Rapport de l'épreuve orale d'informatique fondamentale

École normale supérieure
Concours MP et INFO 2017
Épreuve spécifique Ulm

Jury : Antoine Amarilli et Thibaut Balabonski

Modalités de l'épreuve. L'épreuve orale d'informatique fondamentale décrite dans ce rapport est spécifique au concours d'entrée de l'École normale supérieure de Paris, et est entièrement indépendante de l'épreuve analogue qui figure au concours des autres Écoles normales supérieures. L'épreuve dure une heure, sans préparation, et vise à interroger les candidats sur des questions d'informatique fondamentale au tableau. Elle couvre des notions d'informatique principalement théoriques, mais diffère d'une épreuve de mathématiques en cela que la vaste majorité des sujets conduit à l'étude d'un algorithme et de sa complexité. Cette épreuve ne mesure pas la compétence des candidats en informatique pratique, même si nous avons parfois demandé aux candidats de présenter certains points en pseudocode.

Cette année, les sujets étaient présentés au candidat sous forme imprimée, et quelques minutes étaient laissées au candidat pour prendre connaissance des définitions préliminaires et des premières questions. Les examinateurs ont généralement laissé les candidats traiter le début des sujets par eux-mêmes, en progressant naturellement vers un dialogue interactif pour des questions plus délicates, où quand il s'avérait nécessaire de préciser certains points des réponses proposées. Les sujets étaient toujours composés d'un unique problème formé de plusieurs questions successives ; pour les candidats qui parvenaient à la fin des questions imprimées, les questions suivantes étaient posées directement à l'oral au tableau.

L'épreuve est publique, mais il est demandé aux spectateurs de solliciter l'accord du candidat afin de ne pas le gêner. Les spectateurs doivent rester silencieux pendant l'épreuve et ne peuvent pas interférer avec son déroulement.

Résultats. Cette année, le jury a examiné 88 candidats admissibles aux concours MP et INFO. Le jury n'a pas connaissance de quel concours est présenté par les candidats qu'il auditionne, ainsi les candidats des deux concours sont-ils évalués de la même manière. Conformément aux instructions fournies pour l'harmonisation, les notes se sont réparties de 5 à 20, avec une moyenne de 12,03 et et un écart type de 3,93. Dans l'ensemble, le jury est satisfait de la performance des candidats, et les prestations auxquelles nous avons assisté étaient de niveau global plutôt homogène.

Programme. L'épreuve porte sur le programme de l'option informatique des *deux* années de classes préparatoires (MPSI et MP), ainsi que sur le programme informatique commun, et peut également faire appel à des compétences mathématiques exigibles suivant les programmes de cette discipline. Nous recommandons aux candidats de prendre connaissance de ces programmes et de s'assurer qu'ils maîtrisent effectivement les points qui y figurent. Bien entendu, les sujets proposés aux candidats leur demandaient d'explorer des notions nouvelles, qui allaient au-delà du programme, et qui étaient donc définies rigoureusement dans le sujet que nous leur proposions. Dans l'ensemble, les candidats se sont bien approprié ces notions. Les doutes des candidats à leur sujet, lorsqu'ils étaient explicitement formulés et

relevaient d'une méprise compréhensible, n'étaient généralement pas pénalisés ; nous n'avons sanctionné les candidats que quand leurs questions trahissaient une mauvaise compréhension du programme. On peut au demeurant regretter que de nombreux candidats se soient pénalisés eux-mêmes en comprenant le sujet de travers sans prendre l'initiative de communiquer leurs doutes avec l'examinateur...

Sujets Les sujets proposés cette année se sont concentrés plus particulièrement sur certains thèmes : graphes orientés et non-orientés, arbres, automates finis et langages réguliers, logique propositionnelle, probabilités et comptage.

Par souci de transparence, et pour permettre à tous les candidats de préparer cette épreuve de manière équitable, nous avons inclus, en annexe de ce rapport de concours, l'intégralité des sujets que nous avons posés aux candidats cette année [1]. Soulignons toutefois que certaines questions étaient suffisamment difficiles pour que nous ne nous attendions pas à ce que les candidats puissent les traiter sans aide, même pour les meilleurs d'entre eux ; à l'inverse, nous n'incluons pas ici certaines questions plus délicates que nous avons posées à l'oral une fois que les questions imprimées avaient été résolues.

Critères d'évaluation. Nous avons mesuré la capacité des candidats à comprendre le sujet correctement, si possible sans aide, et à se forger une intuition des notions abstraites qui y étaient présentées de manière formelle. Nous avons évalué leur réponse aux questions, notamment leur aptitude à proposer des idées de résolution, à explorer des directions prometteuses, et à réagir aux indications du jury ; mais aussi à exposer leur raisonnement de façon synthétique et compréhensible à l'oral, ou de manière plus rigoureuse à l'écrit au tableau si cela était demandé. Nous avons évalué la maîtrise du candidat des algorithmes et structures de données au programme, et leur capacité à écrire au tableau certaines routines simples en pseudocode (exploration d'arbre, etc.).

La performance des candidats, quoique homogène dans son ensemble, s'est distinguée suivant certaines dimensions indépendantes. Tous les candidats ont réussi à atteindre une compréhension satisfaisante des définitions préalables du problème étudié, mais certains y sont parvenus seuls, et d'autres ont eu besoin d'être davantage guidés. Face à des questions simples, calculatoires, ou mécaniques, certains candidats savent déterminer rapidement le bon résultat ; d'autres rechignent à faire des calculs qu'ils jugent fastidieux, ou bien cherchent des astuces parfois ingénieuses mais souvent hasardeuses. Pour des questions où nous demandions au candidat de formaliser une preuve, par exemple par récurrence, certains candidats savent articuler la preuve soigneusement à l'oral comme à l'écrit, mais d'autres, alors même qu'ils ont compris l'intuition de la question, ne parviennent pas à la formaliser de façon convaincante. Les meilleurs candidats sont ceux qui parviennent à convaincre à l'oral, de façon synthétique, qu'ils ont compris les arguments clés et qu'ils savent structurer la preuve ; et qui parviennent également à écrire rapidement et rigoureusement une preuve formelle au tableau lorsque l'examinateur en fait la demande.

Pour des questions algorithmiques, nous avons parfois demandé aux candidats d'écrire le pseudocode d'algorithmes simples, ce qui les a parfois désarçonnés. Les meilleurs candidats n'hésitent pas à poser des questions pertinentes (par exemple, comment représente-t-on les arbres, les graphes ?), et adoptent du recul sur le code qu'ils proposent (ils savent notamment reconnaître un parcours en largeur, en profondeur) ; les candidats moins bons se perdent dans ces algorithmes pourtant classiques.

Face à des questions plus ardues, on observe également une grande diversité de réactions. Bien sûr, les meilleurs candidats sont ceux qui proposent d'emblée la bonne approche, ou qui savent interpréter rapidement les indices que nous leur donnons, et parviennent ainsi à régler de telles questions avec très peu d'aide. De manière plus générale, nous avons apprécié que le candidat propose des pistes, avec un recul critique toutefois (c'est-à-dire, en sachant estimer si l'approche a des chances d'aboutir), et si possible avec vivacité et enthousiasme. À défaut d'inspiration, il est toujours préférable d'engager le dialogue avec l'examinateur, ou de proposer des exemples simples à étudier. Les candidats qui restent

1. Des propositions de corrections de ces sujets sont susceptibles d'être mises en ligne par les membres du jury, indépendamment et à titre personnel.

mutiques, et qui bloquent sans communiquer avec l'examinateur sur les difficultés qu'ils rencontrent, ont souvent reçu des notes plus basses.

Un facteur important de la variation entre les performances des candidats était lié à l'intuition que ceux-ci s'étaient formés des notions présentées dans le sujet. Souvent, ces notions étaient définies formellement, mais sans intuition : il était important de s'en former une, mais surtout de savoir en changer si elle s'avérait inadaptée pour les questions qui se posaient ensuite. Certains candidats, après un début prometteur, sont restés bloqués, apparemment parce que leur compréhension intuitive du sujet obscurcissait certains points nécessaires pour traiter les questions posées par la suite.

Recommandations aux candidats. En termes de programme, nous conseillons aux candidats de s'assurer qu'ils maîtrisent les points suivants, sur lequel nous avons identifié certaines lacunes :

- Composantes connexes d'un graphe non-orienté, composantes connexes *et* composantes fortement connexes d'un graphe orienté. Il faut savoir distinguer ces notions et les définir mathématiquement.
- Parcours d'arbres, de graphes. Une fois fixée une représentation des arbres ou des graphes, il faut savoir écrire sans erreur un pseudocode pour les explorer (en profondeur ou en largeur) et pour calculer des quantités simples (par exemple la hauteur d'un arbre).
- Il faut savoir expliquer comment un automate fini lit un mot fourni en entrée ; on s'attend notamment à ce que le candidat sache décrire un algorithme qui détermine, étant donné un mot et un automate fini, si le mot est accepté ou rejeté par l'automate. Il faut pouvoir préciser la complexité de cette tâche, dans le cas où l'automate est déterministe comme dans le cas où il ne l'est pas : cette complexité s'exprime en général en fonction du mot *et* de l'automate.
- Il ne faut pas avoir de doutes sur la complexité (en temps, en espace) de la déterminisation d'un automate fini.
- Un automate fini n'est pas nécessairement déterministe ! Trop de candidats commettent des erreurs parce qu'ils n'imaginent que des exemples d'automates déterministes.
- Algorithmes dynamiques : il faut savoir présenter l'algorithme informellement, en expliquant quelles sont les valeurs calculées, dans quel ordre elles sont calculées, et quel espace mémoire elles occupent ; il faut aussi savoir formaliser ces intuitions en pseudocode.

En termes de méthode, nous formulons les recommandations suivantes :

- Savoir adopter le bon niveau de détail : présenter informellement à l'oral les grandes lignes d'une solution, mais ne pas rechigner à écrire le détail au tableau sur demande de l'examinateur.
- Ne pas se laisser désorienter par des questions sur des points de cours ou sur des points apparemment faciles ou purement mécaniques dans le traitement d'une question. Bien souvent, les candidats qui échafaudent des arguments trop savants ou trop compliqués se retrouvent en difficulté pour répondre à des questions trop simples.
- Ne pas se bloquer sur une unique façon de voir les notions proposées dans le sujet ; essayer d'adopter une autre vision si nécessaire.
- Plutôt que de rester silencieusement bloqué sur une question, réfléchir à voix haute, et communiquer avec l'examinateur sur les difficultés rencontrées. À défaut, étudier des exemples, proposer des affaiblissements de la question, etc.
- Travailler à présenter ses idées à l'oral de façon claire et synthétique, et également au tableau de manière soignée et organisée.

