (4*i+2 < T.taille Et T[2*i] > T[4*i+2]) Ou
           (4*i+4 < T.taille Et T[2*i] > T[4*i+4])
    Si 4*i+4 < T.taille Et T[4*i+4] < T[4*i+2]
      Échanger(T[2*i], T[4*i+4])
      i <- 2*i+2
    Sinon
      Échanger(T[2*i], T[4*i+2])
      i <- 2*i+1
    Si T[2*i] > T[2*i+1]
      Échanger(T[2*i], T[2*i+1])

  i <- 0
  Tant que (4*i+3 < T.taille Et T[2*i+1] < T[4*i+3]) Ou
           (4*i+5 < T.taille Et T[2*i+1] < T[4*i+5])
    Si 4*i+5 < T.taille Et T[4*i+5] > T[4*i+3]
      Échanger(T[2*i+1], T[4*i+5])
      i <- 2*i+2
    Sinon
      Échanger(T[2*i+1], T[4*i+3])
      i <- 2*i+1
    Si T[2*i] > T[2*i+1]
      Échanger(T[2*i], T[2*i+1])
```

Il s'exécute en $O(\log n)$.

(c) C'est évidemment une condition nécessaire. Montrons que c'est une condition suffisante. On suppose donc que T vérifie (1) et (2).

On va montrer que si (3) est vraie à l'indice $i > 0$, alors elle est vraie à l'indice $\left\lfloor \frac{i-1}{2} \right\rfloor$ (c'est-à-dire son parent dans le tas). On suppose donc $T[2i] \leq T[2i+1]$. On a :

$$T\left[2\left\lfloor \tfrac{i-1}{2} \right\rfloor\right] \leq T[2i] \leq T[2i+1] \leq T\left[2\left\lfloor \tfrac{i-1}{2} \right\rfloor + 1\right]$$

D'où $T\left[2\left\lfloor \frac{i-1}{2} \right\rfloor\right] \leq T\left[2\left\lfloor \frac{i-1}{2} \right\rfloor + 1\right]$, c'est à dire (3) à l'indice $\left\lfloor \frac{i-1}{2} \right\rfloor$

Alors, par récurrence, il découle que (3) est vraie partout dès qu'elle est vraie pour les feuilles. C'est ce qu'il faut démontrer, puisque les feuilles sont aux positions $i \in [\![\left\lfloor \frac{n}{2} \right\rfloor, n-1]\!]$.

(d) L'idée de départ est de procéder indépendamment dans le tas des positions paires et dans celui des positions impaires. Procéder ainsi permettra bien entendu de respecter (1) et (2). Cependant, rien ne garantit a priori que (3) serait conservée.

Pour résoudre ce problème, on va utiliser la question précédente : notre algorithme va renvoyer un tableau vérifiant les propriétés (1) et (2) partout et la propriété (3) seulement pour les indices $i \in [\![\left\lfloor \frac{n}{2} \right\rfloor, n-1]\!]$. Il est notable, de plus, que l'algorithme habituel d'insertion dans un tas ne modifie qu'un seul nœud à un tel indice : il s'agit de la nouvelle feuille.

Indication : Dans quel ordre sont disposés les éléments stockés dans les ancêtres de la nouvelle feuille ? Du coup, dans lequel des deux tas doit-on insérer les nouveaux éléments ?

Les éléments stockés dans la branche des ancêtres de la nouvelle feuille sont totalement triés : tous les ancêtres dans le tas min (aux positions paires) sont plus petits que tous les ancêtres dans

le tas max (aux positions impaires). De plus, dans le tas min, les éléments sont triés dans l'ordre décroissant en montant dans la branche, et dans le tas max, il le sont dans l'ordre croissants.

Du coup, on regarde où on doit insérer les deux nouveaux éléments dans cette séquence triée : en notant i l'indice de la nouvelle feuille, si les deux nouveaux éléments sont plus petits que $T\left[2\left\lfloor\frac{i-1}{2}\right\rfloor\right]$, alors il faut les insérer tous les deux dans le tas min. Au contraire, si ils sont plus grands que $T\left[2\left\lfloor\frac{i-1}{2}\right\rfloor+1\right]$, alors ils iront dans le tas max. Si aucune de ces conditions ne s'applique, alors le plus petit va dans le tas min et le plus grand dans le tas max.

En appliquant cette règle, les éléments de la nouvelle branche seront toujours dans le bon ordre, et (3) sera donc respectée à la feuille. Par conséquent, l'arbre renvoyé sera bien un arbre d'intervalle.

L'algorithme obtenu est alors :

```
Fonction insère_min(i, T)
  Tant que i > 0 Et T[2*i] > T[2*((i-1)/2)]
    Échanger(T[2*i], T[2*((i-1)/2)])
    i <- (i-1)/2

Fonction insère_max(i, T)
  Tant que i > 0 Et T[2*i+1] > T[2*((i-1)/2)+1]
    Échanger(T[2*i+1], T[2*((i-1)/2)+1])
    i <- (i-1)/2

Fonction insère(a, b, T)
  Si b < a
    insère(b, a, T)
  Sinon
    i <- T.taille / 2
    T.taille += 2
    Si i > 0 Et b < T[2*((i-1)/2)]
      T[2*i] = b
      insère_min(i, T)
      T[2*i+1] = T[2*i]
      T[2*i] = a
      insère_min(i, T)
    Sinon Si i > 0 Et a > T[2*((i-1)/2)+1]
      T[2*i+1] = b
      insère_max(i, T)
      T[2*i] = T[2*i+1]
      T[2*i+1] = a
      insère_max(i, T)
    Sinon
      T[2*i] = a
      T[2*i+1] = b
      insère_min(i, T)
      insère_max(i, T)
```

La complexité de cet algorithme est $O(\log n)$.

