Banques MP et MPI inter-ENS – Session 2024
Rapport relatif à l'épreuve orale d'informatique

— Épreuve commune à toutes les ENS
— Coefficients (en pourcentage du total des points de chaque concours) :

École	Concours MP	Concours MP - Info	Filière MPI
Paris	23.1%	13.3%	13.3%
Lyon	10.8%	14.1%	14.1%
Paris-Saclay	23.1%	13.2%	13.2%
Rennes	23.1%	N.A.	13.9%

— Membres du jury :
  — Alexandre Debant
  — Romain Demangeon
  — Brice Minaud
  — Raphaël Monat
  — Charles Paperman
  — André Schrottenloher
  — Pierre Senellart

L'épreuve orale d'informatique fondamentale décrite dans ce rapport est commune à toutes les Écoles Normales Supérieures.

Cette année, le jury a interrogé :

— 122 candidat·e·s pour la filière MP. Les notes données s'échelonnent entre 8 et 19, avec une médiane à 13, une moyenne à 12.65 et un écart-type de 2.75. La figure 1 présente l'histogramme complet des notes.

— 103 candidat·e·s pour la filière MPI. Les notes données s'échelonnent entre 6 et 20, avec une médiane à 13, une moyenne à 12.75 et un écart-type de 3.43. La figure 2 présente l'histogramme complet des notes.

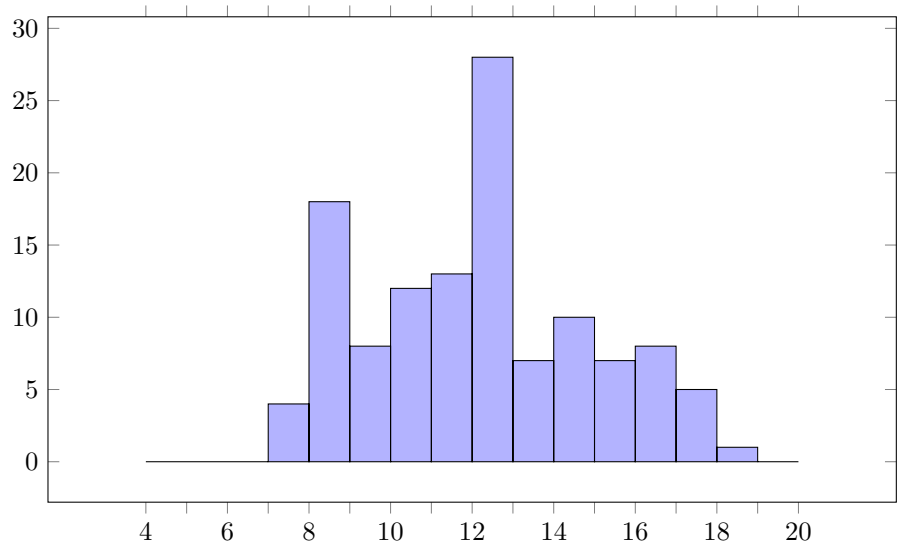


FIGURE 1 – Histogramme des notes de l'épreuve (filière MP). La colonne positionnée entre x et $x+1$ comptabilise les notes comprises entre x exclus et $x+1$ inclus.

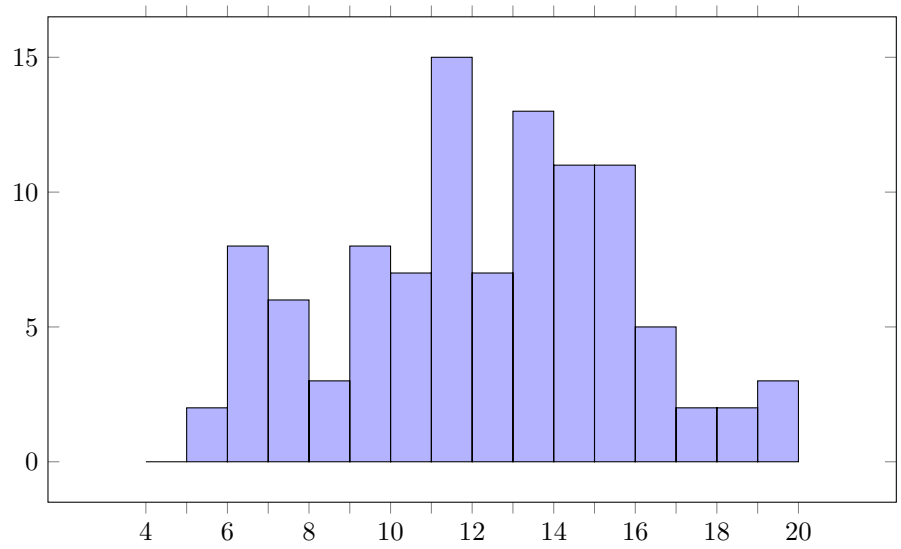


FIGURE 2 – Histogramme des notes de l'épreuve (filière MPI). La colonne positionnée entre x et $x+1$ comptabilise les notes comprises entre x exclu et $x+1$ inclus.