Annexe. Dans la suite de ce document, nous incluons l'intégralité des sujets qui ont été posés, sous la forme des feuilles distribuées aux candidats.

A1 – Séquences d'élimination

On considère un graphe non-orienté $G = (V, E)$ où V est un ensemble non-vide de sommets et E un ensemble de paires de sommets. On appelle *séquence* une suite finie non-vide σ, et sa *longueur* est notée $|\sigma| > 0$. On rappelle qu'un *chemin* dans G est une séquence $\pi = v_1 \dots v_{|\sigma|}$ d'éléments de V telle que $\{v_i, v_{i+1}\} \in E$ pour tout $1 \leq i < |\sigma|$.

On dit que deux séquences $u_1 \dots u_n$ et $v_1 \dots v_n$ de même longueur $n \in \mathbb{N}$ se *rencontrent* s'il existe $1 \leq i \leq n$ tel que $u_i = v_i$. Une *séquence d'élimination* σ pour G est une séquence $\sigma = s_1 \dots s_{|\sigma|}$ d'éléments de V telle que, pour tout chemin π dans G tel que $|\pi| = |\sigma|$, les séquences σ et π se rencontrent.

L'objet du problème est d'étudier quels graphes admettent une séquence d'élimination, et comment ceux-ci peuvent être identifiés par un algorithme.

Question 0. Construire une séquence d'élimination pour le graphe $L_3 = (\{1, 2, 3\}, \{\{1, 2\}, \{2, 3\}\}$ représenté comme suit : $1 \text{ —— } 2 \text{ —— } 3$.

Question 1. Un chemin $\pi = v_1 \dots v_n$ dans un graphe G est dit *simple* s'il n'existe pas $1 \leq i < j < n$ tels que $\{v_i, v_{i+1}\} = \{v_j, v_{j+1}\}$. Un *cycle* dans un graphe G est un chemin simple $\pi = v_1 \dots v_n$ de longueur $n > 1$ tel que $v_1 = v_n$.

Montrer que si G a un cycle alors G n'admet pas de séquence d'élimination.

Question 2. Pour tout $k \in \mathbb{N}^*$, on note L_k le graphe $(\{1, \dots, k\}, \{\{i, i+1\} \mid 1 \leq i < k\})$. Construire une séquence d'élimination pour L_k.
Indication : On distinguera le cas où k est pair et le cas où k est impair.

Question 3. On note 2^V l'ensemble des parties de V. Une *séquence d'élimination partielle* σ pour G et pour un sous-ensemble $X \in 2^V$ est une séquence $\sigma = s_1 \dots s_{|\sigma|}$ d'éléments de V telle que, pour tout chemin π dans G tel que $|\pi| = |\sigma|$, si le dernier élément de π est dans X, alors σ et π se rencontrent.

On note $\phi_E : 2^V \to 2^V$ la fonction définie par :

$$\phi_E(X) := V \setminus \{w \mid \exists w' \in V \setminus X, \{w, w'\} \in E\}.$$

Montrer que, pour tout $X \in 2^V$, pour toute séquence d'élimination partielle σ pour G et X, pour tout $v \in V$, la séquence σ' obtenue en concaténant σ et v est une séquence d'élimination partielle pour G et $\phi_E(X) \cup \{v\}$.

Question 4. Déduire de la question précédente un algorithme pour décider, étant donné un graphe G, si ce graphe admet une séquence d'élimination, et si oui, la calculer. Discuter de l'efficacité de cet algorithme et de la longueur de la séquence obtenue.
Indication : Construire un graphe orienté sur 2^V dont les arêtes sont données par ϕ_E.

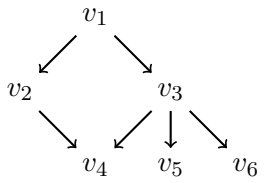
A2 – Étiquetage d'accessibilité dans les graphes

On considère un graphe orienté $G = (V, E)$ où V est un ensemble non-vide de sommets et $E \subseteq V \times V$ est un ensemble de couples de sommets. On rappelle qu'un *chemin* dans G de longueur $n \in \mathbb{N}$ est une suite finie $w_1 \dots w_n$ d'éléments de V telle que, pour tout $1 \leq i < n$, on a $(w_i, w_{i+1}) \in E$. Pour $u, v \in V$, on écrit $u \rightsquigarrow v$ quand il existe $n \in \mathbb{N}$ et un chemin $w_1 \dots w_n$ dans G tel que $u = w_1$ et $w_n = v$.

Un *étiquetage* de G est une fonction $f : V \to \mathbb{N}^2$. Étant donné une paire d'éléments $t_1 = (p_1, q_2)$ et $t_2 = (p_2, q_2)$ de $\mathbb{N}^2$, on écrit $t_1 \leq t_2$ pour indiquer que $p_1 \leq p_2$ et $q_1 \leq q_2$. Un étiquetage f est appelé *étiquetage d'accessibilité* s'il satisfait la propriété suivante : pour tous sommets $u \neq v$ de V, on a $u \rightsquigarrow v$ si et seulement si $f(u) \leq f(v)$.

L'objet du problème est d'étudier quels graphes admettent un étiquetage d'accessibilité, et comment ces graphes peuvent être identifiés et étiquetés par un algorithme.

Question 0. Construire un étiquetage d'accessibilité pour le graphe suivant :



Question 1. Rappeler la définition des composantes connexes et des composantes fortement connexes (CFC) du graphe orienté G, et la reformuler en utilisant $\rightsquigarrow$.

Question 2. On définit le *graphe des CFC* de G comme le graphe orienté $G_C = (V_C, E_C)$, où V_C est l'ensemble des CFC de G et où E_C est défini comme suit :

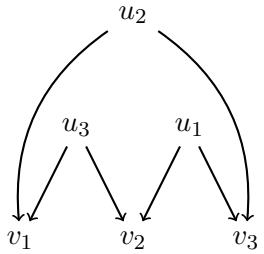
$$E_C := \{(K_1, K_2) \in V_C \times V_C \mid (K_1 \neq K_2) \wedge (\exists v_1 \in K_1, v_2 \in K_2 \text{ tels que } v_1 \rightsquigarrow v_2)\}$$

Expliquer brièvement pourquoi il n'existe pas de *cycle* de G_C, c'est-à-dire qu'il n'y a pas de chemin $w_1 \dots w_n$ de longueur $n \geq 3$ dans G_C tel que $w_1 = w_n$. On dit que G_C est *acyclique*.

Montrer que G admet un étiquetage d'accessibilité si et seulement si G_C admet un étiquetage d'accessibilité.

Question 3. Montrer que G admet un étiquetage d'accessibilité si et seulement si chacune de ses composantes connexes admet un étiquetage d'accessibilité.

Question 4. Montrer que le graphe suivant n'admet pas d'étiquetage d'accessibilité :



A3 – Automates et langages probabilistes

Pour tout ensemble non-vide X, une *distribution* sur X est une fonction π de X dans l'ensemble $\mathbb{Q}_+$ des rationnels positifs. Le *support* de π est $\mathrm{supp}(\pi) := \{x \in X \mid \pi(x) > 0\}$. On supposera toujours que le support d'une distribution est un ensemble *fini*.

Le *produit* de π par $v \in \mathbb{Q}_+$, noté $v \cdot \pi$, est la distribution sur X définie par $(v \cdot \pi)(x) := v \times \pi(x)$ pour tout $x \in X$. Pour π et π' deux distributions sur X, leur *somme*, notée $\pi + \pi'$, est la distribution sur X définie par $(\pi + \pi')(x) := \pi(x) + \pi'(x)$ pour tout $x \in X$. On dit que la distribution π est *normalisée* si on a $\sum_{x \in X} \pi(x) = 1$. On note $\Pi(X)$ l'ensemble des distributions normalisées sur X.

On fixe un alphabet Σ. Un *langage probabiliste* L est une distribution sur l'ensemble Σ^* des mots sur l'alphabet Σ. Pour $a \in \Sigma$, on note aL le langage probabiliste défini par $(aL)(w) := L(w')$ si w peut s'écrire comme aw' (c'est-à-dire que w n'est pas vide et commence par a) et $(aL)(w) := 0$ sinon. On note L_ϵ le langage probabiliste normalisé défini par $L_\epsilon(\epsilon) := 1$ et $L_\epsilon(w) := 0$ pour tout $w \neq \epsilon$.

Un *automate probabiliste* est une paire $A = (Q, \delta)$ où $Q = \{1, \ldots, n\}$ est l'ensemble d'états et δ est une fonction de Q dans $\Pi(\{\$\} \cup (\Sigma \times Q))$ telle que, pour tout $q \in \{1, \ldots, n\}$, pour tout $(a, q') \in \mathrm{supp}(\delta(q))$, on a $q' > q$.

On définit par récurrence le langage probabiliste $\mathcal{L}(A, i)$ accepté par A à l'état $i \in Q$: on a $\mathcal{L}(A, n) := L_\epsilon$, et, pour $1 \leq i < n$, en notant $\pi := \delta(i)$, on a :

$$\mathcal{L}(A, i) := \pi(\$) \cdot L_\epsilon + \sum_{\substack{a \in \Sigma \\ j \in Q}} \pi(a, j) \cdot (a\mathcal{L}(A, j))$$

Le *langage* $\mathcal{L}(A)$ accepté par A est le langage probabiliste $\mathcal{L}(A, 1)$.

Question 0. Construire un automate probabiliste A tel que $\mathcal{L}(A)$ soit le langage probabiliste normalisé L sur $\{a, b\}$ défini par :

- ϵ a probabilité 0.1 dans L ;
- a a probabilité 0.2 dans L ;
- abb a probabilité 0.3 dans L ;
- b a probabilité 0.4 dans L.

Question 1. Montrer que, pour tout automate probabiliste A, le langage probabiliste $\mathcal{L}(A)$ est normalisé.