(e) Si l'ensemble à stocker est de cardinal impair, on met un élément de côté et on se ramène à un cardinal pair. S'il est de cardinal pair, on utilise simplement un tas d'intervalles. Les opérations demandées se programment facilement avec les opérations spécifiées dans les questions précédentes.

Ainsi, les algorithmes qui consistent à lire le maximum ou le minimum sont en $O(1)$, et ceux qui consistent à modifier la structure de données sont en $O(\log n)$.

[Pour plus de détails sur les tas d'intervalles, consulter l'article original [vLW93].]

Références

[vLW93] Jan van Leeuwen and Derick Wood. Interval heaps. *The Computer Journal*, 36(3) :209–216, 1993. https://academic.oup.com/comjnl/article/36/3/209/311525.

J10 – Arithmétique d'Avizienis

Soient a et b deux entiers positifs vérifiant $\frac{b}{2} < a < b$. On introduit le système de numération d'Avizienis : celui-ci s'apparente au système de numération en base b, mais il utilise $2a+1$ chiffres qui vont de $-a$ à a. En pratique, on note $\bar{1}, \bar{2}, \ldots$ pour les chiffres négatifs.

Plus précisément, si $x_n, \ldots, x_0$ sont de tels chiffres (des éléments de $[\![-a, a]\!]$, donc), alors l'interprétation de $x_n \cdots x_0$ est :

$$(x_n \cdots x_0)_{\text{avi}} = \sum_{k=0}^{n} x_k b^k$$

Question 1.

(a) La représentation d'un entier dans le système de numération d'Avizienis est-elle unique ?

(b) Quels nombres peut-on représenter avec n chiffres ?

Question 2.

(a) Quelles sont les représentations de 0 ?

(b) Proposer un algorithme qui détermine le signe d'un nombre non nul représenté avec le système de numération d'Avizienis. Dans quel cas cet algorithme est-il particulièrement efficace ?

Un circuit logique combinatoire est un graphe orienté acyclique tel que :

— Les nœuds avec seulement des arêtes sortantes sont appelés entrées du circuit, on les note $e_1 \ldots e_n$.

— Les nœuds avec seulement des arêtes entrantes sont appelés des sorties du circuit, notés $s_1 \ldots s_m$. Ces nœuds de sortie ont obligatoirement exactement une arête entrante.

— Les nœuds internes (i.e., ayant des arêtes entrantes et sortantes) sont également appelées portes. On en utilise ici de quatre types, étiquetés respectivement NON, ET, OU ou XOU. Les nœuds NON ont une arête entrante exactement, tandis qu'un nœud ET, OU ou XOU a deux arêtes entrantes exactement.

Un calcul du circuit consiste à affecter des 0 ou 1 à chaque nœud d'entrée, puis à propager ces booléens le long des arêtes, en effectuant au passage les calculs indiqués par les nœuds internes. De la sorte, un circuit de n entrées et m sorties peut être associé à une fonction de $\{0,1\}^n$ dans $\{0,1\}^m$.

La profondeur d'un circuit est la longueur du plus long chemin entre une entrée et une sortie.

Question 3.

(a) Donner un circuit prenant 3 booléens (vus comme des éléments de $\{0,1\}$) en entrée et calculant leur somme représentée en base 2 sur 2 bits

(b) En déduire un circuit qui prend en entrée deux entiers positifs ou nuls représentés en base 2 sur n bits et qui calcule la somme représentée en base 2 sur $n+1$ bits. Quelle est sa profondeur, asymptotiquement ?

(c) Proposer un circuit permettant de comparer deux entiers positifs ou nuls représentés en base 2 sur n bits. Quelle est sa profondeur, asymptotiquement ?

Question 4. Montrer qu'il existe un circuit logique combinatoire prenant en entrée deux nombres représentés dans le système d'Avizienis et qui calcule une représentation de leur somme dans le système d'Avizienis. Le circuit aura une profondeur bornée (indépendamment de n), et on choisira une représentation appropriée pour les entrées et sorties sous forme de booléens.

Suite des questions

Question 5. Proposer un circuit logique combinatoire permettant de multiplier deux entiers positifs ou nuls donnés en base 2 sur n bits, en renvoyant un entier sur $2n$ bits. Ce circuit aura une profondeur $O(\log n)$ et contiendra $O(n^2)$ portes.

Corrigé

Question 1.

(a) La représentation d'un entier dans le système de numération d'Avizienis n'est pas unique. C'est en particulier le cas à cause de la possibilité d'avoir des zéros initiaux (cf question 2a), mais c'est le cas même si on interdit ces derniers, i.e., si on impose que la représentation soit normalisée (cf question 2b). Par exemple, $(1\bar{a})_{\text{avi}} = b - a$, où $0 < b - a < \frac{b}{2}$. Donc $b - a$ a au moins deux représentations : c'est lui-même un chiffre, et c'est aussi $(1\bar{a})_{\text{avi}}$

(b) Le plus grand nombre représentable (tous ses chiffres sont égaux à a) est $\sum_{k=0}^{n-1} ab^k = a\frac{b^n-1}{b-1}$, et le plus petit nombre représentable est son opposé, $-a\frac{b^n-1}{b-1}$.

Prenons maintenant un $x \in \left[\!\left[-a\frac{b^n-1}{b-1}, a\frac{b^n-1}{b-1}\right]\!\right]$, et montrons qu'il est représentable. Sans perte de généralité, on peut supposer que $x \geq 0$ (si $x < 0$, on prend la représentation de $-x$ et on prend l'opposé de tous les chiffres). Considérons $y = x + (b - a - 1)\frac{b^n-1}{b-1}$: on obtient facilement $0 \leq y < b^n$, donc y se représente en base b avec n chiffres. Soit $(y_{n-1} \cdots y_0)_b$ cette représentation. Pour $i \in [\![0, n-1]\!]$, on pose $x_i = y_i - (b-a-1)$. On va prouver que $(x_n \cdots x_0)$ est une représentation d'Avizienis de x : tout d'abord, pour tout i, on a $x_i \in [\![a - b + 1, a]\!] \subseteq [\![-a, a]\!]$. Par ailleurs :

$$\sum_{k=0}^{n} x_k b^k = -(b - a - 1)\frac{b^n - 1}{b - 1} + \sum_{k=0}^{n} y_k b^k = -(b - a - 1)\frac{b^n - 1}{b - 1} + y = x$$

Donc x est bien représentable avec n chiffres dans la représentation d'Avizienis.