Après avoir reçu un sujet, les candidat·e·s disposent de 30 minutes de préparation, suivies de 28 minutes d'interrogation devant un des examinateurs. En effet, deux minutes sont utilisées par l'examinateur pour aller chercher la/le candidat·e en salle de préparation. Nous rappelons également que jusqu'à quatre candidat·e·s peuvent passer l'épreuve à la suite sur le même sujet, auquel cas la première/le premier d'entre eux est invité·e à patienter 30 minutes dans la salle de préparation à l'issue de son oral, afin de garantir la confidentialité du sujet.

Comme il a été rappelé en préambule des sujets distribués :

> Le but de cette épreuve est d'évaluer la progression des candidates et candidats dans les questions, mais aussi la qualité de leur exposé des solutions, ainsi que l'autonomie dont elles ou ils font preuve pendant l'oral.

Le jury a proposé 17 sujets originaux (10 pour le concours MP et 7 pour le concours MPI), dont la liste est donnée en annexe de ce document. Entre 12 et 16 candidat·e·s étaient interrogé·e·s sur chaque sujet, ce qui permet d'harmoniser les évaluations d'un sujet donné.

Chaque sujet débute par un énoncé qui présente un problème d'informatique et introduit ses notations, puis comporte des questions de difficulté globalement croissante. Les premières questions permettent de démontrer que l'on a compris l'énoncé, ou d'aider à sa compréhension. Les dernières questions d'un sujet sont souvent des questions d'ouverture plus difficiles. En général, il n'est pas attendu que les candidat·e·s traitent l'intégralité des questions.

Les sujets proposés portent sur des thèmes variés de science informatique : langages, graphes, logique, *etc.*, en lien avec les programmes respectifs des filières MP et MPI. Ils nécessitent de s'approprier des concepts nouveaux, démontrer des résultats théoriques et construire des solutions techniques telles que des algorithmes. Bien que l'épreuve mette l'accent sur les concepts théoriques de l'informatique, on veille à garder à l'esprit le sens des objets que l'on étudie, et un certain sens pratique (par exemple dans la conception d'algorithmes ou l'estimation asymptotique de leur complexité) est apprécié.

Le jury tient à féliciter les candidates et candidats qui ont pour la plupart démontré des acquis très solides, et fait preuve d'idées excellentes malgré les contraintes de temps inhérentes à l'épreuve. Même celles et ceux ayant obtenu de moins bonnes notes ont montré des qualités certaines. En particulier, la plupart des candidat·e·s affichent une bonne maîtrise des concepts mathématiques essentiels (induction, disjonction de cas) ainsi qu'une bonne initiative (par exemple le fait de tester un algorithme sur un petit exemple avant d'en déduire le cas général).

Il est usuel que le jury engage une discussion sur certaines questions afin de guider les candidat·e·s dans leur résolution du problème. Une attitude constructive et positive est appréciée lors de ces échanges. A contrario, certaines attitudes peuvent être fortement pénalisées par le jury, comme par exemple ne pas tenir compte des conseils et indications fournis, ou encore essayer de profiter de ces échanges pour utiliser le jury comme "oracle" pour plusieurs questions.

Le jury adresse les conseils suivants aux futur·e·s candidat·e·s.

Concours et autres voies d'accès. Les Écoles Normales Supérieures sont destinées notamment aux personnes intéressées par la recherche ou l'enseignement. Il peut être utile pour les candidats de connaître les différentes voies d'accès à ces écoles, présentées sur la page suivante :

`https://diplome.di.ens.fr/informatique-ens/`

Gestion du temps et de l'espace. Répéter ou recopier l'énoncé lors de l'oral est une perte de temps, l'examinateur ayant lui aussi le sujet sous les yeux. Le jury déconseille aux candidat·e·s l'effacement du tableau à la main, qui le rend rapidement illisible pour l'examinateur.