Question 2. Montrer que, réciproquement, pour tout langage probabiliste normalisé L, on peut construire un automate probabiliste A_L tel que $\mathcal{L}(A_L) = L$.

Question 3. Un automate probabiliste $A = (Q, \delta)$ est dit *déterministe* si, pour tout $q \in Q$, pour chaque $a \in \Sigma$, il existe au plus un unique $q' \in Q$ tel que $(a, q') \in \mathrm{supp}(\delta(q))$. Montrer que, pour tout langage probabiliste normalisé L, on peut construire un automate probabiliste déterministe A'_L tel que $\mathcal{L}(A'_L) = L$.

Indication : On pourra décomposer L en le mot vide et en les sous-langages, pour $a \in L$, des mots de $\mathrm{supp}(L)$ qui commencent par a.

A4 – Ensembles semilinéaires

Un ensemble $S \subseteq \mathbb{N}$ est dit *linéaire* s'il existe $b \in \mathbb{N}$, $n \in \mathbb{N}$, et $v_1, \ldots, v_n \in \mathbb{N}^*$ tels que :

$$S = \{b + \sum_{1 \leq i \leq n} x_i v_i \mid x_1, \ldots, x_n \in \mathbb{N}\}.$$

Un ensemble $S \subseteq \mathbb{N}$ est dit *semilinéaire* s'il existe $m \in \mathbb{N}$ et des ensembles linéaires $S_1, \ldots, S_m$ tels que $S = \bigcup_{1 \leq i \leq m} S_i$.

Question 0. Expliquer pourquoi tout sous-ensemble fini de $\mathbb{N}$ est semilinéaire.

Question 1. Donner un exemple de sous-ensemble de $\mathbb{N}$ qui n'est *pas* semilinéaire.

Question 2. On dit qu'un ensemble $S \subseteq \mathbb{N}$ est *ultimement périodique* s'il existe $t \in \mathbb{N}$ et $p \in \mathbb{N}^*$ tels que, pour tout $s \geq t$, on a $s \in S$ si et seulement si $s + p \in S$. Montrer que tout ensemble ultimement périodique est semilinéaire.

Question 3. Montrer que, réciproquement, tout ensemble semilinéaire est ultimement périodique. **Indication :** On pourra montrer que la définition de l'ultime périodicité peut se reformuler avec une implication plutôt qu'une équivalence.

Question 4. Un ensemble linéaire est dit *unaire* si on a $n = 1$, et *nullaire* si on a $n = 0$. Déduire des questions précédentes que tout ensemble semilinéaire peut s'écrire comme une union finie d'ensembles linéaires unaires et d'ensembles linéaires nullaires. Proposer un algorithme qui calcule cette représentation.

A5 – Formules de provenance

Pour tout ensemble fini non-vide de *variables* $\mathcal{X} = \{X_1, \ldots, X_m\}$, on appelle *formule sur* $\mathcal{X}$ une formule de la logique propositionnelle ayant $\mathcal{X}$ comme ensemble de variables.

Question 0. Rappeler la définition d'une formule de la logique propositionnelle.

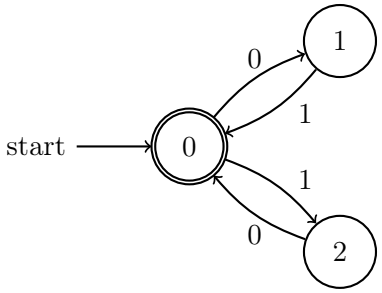
Une *valuation* de $\mathcal{X}$ est une fonction $\nu : \mathcal{X} \to \{0, 1\}$ qui attribue une valeur booléenne à chaque variable de $\mathcal{X}$. Étant donné une valuation ν, la *valeur* sous ν d'une formule Φ sur $\mathcal{X}$ est une valeur dans $\{0, 1\}$ que l'on écrit $\nu(\Phi)$ et que l'on définit comme en cours.

On fixe l'alphabet $\Sigma = \{0, 1, ?\}$. Étant donné un mot $w = a_1 \cdots a_n$ de longueur n sur l'alphabet Σ, son *ensemble de variables* est $\mathcal{X}_n = \{X_1, \ldots, X_n\}$. Une *valuation* de w est une valuation de $\mathcal{X}_n$. Chaque valuation ν de w permet de définir un mot $\nu(w) = a'_1 \cdots a'_n$ sur l'alphabet $\{0, 1\}$ comme suit :

$$\text{pour tout } 1 \leq i \leq n, \text{ on a } a'_i := \begin{cases} a_i & \text{si } a_i \in \{0, 1\} \\ \nu(X_i) & \text{si } a_i = ? \end{cases}$$

Étant donné un mot w de longueur n sur l'alphabet Σ et un automate A sur l'alphabet $\{0, 1\}$, une *formule de provenance pour w sur A* est une formule Φ sur $\mathcal{X}_n$ telle que, pour toute valuation ν de w, on a $\nu(\Phi) = 1$ si et seulement si A accepte $\nu(w)$.

Question 1. Calculer une formule de provenance pour le mot $w = 0??1??$ sur l'automate suivant :



Question 2. Proposer un algorithme naïf qui, étant donné un mot $w \in \Sigma^*$ et un automate A, calcule une formule de provenance pour w sur A. Discuter de l'efficacité de l'algorithme proposé.

Question 3. Étant donné un mot $w \in \Sigma^*$ de longueur n et un automate A ayant Q comme ensemble d'états, pour $q \in Q$ et pour $0 \leq i \leq n$, une *formule partielle de provenance pour w sur A en (i, q)* est une formule $\Phi_{i,q}$ sur $\mathcal{X}_i = \{X_1, \ldots, X_i\}$ telle que, pour toute valuation ν de $\mathcal{X}_i$, on a $\nu(\Phi_{i,q}) = 1$ si et seulement si A peut aboutir à l'état q depuis l'état initial en lisant le préfixe de longueur i de $\nu(w)$.

Pour $0 \leq i < n$ et $q \in Q$, proposer une expression de $\Phi_{i+1,q}$ en fonction des $\Phi_{i,q'}$ pour $q' \in Q$. **Indication :** On distinguera plusieurs cas selon la valeur de a_{i+1}.

Question 4. Déduire de la question précédente un algorithme plus sophistiqué qui, étant donné un mot $w \in \Sigma^*$ et un automate A, calcule une formule de provenance pour w sur A. Discuter de l'efficacité de cet algorithme. Comment pourrait-on rendre l'algorithme plus efficace en modifiant la représentation des formules ?

A6 – Clôture commutative de langages réguliers

On fixe l'alphabet $\Sigma = \{a, b\}$, et on note Σ^* l'ensemble des mots sur Σ. Pour $w \in \Sigma^*$, on note respectivement $|w|_a$ et $|w|_b$ le nombre d'occurrences de a et de b dans w. Un *langage* sur Σ^* est un sous-ensemble non nécessairement fini de Σ^*. La *clôture commutative* d'un langage L sur l'alphabet Σ est le langage $\mathrm{CCl}(L)$ des mots $w \in \Sigma^*$ tels qu'il existe $w' \in L$ avec $|w|_a = |w'|_a$ et $|w|_b = |w'|_b$.

On admettra le théorème de Kleene : un langage peut être décrit par une expression rationnelle si et seulement s'il est reconnu par un automate fini. De tels langages sont appelés *réguliers*. L'objet du problème est d'étudier la clôture commutative de langages réguliers.

Question 0. Calculer la clôture commutative du langage régulier L_0 défini par l'expression rationnelle $a^* + b^*$.

Question 1. Soit L_1 le langage régulier défini par l'expression rationnelle $b(aa)^*$. Proposer une expression rationnelle pour $\mathrm{CCl}(L_1)$. Existe-t-il un langage régulier L_1' tel que $\mathrm{CCl}(L_1') = L_1$?

Question 2. Exhiber un langage régulier L_2 tel que $\mathrm{CCl}(L_2)$ soit le langage $L_2' := \{w \in \Sigma^* \mid |w|_a = |w|_b\}$. En déduire que la clôture commutative d'un langage régulier n'est pas nécessairement un langage régulier.

Question 3. On dit qu'un langage L est *a-borné* s'il existe $k \in \mathbb{N}$ tel que, pour tout $w \in L$, on a $|w|_a \leq k$. Montrer que, pour tout langage régulier L, si L est a-borné, alors $\mathrm{CCl}(L)$ est régulier.

Étendre l'argument aux langages réguliers *ab-bornés*, c'est-à-dire les langages réguliers L où il existe $k \in \mathbb{N}$ tel que, pour tout $w \in L$, on a soit $|w|_a \leq k$, soit $|w|_b \leq k$.

Question 4. Proposer un algorithme qui décide, étant donné un langage régulier L sur Σ, si L est a-borné. Proposer également un algorithme qui décide si L est ab-borné.

A7 – Allocation de candidats à des écoles

Dans ce problème, on considère un ensemble fini d'*écoles d'ingénieur* $\mathcal{E} = E_1, \ldots, E_n$, et un ensemble fini de *candidats* $\mathcal{C}$. Pour $1 \leq i \leq n$, chaque école a un ensemble $A_i \subseteq \mathcal{C}$ de candidats *sur liste principale*, un ordre total $<_i$ sur A_i (qui trie les candidats du meilleur au moins bon pour cet école), un ensemble $A'_i \subseteq \mathcal{C}$ disjoint de A_i de candidats *sur liste complémentaire*, et un ordre total $<'_i$ sur A_i. On s'intéresse à un candidat $c \in \mathcal{C}$, qui accepterait d'intégrer n'importe laquelle des écoles de $\mathcal{E}$. On cherche à déterminer s'il est certain que c pourra être admis dans une école, quels que soient les choix des autres candidats, sachant que :

— chaque candidat ne peut accepter qu'une seule école ;
— chaque école ne peut admettre que des candidats figurant sur l'une de ses listes ;
— chaque école E_i peut admettre un nombre de candidats égal au plus à la longueur $|A_i|$ de sa liste principale ;
— chaque école admet en priorité les candidats les mieux classés dans ses listes principale et complémentaire.

Question 0. Donner un exemple de situation où c ne figure sur aucune liste principale mais où il est certain que c pourra être admis dans une école.