Question 2.

(a) $(0 \cdots 0)_{\text{avi}}$ sont les seules représentations de 0. Supposons, en effet, que $(x_n \cdots x_0)_{\text{avi}} = 0$ avec $x_n \neq 0$. Alors $-x_n b^n = (x_{n-1} \cdots x_0)_{\text{avi}}$. Mais selon 1b, $\left|(x_{n-1} \cdots x_0)_{\text{avi}}\right| \leq (b^n - 1)\frac{a}{b-1} < b^n$ et $|-x_n b^n| \geq b^n$. C'est donc absurde.

(b) Le signe d'un nombre représenté dans le système d'Avizienis est égal au signe de son chiffre non nul de poids le plus fort. En effet, soit $(x_n \cdots x_0)_{\text{avi}}$ un nombre représenté dans ce système avec x_n non nul. Alors, en utilisant le même raisonnement qu'à la question précédente, on a :

$$\left|(x_{n-1} \cdots x_0)_{\text{avi}}\right| < |x_n b^n|$$

Et donc le signe de $(x_n \cdots x_0)_{\text{avi}} = x_n b^n + (x_{n-1} \cdots x_0)_{\text{avi}}$ est bien le signe de x_n.

Cet algorithme est particulièrement efficace (il s'exécute en temps constant) lorsque la représentation d'Avizienis est *normalisée*, c'est-à-dire lorsque le chiffre de poids le plus fort est non nul.

Question 3.

(a) Si x_1, x_2, x_3 sont les trois booléens, alors leur somme est $(y_1 y_0)_2$, où $y_0 = x_1 \oplus x_2 \oplus x_3$ et $y_1 = (x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_1)$. Le circuit logique en découle immédiatement.

(b) On procède par récurrence sur n. L'initialisation $n = 0$ est triviale (vraiment !). Si on a un circuit pour la somme $(c_{n+1} \cdots c_0)_2$ de deux entiers $(a_n \cdots a_0)_2$ et $(b_n \cdots b_0)_2$, alors on calcule la somme de $(a_{n+1} \cdots a_0)_2$ et de $(b_{n+1} \cdots b_0)_2$ en additionnant a_{n+1}, b_{n+1} et c_{n+1} avec le circuit de la question précédente. Les booléens c'_{n+2} et c'_{n+1} obtenus sont alors tels que $(a_{n+1} \cdots a_0)_2 + (b_{n+1} \cdots b_0)_2 = (c'_{n+2}c'_{n+1}c_n \cdots c_0)_2$.
Ce circuit est obtenu en « chaînant » n additionneurs tels que décrits à la question précédente. Il a donc une profondeur $O(n)$. On peut obtenir une profondeur $O(\log n)$ en utilisant par exemple un Carry-lookahead adder [Wik18a], mais ce n'est pas demandé ici.

(c) De la même façon, on procède par récurrence sur n. Le cas $n = 0$ est trivial : les entiers sont toujours égaux. Sinon, on fait un circuit qui détermine si les bits de poids forts sont égaux : s'ils ne le sont pas, alors on peut conclure, et s'ils le sont on se réduit à la comparaison des $n - 1$ bits de poids plus faibles. La profondeur est de $O(n)$.

Question 4. On représente chaque chiffre de la représentation d'Avizienis avec sa représentation en base 2, avec un bit supplémentaire pour le signe, en utilisant soit la représentation en complément à 1 ou la représentation en complément à 2. La question 3 montre qu'on peut facilement avoir des circuits pour additionner et comparer des entiers. Les soustraire ne pose pas plus de difficulté.

Armé de ces circuits pour les opérations élémentaires sur les chiffres de la représentation d'Avizienis, on va construire un circuit de profondeur bornée pour additionner deux nombres dans la représentation d'Avizienis. L'astuce est que la redondance offerte par la représentation d'Avizienis permet de calculer une retenue pour chaque chiffre sans attendre la propagation de la retenue des chiffres de poids plus faibles.

Plus précisément, pour additionner deux nombres $(x_{n-1} \cdots x_0)_{\text{avi}}$ et $(y_{n-1} \cdots y_0)_{\text{avi}}$, on considère que l'addition du chiffre d'indice i fait une retenue dès que le résultat pourrait sortir de $[\![-a, a]\!]$ dans le cas le plus défavorable pour la retenue entrante. Notons que puisque les chiffres sont potentiellement négatifs, les retenues sont potentiellement négatifs elles aussi. On utilise donc la définition suivante pour la retenue sortante à l'indice i :

$$t_{i+1} = \begin{cases} -1 & \text{si } x_i + y_i \leq -a \\ 0 & \text{si } -a < x_i + y_i < a \\ 1 & \text{si } x_i + y_i \geq a \end{cases} \qquad 0 \leq i < n$$
$$t_0 = 0$$

En la représentant par exemple avec deux bits, toutes les t_i peuvent être calculées en fonction des entrées avec un circuit de profondeur indépendante de n.

Maintenant, on peut calculer le résultat de l'addition $(z_n \cdots z_0)_{\text{avi}}$ avec :

$$z_i = x_i + y_i + t_i - bt_{i+1} \qquad 0 \leq i < n$$
$$z_n = t_n$$

Il est alors facile de démontrer que les z_i sont alors bien des chiffres d'Avizienis, c'est-à-dire qu'ils sont dans $[\![-a, a]\!]$. Cela découle de la propriété $\frac{b}{2} < a < b$.

