Sous-ensembles connexes minimaux

On considère un graphe non-orienté pondéré $G = (V, E, w)$ où V est l'ensemble des sommets, E l'ensemble des arêtes, et $w \colon E \to \mathbb{N}_{>0}$ une fonction qui donne un poids strictement positif à chaque arête. On se fixe un ensemble $T \subseteq V$ de *terminaux*. Un *sous-ensemble T-connexe* est un sous-ensemble d'arêtes $E' \subseteq E$ tel que, pour n'importe quelle paire $s \neq t$ de terminaux de T distincts, il y a un chemin qui relie s et t dans le graphe (V, E'). Un sous-ensemble T-connexe E' est *minimal* si la somme du poids des arêtes de E' est minimal parmi les sous-ensembles T-connexes.

Question 1. Est-ce qu'un sous-ensemble T-connexe existe toujours ? Proposer un algorithme permettant de vérifier si un tel sous-ensemble existe, et préciser sa complexité en temps.

Question 2. Proposer un algorithme naïf pour calculer un sous-ensemble T-connexe minimal, et préciser sa complexité en temps.

Question 3. On suppose d'abord que T contient au plus deux sommets. Proposer un algorithme efficace permettant de calculer un sous-ensemble T-connexe minimal. Préciser sa complexité en temps.

Question 4. On suppose à présent que tous les sommets sont des terminaux : $T = V$. Proposer un algorithme efficace permettant de calculer un sous-ensemble T-connexe minimal.

Question 5. On recherche à présent un algorithme efficace pour calculer un ensemble T-connexe minimal dans le cas général. Quelle condition structurelle simple est toujours respectée par un sous-ensemble T-connexe minimal ? Démontrer ce fait rigoureusement.

Question 6. Une idée pour calculer un ensemble T-connexe minimal est de procéder en choisissant, pour chaque paire de terminaux $u \neq v$ de T, un ensemble $\{u, v\}$-connexe $X_{u,v}$ minimal, et de prendre X comme l'union des $X_{u,v}$. Cette approche vous paraît-elle réaliste ? Pourquoi ?

Question 7. On suppose à présent qu'il y a 3 terminaux : $|T| = 3$. Concevoir un algorithme efficace pour calculer un sous-ensemble T-connexe minimal, et préciser sa complexité.

Question 8. On fixe un entier $k \in \mathbb{N}$ et on suppose qu'il y a au plus k terminaux. Généraliser l'algorithme précédent, pour garantir une complexité polynomiale à k fixé.

On s'intéresse maintenant au cas des graphes orientés.

Question 9. Étendre la définition d'un sous-ensemble T-connexe au cas des graphes orientés. Comment caractériser l'existence d'un tel ensemble ?

Question 10. On suppose toujours le graphe orienté, et on considère le cas $|T| = 2$. Caractériser la forme d'un sous-ensemble T-connexe minimal dans ce cas.

Question 11. À l'aide de cette caractérisation, proposer un algorithme efficace pour calculer un sous-ensemble T-connexe minimal dans le cas d'un graphe orienté pour $|T| = 2$.

Sous-ensembles connexes minimaux

On considère un graphe non-orienté pondéré $G = (V, E, w)$ où V est l'ensemble des sommets, E l'ensemble des arêtes, et $w \colon E \to \mathbb{N}_{>0}$ une fonction qui donne un poids strictement positif à chaque arête. On se fixe un ensemble $T \subseteq V$ de *terminaux*. Un *sous-ensemble T-connexe* est un sous-ensemble d'arêtes $E' \subseteq E$ tel que, pour n'importe quelle paire $s \neq t$ de terminaux de T distincts, il y a un chemin qui relie s et t dans le graphe (V, E'). Un sous-ensemble T-connexe E' est *minimal* si la somme du poids des arêtes de E' est minimal parmi les sous-ensembles T-connexes.

Question 1. Est-ce qu'un sous-ensemble T-connexe existe toujours ? Proposer un algorithme permettant de vérifier si un tel sous-ensemble existe, et préciser sa complexité en temps.

Indication : Quel sous-ensemble T-connexe suffit-il de tester ?

Solution : Il est évident qu'il existe un ensemble T-connexe si et seulement si le graphe tout entier peut jouer ce rôle, car la propriété est monotone par ajout d'arêtes. Ainsi, sans perte de généralité, il suffit de vérifier si, dans le graphe fourni, il existe un chemin entre toute paire de sommets de T.

C'est le cas si et seulement si tous les sommets de T sont dans la même composante connexe du graphe d'entrée. Ainsi, on peut calculer les composantes connexes du graphe d'entrée (cette notion est explicitement au programme). Ce calcul est en temps linéaire en le graphe d'entrée. (Cette complexité linéaire n'est pas explicitement au programme, mais comme l'algorithme de Kosaraju est au explicitement au programme, le candidat devrait toujours pouvoir donner une complexité linéaire, quitte à se ramener aux composantes fortement connexes d'un graphe orienté s'il le faut...)

Note : les candidats peuvent aussi vouloir tester l'accessibilité de chaque paire de sommets. Cela fonctionne, mais c'est souvent moins efficace : complexité cubique si on teste naïvement chaque paire, quadratique si on teste chaque point de départ. Les orienter le cas échéant vers la solution linéaire. En revanche l'algo qui fixe un point de départ u et teste que tous les autres points sont accessibles depuis u par un parcours depuis u est correct et sa complexité est également linéaire.

Question 2. Proposer un algorithme naïf pour calculer un sous-ensemble T-connexe minimal, et préciser sa complexité en temps.

Indication : Essayer tous les sous-ensembles d'arêtes.

Solution : On considère tous les sous-ensembles d'arêtes. Pour chaque sous-ensemble, on calcule son poids, et on vérifie que tous les terminaux sont dans la même composante connexe. La complexité est $O(2^M \times (N + M))$, où N est le nombre de sommets et M le nombre d'arêtes : pour chaque sous-ensemble, la somme est en $O(M)$ et le parcours en $O(N + M)$.

Question 3. On suppose d'abord que T contient au plus deux sommets. Proposer un algorithme efficace permettant de calculer un sous-ensemble T-connexe minimal. Préciser sa complexité en temps.

Indication : Quelle est la structure d'une solution minimale pour $|T| = 2$?

Indication : Utiliser un algorithme du cours.

Solution : Si $|T| < 2$, manifestement le sous-graphe vide convient et est optimal. Sinon, on cherche un sous-ensemble qui soit un chemin reliant s et t et de poids total minimal. En effet, étant donné un sous-ensemble T-connexe X, on peut manifestement se ramener à un chemin reliant s et t dans X et son poids est inférieur : donc il suffit bien de rechercher des chemins connectant s et t, et on veut que le poids d'un tel chemin soit minimal.

Pour trouver un chemin de poids minimal reliant s et t, comme les poids sont positifs, on utilise l'algorithme de Dijkstra (qui est au programme). Sa complexité avec une file de priorité implémentée à l'aide d'un tas est en $O(N + M \log N)$ où N est le nombre de sommets de G et M est le nombre d'arêtes.

Question 4. On suppose à présent que tous les sommets sont des terminaux : $T = V$. Proposer un algorithme efficace permettant de calculer un sous-ensemble T-connexe minimal.

Indication : Utiliser un algorithme du cours.

Solution : Manifestement un sous-ensemble T-connexe ne peut exister que si le graphe est connexe (suivant la première question), donc on suppose le graphe connexe. On veut maintenant un sous-ensemble connexe de poids minimal qui contienne tous les sommets du graphe. C'est un arbre couvrant minimal, qui peut être calculé avec l'algorithme de Kruskal (explicitement au programme). Noter que la complexité de l'algorithme de Kruskal n'est pas au programme et qu'elle n'est pas évidente (elle dépend de la structure d'union-find utilisée), donc on ne la demandera pas.

Question 5. On recherche à présent un algorithme efficace pour calculer un ensemble T-connexe minimal dans le cas général. Quelle condition structurelle simple est toujours respectée par un sous-ensemble T-connexe minimal ? Démontrer ce fait rigoureusement.

Indication : Démontrer qu'un tel sous-ensemble est un arbre.

Indication : Procéder par l'absurde.