Question 1. En simplifiant les données introduites dans l'énoncé, proposer une formalisation de ce problème.

Question 2. On suppose qu'il n'y a que deux écoles, c'est-à-dire $n = 2$. Proposer un algorithme simple pour résoudre efficacement le problème.

Question 3. On suppose à présent que chaque candidat excepté c apparaît sur la liste de deux écoles au plus, et que que chaque école peut admettre exactement un candidat.

Expliquer comment on peut construire efficacement une formule de la logique propositionnelle qui est satisfiable si et seulement s'il est possible que c ne soit admis dans aucune école. On construira une formule en *forme normale conjonctive* (CNF), c'est-à-dire une conjonction de *clauses*, où chaque clause est une disjonction de littéraux.

Question 4. Soit $F = C_1 \wedge \cdots \wedge C_n \wedge (x \vee C) \wedge (\neg x \vee C')$ une formule en CNF, où les occurrences de la variable x sont exactement comme indiqué. Montrer que F est satisfiable si et seulement si la formule $C_1 \wedge \cdots \wedge C_n \wedge (C \vee C')$ est satisfiable.

Question 5. En observant la forme de la CNF obtenue en question 3, déduire un algorithme pour résoudre le problème sous les hypothèses de la question 3. Discuter de l'efficacité de cet algorithme.

A8 – Automates à pile et lemme de pompage

Soit Σ un alphabet fini, et Γ un autre alphabet fini appelé *alphabet de pile*. Un *automate à pile* sur Σ est un quintuplet $A = (Q, q_0, \gamma_0, \delta, F)$, où Q est un ensemble fini d'*états*, $q_0 \in Q$ est l'*état initial*, $\gamma_0 \in \Gamma$ est le *symbole de pile initial*, δ est une *fonction de transition* de $Q \times \Sigma \times \Gamma$ vers l'ensemble des parties de $Q \times \Gamma^*$, et $F \subseteq Q$ est un ensemble d'états *finaux*.

Une *configuration* de A est un couple (q, z) où $q \in Q$ est l'*état* et où z est un mot non-vide sur Γ appelé *pile*. La *configuration initiale* est (q_0, γ_0), et une configuration (q, z) est *acceptante* si $q \in F$. Lorsque l'automate est dans une configuration (q, z) et lit une lettre $a \in \Sigma$, il décompose $z = z'\gamma$ avec $\gamma \in \Gamma$ le *sommet de pile*, il choisit un $(q', g) \in \delta(q, a_{i+1}, \gamma)$ tel que $z'g$ soit non-vide, et il aboutit à la configuration $(q', z'g)$. L'automate à pile A *accepte* un mot de Σ^* s'il peut lire ses lettres dans l'ordre à partir de la configuration initiale pour parvenir à une configuration acceptante suivant ces règles.

Question 0. Étant donné un automate A sur Σ sans pile, expliquer comment construire un automate à pile A' sur Σ qui reconnaît le même langage que A.

Question 1. On prend dans cette question $\Sigma = \{a, b\}$. Proposer un automate à pile qui reconnaît le langage $\{a^n b^n \mid n \geq 2\}$. Qu'en déduire ?

Question 2. On prend toujours $\Sigma = \{a, b\}$. Proposer un automate à pile qui reconnaît le langage $\{w \in \Sigma^* \mid |w|_a = |w|_b\}$, où $|w|_a$ et $|w|_b$ désignent respectivement le nombre de a et de b de w.

A9 – Automates pour les valuations de formules booléennes

Soit $\mathcal{X}$ un ensemble fini de variables. Une *valuation* de $\mathcal{X}$ est une fonction de $\mathcal{X}$ dans $\{0,1\}$. Soit Φ une formule de la logique propositionnelle sur $\mathcal{X}$. On dit que la valuation ν *satisfait* Φ si la formule Φ s'évalue à 1 lorsque l'on remplace chaque variable dans Φ par son image par ν.

On fixe l'alphabet $\Sigma = \{0,1\}$. Soit $<$ un ordre total sur $\mathcal{X}$: on écrira en conséquence $\mathcal{X} = x_1, \ldots, x_n$, avec $x_i < x_j$ pour tout $1 \leq i < j \leq n$. Le *mot suivant* $<$ d'une valuation ν de $\mathcal{X}$ est le mot $\nu(x_1) \cdots \nu(x_n)$ de longueur n sur l'alphabet Σ. Un *automate de valuations* pour Φ et $<$ est un automate A sur l'alphabet Σ tel que, pour tout mot $w \in \Sigma^*$, l'automate A accepte w si et seulement si $|w| = n$ et w est le mot suivant $<$ d'une valuation ν qui satisfait Φ.

Question 0. Construire un automate de valuations pour la formule $(x_1 \wedge x_3) \vee x_2$.

Question 1. Proposer un algorithme naïf qui, étant donné une formule Φ de la logique propositionnelle, construit un automate de valuations pour Φ. Discuter de la complexité de cet algorithme.

Dans les deux prochaines questions, on cherche à améliorer l'efficacité de cet algorithme.

Question 2. Pour tout $n \in \mathbb{N}$, on appelle L_n le langage sur Σ défini par $L_n := \{ww \mid w \in \Sigma^n\}$. Montrer que, pour tout $n \in \mathbb{N}$, pour tout automate A qui reconnaît L_n, l'automate A a au moins 2^n états.

Question 3. En utilisant la question précédente, montrer que, pour tout ensemble de variables totalement ordonné de taille paire $\mathcal{X} = x_1, \ldots, x_{2n}$, on peut construire une formule Φ_{2n} de taille $O(n)$ telle que tout automate de valuations pour Φ_{2n} et $<$ ait au moins 2^n états.

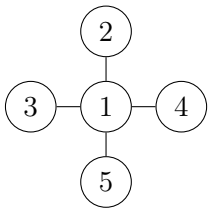
Qu'en déduire quant à l'algorithme de la question 1 ?

A10 – Ampleur de tris sur des graphes

Dans ce problème, on appelle *graphe* un graphe non-orienté et non-vide. Soit $G = (V, E)$ un graphe, et posons $n := |V|$ son nombre de sommets. Pour toute partition de V en deux ensembles disjoints X et Y tels que $X \cup Y = V$, la *frontière* de (X, Y) est le sous-ensemble de E défini par $\mathcal{F}_G(X, Y) := \{\{u, v\} \in E \mid (u, v) \in (X \times Y) \cup (Y \times X)\}$. Pour toute bijection $\sigma : \{1, \ldots, n\} \to V$, on appelle *tri* de G la séquence $S = \sigma(1), \ldots, \sigma(n)$, et pour tout $1 \leq i < n$ on définit $S_{\leq i} := \{\sigma(1), \ldots, \sigma(i)\}$ et $S_{>i} := \{\sigma(i+1), \ldots, \sigma(n)\}$. L'*ampleur* du tri S est :

$$\mathcal{A}_G(S) := \max_{1 \leq i < n} |\mathcal{F}_G(S_{\leq i}, S_{>i})|$$

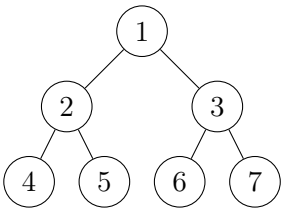
Question 0. Montrer que, pour le graphe G suivant, tout tri de G est d'ampleur au moins 2 :



Question 1. Montrer que, pour tout $n \in \mathbb{N}^*$, il existe un graphe B_n à $n+1$ sommets et n arêtes tel que tout tri de B_n est d'ampleur au moins $\lceil n/2 \rceil$.

Question 2. Montrer que, pour tout $n \in \mathbb{N}^*$, il existe un graphe C_n à n sommets tel que tout tri de C_n est d'ampleur $\lfloor n/2 \rfloor \times \lceil n/2 \rceil$.

Question 3. Pour tout $n \in \mathbb{N}^*$, on appelle A_n l'arbre binaire complet de hauteur n, vu comme un graphe. Par exemple, A_2 est comme suit :

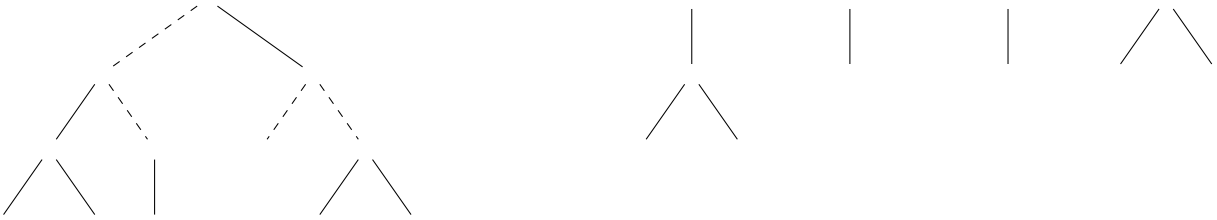


Pour tout $n \in \mathbb{N}^*$, construire un tri de A_n d'ampleur $2^{n+1} - 2$.

Question 4. Construire un autre tri de A_n d'ampleur n.

A11 – Comptage de sous-arbres

On considère dans ce sujet des arbres enracinés binaires (chaque nœud a au plus 2 enfants). Étant donné un arbre T, tout sous-ensemble S des arêtes de T définit une forêt, notée $S(T)$, obtenue en conservant les arêtes de T présentes dans S et en retirant les autres (on retire également les nœuds qui n'ont plus d'arêtes incidentes). Par exemple, pour l'arbre représenté ci-dessous à gauche, pour S contenant toutes les arêtes sauf celles en pointillés, on obtient la forêt représentée à droite :



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

Question 0. Pour l'arbre d'exemple T et la forêt d'exemple F ci-dessus, combien de sous-ensembles d'arêtes de T permettent d'obtenir la même forêt que F ? Combien y a-t-il de sous-ensembles S d'arêtes de T telles que $S(T)$ consiste exactement en deux arêtes isolées ?

Question 1. On cherche à déterminer, étant donné un arbre T et une hauteur $h \in \mathbb{N}^*$, le nombre de sous-ensembles S de T tels que $S(T)$ contienne un arbre de hauteur au moins h. Proposer un algorithme naïf pour cette tâche, et discuter de sa complexité en fonction de T et de h.

Question 2. Proposer un algorithme plus efficace pour cette tâche à base de programmation dynamique, et discuter de sa complexité en fonction de T et de h.