Question 5. Pour multiplier $(x_{n-1} \cdots x_0)_2$ et $(y_{n-1} \cdots y_0)_2$, on écrit :

$$\left(\sum_{i=0}^{n-1} x_i 2^i\right)\left(\sum_{j=0}^{n-1} y_j 2^j\right) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(x_i \wedge y_j)2^{i+j}$$

Multiplier ces deux nombres, c'est donc additionner les nombres $\sum_{j=0}^{n-1}(x_i \wedge y_j)2^{i+j}$ (pour $i \in [\![0, n-1]\!]$), dont la représentation en base deux se calcule facilement avec un circuit de profondeur bornée et un nombre quadratique de portes.

Pour calculer cette addition, on pourrait utiliser $n-1$ fois successives l'algorithme de la question 3b. Cependant, on obtiendrait une profondeur totale $\Theta(n^2)$, ce qui n'est pas satisfaisant. On va améliorer cet algorithme de deux manières : premièrement, au lieu d'utiliser l'addition en base 2 décrite à la question 3b, on utilise l'addition dans la représentation d'Avizienis. Mais comme $b = 2$ n'est pas une base possible dans la représentation d'Avizienis, on regroupe les bits deux par deux et on effectue des additions d'Avizienis avec $b = 4$ et $a = 3$. En utilisant cette idée, on peut effectuer chaque addition en profondeur $O(1)$.

La deuxième amélioration consiste à changer sur l'ordre dans lequel les additions sont effectuées : en utilisant un algorithme diviser pour régner, on peut effectuer toutes ces additions en profondeur $O(\log n)$. Plus précisément, on sépare les n nombres en deux groupes de cardinaux égaux (à 1 près), on effectue récursivement les sommes des deux sous-groupes, puis on calcule la sommes des deux sous-sommes avec l'algorithme d'addition d'Avizienis. Ainsi, on peut effectuer l'addition de n nombres dans la représentation d'Avizienis en une profondeur $P(n)$ vérifiant la récurrence :

$$P(n) = P(\lceil \tfrac{n}{2} \rceil) + O(1)$$

D'où $P(n) = O(\log n)$.

Le nombre de portes nécessaires pour calculer cette somme reste linéaire en fonction de la taille des nombres à additionner, c'est-à-dire quadratique par rapport à n.

Le seul problème restant est que la somme finale est représentée dans la représentation d'Avizienis, alors qu'on la veut en binaire. Il reste donc à prouver qu'on peut effectuer la conversion vers une représentation binaire en profondeur $O(\log n)$ et avec $O(n)$ portes. Pour cela, on utilise (encore !) un algorithme diviser pour régner, qui prend en entrée un nombre $a = (a_{n-1} \cdots a_0)_{\text{avi}}$ dans la représentation d'Avizienis (avec toujours $b = 4$ et $a = 3$), et qui calcule à la fois son écriture en base 2 et l'écriture en base 2 de $a - 1$ sur $2n + 1$ bits (en utilisant la convention du complément à 2 pour les nombres négatifs). Le cas $n = 1$ est facile. Si $n > 1$, alors on effectue indépendamment la conversion des $\lfloor \frac{n}{2} \rfloor$ bits de poids faibles et des $\lceil \frac{n}{2} \rceil$ bits de poids forts. On sélectionne parmi le bon résultat pour le poids fort en fonction de la valeur du bit de poids le plus fort de la conversion des bits de poids faibles.

Pour plus d'informations sur la numération d'Avizienis, on peut par exemple aller sur la page Wikipedia associée [Wik18b]

Références

[Wik18a] Wikipédia. Carry-lookahead adder, 2018. `https://en.wikipedia.org/wiki/Carry-lookahead_adder`.

[Wik18b] Wikipédia. Système de numération d'Avizienis, 2018. `https://fr.wikipedia.org/wiki/Syst%C3%A8me_de_num%C3%A9ration_d%27Avizienis`.

J11 – Constitution d'échantillons aléatoires

On suppose que notre langage de programmation fournit une primitive `pièce`, qui renvoie un booléen tiré selon la distribution de Bernoulli de paramètre $\frac{1}{2}$.

On définit la *complexité moyenne en temps* d'un programme utilisant `pièce` comme étant l'espérance, sur tous les résultats des tirages de `pièce`, de la complexité en temps de l'exécution du programme au sens habituel.

Par ailleurs, on suppose que les opérations arithmétiques usuelles sur les réels peuvent être calculées de manière exacte en temps constant. (Ces hypothèses sont fausses en pratique, mais de bonnes approximations sont possibles.)

Question 1 Proposer un algorithme qui prend en paramètre un réel $p \in [0, 1]$, et qui renvoie un booléen tiré selon la distribution de Bernoulli de paramètre p. Quelle est sa complexité moyenne en temps ? Quelle est sa complexité en espace ?

Question 2 Soit L un langage régulier sur l'alphabet $\{a, b\}$ décrit par un automate déterministe $\mathcal{A}$ et soit $n \in \mathbb{N}$. On note L_n l'ensemble des mots de L de longueur n, et on pose $\gamma_n = \frac{|L_n|}{2^n}$. On suppose $\frac{1}{\gamma_n}$ borné par une constante. Donner un algorithme qui tire aléatoirement uniformément un mot dans L_n, avec une complexité moyenne en temps $O(n)$. Quelle est sa complexité en espace ?

Question 3

(a) Déduire de la question 2 un algorithme qui prend en paramètre un entier $x \in \mathbb{N}^*$ et qui renvoie un entier naturel tiré selon la distribution uniforme sur $\{0, \dots, x-1\}$, avec une complexité moyenne en temps $O(\log x)$.