Solution : On va démontrer que tout sous-ensemble T-connexe est un arbre (potentiellement sans arêtes, si $|T| < 2$). (Ne pas laisser le candidat ou la candidate se lancer dans une caractérisation plus fine.) *Remarque : le nom standard des "sous-ensembles connexes minimaux" est en réalité "arbre de Steiner", le sujet n'emploie pas ce nom afin de leur permettre de parvenir indépendamment à cette observation.*

On procède par l'absurde. Soit X un sous-ensemble T-connexe minimal, et supposons que ce n'est pas un arbre. Ainsi X contient un cycle. Considérons X' obtenu en supprimant l'une quelconque des arêtes e du cycle. Ainsi X' est de poids total strictement inférieur. Or, X' est toujours T-connexe. En effet, prenons une paire $u \neq v$ de terminaux, et considérons un chemin qui les relie dans X. Si ce chemin n'utilise pas e, c'est également un chemin dans X'. Sinon, on modifie le chemin pour remplacer e par le chemin entre les deux extrémités de e qui fait le tour du cycle. Dans tous les cas, on obtient des chemins qui témoignent du fait que X' est T-connexe et de poids strictement moindre que X qui était supposé minimal, contradiction.

Question 6. Une idée pour calculer un ensemble T-connexe minimal est de procéder en choisissant, pour chaque paire de terminaux $u \neq v$ de T, un ensemble $\{u, v\}$-connexe $X_{u,v}$ minimal, et de prendre X comme l'union des $X_{u,v}$. Cette approche vous paraît-elle réaliste ? Pourquoi ?

Indication : Cette approche ne peut pas fonctionner, elle ne donnera pas toujours une solution correcte.

Indication : Construire un exemple où il vaut mieux connecter des terminaux en passant par un seul point central.

Solution : Une réponse très simple : si on prend un triangle où les 3 sommets sont terminaux, alors l'approche proposée dans la question prendra les 3 arêtes du triangle, alors que la solution optimale est un arbre. Ceci montre que l'approche, telle quelle, ne marchera pas.

Mais il n'est pas non plus vrai qu'une solution optimale est toujours une union de $X_{u,v}$ pour certaines paires de terminaux u, v. Ce qui suit explique pourquoi c'est vrai, mais on ne l'exigera pas forcément des candidates et candidats.

Il suffit de prendre un graphe avec un sommet central u et des terminaux $t_1, \ldots, t_n$ pour un $n > 3$ quelconque, et de mettre des arêtes de poids 2 reliant u à chaque t_i et des arêtes de poids 3 reliant les t_i entre eux. La solution optimale prend les arêtes centrales pour un poids total de $2n$, avec aucune arête de poids 3. On se convainc sans peine que les solutions qui prennent une arête de poids 3 sont moins bonnes.

Autre construction possible : on a n terminaux, $n > 2$, avec un chemin dont les arêtes ont poids 1, et toutes les autres arêtes ont poids $1 + \varepsilon$ pour un $\varepsilon > 0$ arbitraire. L'unique solution optimale prend les $n - 1$ arêtes de poids 1 et aucune arête de poids $1 + \varepsilon$. Pourtant la paire extrémale est reliée dans cette solution par un chemin de poids $n - 1$ qui est bien plus long que l'arête de poids $1 + \varepsilon$ qui relie directement les extrémités.

Autre réponse suivant la question précédente : l'algorithme de Kruskal optimise le poids de l'arête sélectionnée la plus grande, pas la longueur des chemins. Ceci aide à voir que sur l'exemple du paragraphe d'avant, l'algorithme de Kruskal ne prendra pas les arêtes de poids $1 + \varepsilon$.

Question 7. On suppose à présent qu'il y a 3 terminaux : $|T| = 3$. Concevoir un algorithme efficace pour calculer un sous-ensemble T-connexe minimal, et préciser sa complexité.

Indication : Il y a 2 possibilités : soit un des terminaux est un nœud interne de l'arbre, soit il y a un seul nœud interne de degré > 2.

Solution : La solution naïve consistant à relier les points deux à deux ne marche pas, c'est l'objet de la question précédente. La solution ci-dessous donne directement une réponse pour n'importe quel nombre constant de sommets (la question suivante), mais les candidats et candidates la trouveront peut-être plus facilement dans le cas particulier de 3 terminaux.

On a déjà montré que la structure de la solution optimale est un arbre. On remarque au demeurant deux propriétés.

Premièrement, chaque feuille de l'arbre est un terminal (mais les terminaux ne sont pas forcément des feuilles, ça peut aussi être des nœuds internes). En effet, si une feuille de l'arbre n'est pas un terminal, alors on la retire ainsi que l'arête qui lui est incidente, et l'ensemble reste T-connexe et son poids décroit, ce qui serait absurde.

Deuxièmement, si on appelle sommet *important* de l'arbre un sommet qui est une feuille ou de degré strictement supérieur à 2, si l'on considère tout chemin π dans l'arbre reliant deux sommets importants en ne passant que par des sommets non-importants, alors π est forcément un plus court chemin de G. En effet, si on a un chemin π' plus court dans G entre ces deux sommets, on peut remplacer π par π et rendre l'ensemble plus léger sans affecter sa T-connexité.

Ces deux propriétés suggèrent un algorithme simple. Comme il y a k terminaux dans T, l'arbre recherché a au plus k feuilles (qui sont des terminaux), et il a au plus $2k - 2$ sommets importants au total : les feuilles, et les sommets importants non-feuilles qu'on appellera *sommets importants internes* qui sont en nombre au plus $k - 2$. (Dans le cas $k = 3$, il y a au plus 4 sommets importants : soit il y a 3 feuilles (qui sont les terminaux) et un sommet de degré > 2 qui n'est pas un terminal, soit il y a 2 feuilles et l'arbre est un chemin sur lequel doit se trouver le 3e terminal.)

Donc on peut d'abord deviner les sommets importants internes (au plus $k-2$, certains sont des terminaux d'autres non), prendre les terminaux restants comme des feuilles (ceci donne les sommets importants), puis deviner une structure d'arbre sur les sommets importants (grossièrement on peut tester les $\binom{a-1}{b^2}$ combinaisons avec a le nombre de feuilles de l'arbre et b le nombre de nœuds importants), choisir pour chaque paire de sommets importants adjacents un chemin témoin par un algorithme de plus court chemin (il vaut mieux précalculer à l'avance les plus

courts chemins de chaque sommet à chaque sommet avec l'algorithme de Floyd-Warshall par exemple), et calculer le poids total de l'arbre.

On trouvera ainsi l'arbre optimal en temps polynomial en le graphe d'entrée (mais avec une dépendance exponentielle en k).

Dans le cas particulier de 3 sommets, qui est l'objet de la présente question, on obtient une complexité de $O(|V|^3)$ pour calculer toutes les distances de plus court chemin (avec l'algorithme de Floyd-Warshall, puis on teste, pour chaque sommet central (potentiellement un terminal), quelle est la somme des distances de ce sommet central aux autres terminaux (la distance peut être nul si le sommet central est l'un des terminaux). Ainsi la complexité obtenue est de $O(|V|^3)$ au total. On peut aussi appliquer l'algorithme de Dijkstra depuis chaque choix de sommet central, on obtient alors $O(|V| \times (|V| + |E|) \times \log |V|)$. C'est moins bien que $O(|V|^3)$ si le graphe est dense, mais meilleur s'il n'est pas dense.

Une façon plus élégante de procéder est l'algorithme de Dreyfus-Wagner. On la présente à présent, mais toute approche garantissant une complexité polynomiale à k fixé sera acceptée. Dans cet algorithme, on calcule récursivement un sous-ensemble $(T' \cup \{u\})$-connexe optimal pour chaque $T' \subseteq T$ et chaque sommet arbitraire u (qui jouera le rôle de sommet important interne qui n'est pas un terminal). Quand on veut un sous-arbre $(T' \cup \{u\})$-connexe, la réponse est triviale si T' est vide. Sinon, on devine le point d'attache de u à un arbre optimal pour T', c'est-à-dire qu'on teste chaque sommet possible v (y compris $u = v$). Entre u et v, il y a un plus court chemin de u à v (possiblement vide). Ensuite, de deux choses l'une : soit v était une feuille de l'arbre, soit non. Si c'est une feuille, alors v est forcément un terminal et on avait $v \in T'$: on peut alors calculer récursivement un sous-ensemble $(T'' \cup \{v\})$-connexe optimal où $T'' := T' \setminus \{v\}$, et on a $|T''| < |T'|$ donc on a progressé. Si u n'est pas une feuille, alors il partitionne l'arbre en (au moins) deux parties, avec un ensemble non-vide de terminaux de T' de chaque côté : on devine donc une partition non-triviale de T' en T'_1 et T'_2 et on calcule récursivement un sous-ensemble $(T'_1 \cup \{v\})$-connexe et un sous-ensemble $(T'_2 \cup \{v\})$-connexe qu'on combine. À nouveau $|T'_1| < |T'|$ et $|T'_2| < |T'|$ donc la récursion progresse.