Question 3. Comment peut-on compter le nombre de sous-ensembles S tels que $S(T)$ contienne un arbre de hauteur *exactement* h ?

Question 4. Le *diamètre* d'une forêt F est le plus grand entier $d \in \mathbb{N}^*$ tel qu'il existe un chemin simple non-orienté de longueur d dans F. Proposer un algorithme qui, étant donné T et d, calcule le nombre de sous-ensembles S de T tels que $S(T)$ soit de diamètre au moins d.

B1 – Unification

Soit un alphabet Γ dont chaque lettre α est associée à un nombre entier `arité(`α`)` positif ou nul. On fixe également un ensemble $\mathcal{V}$ disjoint de Γ de symboles appelés *variables*. Pour tout $x \in \mathcal{V}$ on fixe `arité(`x`)`$= 0$.

Un *terme ouvert* sur l'alphabet Γ est un arbre fini dont chaque nœud est étiqueté par une lettre $\alpha \in \Gamma$ ou une variable $x \in \mathcal{V}$, dans lequel un nœud étiqueté par une lettre $\alpha \in \Gamma$ possède exactement `arité(`α`)` fils et chaque nœud étiqueté par une variable est une feuille. Le terme formé par un nœud α d'arité k ayant pour fils les termes t_1 à t_k est noté $\alpha(t_1, ..., t_k)$, ou juste α si $k = 0$.

Une *substitution* σ est une fonction de $\mathcal{V}$ vers les termes sur Γ. L'application $\sigma(t)$ d'une substitution σ à un terme t remplace chaque nœud étiqueté par une variable x par l'arbre $\sigma(x)$.

On veut *résoudre* un système d'équations de la forme $\{t_1 = u_1, ..., t_n = u_n\}$, c'est-à-dire qu'on cherche une substitution σ telle que pour tout i, $\sigma(t_i) = \sigma(u_i)$.

Question 0. On se donne l'alphabet $\{i, *\}$ avec `arité(`i`)`$= 1$ et `arité(`$*$`)`$= 2$. On prend un ensemble de variables $\mathcal{V} = \{x, y, z, ...\}$.

Peut-on résoudre le système $\{*(i(x), x) = *(y, *(z, u))\}$? Et le système $\{*(i(x), x) = *(y, y)\}$?

Question 1. Une équation dans un système S est *résolue* si elle est de la forme $x = t$ ou $t = x$ avec t un terme quelconque et x une variable qui n'a pas d'autre occurrence dans S (et en particulier pas dans t). Un système est *résolu* si toutes ses équations le sont.

Étant donné un système résolu S, définir une substitution σ_S qui résout effectivement S.

Question 2. On étudie l'algorithme de résolution consistant à appliquer autant que possible les trois règles suivantes aux équations non résolues :

- Si une équation a la forme $t = t$, la supprimer.
- Si une équation a la forme $\alpha(t_1, ..., t_n) = \alpha(u_1, ..., u_n)$, la remplacer par les n équations $t_i = u_i$.
- Si une équation a la forme $x = t$ ou $t = x$ et x n'apparaît pas dans t, alors appliquer à toutes les autres équations du système la substitution $\{x \mapsto t\}$.

Expliciter les différentes conditions auxquelles il n'est pas possible d'appliquer l'une de ces trois règles. Préciser quelles conditions correspondent au succès ou à l'échec de la résolution.

Question 3. Montrer la correction de l'algorithme : si, partant d'un système S, il produit un système résolu S', alors la substitution $\sigma_{S'}$ correspondant à S' résout le système d'origine S.

Échantillon de questions supplémentaires

Question 4. Montrer que l'algorithme termine : quel que soit le système d'équations fini de départ, les règles ne peuvent être appliquées qu'un nombre fini de fois.
Indication : on pourra d'abord montrer que l'ordre lexicographique $\preceq_n$ sur $\mathbb{N}^n$ n'admet pas de séquence infinie strictement décroissante.

Question 5. Montrer la complétude de l'algorithme : s'il existe une substitution résolvant le système de départ, alors l'algorithme aboutira à un système résolu.

Les questions suivantes concernaient la complexité de cet algorithme et la mise en place d'un algorithme amélioré représentant les termes à unifier par un graphe (algorithme de Robinson amélioré).

B2 – Allocation dynamique de mémoire

Étant donné un tableau d'entiers T de longueur n, et un entier $0 \leq i < n$, on note $T[i]$ le contenu de la case d'indice i de T. Avec de plus $i < j \leq n$, on désigne par $T[i..j[$ le segment du tableau compris entre les indices i inclus et j exclu. La *taille* de ce segment est $j - i$. Étant donné un entier naturel non nul a, un *a-segment* est un segment dont la taille est un multiple de a, et une *a-partition* de T est une suite de k a-segments $T[i_0..i_1[$, $T[i_1..i_2[$, ..., $T[i_{k-1}..i_k[$ avec $i_0 = 0$ et $i_k = n$.

On s'intéresse à un tableau a-partitionné en segments, dans lequel chaque segment est soit *libre* soit *réservé*, et on veut fournir les deux opérations suivantes :

- `réserver(`n`)`, en supposant que n est un multiple de a, trouve un segment libre de taille m supérieure ou égale à n et le découpe en, dans l'ordre : un segment de taille n réservé et un segment de taille $m - n$ libre. Cette opération renvoie comme résultat l'indice de début i du segment qui a été réservé. L'opération échoue si aucun segment de taille suffisante n'est trouvé.
- `libérer(`i`)` s'applique uniquement si l'indice i est l'indice de début d'un segment réservé. Ce segment est alors libéré.

Question 0. On prend $n = 12$ et les trois séquences suivantes :

s_1 = `réserver(8)`	s_1 = `réserver(4)`	s_1 = `réserver(8)`
s_2 = `réserver(4)`	s_2 = `réserver(4)`	`libérer(`s_1`)`
`libérer(`s_1`)`	`libérer(`s_1`)`	s_2 = `réserver(4)`
`libérer(`s_2`)`	`libérer(`s_2`)`	`libérer(`s_2`)`
s_3 = `réserver(8)`	s_3 = `réserver(8)`	s_3 = `réserver(8)`

L'une de ces séquences réussit-elle systématiquement ? L'une de ces séquences échoue-t-elle systématiquement ? Commenter.

Question 1. On suppose que les segments libres sont reliés par une structure de données semblable à une liste chaînée (chaque élément dispose d'un pointeur vers son successeur), avec les deux caractéristiques supplémentaires suivantes : chaque élément dispose également d'un pointeur vers son prédécesseur, et on a un pointeur successeur du dernier vers le premier élément (et un prédécesseur correspondant). On suppose également, pour chaque segment, être capable d'obtenir sa taille et son statut libre ou réservé.

Proposer des algorithmes pour les opérations `réserver` et `libérer`. Quelle est la complexité de ces opérations, et de quoi dépend-elle ?

Question 2. On veut maintenant réduire au maximum les structures de données annexes, en codant toutes les informations de taille, toutes les informations de liberté, et le plus possible d'informations de L dans le tableau T lui-même. Pour cela on se donne le droit d'utiliser la première case des segments réservés, et l'intégralité des segments libres, et on se limite à des a-segments pour un a à déterminer.

Proposer une manière de faire, et préciser un a correspondant le plus petit possible ainsi que les informations annexes qui restent en dehors de T.

Échantillon de questions supplémentaires

Question 3. L'objectif de cette question est de régler le problème de la deuxième séquence exemple, en fusionnant systématiquement les segments libres.

Comment adapter le codage de T et les opérations `réserver` et `libérer` pour faire cela avec un surcoût de temps et d'espace minimal ? Préciser et justifier l'invariant sur les segments libres préservé par ces algorithmes.

Les questions suivantes concernaient le système des segments frères ("buddy systems"), basé sur des blocs dont la taille est une puissance de 2, et une estimation de la fragmentation induite par cette contrainte.

B3 – Sélection d'instructions

Soit un alphabet Γ dont chaque lettre α est associée à un nombre entier `arité(`α`)` positif ou nul. Un *terme* sur l'alphabet Γ est un arbre dont chaque nœud est étiqueté par une lettre α et possède un nombre de fils égal à `arité(`α`)`. Le terme formé par un nœud α d'arité k ayant pour fils les termes t_1 à t_k est noté $\alpha(t_1, ..., t_k)$, ou juste α si $k = 0$.

On appelle *tuile* sur l'alphabet Γ un terme sur l'alphabet $\Gamma \cup \{\bot\}$, où la lettre $\bot$ est telle que `arité(`$\bot$`)` $= 0$. On définit l'*arité* d'une tuile t comme le nombre d'occurrences de $\bot$ dans t. Étant donnée une tuile t d'arité k et k tuiles u_1, ..., u_k, on note $t[u_1, ..., u_k]$ la tuile obtenue en remplaçant les k occurrences de $\bot$ dans t par les u_i, de gauche à droite. Étant donné un ensemble Θ de tuiles sur Γ, un terme c sur Θ peut être *aplati* en un terme $[\![c]\!]$ sur Γ, en remplaçant chaque $t(u_1, ..., u_k)$ par $t[[\![u_1]\!], ..., [\![u_k]\!]]$. On appelle c une Θ-*couverture* de $[\![c]\!]$. La *taille* d'une Θ-couverture est son nombre de nœuds.

Étant donné un ensemble de tuiles Θ et un terme a, une Θ-couverture de a est *localement optimale* s'il n'est pas possible de fusionner des tuiles adjacentes pour former une nouvelle tuile de Θ. Une couverture est *globalement optimale* si elle utilise un nombre de tuiles minimal.

Question 0. On se donne l'alphabet $\Gamma = \{m, a, p, c\}$ où les lettres ont dans l'ordre les arités 2, 1, 2 et 0, ainsi que l'ensemble de tuiles $\Theta = \{m(a(p(\bot, c)), \bot),\ m(a(\bot), a(\bot)),\ p(\bot, c),\ a(c),\ c\}$ et le terme $t = m(a(p(c, c)), a(c))$.

Donner pour t une couverture globalement optimale, et une couverture localement optimale mais pas globalement optimale.