Raisonnements. Afin de démontrer leur maîtrise du programme d'informatique de leur filière, mais aussi leur compréhension des nouveaux concepts introduits par le sujet traité, le jury invite les candidat·e·s à s'assurer de la cohérence de ce qu'ils décrivent.

Par exemple, si le jury demande un identifiant de variable OCaml, celui-ci ne peut pas, par définition, être un mot clé du langage, ou une application de fonction. Des erreurs d'attention ou de compréhension

peuvent arriver, mais la répétition de ces erreurs de syntaxe ou de typage donne l'impression très négative que la/le candidat·e n'a pas compris le sujet.

De manière générale, le jury conseille aux candidat·e·s d'utiliser les raisonnements par l'absurde uniquement lorsque ceux-ci sont nécessaires, et de préférer les inductions structurelles (par exemple sur un arbre) aux récurrences sur un critère numérique (hauteur de l'arbre). Celles-ci sont usuellement plus efficaces et / ou plus adaptées aux objets étudiés.

Pédagogie, intuition, formalisme. Si la situation le permet, il est souhaitable de décrire (en général oralement ou/et par un dessin) une preuve dans les grandes lignes avant de se lancer dans la preuve formelle. Cela permet à l'examinateur de s'assurer que la/le candidat·e a compris le principe de la preuve. De même, il est souhaitable de décrire un algorithme dans les grandes lignes avant de se lancer dans l'écriture du pseudo-code.

Si une question attend une réponse oui/non, il est conseillé d'annoncer cette réponse avant de se lancer dans la preuve. De même, si une question contient plusieurs résultats à prouver, annoncer lequel sera prouvé en premier, *etc.*

Certains sujets demandent d'absorber plusieurs notions nouvelles, parfois au moyen d'un formalisme assez lourd. On attend alors des candidat·e·s une capacité à s'affranchir du formalisme pour se concentrer sur la signification des objets. Cependant, si l'examinateur demande des précisions, la/le candidat·e doit être en mesure de justifier formellement son raisonnement.

Langage OCaml, programmation fonctionnelle. Lorsqu'une fonction OCaml doit être décrite, le jury suggère fortement aux candidat·e·s de réfléchir en premier abord à la signature de type de la fonction, qui leur permettra de clarifier leurs idées avant de proposer une implémentation.

Pour la filière MPI, le jury a remarqué les candidat·e·s voyaient difficilement le parallèle entre variables libres en logique et dans des expressions OCaml.

Remarques diverses. Le jury tient à souligner, comme l'an dernier, que la majorité des candidat·e·s oublient qu'il y a une différence entre entiers machine de langages comme OCaml et C, et les entiers mathématiques.

Plusieurs sujets demandaient des manipulations de formules logiques qui ont posé problème à certain·e·s candidat·e·s. Par exemple, lors d'un sujet manipulant des formules logiques construites à partir de conjonctions et d'implications, il était important de remarquer que $(a \wedge b) \implies c$ est équivalent à $a \implies b \implies c$. Certain·e·s candidat·e·s ont essayé de revenir à la définition de l'implication. Cela n'était malheureusement pas intéressant, en particulier car la négation et la disjonction n'apparaissaient pas dans les constructions du sujet. Un autre sujet demandait de manipuler $\oplus$ (OU exclusif) et $\wedge$ (ET) en utilisant leur associativité et le fait que $\wedge$ est distributif sur $\oplus$ (ce qui était rappelé). Le parallèle entre ces opérateurs logiques et les opérations d'addition et multuplication dans un corps fini à 2 éléments ne semblait pas connu.

Le concours des ENS s'intéresse à sélectionner des candidat·e·s qui ont majoritairement vocation à faire de la recherche ou de l'enseignement au terme de leurs études, devant un public divers. À ce titre, certains membres du jury souhaitent souligner qu'il est préférable d'utiliser les termes « enfants » et « parent » plutôt que « fils » et « père » lors de raisonnements sur des arbres.

Le jury s'est étonné du fait que certain·e·s candidat·e·s confondent les lettres grecques Φ (phi) et Ψ (psi), ce qui peut induire en erreur l'examinateur lorsque ces deux lettres interviennent dans le même raisonnement.

Annexe : sujets proposés pour la filière MP

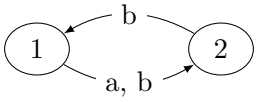
Certains sujets sont accompagnés d'*ébauches* de solution.