(b) En espérance et à une constante *additive* près, combien d'appels à `pièce` sont effectués par cet algorithme ?

(c) Proposer un algorithme qui effectue moins d'appels à `pièce` en moyenne.

Corrigé

Question 1 *Pseudo-code demandé.*

Considérons l'algorithme suivant :

```
Fonction bern(p)
  Si p > 1/2
    Renvoyer Non(bern(1-p))
  Sinon Si pièce()
    Renvoyer bern(2*p)
  Sinon
    Renvoyer Faux
```

Tout d'abord, la fonction termine presque sûrement : En effet, la probabilité qu'elle ne termine pas correspond à la probabilité que `pièce` ne renvoie jamais `Faux`, et cette probabilité est nulle.

Si $p > \frac{1}{2}$, alors on se réduit à tirer une variable de Bernoulli de paramètre $1-p$ en prenant la négation du résultat. Sinon, la probabilité de renvoyer vrai est $\frac{1}{2} \cdot 2p = p$

Par ailleurs, le nombre d'appels à `pièce` suit une loi géométrique de paramètre $\frac{1}{2}$. Son espérance est donc 2. La fonction s'exécute donc en temps $O(1)$ en moyenne, et la complexité en espace est constante (on peut faire en sorte que la récursivité soit terminale).

Question 2 *Pseudo-code demandé.*

On tire aléatoirement un mot w de longueur n, et on teste son appartenance à L. Si w est effectivement dans L, alors il est tiré dans la distribution uniforme sur Σ^n, conditionnellement à son appartenance à L : c'est exactement la distribution uniforme sur L. S'il n'est pas dans L, on oublie tout et on recommence.

Le nombre total d'exécutions de la boucle principale suivra alors une distribution géométrique de paramètre γ_n. Son espérance $\frac{1}{\gamma_n}$ est donc bornée.

Par conséquent, la complexité en temps de l'algorithme est $O(\frac{n}{\gamma_n}) = O(n)$, et la complexité en espace est clairement $O(n)$ également.

Question 3

(a) Soit $n = \lceil \log_2 x \rceil$. Démontrons que le langage L_x des représentations en base 2 sur n bits des entiers naturels strictement inférieurs à x est régulier, et qu'on peut construire pour ce langage un automate déterministe complet à $2n+1$ états.

Si $x = 2^n$, on a $L_x = \Sigma^n$, et le résultat est trivial.

Sinon, soit $x = (x_{n-1} \cdots x_0)_2$ la représentation en base 2 de x. Un mot $y_{n-1} \cdots y_0$ est la représentation en base 2 d'un nombre $y < x$ si et seulement s il existe un indice i tel que $x_{n-1} = y_{n-1}, \ldots, x_{i+1} = y_{i+1}$, $y_i = 0$ et $x_i = 1$. On construit donc l'automate avec $2n+1$ états $s_n, s_{n-1}, s'_{n-1}, \ldots, s_0, s'_0$ comme suit :

- s_n est l'état initial ;
- s_0 est un état puits : on a deux transitions $s_0 \xrightarrow{0,1} s_0$;
- s'_0 est le seul état final, et on a deux transitions $s'_0 \xrightarrow{0,1} s_0$;
- pour tout $0 < i < n$, on a deux transitions $s'_i \xrightarrow{0,1} s'_{i-1}$;
- pour tout $0 \leq i < n$ avec $x_i = 1$, on a une transition $s_{i+1} \xrightarrow{0} s'_i$, et une transition $s_{i+1} \xrightarrow{1} s_i$;
- pour tout $0 \leq i < n$ avec $x_i = 0$, on a une transition $s_{i+1} \xrightarrow{0} s_i$, et une transition $s_{i+1} \xrightarrow{1} s_0$.

Il est facile de démontrer que cet automate reconnaît bien le langage voulu.

Du coup, tirer un entier $u < x$ revient à tirer un mot binaire dans L_x, ce que l'on peut faire en $O(n) = O(\log_2 x)$ selon la question précédente, puisqu'on a alors $\gamma_n > \frac{1}{2}$.

(b) On fait $\frac{1}{\gamma_n}\lceil \log_2 x \rceil$ appels à `pièce` en moyenne. Si x est le successeur d'une puissance de 2, on fait $2\lceil \log_2 x \rceil + O(1)$ appels à `pièce`.

(c) L'idée générale de l'algorithme est qu'à chaque appel de `pièce`, on exclut la moitié des résultats possibles. Cependant, il est possible que l'intervalle des résultats possibles ne puisse pas être divisé en deux parties de même longueur. Dans ce cas, certaines valeurs possibles du résultat doivent être partagées entre les deux moitiés : il est alors nécessaire de changer leur poids pour conserver l'uniformité de la distribution. Ainsi, nous allons écrire une fonction récursive prenant en paramètre un entier n et deux réels $x_{-1}, x_n \in [0, 1[$. Cette fonction tirera aléatoirement un entier dans $[\![-1, n]\!]$, avec des poids $x_{-1}, 1, \ldots, 1, x_n$. Si $n = 0$, il suffit de tirer une variable de Bernoulli en utilisant la méthode de la question 1. Sinon, on appelle `pièce`, on divise n en deux, et on peut recommencer. On obtient le pseudo-code suivant :

```
Fonction unif_aux(n, xm1, xn)
  Si n = 0
    Si bern(xm1/(xm1+xn))
      Renvoyer -1
    Sinon
      Renvoyer 0
  # n-xm1+xn est le poids total à considérer pour cet appel
  mid <- (n-xm1+xn)/2
  Si pièce()
    Renvoyer unif_aux(floor(mid), xm1, mid-floor(mid))
  Sinon
    Renvoyer unif_aux(n-ceil(mid), 1-mid+floor(mid), xn) + ceil(mid)

Fonction unif(n)
  Renvoyer unif_aux(n, 0, 0)
```

Cet algorithme effectue $\log_2 n + O(1)$ appels à `pièce` en moyenne. Cette complexité est optimale à une constante additive près : en effet, il faut au moins $\lceil \log_2 n \rceil$ appels à `pièce` pour espérer avoir n valeur possibles différentes en sortie de l'algorithme.