Question 8. On fixe un entier $k \in \mathbb{N}$ et on suppose qu'il y a au plus k terminaux. Généraliser l'algorithme précédent, pour garantir une complexité polynomiale à k fixé.

Indication : On a vu qu'un sous-ensemble T-connexe est un arbre : que peut-on dire des feuilles ?

Indication : Que peut-on dire des chemins dans l'arbre ?

Solution : Cf question précédente.

On s'intéresse maintenant au cas des graphes orientés.

Question 9. Étendre la définition d'un sous-ensemble T-connexe au cas des graphes orientés. Comment caractériser l'existence d'un tel ensemble ?

Indication : Pas d'indication.

Solution : La définition attendue est que, pour chaque *couple* (s, t) de terminaux distincts, il y a un chemin orienté allant de s à t. Pour qu'une solution existe, les terminaux doivent tous être dans la même composante *fortement connexe* du graphe orienté, ce qui peut se vérifier en appliquant l'algorithme de Kosaraju (au programme).

Question 10. On suppose toujours le graphe orienté, et on considère le cas $|T| = 2$. Caractériser la forme d'un sous-ensemble T-connexe minimal dans ce cas.

Indication : Fixer un chemin, et étudier ce que fait l'autre chemin.

Solution : Une solution se compose d'un chemin de s à t, et d'un chemin de t à s, qui peuvent partager des arêtes. Ni l'un ni l'autre n'est nécessairement un plus court chemin. En revanche chacun de ces chemins peut bien sûr être supposé simple individuellement.

Du reste, une fois fixé le chemin π de s à t, en notant X l'ensemble des sommets intermédiaires de π et en le voyant comme ordonné par l'ordre de visite dans π, le chemin π' qui part de t peut aller à s sans utiliser de sommet de X, ou bien il peut atteindre un sommet u_1 de X. Il peut alors suivre un certain nombre d'arêtes de π, puis ressortir à un sommet v_1 de X en franchissant une arête qui n'est pas dans π ; noter que $u_1 \leq v_1$. S'il entre à nouveau dans un sommet u_2 de X, on voit alors que forcément $u_2 < u_1$; en effet sinon on a un chemin potentiellement vide de u_1 à u_2 dans π et on peut ainsi éliminer les arêtes entre v_1 et u_2 de π' et rendre la solution plus légère. Du reste le chemin va devoir à nouveau franchir une arête qui n'est pas dans π (puisque π est simple donc ne repasse pas par la source s), et si on note le sommet v_2 où ceci se produit on a également $v_2 < u_1$.

En répétant le raisonnement, on voit qu'il y a un entier $m \in \mathbb{N}$ et une séquence $u_m \leq v_m < \cdots < u_1 \leq v_1$ de sorte que π' soit un chemin qui visite dans l'ordre $t, u_1, v_1, u_2, v_2, \cdots, u_m, v_m, s$, les parties entre t et u_1 et entre v_m et s et entre v_i et u_{i+1} pour chaque i ne visitant aucun sommet de X, et les parties entre u_i et v_i pour chaque i réutilisant des arêtes de π.

Malheureusement, m n'est pas constant, donc on ne peut pas facilement deviner cette structure ; il faut un algorithme plus sophistiqué ce qui est l'objet de la question suivante.

Question 11. À l'aide de cette caractérisation, proposer un algorithme efficace pour calculer un sous-ensemble T-connexe minimal dans le cas d'un graphe orienté pour $|T| = 2$.

Indication : Faire une exploration de graphe sur les paires de sommets.

Solution : L'algorithme, élémentaire mais astucieux, est présenté dans Feldman et Ruhl, SICOMP, 2006, section 2.

L'idée de l'algorithme est de calculer un chemin π partant de s et franchissant les arêtes vers l'avant, et un chemin π' arrivant dans s et franchissant les arêtes dans le sens inverse, simultanément, en se souvenant de où en est chaque chemin, et en comptabilisant chaque arête franchie par l'un des chemins. On peut à chaque étape soit faire avancer π seul, soit faire reculer π' seul, soit les faire avancer "ensemble". Dans ce cas spécifique, π est avancé jusqu'à un sommet u et π' a reculé jusqu'à un sommet t, et il y a un chemin de u à t : on comptabilise alors le coût d'un tel chemin de poids minimal, et on échange u et t (c'est-à-dire que u avance jusqu'à t, et t recule jusqu'à u, et on ne franchit le chemin témoin qu'une seule fois). On cherche à faire en sorte que π parvienne à t et π' parvienne aussi à reculer jusqu'à t. Cet algorithme dynamique s'implémente simplement pour avoir une complexité en $O(N^3)$ qui correspond au précalcul de tous les plus courts chemins (avec Floyd-Warshall par exemple) puis à explorer au plus $O(N^2)$ couples d'états avec à chaque fois au plus $O(N)$ voisins possibles.

L'algorithme se généralise à un nombre constant de terminaux, mais c'est encore plus compliqué ; cf l'article de Feldman et Ruhl. Ceci peut être demandé à l'oral pour discuter avec les meilleurs candidats et meilleures candidates.

Codage par couleur

On considère des graphes orientés, supposés sans boucle. On cherchera dans ces graphes un chemin d'une longueur spécifique (sans imposer de point de départ et d'arrivée), où la *longueur* d'un chemin est le nombre de sommets qui apparaissent dans le chemin. Sauf mention explicite du contraire, on considère toujours des chemins *simples*, c'est-à-dire qui ne passent pas deux fois par le même sommet.

Question 1. Y a-t-il des graphes fortement connexes arbitrairement grands sans chemin de longueur 4 ?

Question 2. Proposer un algorithme naïf pour déterminer, étant donné un graphe G et un entier $k \in \mathbb{N}$, si G contient un chemin de longueur k. Quelle est la complexité de cet algorithme ?

Question 3. Dans cette question seulement, on s'intéresse à des chemins *non nécessairement simples*. Proposer un algorithme efficace pour déterminer, étant donné un graphe G et un entier $k \in \mathbb{N}$, si G contient un tel chemin de longueur k.

Question 4. Dans cette question seulement, on suppose que les sommets du graphe d'entrée G portent chacun une couleur parmi l'ensemble $\{1, \ldots, k\}$, et on souhaite savoir si G admet un chemin *multicolore* de longueur k, c'est-à-dire un chemin $v_1, \ldots, v_k$ où les couleurs des sommets sont deux à deux distinctes.

Proposer un algorithme pour résoudre ce problème, et en expliciter la complexité.

Pour améliorer le temps d'exécution de l'algorithme de la question 2, on va concevoir un algorithme *probabiliste*. Un tel algorithme a la possibilité de tirer au hasard certaines valeurs au cours de son exécution ; et on regarde, sur chaque entrée, quelle est la probabilité que l'algorithme réponde correctement, en fonction de ces tirages aléatoires.
On veut résoudre le problème suivant : étant donné un graphe G et un entier k, on veut savoir si G a un chemin de longueur k. On considère d'abord l'algorithme (1) que voici. On répète M fois l'opération suivante (où M sera déterminé ensuite) : tirer k sommets au hasard et vérifier si ces sommets forment un chemin. On répond OUI si l'un de ces tirages est réussi et NON dans le cas contraire.

Question 5. On suppose que le graphe G a n sommets et contient précisément $1 \leq c \leq n^k$ chemins de longueur k. Exprimer la probabilité que l'algorithme (1) réponde OUI, en fonction de M, de k, de n, et de c.

Question 6. Si on suppose que G n'a pas de chemin de longueur k, que répond l'algorithme (1) ?

Question 7. Pour $M = n^k$, montrer que l'algorithme répond correctement dans les deux cas avec probabilité au moins $1/2$.

Question 8. Quel est le temps d'exécution de l'algorithme (1) pour ce choix de M ? Commenter.

On considère à présent l'algorithme (2) dont le principe est le suivant. On répète M fois l'opération suivante (où M sera déterminé ensuite) : tirer un ordre total aléatoire $v_1 < \cdots < v_n$ sur les sommets de G, retirer les arêtes (u, v) où on a $u > v$, et vérifier si le graphe résultant $G_<$ a un chemin de longueur k (d'une manière qui reste à déterminer).