Automates sur les langages de chemins

Σ-graphes

Dans ce sujet, on appelle *langage régulier* un ensemble de mots reconnu par un automate fini déterministe. Dans la suite on notera ε le mot vide.
Un Σ-graphe $G = (S, A)$ est un graphe dirigé dont les arêtes sont étiquetées par des lettres de l'alphabet Σ. Formellement, S est un ensemble fini, $A \subseteq S \times \Sigma \times S$. Pour un élément $x = (s_1, a, s_2)$ de A on adopte la notation x.source pour s_1, x.label pour a et x.cible pour s_2.
Un chemin de G est une séquence finie $x_1, \ldots, x_n$ d'éléments A tel que x_i.cible et x_{i+1}.source sont égaux. Dans la suite on note $\mathrm{Path}(G)$ l'ensemble des chemins de G.

Question 1. Montrer que $\mathrm{Path}(G)$ est un langage régulier et donner un automate fini déterministe pour le graphe suivant :



Pour $G = (S, A)$ un Σ-graphe, on note $\pi_G \colon A^* \to \Sigma^*$ l'application qui *oublie* les sources et cibles du chemin pour ne garder que les étiquettes. Formellement, pour $u = x_1 \cdots x_n \in A^n$, alors $\pi_G(u)$ est un élément de Σ^n tel que $\pi_G(u)_i = x_i$.label.

Question 2. Soit $G = (S, A)$ un Σ-graphe et K un langage régulier sur l'alphabet A. Montrer que $\pi_G(K)$ est un langage régulier.

On note $\alpha \colon \Sigma^* \to \mathcal{P}(\Sigma)$ l'alphabet d'un mot, c'est-à-dire, l'ensemble des lettres qui apparaissent dans l'écriture d'un mot. Plus formellement, $\alpha(u_1 \cdots u_n) = \{a \in \Sigma \mid \exists i, u_i = a\}$. Un langage L est testable alphabétiquement s'il vérifie pour tout $u, v \in \Sigma^*$, $u \in L$ et $\alpha(v) = \alpha(u)$ implique $v \in L$. Pour $G = (S, A)$ un Σ-graphe, on dit qu'un langage L est G-testable alphabétiquement s'il existe un langage K sur l'alphabet A testable alphabétiquement tel que $\pi_G(K \cap \mathrm{Path}(G)) = L$.

Question 3. On pose dans cette question $\Sigma = \{a, b\}$. On note $M_2 = (\{-1, 0, 1\}, A)$ avec $(-1, x, 0)$, $(0, x, 1)$ et $(1, x, 0)$ dans A pour toute lettre $x \in \Sigma$. Montrer que $(\Sigma^2)^+ a \Sigma^*$ n'est pas testable alphabétiquement mais est M_2-testable alphabétiquement.

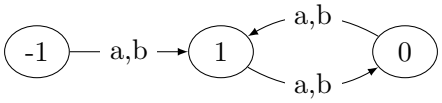


FIGURE 1 – Graphe M_2

Le graphe des suffixes

On note $\Sigma^{<n}$ l'ensemble des mots de taille au plus n. Pour $u_1 \cdots u_p \in \Sigma^{<n}$, et $a \in \Sigma$, on note $u \cdot_n a$ le mot qui est égal à $u_1 \cdots u_p \cdot a$ si $p < n-1$ et $u_2 \cdots u_p a$ si $p = n-1$. Dans la suite on note $S_n = (\Sigma^{<n} \cup \{\bot\}, A_n)$ le Σ-graphe défini par :

$$A_n = \{(u,a,v) | u,v \in \Sigma^{<n}, a \in \Sigma, u \cdot_n a = v\} \cup \{(u,a,\bot) \mid u \in \Sigma^{n-1}, a \in \Sigma\}$$

Un langage régulier L est *testable par fenêtre glissante* s'il existe n tel que L est S_n-testable alphabetiquement.

Question 4. Dessinez le graphe S_2 pour $\Sigma = \{a,b\}$ et montrer que $(ab)^*$ est testable par fenêtre glissante.

Question 5. Montrez que $(ac^*b + c)^*$ n'est pas testable par fenêtre glissante.