Question 1. On considère l'algorithme de recherche de couverture qui commence par placer à la racine la plus grande tuile t compatible (celle contenant le plus grand nombre de nœuds non étiquetés par $\bot$), puis s'appelle récursivement sur les sous-arbres définis par les feuilles de t.

Donner un ensemble de tuiles Θ et un arbre a admettant une Θ-couverture, mais pour lequel l'algorithme échoue.

Question 2. On appelle *tuile élémentaire* une tuile avec un seul nœud différent de $\bot$. Pour assurer que l'algorithme précédent trouve toujours une couverture, on suppose que l'ensemble Θ contient toutes les tuiles élémentaires.

Démontrer que dans ce cas, l'algorithme de la question précédente réussit et produit toujours une couverture localement optimale.

Échantillon de questions supplémentaires

Question 3. Donner un ensemble Θ et un terme t pour lequel la solution donnée par l'algorithme n'est pas globalement optimale, et évaluer le rapport maximal entre la taille de la couverture donnée par l'algorithme et la taille d'une couverture globalement optimale.

Question 4. Estimer la complexité de l'algorithme, en fonction des différents paramètres du problème.

Question 5. Proposer un algorithme construisant une couverture globalement optimale d'un terme a, en utilisant une quantité de mémoire essentiellement proportionnelle à la taille de a. Justifier avec soin la correction de l'algorithme et évaluer sa complexité en temps et en espace.

La suite des questions consistait à établir des conditions auxquelles l'algorithme glouton proposé au début donne à coup sûr une couverture globalement optimale.

B4 – Permutations

Une *permutation* d'ordre n est une bijection de l'ensemble $\{1, ..., n\}$ dans lui-même. Une permutation σ d'ordre n peut être représentée comme un mot $a_1...a_n$ avec pour tout i, $a_i = \sigma(i)$.

Une *inversion* de σ est une paire i, j telle que $i < j$ et $\sigma(j) < \sigma(i)$. La *table des inversions* de σ est la séquence $\iota(\sigma) = b_1...b_n$ telle que pour tout k, b_k est le nombre d'inversions (i, j) de σ telles que $\sigma(j) = k$.

Question 0. Donner la table des inversions de la permutation 15342. Donner une permutation admettant 31200 comme table des inversions.

Question 1. Étant donnée une permutation σ et sa table des inversions $\iota(\sigma) = b_1...b_n$, donner des encadrements pour les b_k.

Question 2. Considérons un mot $a_1...a_n$ décrivant une permutation σ, et appliquons lui le traitement suivant :

— Pour i de 1 à $n - 1$, si $a_i > a_{i+1}$ alors permuter a_i et a_{i+1}.

Comment évolue la table des inversions de σ ?

Question 3. Soit une séquence d'entiers $b_1, \ldots, b_n$ vérifiant les encadrements de la question 1. On peut construire une permutation ayant $b_1 \ldots b_n$ comme table des inversions en plaçant d'abord n, puis en insérant $n - 1$ en fonction de b_{n-1}, puis $n - 2$ en fonction de b_{n-2}, et ainsi jusqu'à 1.

Préciser l'algorithme correspondant et estimer sa complexité.

Échantillon de questions supplémentaires

Question 4. Proposer un algorithme de complexité $O(n \log n)$ pour la construction de la permutation σ correspondant à une table des inversions τ.
Indication : Si on place les éléments dans l'ordre de 1 à n, où placer i quand arrive son tour ?
Indication : Un arbre binaire équilibré à n feuilles a une hauteur proportionnelle à $\log n$.

Question 5. Proposer un algorithme de complexité $O(n \log n)$ pour la construction de la table des inversions d'une permutation σ.
Indication : Vous connaissez déjà des liens entre $O(n \log n)$ et une notion de permutation.

La suite des questions consistait à établir un encadrement liant le nombre d'inversions d'une permutation à une quantité calculable en temps linéaire (la mesure de Spearman).

B5 – Codes convolutifs

Étant donné un alphabet Γ, on note Γ^* l'ensemble des mots sur Γ et Γ^k l'ensemble des mots sur Γ de longueur k. Un mot m est un *facteur* d'un mot w s'il existe deux mots u, v tels que $w = umv$.

Soient deux alphabets Γ et Σ. Une *k-traduction* f est une fonction $\Gamma^* \longrightarrow \Sigma^*$ définie à partir d'une fonction $\varphi : \Gamma^{k+1} \longrightarrow \Sigma$, et de k lettres $e_{-k}, \ldots, e_{-1} \in \Gamma$ par $f(e_0 \ldots e_n) = s_0 \ldots s_n$ avec pour tout i, $s_i = \varphi(e_{i-k} \ldots e_i)$.

Question 0. Considérons la 2-traduction f définie par : $\Gamma = \{0, 1\}$, $\Sigma = \{a, b, c, d\}$, $k = 1$, $e_{-2} = e_{-1} = 0$, et φ définie par le tableau suivant :

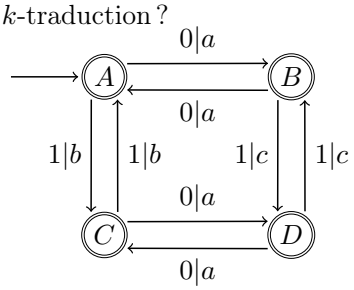
000	a	100	c
001	c	101	a
010	d	110	b
011	b	111	d

Traduire le mot 10110. Identifier parmi les mots suivants ceux qui sont dans l'image de f ou sont un facteur d'un mot de l'image de f : adc, acd, adb.

On appelle *transducteur* un automate fini A doté d'un alphabet d'entrée Γ et d'un alphabet de sortie Σ, et dans lequel chaque transition est étiquetée par une lettre de Γ et une lettre de Σ, appelées respectivement étiquette d'entrée et étiquette de sortie. L'automate d'entrée $E(A)$ (resp. de sortie $S(A)$) est l'automate obtenu en ne conservant que les étiquettes d'entrée (resp. de sortie).

Un transducteur A définit une relation $\mathcal{R}_A$ entre Γ^* et Σ^*, telle que $m_e \mathcal{R}_A m_s$ si et seulement si m_e et m_s sont le mot d'entrée et le mot de sortie d'un même chemin acceptant de A.

Question 1. Étant donnée une k-traduction, expliquer comment construire un transducteur correspondant. Le transducteur suivant définit-il une k-traduction ?



Question 2. Donner des conditions suffisantes sur les automates d'entrée et/ou de sortie d'un transducteur A pour que $\mathcal{R}_A$ définisse :

— une fonction
— une fonction totale
— une fonction injective

Sous les conditions d'une fonction totale injective, donner un algorithme de décodage qui, à partir d'un mot m_s, renvoie son unique antécédent par f ou échoue si m_s n'est pas dans l'image de f. Préciser sa complexité.

On généralise la notion d'antécédent de la manière suivante : un mot m_e est un *antécédent généralisé* d'un mot m_s s'il existe un chemin étiqueté par $m_e|m_s$ dans le transducteur (ce chemin ne commençant pas nécessairement à un état initial).

Question 3. Expliquer comment adapter l'algorithme précédent pour, étant donné un mot de sortie m_s, calculer l'ensemble des antécédents généralisés de m_s. En supposant f injective, donner une borne sur le nombre d'antécédents généralisés d'un mot. Donner des conditions sur le transducteur pour que seul un nombre fini de mots aient plus d'un antécédent généralisé.

La suite des questions consistait à établir et analyser un algorithme de décodage avec correction d'erreurs : partant d'un mot m_s de Σ^n, trouver un mot m_e de Γ^n minimisant la distance de Hamming entre $f(m_e)$ et m_s (programmation dynamique, algorithme de Viterbi).

B6 – Plus proche ancêtre commun

Dans cet exercice on considère des arbres binaires. Étant donnés un arbre et un nœud x, un *ancêtre* de x est un nœud appartenant à l'unique chemin simple de x à la racine de l'arbre (extrémités incluses). La *profondeur* d'un nœud x est la longueur du chemin de x à la racine, comptée en nombre d'arêtes. Étant donnés deux nœuds x et y de l'arbre, le *plus proche ancêtre commun* de x et y est le nœud de profondeur maximale qui est à la fois ancêtre de x et de y.

On s'intéresse au cas où un même arbre va être soumis à de nombreux calculs de plus proche ancêtre commun, de sorte qu'il est intéressant de recourir à des algorithmes avec pré-traitement : une première phase analyse l'arbre et mémorise éventuellement un certain nombre d'informations afin de faciliter la deuxième phase (les recherches de plus proches ancêtres communs).

Question 0. Démontrer que pour tous deux nœuds x et y d'un arbre, le plus proche ancêtre commun existe et est unique.

Question 1. Proposer un algorithme simple sans pré-traitement pour la recherche du plus proche ancêtre commun et évaluer sa complexité.

Question 2. Proposer un pré-traitement qui rend la recherche des plus proches ancêtres communs triviale, et évaluer sa complexité mémoire.

Pour aboutir à de meilleurs algorithmes, on va se réduire à un problème différent, la recherche de minimum : étant donnés un tableau d'entiers A et des bornes i et j, renvoyer un indice auquel apparaît l'élément minimum de $A[i..j]$, la portion de tableau comprise entre i et j (bornes incluses).

Question 3. Expliquer comment pré-calculer les indices des minimums pour toute paire (i, j) en un temps quadratique en la longueur du tableau.

Question 4. Pour économiser du temps et de l'espace, on va limiter les précalculs aux fragments de la forme $A[i..i + 2^k - 1]$ pour tous i, k positifs ou nuls tels que $i + 2^k - 1$ est strictement inférieur à la taille du tableau. Détailler l'algorithme effectuant ce précalcul et donner les complexités en temps et en mémoire.

Expliquer comment ce précalcul suffit à trouver le minimum de tout tableau $A[i..j]$ en temps constant.

Échantillon de questions supplémentaires

Question 5. Étant donné un arbre, on construit un tableau d'entiers de la manière suivante : on remplace chaque arête de l'arbre par deux arêtes orientées (une dans chaque direction) et on considère un circuit eulérien π partant de la racine (c'est-à-dire un chemin passant une et une seule fois par chaque arête orientée). La i-ème case de A contient la profondeur du sommet auquel aboutit π après i arêtes

Étant donnés un arbre et deux nœuds x et y, montrer que l'on peut se servir du tableau décrit ci-dessus pour résoudre les recherches de plus proche ancêtre commun.