Question 9. Quelle propriété ont les graphes $G_<$ construits par cet algorithme ? Comment exploiter cette propriété pour trouver efficacement les chemins de longueur k ?

Question 10. Proposer un M qui garantisse que cet algorithme réponde correctement avec une probabilité $\geq 1/2$. Quelle complexité obtient-on ? commenter.

Question 11. En utilisant les chemins multicolores, proposer un algorithme probabiliste qui répond correctement avec probabilité au moins $1/2$ et s'exécute en $O((2e)^k \times (n + m))$ sur un graphe à n sommets et m arêtes.

Codage par couleur

On considère des graphes orientés, supposés sans boucle. On cherchera dans ces graphes un chemin d'une longueur spécifique (sans imposer de point de départ et d'arrivée), où la *longueur* d'un chemin est le nombre de sommets qui apparaissent dans le chemin. Sauf mention explicite du contraire, on considère toujours des chemins *simples*, c'est-à-dire qui ne passent pas deux fois par le même sommet.

Question 1. Y a-t-il des graphes fortement connexes arbitrairement grands sans chemin de longueur 4 ?

Indication : La réponse est positive.

Solution : Oui, prendre un graphe avec un sommet central u et des arêtes bidirectionnelles reliant u à des sommets $v_1, \dots, v_n$ pour n arbitrairement grand. Les chemins simples les plus longs sont de longueur 3, de la forme v_i, u, v_j avec $i \neq j$. Ainsi, la question de déterminer si un graphe admet un grand chemin simple n'est pas triviale, vu qu'elle peut être vraie ou fausse sur de grands graphes.

Question 2. Proposer un algorithme naïf pour déterminer, étant donné un graphe G et un entier $k \in \mathbb{N}$, si G contient un chemin de longueur k. Quelle est la complexité de cet algorithme ?

Indication : Tester toutes les combinaisons.

Solution : *Note préliminaire : le sujet est inspiré de l'exposé suivant, par Marcin Pilipczuk (transparents de Łukasz Kowalik) :* `https://www.youtube.com/watch?v=PDp_JmpiSr8`

On teste simplement chaque choix possible de k sommets $v_1, \dots, v_k$ et on vérifie que les arêtes (v_i, v_{i+1}) existent pour chaque $1 \leq i < k$. La complexité de cet algorithme est en $O(n^k \times m)$ si le graphe a n sommets et m arêtes.

Alternativement, on peut tester chaque sommet de départ et lancer un DFS depuis le sommet où on s'interdit de revisiter les sommets déjà sur la pile (pour garantir que le chemin est simple) mais on ne s'interdit pas de revisiter des sommets déjà visités (en effet on les atteint potentiellement par d'autres sommets et cela permettrait de faire un chemin plus long). La complexité est analogue.

(Au demeurant, le problème est NP-difficile vu que la recherche d'un chemin Hamiltonien en est un cas particulier. Mais on ne le demande pas.)

Question 3. Dans cette question seulement, on s'intéresse à des chemins *non nécessairement simples*. Proposer un algorithme efficace pour déterminer, étant donné un graphe G et un entier $k \in \mathbb{N}$, si G contient un tel chemin de longueur k.

Indication : Créer plusieurs copies du graphe.

Solution : Étant donné le graphe $G = (V, E)$, on crée le graphe $G \times [k]$ dont les sommets sont $V \times \{1, \dots, k\}$ et les arêtes sont $\{((u, i), (v, i+1)) \mid (u, v) \in E, 1 \leq i < k\}$. On teste ensuite simplement s'il existe un chemin de la première copie (sommets de la forme $\{(u, 1) \mid u \in V\}$) à la dernière (sommets de la forme $\{(u, k) \mid u \in V\}$) ce qu'on peut faire avec un parcours en largeur par exemple. La complexité est en $O((n + m) \times k)$.

On peut aussi voir cela comme une exploration dans G qui s'autorise à repasser par un sommet si on l'a atteint avec une distance plus grande ; tester tous les sommets de départ, etc., ce qui mène à des complexités analogues.

En réalité un algorithme plus efficace pour ce problème est de tester en temps linéaire si G est acyclique. S'il a un cycle, alors on sait qu'il admet un chemin non nécessairement simple de n'importe quelle longueur. Sinon, on peut utiliser l'algorithme en temps linéaire sur les graphes acycliques orientés qui est traité plus loin dans le sujet.

Question 4. Dans cette question seulement, on suppose que les sommets du graphe d'entrée G portent chacun une couleur parmi l'ensemble $\{1, \ldots, k\}$, et on souhaite savoir si G admet un chemin *multicolore* de longueur k, c'est-à-dire un chemin $v_1, \ldots, v_k$ où les couleurs des sommets sont deux à deux distinctes.

Proposer un algorithme pour résoudre ce problème, et en expliciter la complexité.

Indication : Un chemin multicolore peut-il ne pas être simple ?

Indication : Créer plusieurs copies du graphe d'une manière analogue à la question précédente.

Solution : On crée 2^k copies du graphe $G = (V, E)$, c'est-à-dire qu'on considère le graphe $(V \times 2^k, E')$ avec E' défini comme suit : pour chaque arête $(u, v) \in E$, en notant c la couleur de v, pour chaque $S \subseteq 2^k$, on crée $((u, S), (v, S \cup \{c\}))$.

On cherche ensuite à savoir s'il existe un chemin depuis un sommet de la forme $(u, \{c\})$ pour c la couleur de u à un sommet de la forme $(v, 2^k)$. Si un tel chemin existe, c'est forcément un chemin dans G (par projection sur la première composante), et par définition des arêtes il a forcément visité un sommet de chaque couleur. En particulier le chemin est forcément simple. Donc cet algorithme identifie bien correctement si G contient un chemin multicolore.

La complexité est linéaire en la taille du graphe produit c'est-à-dire $O((n+m)2^k)$. (On remarque que le coloriage nous aide en nous dispensant de se souvenir des sommets visités, ce qui serait nécessaire sinon pour savoir si le chemin est simple : c'est pourquoi on a un facteur 2^k plutôt que m^k ou n^k ce qui est bien sûr meilleur.)

Pour améliorer le temps d'exécution de l'algorithme de la question 2, on va concevoir un algorithme *probabiliste*. Un tel algorithme a la possibilité de tirer au hasard certaines valeurs au cours de son exécution ; et on regarde, sur chaque entrée, quelle est la probabilité que l'algorithme réponde correctement, en fonction de ces tirages aléatoires.
On veut résoudre le problème suivant : étant donné un graphe G et un entier k, on veut savoir si G a un chemin de longueur k. On considère d'abord l'algorithme (1) que voici. On répète M fois l'opération suivante (où M sera déterminé ensuite) : tirer k sommets au hasard et vérifier si ces sommets forment un chemin. On répond OUI si l'un de ces tirages est réussi et NON dans le cas contraire.

Question 5. On suppose que le graphe G a n sommets et contient précisément $1 \leq c \leq n^k$ chemins de longueur k. Exprimer la probabilité que l'algorithme (1) réponde OUI, en fonction de M, de k, de n, et de c.

Indication : Utiliser l'indépendance des M tirages.

Indication : Montrer que l'algorithme répond OUI avec probabilité $1 - (1 - c/n^k)^M$.

Solution : Il y a n^k tirages possibles et c de ces tirages réussissent. Donc chaque tirage a une probabilité de c/n^k de réussir, et une probabilité de $1 - c/n^k$ d'échouer. On effectue M tirages indépendants, donc avec probabilité $(1 - c/n^k)^M$ aucun tirage ne réussit et l'algorithme répond NON (la réponse incorrecte) ; et avec probabilité $1 - (1 - 1/c^k)^M$ un tirage au moins réussit et l'algorithme répond OUI (la réponse correcte).

Question 6. Si on suppose que G n'a pas de chemin de longueur k, que répond l'algorithme (1) ?

Indication : Pas d'indication.

Solution : Dans ce cas chaque tirage échoue nécessairement et l'algorithme répond toujours NON.

Question 7. Pour $M = n^k$, montrer que l'algorithme répond correctement dans les deux cas avec probabilité au moins $1/2$.

Indication : On pourra établir l'inégalité : pour tout $x \geq 1$, on a $(1 - 1/x)^x \leq e^{-1}$.