J12 – Énumérations

On fixe $\mathcal{A}$ un automate déterministe sur un alphabet Σ, et un entier $m \in \mathbb{N}^*$.

Question 1

(a) Proposer un algorithme qui calcule la longueur minimale des mots acceptés par $\mathcal{A}$, si cette quantité est bien définie, et qui détecte si elle ne l'est pas. Quelle est sa complexité ?

(b) On suppose dans cette question que $\mathcal{A}$ n'a pas de cycle. Proposer un algorithme qui calcule la longueur maximale des mots acceptés par $\mathcal{A}$, si cette quantité est bien définie, et qui détecte si elle ne l'est pas. Quelle est sa complexité ?

(c) Que se passe-t-il lorsque $\mathcal{A}$ a un cycle ?

Question 2

(a) Proposer un algorithme permettant de calculer le nombre modulo m de mots de longueur n dans $L(\mathcal{A})$. Quelle est sa complexité en temps et en espace ?

(b) Le problème devient-il plus difficile si on demande de calculer le résultat exact (sans modulo) ?

(c) L'hypothèse que $\mathcal{A}$ est déterministe est-elle importante ?

Corrigé

Question 1 *Pseudo-code demandé pour au moins l'un des algorithmes.*

(a) La longueur du mot le plus court peut s'obtenir en faisant un parcours en largeur en partant de l'état initial de l'automate, et en calculant la distance minimale d'un état final. Si aucun état final n'est atteint, alors le langage est vide et la longueur minimale n'est pas définie. Sa complexité est $O(|\delta|)$

(b) On peut calculer la longueur du plus long mot à l'aide d'une fonction récursive :

```
# Renvoie -oo si aucun mot n'existe
Fonction plusLongMot(s)
  Si s est final
    r <- 0
  Sinon
    r <- -oo
  Pour chaque s' successeur de s
    r <- max(r, plusLongMot(s') + 1)
  Renvoyer r

Réponse : plusLongMot(init)
```

Il faut mémoïser cette fonction pour avoir la bonne complexité.

Pour représenter `-oo`, on peut utiliser n'importe quel entier plus petit que $-|Q|$, et le résultat est négatif si et seulement si le langage est vide. Attention, même si `s` est final, il faut tout de même essayer des mots plus longs.

La complexité est la même : $O(|\delta|)$.

(c) Si l'automate a un cycle, alors la terminaison de l'algorithme de la question b) n'est plus garantie. Il faut alors commencer par vérifier si l'un de ces cycles est accessible et co-accessible à l'aide de deux parcours en profondeur. Si oui, alors il existe des mot acceptés par $\mathcal{A}$ de longueur arbitrairement longue, et le maximum n'est pas défini. Si non, alors on peut supprimer ces cycles de l'automate sans modifier le langage qu'il reconnaît, puis réutiliser l'algorithme de b).

Question 2

(a) On suppose ici que l'automate $\mathcal{A}$ n'a pas de cycle. On utilise un algorithme de programmation dynamique :

```
Fonction compteMots(s, i)
  Si le calcul de compteMots(s, i) a déjà été fait
    Renvoyer le résultat déjà calculé
  Si i = 0
    Renvoyer 1 si s est final, 0 sinon
  r = 0
  Pour chaque successeur s' de s
    r = (r + compteMots(s', i-1)) mod m
  Renvoyer r

Réponse : compteMots(init, n)
```

La complexité en temps est de $O(n|\delta|)$, et de $O(n|Q|)$ en espace. On peut réduire la complexité en espace à $O(|Q|)$ en ne stockant simultanément les résultats de `compteMots` que pour deux valeurs simultanées de `n`.

(b) Si on demande de calculer le résultat sans modulo, alors celui-ci peut croître de manière exponentielle par rapport à n. Par conséquent, le résultat dépassera rapidement la capacité du "mot

machine" lorsque n augmente, et la complexité des opérations arithmétiques pour calculer ces nombres ne peut plus être considérée comme constante.

(c) Si $\mathcal{A}$ n'est pas déterministe, alors plusieurs chemins peuvent reconnaître le même mot. Puisque notre algorithme ne compte en fait que les chemins de longueur donnée depuis l'état initial vers les états finaux, il n'est plus garanti que celui-ci renvoie le résultat attendu.

J13 – Réseaux de tri

Dans ce sujet, on se propose d'étudier les *réseaux de tri*. Il s'agit d'une certaine catégorie d'algorithmes de tri, pour lesquels on fixe à l'avance une série de comparaisons et d'échanges constituant ledit réseau.

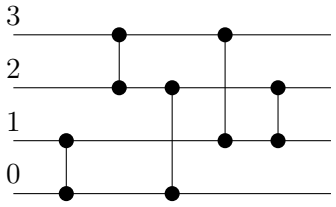
Plus précisément, dans ce sujet, un réseau de tri agit sur un tableau T de n entiers, et est décrit par une séquence de *modules de comparaison*. Un module de comparaison est décrit par la donnée d'une paire (i, j) de deux indices tels que $0 \leq i < j < n$. Un tel module de comparaison a pour effet d'échanger les contenus des cellules $T[i]$ et $T[j]$ si et seulement si $T[j] < T[i]$.

On représentera un réseau de tri de la façon suivante :

— Chaque cellule du tableau est représenté par une ligne horizontale allant de l'extrême gauche à l'extrême droite du schéma.

— Les modules de comparaison sont alors représentés de gauche à droite. Chacun d'entre eux est représenté par une ligne verticale reliant les lignes correspondant aux deux cellules concernées.