Les questions suivantes visaient à obtenir un temps de pré-traitement linéaire, tout en permettant toujours des recherches en temps constant. Pour cela, le tableau était partitionné en blocs de taille logarithmique en la taille totale.

B7 – Problème du mot

On se donne un alphabet Σ, et un ensemble d'*équations orientées* $E = \{x_1 \mapsto y_1, ..., x_n \mapsto y_n\}$ entre mots de Σ^*. On construit le graphe G_E dont les sommets sont les mots de Σ^* et dans lequel on a une arête de m vers m' s'il existe u, v dans Σ^* et i tels que $m = ux_iv$ et $m' = uy_iv$. On note alors $m \rightsquigarrow m'$. On note de plus $m \rightsquigarrow^* m'$ s'il existe un chemin (éventuellement vide) de m vers m' et on appelle m' un *descendant* de m.

Étant donnés deux mots x et y, on veut résoudre le *problème du mot* : déterminer si $x \cong_E y$, où $\cong_E$ est définie comme la plus petite relation d'équivalence contenant $\rightsquigarrow$.

Question 0. On se donne l'alphabet $\Sigma = \{a, b, c\}$ et les équations $E = \{ba \mapsto ab; ac \mapsto b\}$. A-t-on $abac \cong_E abca$?

Question 1. Pourquoi ne peut-on pas résoudre ce problème à l'aide des algorithmes habituels de recherche de chemins dans un graphe ? Énoncer une condition suffisante sur E pour qu'un algorithme de recherche de chemins fonctionne.

En supposant que l'alphabet Σ est totalement ordonné, on définit l'*ordre hiérarchique* $\prec$ sur les mots par : $m \prec m'$ si m est plus court que m' ou si ils ont la même taille et, en notant $m = a_1...a_n$ et $m' = b_1...b_n$, s'il existe $i \in \{1...n\}$ tel que $a_i < b_i$ et pour tout $j < i$, $a_j = b_j$. Un ensemble d'équations est *bien orienté* si pour toute équation $x_i \mapsto y_i$ dans E on a $x_i \succ y_i$.

Question 2. Montrer qu'il existe une fonction strictement monotone de $(\Sigma^*, \prec)$ dans $(\mathbb{N}, <)$, et en déduire qu'il n'existe pas de séquence infinie strictement décroissante pour $\prec$ dans Σ^*.

En supposant que E est bien orienté, en déduire que pour tout mot m n'admet qu'un nombre fini de descendants.

Un système d'équations orientées est *confluent* si pour tous $m_0, m_1, m_2 \in \Sigma^*$ avec $m_0 \rightsquigarrow^* m_1$ et $m_0 \rightsquigarrow^* m_2$ il existe $m_3 \in \Sigma^*$ tel que $m_1 \rightsquigarrow^* m_3$ et $m_2 \rightsquigarrow^* m_3$.

Une *forme normale* est un mot $m \in \Sigma^*$ qui n'a d'autres descendants que lui-même.

Question 3. Étant donné un système d'équations bien orienté et confluent, montrer que tout mot admet parmi ses descendants une unique forme normale. En déduire un algorithme qui réussit toujours et résout le problème du mot.

Échantillon de questions supplémentaires

Un système d'équations orientées est *localement confluent* si pour tous $m_0, m_1, m_2 \in \Sigma^*$ avec $m_0 \rightsquigarrow m_1$ et $m_0 \rightsquigarrow m_2$ il existe $m_3 \in \Sigma^*$ tel que $m_1 \rightsquigarrow^* m_3$ et $m_2 \rightsquigarrow^* m_3$.

Question 4. Considérons le système d'équations donné par les règles $\{a \mapsto b; b \mapsto a; a \mapsto c; b \mapsto d\}$.
Ce système est-il localement confluent ? Est-il confluent ?

Question 5. Montrer qu'un système bien orienté localement confluent est confluent.

Les questions suivantes introduisaient la notion de paire critique et étudiaient un algorithme permettant de compléter un système d'équations E en un système localement confluent générant la même équivalence sur les mots (algorithme de Knuth-Bendix).

B8 – Arbretas

L'objectif de cet exercice est, étant donné un ensemble E de points du plan $\mathbb{R}^2$, et deux abscisses $x_g \leq x_d \in \mathbb{R}$, de trouver un point de E d'ordonnée minimale parmi les points dont l'abscisse est dans l'intervalle ouvert $]x_g, x_d[$.

On utilise pour cela des arbres binaires dits *cartésiens*, dont chaque nœud est étiqueté par un point du plan. On dit qu'un arbre cartésien *représente* l'ensemble E des points qui étiquettent ses nœuds. Un arbre cartésien définit un *arbre des abscisses* et un *arbre des ordonnées*, obtenus respectivement en ne gardant que l'abscisse ou que l'ordonnée de chaque point (ces deux arbres ont donc la même forme).

On demande que nos arbres cartésiens respectent de plus les contraintes suivantes :

— L'arbre des abscisses a une structure d'arbre binaire de recherche : pour chaque nœud binaire étiqueté par x, tous les éléments du sous-arbre gauche sont plus petits que x et tous les éléments du sous-arbre droit sont plus grands que x.

— L'arbre des ordonnées a une structure de tas binaire : pour chaque nœud binaire, étiqueté par y, les étiquettes y_g et y_d des racines des sous-arbres gauche et droit s'ils existent sont plus grandes que y.

Pour refléter cette combinaison de propriétés, ces arbres cartésiens sont aussi appelés *arbretas*.

Question 0. On considère l'ensemble de points $E = \{(0,3); (1,1); (2,5); (3,2); (4,2)\}$. Donner deux arbres cartésiens distincts représentant l'ensemble E.

Question 1. Montrer que, si E est un ensemble de points dans lequel tous les points ont des abscisses deux à deux distinctes et tous les points ont des ordonnées deux à deux distinctes, alors il existe un unique arbre cartésien représentant E.

Question 2. Soit un arbre binaire dont les nœuds ne sont pas étiquetés. Montrer qu'il existe au moins une manière d'associer un point à chaque nœud pour en faire un arbre cartésien.

Question 3. Définir un algorithme qui prend en entrée un arbre cartésien a représentant un ensemble E et un nombre réel x_s, et qui renvoie deux arbres cartésiens, représentant respectivement $\{(x,y) \in E \mid x < x_s\}$ et $\{(x,y) \in E \mid x > x_s\}$. Quelle est la complexité de cet algorithme ?

Question 4. Définir un algorithme qui prend en entrée un arbre cartésien a représentant un ensemble E de points, deux réels x_{min} et x_{max}, et qui renvoie un point de E d'ordonnée minimale parmi les points dont l'abscisse est dans l'intervalle ouvert $]x_{min}, x_{max}[$. Quelle est la complexité de cet algorithme ?

Échantillon de questions supplémentaires

Question 5. Définir un algorithme qui prend en entrée un arbre cartésien a et un point p, et qui insère p dans l'arbre a en respectant la structure d'arbre cartésien. Quel est le coût de la création à partir de E d'un arbre cartésien représentant E en insérant un à un les éléments ?

Question 6. Question rapide : supposons qu'on a défini en premier l'algorithme d'insertion. Comment s'en servir pour effectuer la séparation facilement ?

Question 7. Proposer un algorithme pour construire l'arbre cartésien d'un ensemble E en temps $O(n \log n)$.
Indication : Commencer par trier E.

Les questions suivantes visaient à estimer la profondeur d'un arbre cartésien représentant un ensemble de points aléatoires.

B9 – Tablistes binaires

L'objectif de ce sujet est d'étudier une structure de données qui fournit à la fois des opérations de type tableau (accéder à un élément à partir de son indice) et de type liste (ajouter ou retirer un élément en tête).

On appelle *arbre binaire d'ordre k* un arbre binaire complet dont toutes les feuilles sont séparées de la racine par k arêtes. Un arbre binaire d'ordre 0 est donc constitué d'un unique nœud qui est aussi une feuille, et un arbre binaire d'ordre 1 est constitué de trois nœuds dont deux feuilles. Pour $k > 0$, un *entonnoir d'ordre k* est un arbre formé d'un nœud racine ayant pour unique fils un arbre binaire d'ordre $k-1$. Un entonnoir d'ordre 0 est formé d'un unique nœud qui est aussi une feuille.

Une *tabliste* est une liste de paires $(k_0, E_0)...(k_m, E_m)$ d'un entier et d'un entonnoir, dans lesquelles l'entonnoir E_i est d'ordre k_i et chaque nœud de chaque entonnoir est étiqueté par un élément, feuilles comprises. Une tabliste représente un tableau, dont les cases sont les nœuds de la tabliste considérés dans l'ordre suivant : chaque entonnoir E_i contient les indices de a à $a + 2^{k_i} - 1$, l'indice a étant le premier indice non utilisé par les entonnoirs précédents, et E_0 commençant à 0. Un entonnoir contenant les indices à partir de a est organisé comme suit : la racine correspond à l'indice a, et l'arbre binaire a les indices à partir de $a + 1$, attribués à chaque nœud selon l'ordre de parcours infixe (d'abord le sous-arbre gauche, puis la racine, et enfin le sous-arbre droit). La *taille* d'une tabliste est le nombre d'éléments qu'elle contient, c'est-à-dire son nombre de nœuds.

Question 0. Donner une tabliste représentant le tableau $[1, 1, 1, 2, 3, 6, 11, 23, 47, 106, 235, 551, 1301]$.

Question 1. Donner un algorithme qui, étant donnée une tabliste $(k_0, E_0)...(k_m, E_m)$ et un entier i trouve l'élément contenu par le nœud d'indice i, et évaluer sa complexité en fonction de la taille de la tabliste.

On appelle *tabliste binaire* une tabliste $(k_0, E_0)...(k_m, E_m)$ telle que $\forall 0 \leq i < m,\ k_i < k_{i+1}$.

Que devient la complexité si on se restreint à des tablistes binaires ?

Question 2. Prenons une tabliste $(k, E_1)(k, E_2)$. Montrer comment fusionner E_1 et E_2 en un entonnoir E' d'ordre $k+1$ tel que chaque nœud conserve l'indice qu'il avait dans la tabliste d'origine.