Indication : Pour démontrer cette inégalité, on pourra partir de l'inégalité $e^x \geq 1 + x$ pour tout réel x.

Solution : Si le graphe d'entrée n'a pas de chemin, il n'y a rien à montrer par la question précédente. S'il y en a un, il faut montrer que la probabilité d'échec est d'au plus $1/2$. C'est-à-dire montrer que $(1 - 1/n^k)^{n^k} \leq 1/2$. Il suffit donc d'établir l'inégalité donnée en indication : pour tout $x \geq 1$, on a $(1 - 1/x)^x \leq e^{-1}$. En effet on a $e^{-1} < 1/2$ (vu que $e > 2$).

Pour démontrer l'inégalité, on peut utiliser l'inégalité $e^x \geq 1 + x$ donnée en indication (convexité de la fonction exponentielle, ou facilement démontrable en dérivant et en faisant le tableau de variations de la fonction comme au lycée). À partir de l'inégalité, on substitue x par $-1/x$ et on obtient $e^{-1/x} \geq 1 - 1/x$. Puis, comme $x \geq 1$, mettre à la puissance x préserve l'inégalité, donc $e^{-1} \geq (1 - 1/x)^x$. Voir aussi cette discussion : `https://math.stackexchange.com/a/544696`

Autre preuve possible de l'inégalité : on a $(1 - 1/x)^x = \exp(x \ln(1 - 1/x))$. Par concavité du logarithme qui est sous sa tangente en 1, on a $\ln(1 - 1/x) \leq -1/x$, donc on peut majorer par $\exp(1)$.

Question 8. Quel est le temps d'exécution de l'algorithme (1) pour ce choix de M ? Commenter.

Indication : Pas d'indication.

Solution : L'algorithme (1) s'exécute en $O(M \times m)$. Donc pour le M indiqué on retrouve la même complexité qu'en question 2. Ce n'est donc pas très intéressant pour le moment : ces questions servent à introduire le cadre général d'algorithmes probabilistes qui est étudié dans la suite.

On considère à présent l'algorithme (2) dont le principe est le suivant. On répète M fois l'opération suivante (où M sera déterminé ensuite) : tirer un ordre total aléatoire $v_1 < \cdots < v_n$ sur les sommets de G, retirer les arêtes (u, v) où on a $u > v$, et vérifier si le graphe résultant $G_<$ a un chemin de longueur k (d'une manière qui reste à déterminer).

Question 9. Quelle propriété ont les graphes $G_<$ construits par cet algorithme ? Comment exploiter cette propriété pour trouver efficacement les chemins de longueur k ?

Indication : Utiliser le fait que ces graphes sont acycliques.

Indication : On peut chercher les chemins les plus longs dans un graphe acyclique avec un tri topologique.

Solution : (Les tris topologiques sont au programme, donc exigibles. Mais on acceptera qu'un candidat ou une candidate propose le bon algorithme sans parler explicitement de tri topologique.)

Le graphe $G_<$ est acyclique pour tout ordre total $<$, en effet l'ordre $<$ peut servir de tri topologique (autrement dit un cycle donnerait un cycle dans l'ordre total ce qui par transitivité impliquerait qu'il n'est pas asymmétrique).

Or, on peut résoudre efficacement sur des graphes acycliques le problème de savoir s'ils admettent un chemin de longueur k. En effet, on considère les sommets de $G_<$ dans l'ordre inverse de $<$, et on calcule pour chaque sommet v la longueur $l(v)$ du plus long chemin qui part de v. Si v n'a aucune arête sortante, cette longueur est de 0. Si v a des arêtes sortantes vers $w_1, \dots, w_k$, les valeurs $l(w_1), \dots, l(w_k)$ étant déjà calculées car $v < w_i$ pour tout i, on pose $l(v) = 1 + \max_i l(w_i)$.

Cet algorithme s'exécute en temps linéaire, $O(n+m)$ sur un graphe à n sommets et m arêtes (indépendant de k).

Question 10. Proposer un M qui garantisse que cet algorithme réponde correctement avec une probabilité $\geq 1/2$. Quelle complexité obtient-on ? commenter.

Indication : Si le graphe a un chemin de longueur k, combien de tirages permettront de l'identifier ?

Solution : Encore une fois, l'algorithme est biaisé vers le faux : si le graphe d'entrée n'a pas de chemin il répond toujours faux. Donc il suffit de majorer la probabilité que l'algorithme réponde NON alors que le graphe contient un chemin de longueur k. Or, si le graphe d'entrée a un chemin de longueur k, alors il est identifié si chacune de ses $k-1$ arêtes sont conservées, et c'est le cas si et seulement si les sommets sont triés dans le bon ordre par l'ordre total qu'on a tiré. Ainsi chaque tirage a une probabilité de $1/k!$ d'identifier le chemin, et la probabilité d'échec est d'au plus $(1-1/k!)^M$.

On peut donc prendre $M = k!$ pour obtenir une probabilité d'échec constante, pour la même raison que pour l'algorithme (1). La complexité de l'algorithme est alors de $O(k! \times (n+m))$ si on suppose qu'on peut tirer un ordre aléatoire en temps $O(m)$ (c'est le cas, par exemple en $O(n)$ avec l'algorithme de Fisher-Yates pour tirer une permutation aléatoire, mais on n'exigera pas de tels détails).

On remarque que la complexité de $O(k! \times (n+m))$ est bien meilleure que $O(m^k)$ lorsque k n'est pas trop grand.

Question 11. En utilisant les chemins multicolores, proposer un algorithme probabiliste qui répond correctement avec probabilité au moins $1/2$ et s'exécute en $O((2e)^k \times (n+m))$ sur un graphe à n sommets et m arêtes.

Indication : Tirer un coloriage aléatoire des sommets du graphe.

Solution : On va répéter M fois l'opération suivante : tirer un k-coloriage aléatoire du graphe, et vérifier si on trouve un chemin multicolore de longueur k avec l'algorithme de la question 4 en temps $O(2^k(n+m))$.

Quand il n'y a pas de chemin de longueur k, alors il n'y a pas de tel chemin qui soit multicolore et l'algorithme répond correctement NON.

Quand il y a un tel chemin, la probabilité de le trouver est la probabilité que ce chemin soit multicolore dans le coloriage tiré. Il y a k^k coloriages possibles dont $k!$ sont corrects. Admettons pour l'instant que $k! \geq (k/e)^k$ (on le montrera ensuite). Ainsi, pour un coloriage aléatoire, l'algorithme répond OUI avec probabilité au moins $1/e^k$.

Ainsi, si on prend $M = e^k$, on aura à nouveau une probabilité constante de succès. Le temps d'exécution sera alors bien de $O(e^k 2^k (n+m))$, ce qui est encore mieux qu'à la question précédente.

Pour démontrer que $k! \geq (k/e)^k$, on sait par la formule de Stirling que $k! \sim \sqrt{2\pi k}(k/e)^k$, ce qui donne une idée du fait que c'est une bonne estimation. Pour le montrer formellement, on peut adapter la technique utilisée pour démontrer la formule de Stirling (la démonstration de la formule n'est pas au programme mais retrouvable si besoin avec des indications). Voici l'argument : on cherche à donner une borne inférieure de $\ln k! = \sum_{2 \leq i \leq k} \ln(i)$.

Mais cette somme est au moins l'intégrale $\int_1^k \ln(x)\mathrm{d}x$ puisque ln est croissante. Une primitive du logarithme est $x\ln x - x$ donc on obtient la borne inférieure $\ln(k!) \geq k\ln k - k + 1$. Et en exponentiant : $k! \geq e(k/e)^k \geq (k/e)^k$ comme affirmé.

Cette technique du codage par couleur est utilisable pour d'autres problèmes : elle a été introduite par Alon, Yuster, et Zwick. Voir par exemple : `https://en.wikipedia.org/wiki/Color-coding`.

Universalité pour les facteurs et les sous-mots

On considère des automates déterministes et non-déterministes sur un alphabet Σ^*. Un automate est dit *universel* s'il accepte tous les mots de Σ^*.

Question 1. Donner un exemple d'un automate déterministe émondé sur l'alphabet $\Sigma = \{a, b\}$ qui soit universel et ait au moins deux états.

Question 2. Expliquer comment on peut décider, étant donné un automate déterministe, s'il est universel ou non. Quelle est la complexité en temps de l'algorithme ?