Par exemple, le réseau de tri $(0,1)(2,3)(0,2)(1,3)(1,2)$ est représenté par le schéma suivant :



Question 1 Exécuter le réseau de tri donné en exemple sur le tableau $[2, 4, 5, 1]$. Ce réseau permet-il de trier n'importe quel tableau de taille 4 ?

Question 2 Donner un réseau de tri qui trie tout tableau d'entiers de taille n. On ne demande pas que ce réseau de tri soit optimal selon quelque critère que ce soit. Combien de modules de comparaison contient-il, asymptotiquement ?

Question 3 Montrer qu'un réseau de tri permet de trier tout tableau si et seulement si il permet de trier tout tableau de booléens.

Question 4 Dans cette question, on suppose que $n = 2^k$ est une puissance de 2 avec $k > 1$. Par ailleurs, on ne considère que les tableaux T tels que les sous-tableaux $T[0] \dots T[\frac{n}{2} - 1]$ et $T[\frac{n}{2}] \dots T[n-1]$ sont déjà triés. Construire un réseau de tri qui trie ces tableaux, en utilisant $O(n \log n)$ modules de comparaison.

Indication : on pourra utiliser une approche diviser-pour-régner, en commençant par trier les positions paires et les positions impaires indépendamment.

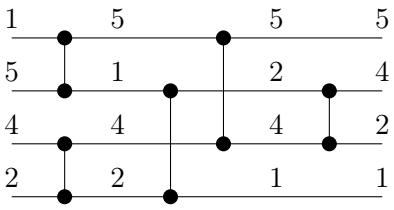
Question 5 En déduire un réseau de tri efficace pour trier tout tableau de taille quelconque. Asymptotiquement, combien de modules de comparaison contient-il ?

On définit la profondeur d'un réseau de tri par le nombre maximal de modules de comparaison qu'une valeur doit traverser avant d'arriver à la fin du réseau.

Question 6 Quelle est, asymptotiquement, la profondeur du réseau de tri de la question 5 ?

Corrigé

Question 1



Ce réseau de tri trie tous les tableaux de taille 4 : en effet, en notant T' le tableau à la fin de l'exécution, on a $T'[0] = \min(\min(T[0], T[1]), \min(T[2], T[3]))$ et $T'[3] = \max(\max(T[0], T[1]), \max(T[2], T[3]))$. Ainsi, $T'[0]$ et $T'[3]$ sont bien respectivement le minimum et le maximum du tableau. Le dernier module de comparaison permet bien de s'assurer que $T'[1] \leq T'[2]$.

Question 2 On peut effectuer un tri par sélection en utilisant un réseau de tri. Si $n \leq 1$, alors il n'y a rien à faire. Sinon avec $n-1$ modules de comparaison, on place le maximum du tableau à la position $n-1$. Ensuite, il reste à trier les $n-1$ éléments restants, ce que l'on peut faire par hypothèse de récurrence.

On peut prouver facilement que le réseau de tri ainsi construit contient $\frac{n(n-1)}{2} = O(n^2)$ modules de comparaison.

Question 3 On va commencer par démontrer que si f est une fonction croissante, alors f commute avec le réseau de tri. C'est-à-dire que si, à travers un réseau de tri, T est transformé en T', alors le tableau contenant $f(T[0]), \ldots, f(T[n-1])$ est transformé en $f(T'[0]), \ldots, f(T'[n-1])$. En effet, il est facile de démontrer que les modules de comparaison commutent avec f, et ensuite, par un récurrence évidente, on peut étendre le résultat aux réseaux de tri.

On suppose maintenant qu'on dispose d'un réseau de tri qui trie tous les tableaux de booléens. Soient T un tableau, et i et j deux indices tels que $T[i] < T[j]$. On va démontrer que les positions correspondantes i' et j' dans le tableau final T' sont telles que $i' < j'$. Posons $f(x) = \mathbb{1}_{x>T[i]}$. On a f croissante, $f(T[i]) = 0$ et $f(T[j]) = 1$. Puisque f est à valeur dans $\{0, 1\}$, le réseau de tri trie correctement le tableau $f(T[0]), \ldots, f(T[n-1])$, et donc, en associant le lemme précédent, on obtient donc $i' < j'$. Par conséquent, toutes paires de cases contenant des valeurs différentes dans le tableau sont bien ordonnées par le réseau : cela implique directement que le réseau ordonne correctement le tableau.

Question 4 On commence par remarquer que la démonstration de la question 3 s'applique aussi au cas où le tableau d'entrée vérifie la nouvelle hypothèse de cette question. Il suffira donc de prouver que le réseau de tri que nous allons construire fonctionne pour des tableaux de booléens.

Posons, pour simplifier les notations $\tilde{T}[0] = T[2^{k-1}], \ldots, \tilde{T}[2^{k-1}-1] = T[2^k - 1]$. Il s'agit donc, avec ces nouvelles notations, de fusionner les tableaux triés $T[0..2^{k-1}]$ et $\tilde{T}[0..2^{k-1}]$.

Si $k = 1$, c'est trivial.

Sinon, on commence par faire la fusion des indices pairs et des indices impairs, en appelant l'algorithme récursivement.

On va fusionner :

— le tableau formé par les valeurs $T[0], T[2], \ldots, T[2^{k-1} - 2]$ avec le tableau formé par les valeurs $\tilde{T}[0], \tilde{T}[2], \ldots, \tilde{T}[2^{k-1} - 2]$;

— le tableau formé par les valeurs $T[1], T[3], \ldots, T[2^{k-1} - 1]$ avec le tableau formé par les valeurs $\tilde{T}[1], \tilde{T}[3], \ldots, \tilde{T}[2^{k-1} - 1]$.