Résolution du problème 3-SAT

Soit X un ensemble de variables qui seront notées avec des lettres minuscules : x, y, etc. Un *littéral* est une variable x ou sa négation $\neg x$. Une formule booléenne est en *forme normale conjonctive* (CNF) si elle s'écrit sous la forme d'une conjonction de *clauses*, qui sont des disjonctions de littéraux. On l'appellera alors « une CNF » par abus de langage, et de plus, une « 3-CNF » lorsque chaque clause contient au plus trois littéraux. Une *valuation* est une fonction de X dans $\{0, 1\}$.
On considère le problème 3-SAT :

— Entrée : une 3-CNF contenant n littéraux et m clauses. On supposera $m = \mathcal{O}(n)$.

— Sortie : Vrai si la formule est satisfiable, Faux sinon.

On s'intéresse à des algorithmes efficaces pour résoudre ce problème, en temps $\widetilde{\mathcal{O}}(2^{\alpha n})$ où α est une constante et la notation $\widetilde{\mathcal{O}}$ omet les facteurs polynomiaux en n.

Question 1. Donner un algorithme en temps $\widetilde{\mathcal{O}}(2^n)$ pour résoudre 3-SAT.

Solution : Par « temps » ici on désigne le nombre d'opérations binaires (calculer un AND, OR, XOR, etc.) On utilise un algorithme de recherche exhaustive : tester toutes les évaluations possibles des variables. Chaque évaluation possible est testée en temps $\mathcal{O}(m)$, et il y a 2^n évaluations différentes.

Pour toute 3-CNF Φ et toute variable $x \in X$, on note $\Phi|x$ la 3-CNF obtenue en évaluant x à 1 (« Vrai ») dans Φ (respectivement, $\Phi|\neg x$ en évaluant x à 0). On étend cette notation à un ensemble de variables : par exemple $\Phi|xy$ évalue x et y à 1.

On admet que le polynôme $X^3 - X^2 - X - 1$ admet une seule racine réelle α et que $\alpha < 1.84$. On admet que la plus grande racine β du polynôme $X^3 - 2X - 2$ vérifie $\beta < 1.77$.

Question 2.
1. Soit x une variable apparaissant dans Φ. Montrer que Φ est satisfiable si et seulement si $\Phi|x$ est satisfiable ou $\Phi|\neg x$ l'est.
2. Soit x une variable telle que le littéral $\neg x$ n'apparaît pas dans Φ. Montrer que Φ est satisfiable si et seulement si $\Phi|x$ est satisfiable.

Solution : 1. Si Φ est satisfiable, alors par définition il existe une valuation des variables qui satisfait toutes ses clauses. Si x est évaluée à 0 alors cette valuation fonctionne pour $\Phi|\neg x$, sinon elle fonctionne pour $\Phi|x$.

Inversement, si $\Phi|x$ est satisfiable alors il existe une valuation qui satisfait toutes ses clauses, que l'on étend en assignant 1 à x. Si $\Phi|\neg x$ est satisfiable on l'étend par 0.

2. Après la première partie de la question, une seule implication reste à démontrer : si Φ est satisfiable alors $\Phi|x$ l'est. On prend d'abord une valuation f de Φ, que l'on modifie en f' où $f'(x) = 1$. Cela ne peut pas rendre une clause de Φ fausse (car elles ne contiennent pas $\neg x$). Donc par définition f' est une valuation qui satisfait $\Phi|x$.

Question 3.
1. Soit $(x \vee y \vee z)$ une clause de Φ. Montrer que Φ est satisfiable si et seulement si $\Phi|x$ ou $\Phi|\neg xy$ ou $\Phi|\neg x\neg yz$ l'est.
2. En déduire un algorithme pour résoudre 3-SAT. Soit $T(n)$ sa complexité, en fonction du nombre de variables n. Montrer que $T(n) = \widetilde{\mathcal{O}}(1.84^n)$.

Solution : 1. On peut procéder par équivalences en faisant d'abord une disjonction de cas sur x, puis sur y :

$$\begin{aligned} \Phi \text{ est SAT} &\iff \Phi|x \text{ ou } \Phi|\neg x \text{ est SAT} \\ \Phi|\neg x \text{ est SAT} &\iff \Phi|\neg xy \text{ est SAT ou } \Phi|\neg x\neg y \text{ est SAT} \end{aligned}$$

dans le dernier cas, on remarque que $\Phi|\neg x\neg y$ contient la clause z, ce qui signifie que $\Phi|\neg x\neg y$ est SAT si et seulement si $\Phi|\neg x\neg yz$ l'est aussi. On obtient bien la disjonction de cas voulue.