Dans les trois prochaines questions, on va s'intéresser au problème de décider si un automate nondéterministe est universel. On admettra que le problème SAT est NP-difficile, où SAT est défini comme suit : étant donné une formule de la logique propositionnelle en forme normale conjonctive, décider si elle admet une valuation satisfiante.

Question 3. À l'aide du résultat admis, montrer que le problème suivant est NP-difficile : étant donné une formule φ en forme normale disjonctive, décider s'il y a une valuation qui ne satisfait pas φ.

Question 4. On travaille avec des formules sur l'ensemble de variables $X_n = \{x_1, \ldots, x_n\}$, et on associe à chaque valuation $\nu \colon X_n \to \{0, 1\}$ le mot $w_\nu = \nu(x_1) \cdots \nu(x_n)$ de longueur n sur l'alphabet $\Sigma = \{0, 1\}$.

Montrer que, pour toute conjonction φ de littéraux sur un ensemble $X \subseteq X_n$ de variables, on peut efficacement construire un automate A_φ sur l'alphabet $\Sigma = \{0, 1\}$ avec la propriété suivante : pour chaque valuation ν de X_n, on a que ν satisfait φ si et seulement A_φ accepte w_ν.

Question 5. Que conclure du problème de décider, étant donné un automate nondéterministe A, si A est universel ?

Un automate A est dit *sous-mot-universel* si tout mot de Σ^* apparaît comme sous-mot d'un mot du langage de A.

Question 6. Étant donné un automate nondéterministe A, expliquer comment construire un automate A' qui reconnaisse le langage des sous-mots du langage de A.

Question 7. Étant donné un automate nondéterministe A, on souhaite construire un automate *déterministe* A' qui reconnaisse le langage des sous-mots du langage de A. Montrer que A' nécessite en général un nombre d'états exponentiel en la taille de A. On ne supposera pas que la taille de l'alphabet est constante.

Question 8. Proposer un algorithme efficace pour déterminer, étant donné un automate nondéterministe A, si A est sous-mot-universel.

Un automate A est dit *facteur-universel* si tout mot de Σ^* apparaît comme facteur d'un mot du langage de A.

Question 9. Montrer qu'il est difficile de déterminer si un automate nondéterministe est facteur-universel.

Question 10. Proposer un algorithme en temps polynomial pour déterminer si un automate déterministe est facteur-universel.

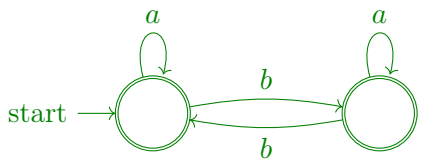
Universalité pour les facteurs et les sous-mots

On considère des automates déterministes et non-déterministes sur un alphabet Σ^*. Un automate est dit *universel* s'il accepte tous les mots de Σ^*.

Question 1. Donner un exemple d'un automate déterministe émondé sur l'alphabet $\Sigma = \{a, b\}$ qui soit universel et ait au moins deux états.

Indication : Partir d'un tel automate avec un seul état.

Solution : Prendre par exemple :



Question 2. Expliquer comment on peut décider, étant donné un automate déterministe, s'il est universel ou non. Quelle est la complexité en temps de l'algorithme ?

Indication : Passer au complémentaire.

Solution : Étant donné un automate déterministe A, il est universel si et seulement si le complémentaire de son langage est vide. Or, on peut construire en temps linéaire en A un automate A' reconnaissant le langage complémentaire : il faut d'abord compléter A si nécessaire en ajoutant un puits, puis échanger les états finaux et non-finaux. Ensuite, pour tester si A' accepte le langage vide, il faut tester s'il existe un chemin de l'état initial à l'un des états finaux : ceci peut se faire également en temps linéaire par un parcours de graphe, par exemple en profondeur.

Un autre argument consiste à rendre l'automate complet et à vérifier que tous les états accessibles sont finaux. Si c'est le cas, alors manifestement tout mot est accepté. À l'inverse, si un état accessible n'est pas final, alors un mot qui nous mène à cet état n'est pas accepté. La complexité est également linéaire avec cette approche.

Un autre argument encore est de dire qu'on peut minimiser l'automate et vérifier qu'on obtient un automate avec un état qui est final et a une boucle pour chaque transition. C'est correct, mais la complexité de la minimisation est plus importante que celle de l'algorithme précédent ; ainsi on demandera le cas échéant aux candidates et candidats si un algorithme plus efficace est possible.

Dans les trois prochaines questions, on va s'intéresser au problème de décider si un automate nondéterministe est universel. On admettra que le problème SAT est NP-difficile, où SAT est défini comme suit : étant donné une formule de la logique propositionnelle en forme normale conjonctive, décider si elle admet une valuation satisfiante.

Question 3. À l'aide du résultat admis, montrer que le problème suivant est NP-difficile : étant donné une formule φ en forme normale disjonctive, décider s'il y a une valuation qui ne satisfait pas φ.

Indication : Utiliser les lois de De Morgan.

Solution : Étant donné une formule φ en forme normale conjonctive (CNF), la loi de De Morgan nous permet d'exprimer $\neg\varphi$ comme une formule en forme normale conjonctive (DNF). Une valuation satisfait φ si et seulement si elle falsifie $\neg\varphi$. Ainsi, φ a une valuation satisfaisante si et seulement si $\neg\varphi$ a une valuation falsifiante.

Question 4. On travaille avec des formules sur l'ensemble de variables $X_n = \{x_1, \ldots, x_n\}$, et on associe à chaque valuation $\nu\colon X_n \to \{0,1\}$ le mot $w_\nu = \nu(x_1)\cdots\nu(x_n)$ de longueur n sur l'alphabet $\Sigma = \{0,1\}$.

Montrer que, pour toute conjonction φ de littéraux sur un ensemble $X \subseteq X_n$ de variables, on peut efficacement construire un automate A_φ sur l'alphabet $\Sigma = \{0,1\}$ avec la propriété suivante : pour chaque valuation ν de X_n, on a que ν satisfait φ si et seulement A_φ accepte w_ν.

Indication : Considérer d'abord le cas d'une clause vide (sans aucun littéral).

Solution : La question est facile une fois qu'on a compris ce qui est demandé. On exclut d'abord le cas où φ n'est pas satisfiable, c'est-à-dire le cas où elle contient deux littéraux de polarité opposée pour la même variable : si c'est le cas, on peut prendre A_φ un automate sans état final, qui accepte le langage vide.

Sinon, pour chaque variable x_i avec $1 \leq i \leq n$, soit elle apparaît positivement dans φ et doit être vraie dans toute valuation qui satisfait φ, soit elle apparaît négativement et doit être fausse dans toute valuation, soit elle n'apparaît pas et il n'y a pas de contrainte sur sa valeur.

On construit donc un automate (déterministe incomplet) avec $n+1$ états $\{q_0, \ldots, q_{n+1}\}$. L'état q_0 est initial, l'état q_{n+1} est final, et pour chaque état $1 \leq i \leq n$:

- Si x_i apparaît positivement dans φ, il y a une transition de q_{i-1} à q_i étiquetée par 1.
- Si x_i apparaît négativement dans φ, il y a une transition de q_{i-1} à q_i étiquetée par 0.
- Si x_i n'apparaît pas dans φ, il y a une transition de q_{i-1} à q_i étiquetée par 0 et par 1.

Il est alors immédiat que A_φ accepte précisément les mots de longueur n qui correspondent à une valuation qui satisfait φ.

Question 5. Que conclure du problème de décider, étant donné un automate nondéterministe A, si A est universel ?

Indication : Étant donné des automates nondéterministes, comment construire un automate reconnaissant l'union de leurs langages ?

Solution : On montre que déterminer si un automate *n'est pas* universel est NP-difficile, par réduction depuis le problème de vérifier si une formule en DNF a une valuation falsifiante. Étant donné une telle formule $\varphi = \bigwedge_i \psi_i$, on construit les automates A_{ψ_i} comme en question précédente. On les modifie d'abord pour accepter tous les mots de longueur plus grande que n, et pour accepter les préfixes des mots déjà acceptés : on fait cela en rendant tous les états finaux et en rajoutant sur le dernier état une boucle pour tous les symboles de l'alphabet. En notant A'_{ψ_i} les automates résultants, ces automates acceptent tous les mots de $\{0,1\}^*$ à part les mots u qui ne peuvent pas être étendus en un mot $u' = uv$ (pour un certain $v \in \{0,1\}^*$) tel que les n premiers caractères de u' décrivent une valuation satisfaisante de ψ_i.