En déduire un algorithme pour ajouter un élément e en tête d'une tabliste binaire T. L'élément ajouté doit être au nouvel indice 0, et les indices de tous les éléments suivants doivent être décalés de 1. Quelle est la complexité de cet algorithme ?

Échantillon de questions supplémentaires

Une intuition importante ici est que la structure d'une tabliste binaire à n éléments suit la décomposition en base 2 de n (avec le bit de poids faible à gauche). L'opération de fusion précédente correspond donc à la propagation d'une retenue. Cette indication était fournie aux candidats qui ne faisaient pas le lien eux-mêmes.

Question 3. On considère une tabliste binaire de départ quelconque T et une suite de n éléments. On veut successivement ajouter en tête de T les n éléments. Estimer la complexité totale de l'opération. *Autrement dit : Quelle est la complexité amortie de l'algorithme d'insertion ?*

Les question suivantes considéraient à la fois l'insertion et la suppression d'un élément, et visaient à enrichir la structure de telle sorte qu'une séquence quelconque de n opérations d'insertion ou de suppression ait un coût de $O(n)$. Cela passait par une séquence d'arbres correspondant à une écriture redondante des nombres, dans laquelle le chiffre 2 était autorisé pour matérialiser une retenue pas encore propagée.

B10 – Solutions aléatoires

On se donne un ensemble $\mathcal{X} = \{x_1, ..., x_n\}$ de variables. Étant donnée une formule logique φ sur les variables de $\mathcal{X}$, on souhaite générer aléatoirement une valuation $\nu : \mathcal{X} \to \{0, 1\}$ satisfaisant φ. On souhaite en outre faire en sorte que les valuations satisfaisant φ soient générées selon une probabilité uniforme.

Question 0. Considérons la formule $\varphi = x_1 \vee \neg(x_2 \vee x_1) \vee x_3$. Quelle est la probabilité qu'une valuation ν satisfaisant φ vérifie $\nu(x_1) = 1$? Quelle est la probabilité qu'elle vérifie $\nu(x_2) = 1$ si on sait de plus que $\nu(x_1) = 0$?

Pour toutes formules φ, φ' et toute variable x, on note $\varphi[x := \varphi']$ la formule obtenue en remplaçant chaque occurrence de x dans φ par φ'.

Question 1. Étant données trois formules φ_0, φ_1 et φ_2, construire avec les connecteurs logiques de base une formule signifiant `si` φ_0 `alors` φ_1 `sinon` φ_2, et montrer que pour toute formule φ et toute variable x, la formule `si` x `alors` $\varphi[x := 1]$ `sinon` $\varphi[x := 0]$ est équivalente à φ.

En déduire que toute formule φ est équivalente à une formule φ' utilisant exclusivement le connecteur `si . alors . sinon ..` Quelle est la taille de φ' ? Cette formule peut-elle aider pour la génération d'une valuation satisfaisante aléatoire ?

La formule expansée précédente est appelée *arbre de décision*. Il s'agit d'un cas particulier de *diagramme de décision*, c'est-à-dire un graphe orienté acyclique dont chaque feuille est étiquetée par 0 ou 1, et dont chaque nœud interne est étiqueté par une variable x et a un degré sortant de 2, l'une des arêtes étant étiquetée par 0 et l'autre par 1. On pourra noter $[x, n_0, n_1]$ le nœud étiqueté par x dont les deux successeurs sont les nœuds n_0 (via l'arête étiquetée par 0) et n_1 (via l'arête étiquetée par 1).

Un diagramme de décision est *réduit* si les deux conditions suivantes sont vérifiées :

1. En aucun nœud $n = [x, n_0, n_1]$ on n'a $n_0 = n_1$
2. Il n'existe pas deux nœuds identiques

Un diagramme de décision est *ordonné* pour un ordre $<$ sur $\mathcal{X}$ si pour tout chemin dans le diagramme, les variables étiquetant les nœuds sont rencontrées dans l'ordre croissant suivant $<$, et chaque variable est rencontrée au plus une fois.

Question 2. Proposer un algorithme pour calculer un diagramme de décision réduit pour une formule et discuter sa complexité en temps et en espace.

Question 3. Montrer que les feuilles 0 et 1 sont atteignables depuis tout nœud interne d'un diagramme de décision réduit.

Question 4. À partir du diagramme de décision ordonné et réduit d'une formule φ, donner un algorithme efficace pour générer une valuation satisfaisante aléatoire.

Échantillon de questions supplémentaires

Question 5. Considérons une fonction booléenne f de n variables $x_1, ..., x_n$ et l'ordre total sur les variables $x_i < x_j$ si $i < j$. Montrer qu'il existe un unique diagramme de décision ordonné réduit caractérisant f.

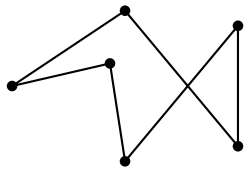
Les questions suivantes concernaient la combinaison de deux diagrammes de décision ordonnés réduits, ainsi que sur l'impact sur la taille du diagramme de l'ordre choisi sur les variables.

B11 – Tournées générales à faible coût

On considère un ensemble E de points du plan euclidien, qui définit un graphe complet G_E dont les sommets sont les points de E, chaque paire de sommets étant liée par une arête dont la *longueur* est la distance euclidienne entre ses extrémités.

Une *tournée* de E est un chemin qui revient à son point de départ après être passé au moins une fois par chaque point de E. Une *tournée parfaite* est une tournée qui passe exactement une fois par chaque point. La *longueur* d'un chemin est la somme des longueurs de ses arêtes. On cherche à construire des tournées parfaites de longueur minimale.

Question 0. Voici un ensemble de points du plan, et les arêtes qui ont été retenues pour une tournée. Par quelles petites modifications peut-on diminuer la longueur de cette tournée ?



Question 1. Proposer un algorithme naïf et estimer sa complexité.

Un *arbre couvrant* de G_E est donné par un ensemble d'arêtes A_{AC} tel que $G_{AC} = (E, A_{AC})$ est un arbre. Le *poids* d'un arbre couvrant est la somme des longueurs de ses arêtes.

Question 2. En supposant qu'on sache construire efficacement un arbre couvrant de poids minimal pour E, proposer une manière de construire une tournée parfaite dont la longueur est au plus deux fois la longueur d'une tournée minimale.
Indication : Inégalité triangulaire.

Un *couplage parfait* d'un graphe $G = (S, A)$ est donné par un ensemble d'arêtes A_{CP} tel que tout sommet de $G_{CP} = (S, A_{CP})$ est de degré 1. Le *poids* d'un couplage est la somme des longueurs de ses arêtes.

Question 3. Dans un graphe avec un nombre pair de sommets, comparer le poids minimal d'un couplage parfait à la longueur d'une tournée minimale.

Une *tournée eulérienne* d'un graphe $G = (S, A)$ est une tournée qui passe exactement une fois par chaque arête de A.

Question 4. Montrer qu'un graphe $G = (S, A)$ connexe admet une tournée eulérienne si et seulement si tous ses sommets ont un degré pair.

Question 5. En supposant que l'on possède des algorithmes pour construire efficacement l'arbre couvrant minimal et le couplage parfait minimal, combiner les éléments précédents pour obtenir un algorithme produisant une tournée parfaite dont la longueur est au plus une fois et demie la longueur d'une tournée minimale. *Il s'agit de l'algorithme de Christofides.*

Les questions suivantes visaient à démontrer que, grâce aux propriétés géométriques du problème, la recherche de l'arbre couvrant minimal pouvait se restreindre à un sous-graphe de G_E avec un nombre d'arêtes linéaire en le nombre de sommets (graphe de Delaunay).

B12 – Comptage de motifs dans des arbres

Soit un alphabet Σ dont chaque lettre a est associée à un nombre entier `arité(a)` positif ou nul. On s'intéresse aux arbres dont chaque nœud est étiqueté par une lettre a et possède un nombre de fils égal à `arité(a)`. L'arbre formé par un nœud a d'arité k ayant pour fils les arbres t_1 à t_k est noté $a(t_1, ..., t_k)$, ou juste a si $k = 0$.

On se donne un symbole spécial $\perp$ d'arité 0. On appelle *motif* un arbre sur $\Sigma \cup \{\perp\}$. On dit qu'un arbre $a(t_1, ..., t_k)$ *correspond* au motif $a(m_1, ..., m_k)$ si chaque t_i correspond à m_i. Tout arbre t correspond au motif $\perp$.

Étant donné un arbre A et un motif M, on veut compter le nombre de sous-arbres de A correspondant à M.

Question 0. Indiquer dans l'arbre $b(a(a(a(c, c), c), c), a(a(c, b(c, c)), c))$ les sous-arbres correspondant au motif $a(a(\perp, \perp), c)$.

Question 1. Étant donnés un arbre A et un motif M, donner un algorithme naïf pour compter le nombre de sous-arbres de A correspondant à M et estimer sa complexité.

Question 2. On veut généraliser la question précédente au comptage du nombre de sous-arbres de A correspondant à l'un des motifs d'un ensemble $M_1, ..., M_k$. Proposer un pré-traitement de l'ensemble $M_1, ..., M_k$ pour que cela ne multiplie pas la complexité par k.

Question 3. Donner un autre algorithme de comptage, travaillant des feuilles vers la racine, en reprenant les principes de la programmation dynamique. Discuter la complexité de cet algorithme.

Un *automate d'arbres* sur l'alphabet Σ est un triplet (Q, F, δ) où Q est un ensemble d'états, $F \subseteq Q$ est l'ensemble des états acceptants, et $\delta \subseteq \Sigma \times Q^* \times Q$ est une relation de transition qui à une lettre a d'arité k et k états $q_1, ..., q_k$ associe un état q.

La reconnaissance d'un arbre par un automate d'arbres se fait des feuilles vers la racine.

Question 4. Utiliser un automate d'arbres pour reformuler l'algorithme précédent. Discuter la complexité de cette reformulation.

Échantillon de questions supplémentaires

Question 5. Montrer que le nombre d'états de l'automate d'arbres déterministe peut être exponentiel en le nombre de motifs recherchés (un automate est déterministe si la relation de transition est une fonction).

Question 6. Modifier la construction des automates d'arbres pour compter les occurrences de motifs à k erreurs près (on comptera comme une erreur le remplacement d'un symbole par un autre de même arité).