On obtient alors les sous-tableaux S_0 et S_1, chacun de taille 2^{k-1}. Ces sous-tableaux sont exactement entrelacés : S_0 occupe les position paires, et S_1 occupe les positions impaires. La sortie de ce processus est donc le tableau S tel que $S[2i] = S_0[i]$ et $S[2i+1] = S_1[i]$ pour tout $0 \leq i < 2^{k-1}$.

Si les tableaux $T[0..2^{k-1}]$ et $\tilde{T}[0..2^{k-1}]$ ne contenaient que des 1 et des 0, posons l et $\tilde{l}$ le nombre de 0 dans T et dans $\tilde{T}$, respectivement. Alors, S_0 est constitué de exactement $\lceil \frac{l}{2} \rceil + \left\lceil \frac{\tilde{l}}{2} \right\rceil$ cases contenant 0, et S_1 de exactement $\lfloor \frac{l}{2} \rfloor + \left\lfloor \frac{\tilde{l}}{2} \right\rfloor$ cases contenant 0. Ainsi, sachant que les tableaux S_0 et S_1 sont exactement entrelacés, on a trois possibilités :

— Soit l et $\tilde{l}$ sont tous les deux pairs. Alors $\lceil \frac{l}{2} \rceil + \left\lceil \frac{\tilde{l}}{2} \right\rceil = \lfloor \frac{l}{2} \rfloor + \left\lfloor \frac{\tilde{l}}{2} \right\rfloor$, et S est déjà trié : il commence par exactement $l + \tilde{l}$ fois 0, le reste contient 1.

— Soit l et $\tilde{l}$ sont de parités différentes. Alors $\lceil \frac{l}{2} \rceil + \left\lceil \frac{\tilde{l}}{2} \right\rceil = 1 + \lfloor \frac{l}{2} \rfloor + \left\lfloor \frac{\tilde{l}}{2} \right\rfloor$ et S est aussi déjà trié : il commence par $2\left(\lfloor \frac{l}{2} \rfloor + \left\lfloor \frac{\tilde{l}}{2} \right\rfloor\right)$ fois 0, puis :

$$S\left[2\left(\left\lfloor \frac{l}{2} \right\rfloor + \left\lfloor \frac{\tilde{l}}{2} \right\rfloor\right)\right] = S_0\left[\left\lfloor \frac{l}{2} \right\rfloor + \left\lfloor \frac{\tilde{l}}{2} \right\rfloor\right] = S_0\left[\left\lceil \frac{l}{2} \right\rceil + \left\lceil \frac{\tilde{l}}{2} \right\rceil - 1\right] = 0$$

— Soit l et $\tilde{l}$ sont tous deux impairs. Alors $\lceil \frac{l}{2} \rceil + \left\lceil \frac{\tilde{l}}{2} \right\rceil = 2 + \lfloor \frac{l}{2} \rfloor + \left\lfloor \frac{\tilde{l}}{2} \right\rfloor$. Le seul défaut de tri se trouve alors aux positions :

$$S\left[2\left(\left\lfloor \frac{l}{2} \right\rfloor + \left\lfloor \frac{\tilde{l}}{2} \right\rfloor\right) + 1\right] = S_1\left[\left\lfloor \frac{l}{2} \right\rfloor + \left\lfloor \frac{\tilde{l}}{2} \right\rfloor\right] = 1$$
$$S\left[2\left(\left\lfloor \frac{l}{2} \right\rfloor + \left\lfloor \frac{\tilde{l}}{2} \right\rfloor\right) + 2\right] = S_0\left[\left\lfloor \frac{l}{2} \right\rfloor + \left\lfloor \frac{\tilde{l}}{2} \right\rfloor + 1\right] = S_0\left[\left\lceil \frac{l}{2} \right\rceil + \left\lceil \frac{\tilde{l}}{2} \right\rceil - 1\right] = 0$$

Dans ce cas, il faut s'assurer qu'un module de comparaison soit présent entre les positions en question pour que le tableau soit trié.

Finalement, on voit qu'en ajoutant un module de comparaison entre les positions $2i+1$ et $2i+2$ pour tout $i \in [\![0, 2^{k-1} - 2]\!]$, on trie tous les tableaux constitués de 0 et de 1 vérifiant la propriété de la question 4. Selon la généralisation de la propriété de la question 3, on obtient un réseau de tri pour tout tableau vérifiant la propriété de la question 4.

Le nombre de modules de comparaisons utilisés par ce réseau de tri vérifie la récurrence $M(k) = 2M(k-1) + O(2^k)$. Il utilise donc $O(n \log n)$ modules de comparaison. Par ailleurs, la profondeur du réseau de tri utilisé vérifie la récurrence $P(k) = 1 + P(k-1)$. La profondeur est donc en $O(\log n)$.

Questions 5 et 6 On commence par traiter le cas où la taille est une puissance de deux. Pour cela, on fait un tri fusion en utilisant le réseau de la question 4 pour effectuer chaque fusion. On obtient alors un réseau avec $O(\log n)$ étages, chacun contenant $O(n \log n)$ modules de comparaison et de profondeur $O(\log n)$. Le réseau contient donc $O(n \log^2 n)$ modules de comparaison et a une profondeur $O(\log^2 n)$.

Ensuite, pour un tableau de taille n quelconque, on considère le réseau de tri fusion correspondant à la puissance de 2 immédiatement supérieure $2^{\lceil \log_2 n \rceil}$. On nettoie alors les $2^{\lceil \log_2 n \rceil} - n$ dernières lignes de ce réseau de tri en les interprétant comme si elles contenaient une valeur "infinie", plus grande que toutes celles dans le tableau. Ainsi, aucun module de comparaison connecté à l'une de ces lignes ne fera un échange, et on peut les supprimer sans risquer de casser la correction de l'algorithme de tri.

Pour conclure, on vient de construire un réseau de tri pour trier un tableau de taille n contenant $O(n \log^2 n)$ modules de comparaison et avec une profondeur $O(\log^2 n)$, quel que soit n.