On définit maintenant A_φ comme la disjonction des A'_{ψ_i}. C'est-à-dire que, si on autorise les automates avec plusieurs états initiaux, il suffit de prendre leur union en gardant tous les états initiaux. Si le candidat ou la candidate tient à ce qu'il y ait un unique état initial, on peut soit l'ajouter en le reliant par des epsilon-transitions aux états initiaux et éliminer ensuite ces ε transitions, ou bien l'ajouter avec un caractère inutile en début de mot (tenant

lieu d'ε-transition) et modifier l'automate pour accepter le mot vide et les mots qui commencent par un autre caractère.

Cette réduction est manifestement en temps polynomial. Montrons qu'elle est correcte. Si φ a une valuation falsifiante ν, alors il existe un mot w_ν qui est rejeté par tous les A_{ψ_i}. Il est clairement toujours rejeté par les A'_{ψ_i}. Ainsi, il est rejeté par A_φ, et A_φ n'est pas universel. À l'inverse, si A_φ rejette un mot w, alors tous les A'_{ψ_i} le rejettent. Pour chaque i, comme tous les états de A'_{ψ_i} sont finaux, on sait que le mot w est rejeté parce que dans son unique run dans A'_{ψ_i} on tente d'emprunter une transition qui n'existe pas. Ainsi, quitte à rajouter des symboles à la fin de w si nécessaire, on obtient un mot w' de longueur n qui est rejeté par tous les A'_{ψ_i}, donc par chacun des A_{ψ_i}. Ainsi, la valuation ν correspondant à w ne satisfait pas chacun des ψ_i. Ainsi, ν ne satisfait pas φ donc φ a une valuation falsifiante.

Pour être précis : cet argument montre que le problème de l'universalité pour les automates nondéterministes est coNP-difficile, c'est-à-dire que c'est le complémentaire d'un problème (la non-universalité des automates non-déterministes) qu'on vient de montrer NP-difficile. Selon la notion de réduction utilisée par les candidates et candidats, ils et elles peuvent affirmer que le problème est NP-difficile (en utilisant une réduction de Cook). En tout cas, il ne faut pas exiger de précision à ce sujet.

Un automate A est dit *sous-mot-universel* si tout mot de Σ^* apparaît comme sous-mot d'un mot du langage de A.

Question 6. Étant donné un automate nondéterministe A, expliquer comment construire un automate A' qui reconnaisse le langage des sous-mots du langage de A.

Indication : Utiliser des transitions spontanées.

Solution : La construction ci-dessous est donnée dans [KS14, Section 3.1], on la récapitule ci-dessous.

Pour chaque paire d'états q et q' de l'automate A telle qu'il y ait une transition de q à q', on ajoute une transition spontanée de q à q'. Ceci donne l'automate A'. On peut ensuite éliminer ces transitions spontanées (le fait que ce soit possible est exigible) pour obtenir l'automate final. (On demandera un automate sans transitions spontanées pour vérifier que le candidat ou la candidate sait dire qu'on peut les éliminer, mais on ne demandera pas le détail.) L'automate A' reconnaît le bon langage, en effet chaque mot accepté correspond manifestement à un sous-mot d'un mot accepté, et à l'inverse étant donné un sous-mot d'un mot accepté on peut construire un run qui accepte le sous-mot.

Noter que l'automate résultant est un automate nondéterministe, ce qui mène à la question suivante.

Question 7. Étant donné un automate nondéterministe A, on souhaite construire un automate *déterministe* A' qui reconnaisse le langage des sous-mots du langage de A. Montrer que A' nécessite en général un nombre d'états exponentiel en la taille de A. On ne supposera pas que la taille de l'alphabet est constante.

Indication : Utiliser le langage des mots qui ne contiennent pas toutes les lettres de l'alphabet.

Solution : Ce résultat est démontré dans [KS14, Section 3.2], qui montre par ailleurs un résultat plus fort : le résultat reste vrai même si on impose que l'alphabet ne contienne que deux symboles (mais la démonstration est alors plus difficile). On récapitule ci-dessous la preuve pour un alphabet de taille non-constante comme demandé.

Prenons $\Sigma = \{a_1, \ldots, a_k\}$ un alphabet de taille k. On pose le langage L sur Σ formé des mots qui n'utilisent pas toutes les lettres de l'alphabet. Ce langage peut être reconnu par un automate nondéterministe avec $O(k)$ états où on devine d'abord quelle lettre n'est pas utilisée, puis on lit le mot en vérifiant que cette lettre est absente.

Or, le langage L' des sous-mots de L est exactement L : en effet tout mot de L est manifestement dans L', à l'inverse tout sous-mot d'un mot de L est lui-même dans L ainsi $L' \subseteq L$. Il suffit donc de montrer une borne inférieure sur la taille d'un automate déterministe reconnaissant L. Or, après avoir lu chaque mot formé de $k/2$ lettres distinctes (en ordonnant Σ arbitrairement), il faut parvenir à un état différent de l'automate. En effet, supposons sinon qu'il y a deux ensembles $S \neq S'$ tels que les mots w_S et $w_{S'}$ correspondants mènent au même état q, on sait que la lecture de $w_{\Sigma \setminus S}$ depuis q doit être rejetée car $w_S w_{\Sigma \setminus S}$ contient toutes les lettres de Σ, alors que la lecture de $w_{S'} w_{\Sigma \setminus S}$ devait être acceptée (si on prend une lettre qui est dans $S \setminus S'$ alors elle n'est ni dans S' ni dans $\Sigma \setminus S$), contradiction.

Ainsi A doit avoir au moins $\binom{k/2}{k}$ états, ce qui établit la borne demandée.

Question 8. Proposer un algorithme efficace pour déterminer, étant donné un automate nondéterministe A, si A est sous-mot-universel.

Indication : Étudier d'abord l'alphabet $\Sigma = \{a, b\}$.

Indication : Que dire sur les composantes fortement connexes de l'automate ?

Solution : On suppose que l'automate est émondé ; sinon, cela peut s'assurer en temps linéaire. Le résultat est démontré dans [RSX12, Theorem 12], la preuve est récapitulée ci-dessous.

Il faut remarquer et démontrer la caractérisation suivante : un automate A est sous-mot-universel si et seulement s'il a une composante fortement connexe (CFC) qui contienne une transition pour chaque lettre de l'alphabet.

En effet, si c'est le cas, alors soit $w \in \Sigma^*$ un mot quelconque. On construit un mot w' du langage de A dont w soit sous-mot : on commence par un mot qui étiquette un chemin d'un état initial de A à un état de la CFC C que l'on a identifiée : ceci existe car l'automate est émondé. Ensuite, pour chaque lettre a de w, on prend un chemin dans C pour aller à une transition pour la lettre a, et on la franchit en restant dans C. Enfin, on prend un mot qui nous emmène vers un état final, ce qui existe car l'automate est émondé. Le mot w' construit est un mot du langage de l'automate où w apparaît comme facteur.

La direction réciproque est un peu plus difficile. Supposons que l'automate A soit sous-mot-universel, et soit n son nombre de CFC (c'est au plus le nombre d'états). Soit u un mot comprenant chaque lettre de l'alphabet au moins une fois, et considérons le mot u^n. Par hypothèse, A accepte un mot v dont u^n soit facteur. On écrit $v = v_1 \cdots v_n$, où chaque v_i admet u comme facteur. Considérons un run ρ acceptant de v, et considérons son passage dans chaque CFC. Par définition d'une CFC, quand ρ quitte une CFC, il ne revient jamais dans cette CFC. Ainsi, il doit exister un $1 \leq i \leq n$ tel que l'automate A reste dans une certaine CFC C en lisant v_i tout entier. Comme u est facteur de v_i, on voit ainsi que A franchit une transition de chaque lettre dans la CFC C, donc C témoigne du fait que A satisfait la condition.

La condition identifiée se teste en temps linéaire. En effet, étant donné un automate A, on calcule ses CFC en temps linéaire (par exemple avec l'algorithme de Kosaraju, exigible). On vérifie ensuite, pour chaque CFC, si elle contient bien une transition de chaque lettre, avec un tableau de booléens par exemple.

> Un automate A est dit *facteur-universel* si tout mot de Σ^* apparaît comme facteur d'un mot du langage de A.

Question 9. Montrer qu'il est difficile de déterminer si un automate nondéterministe est facteur-universel.

Indication : Adapter la preuve utilisée pour l'universalité d'un NFA.

Solution : Bien sûr un NFA peut être facteur-universel sans être universel, donc on ne peut pas simplement utiliser la preuve précédente. Le résultat est démontré dans [RSX12, Theorem 9], on redonne la preuve en question ci-dessous.

On sait qu'il est difficile de déterminer si un NFA A sur l'alphabet $\Sigma = \{0, 1\}$ est universel. On construit un nouveau NFA A' sur l'alphabet $\Sigma' = \{0, 1, b, e\}$ (où b et e signifient respectivement "begin" et "end") à partir de A de la façon suivante (on utilise une ε-transition qu'on peut ensuite éliminer) : on ajoute deux états q et q' initiaux et finaux, il y a une boucle $0, 1, b$ sur q, une boucle $0, 1, e$ sur q', et une transition ε de q à q', et il y a des transitions étiquetées b de q et q' vers tous les états initiaux de A et des transitions étiquetées e de tous les états finaux de A' vers q et q'.

L'idée de la construction de A est que, si on note L le langage de A, le langage de A' est le langage $L' : ((0 + 1 + e)^*(0 + 1 + b)^*bLe)^*(0 + 1 + e)^*(0 + 1 + b)^*$. Montrons la double inclusion. Pour un mot de cette forme factorisé de la façon évidente, on peut lire les termes de la forme $(0 + 1 + e)^*(0 + 1 + b)^*bLe$ en passant par q puis par q' puis en allant dans A et en en revenant, et faire le dernier terme $(0 + 1 + e)^*(0 + 1 + b)^*$ ensuite en passant par q puis q'. À l'inverse, tout mot accepté par l'automate doit passer au mieux par q puis q', puis faire un mot de la forme bLe, et ainsi de suite, donc on peut décomposer les mots acceptés suivant cette forme.

On prétend que A' est facteur-universel si et seulement si A est universel. En effet, si A' est facteur-universel, on peut y trouver n'importe quel mot bwe comme facteur avec $w \in \{0, 1\}^*$, et dans L les facteurs de la forme bwe ne peuvent s'obtenir qu'avec $w \in L$, donc L est universel. À l'inverse, montrons que, si A est universel, alors A' est facteur-universel. Montrons en fait que A' est universel et que tout mot sur Σ' peut se lire dans A' en allant de q à q'. Soit w le mot à atteindre, on fait une induction sur la longueur de w pour montrer qu'il est accepté par A'. Si aucun e ne suit un b dans w, alors w est de la forme $(0 + 1 + e)^*(0 + 1 + b)^*$ et il est dans L' et peut être lu dans A' en allant de q à q'. Sinon, écrivons $w = ubvew'$ où on distingue le premier e qui est précédé d'un b, et le b le plus proche qui précède ce e. On sait alors que u ne contient pas de e qui suit un b donc il est de la forme $(0 + 1 + e)^*(0 + 1 + b)^*$, et que v ne contient ni b ni e donc il est de la forme $\{0, 1\}^*$. On peut lire u en allant de q à q', puis lire bve en allant à q car L est universel, et ensuite lire w' par hypothèse d'induction. Ceci conclut que A' est universel, donc il est facteur-universel.

Question 10. Proposer un algorithme en temps polynomial pour déterminer si un automate déterministe est facteur-universel.

Indication : Exclure le cas d'un état à partir duquel on ne peut pas atteindre de puits.

Indication : Utiliser la notion de *mot synchronisant*, à savoir un mot qui mène à un même état q peu importe depuis où on le lit dans l'automate. On peut admettre dans un premier temps qu'on peut tester en temps polynomial si un automate déterministe admet un mot synchronisant.

Solution : Ce résultat est démontré dans [RSX12, Theorem 5]. La démonstration utilise la notion de *mots synchronisants*, dont on ne rappelle pas la définition. On présente ci-dessous la démonstration de [RSX12].

On suppose que A est émondé. Un état *universel* q de A est un état à partir duquel l'automate est complet, c'est-à-dire que tous les états accessibles à partir de q ont une transition sortante pour chaque lettre. Clairement, si A a un état universel q, alors il est facteur-universel : pour tout mot w, on peut l'obtenir en prenant u l'étiquette d'un chemin de l'état initial vers q (qui existe car A est émondé), puis on peut lire w depuis q sans sortir de l'automate, puis on peut prendre un chemin de l'état atteint vers un état final (qui existe car A est émondé).

On suppose donc dans la suite que A n'a pas d'état universel. On complète à présent A pour ajouter un unique puits q_p, toutes les transitions manquantes mènent vers le puits. Comme A n'a pas d'état universel, on sait que le puits est accessible depuis tout état de A.

Un *mot synchronisant* w de A est un mot qui aboutit au même état q quel que soit l'état depuis lequel on lit w. Si A a un mot synchronisant w, alors il n'est pas facteur-universel. En effet, comme q n'est pas universel, prenons

u un mot dont la lecture depuis q nous emmène au puits q_p. Le mot wu n'est pas facteur d'un mot accepté par A, puisque lire w nous emmène à l'état q (peu importe ce qui précède) et lire u ensuite nous fait échouer (peu importe ce qui suit).

À l'inverse, si A n'est pas facteur-universel, alors il y a un mot u qui n'est facteur d'aucun mot accepté par A. On prétend que la lecture de u depuis n'importe quel état de A nous emmène à q_p. En effet, si on suppose qu'il y a un état q depuis lequel lire u nous emmène à un état q' différent du puits, alors en prenant s un mot qui nous emmène à q (qui existe car tous les états de A sont accessibles), et t un mot qui nous emmène de q' à un état final (qui existe car tous les états de A sauf q_p sont co-accessibles), alors A accepterait sut dont u est facteur, une contradiction. Donc u nous emmène à q_p quand on le lit depuis n'importe quel état, c'est donc un mot synchronisant.

On a donc ramené le problème à déterminer si A admettait un mot synchronisant ; il reste à expliquer comment déterminer cela algorithmiquement. Or, on prétend qu'un automate admet un mot synchronisant si et seulement si chaque paire d'états est *synchronisable* en le sens suivant : q et q' sont synchronisables s'il existe un mot w tel que lire w depuis q ou depuis q' nous mène au même état. Ce résultat est démontré par exemple dans [BP12, Proposition 1], une démonstration est donnée dans le paragraphe qui suit.

Une direction de l'affirmation est facile : un mot synchronisant w témoigne du fait que chaque paire d'états est synchronisable (par w). À l'inverse, si chaque paire d'états est synchronisable, supposons par l'absurde que l'automate n'a pas de mot synchronisant. Soit S un ensemble d'états de la plus petite cardinalité possible tel qu'il existe un mot w après la lecture duquel on est sûr d'être dans un état de S peu importe de quel état on est parti : on a supposé que A n'a pas de mot synchronisant donc $|S| > 1$. Maintenant, prenons q et q' deux états distincts de S, et soit w' un mot qui les synchronise. On sait qu'après la lecture de w on est dans un état de S, et qu'après la lecture de ww' on aboutit à un état commun q'' pour q et q' et à un état pour chaque autre état de S (ces états n'étant pas forcément distincts, ni entre eux ni avec q''). Ainsi après la lecture de ww' on ne peut se trouver que dans à $|S| - 1$ états au plus ; ceci contredit la minimalité de S.

À présent, on peut déterminer si une paire d'états (q, q') est synchronisable en temps polynomial, simplement par une exploration en largeur dans l'automate produit : à partir du couple (q, q'), on détermine si on peut atteindre un état de la forme (q'', q'') pour un certain q''. Voir par exemple [Vol08].

Bibliographie

[BP12] Marie-Pierre Béal and Dominique Perrin. Synchronized automata, 2012. Notes de cours. `http://www-igm.univ-mlv.fr/~perrin/Enseignement/Master2013/CoursOption/synchronizedAutomata.pdf`.

[KS14] Prateek Karandikar and Philippe Schnoebelen. On the state complexity of closures and interiors of regular languages with subwords. In *DFCS*, 2014.

[RSX12] Narad Rampersad, Jeffrey Shallit, and Zhi Xu. The computational complexity of universality problems for prefixes, suffixes, factors, and subwords of regular languages. *Fundamenta Informaticae*, 116(1-4), 2012.

[Vol08] Mikhail V Volkov. Synchronizing automata and the Černỳ conjecture. In *ICALP*, 2008. (Malheureusement, pas de version apparemment disponible en libre accès.).