2. L'algorithme consiste à examiner la première clause de Φ et à faire la disjonction vue précédemment. On retombe sur plusieurs cas de formules à moins de variables. La récursion donne :

$$T(n) \leq poly(n) + T(n-1) + T(n-2) + T(n-3) \ .$$

Le polynôme caractéristique est $X^3 - X^2 - X - 1$. L'idée est qu'on peut démontrer facilement par récurrence sur n : $T(n) \leq poly'(n)\alpha^n$. En effet, c'est évidemment vrai pour $n = 1$ en sélectionnant un facteur polynomial $poly'$ approprié, et si c'est vrai pour $n-1, n-2, n-3$, on obtient :

$$T(n) \leq poly(n) + poly'(n-1)\alpha^n \left(\alpha^{-1} + \alpha^{-2} + \alpha^{-3}\right)$$

or par définition de α, on a $\alpha^{-1} + \alpha^{-2} + \alpha^{-3} = 1$. Il suffit ensuite de s'assurer que $poly'(n-1) + poly(n) \leq poly'(n)$, ce qui est vrai en sélectionnant un $poly'$ de degré suffisamment élevé.

Question 4. Soit x une variable telle que le littéral x et sa négation apparaissent tous les deux dans Φ. En considérant une nouvelle disjonction de cas, donner un algorithme plus efficace pour 3-SAT.

Solution : Par définition on peut écrire :

$$\Phi = (x \vee y \vee z) \wedge (\neg x \vee u \vee v) \wedge \Psi$$

pour une formule Ψ plus petite (mais qui peut encore contenir x et les autres variables). Par conséquent :

$$\begin{aligned} \Phi \text{ est SAT} &\iff \Phi|x \text{ est SAT ou } \Phi|\neg x \text{ est SAT} \\ &\iff \Phi|x \wedge (u \vee v) \text{ est SAT ou } \Phi|\neg x \wedge (y \vee z) \text{ est SAT} \end{aligned}$$

Dans le premier cas, on fait une nouvelle disjonction sur u, et dans le cas $\neg u = 1$ on déduit v. C'est symétrique pour y et z. Par conséquent :

Φ est satisfiable si et seulement si $\Phi|xu$ ou $\Phi|x\neg uv$ ou $\Phi|\neg xy$ ou $\Phi|\neg x\neg yz$ le sont.

La nouvelle récurrence sélectionne donc d'abord une variable x de Φ, et vérifie si sa négation apparaît. Si non, on enlève x (par la question 2.2, on peut évaluer x à 1). Si oui, on retombe dans un des cas plus haut.

La complexité $T(n)$ vérifie donc : $T(n) \leq poly(n) + \max(T(n-1), 2T(n-2) + 2T(n-3))$.

Cependant, le terme $T(n-1)$ va disparaître, en effet : $T(n-1) \leq 2T(n-2)$ car on peut toujours deviner une variable. On a donc : $T(n) \leq poly(n) + 2T(n-2) + 2T(n-3)$.

Avec la même méthode que précédemment, on en déduit que $T(n) \leq poly''(n)1.77^n$.

La *distance de Hamming* h entre deux n-uplets de $\{0,1\}^n$ est le nombre de positions où ils diffèrent.

On considère un nouvel algorithme récursif pour 3-SAT qui ne modifie pas la formule Φ, mais une valuation f :

Local(Φ, f, r)
1 : **Si** f satisfait Φ renvoyer Vrai
2 : **Si** $r = 0$ renvoyer Faux
3 : Soient x, y, z les variables de la première clause de Φ non satisfaite par f (on aura pris un ordre arbitraire des clauses de Φ)
4 : Définir f_x par : $\forall t \neq x, f_x(t) = f(t)$ et $f_x(x) = 1 - f(x)$
5 : Définir f_y par : $\forall t \neq y, f_y(t) = f(t)$ et $f_y(y) = 1 - f(y)$
6 : Définir f_z par : $\forall t \neq z, f_z(t) = f(t)$ et $f_z(z) = 1 - f(z)$
7 : Si Local($\Phi, f_x, r-1$) = Vrai renvoyer Vrai
8 : Si Local($\Phi, f_y, r-1$) = Vrai renvoyer Vrai
9 : Renvoyer Local($\Phi, f_z, r-1$)

Question 5. 1. Montrer que s'il existe une valuation qui satisfait Φ et qui est à distance de Hamming $\leq r$ de f, **Local(**Φ, f, r) renvoie Vrai

2. Donner un algorithme de complexité $\widetilde{\mathcal{O}}\left(3^{n/2}\right)$ pour 3-SAT

Solution : 1. On prouve par récurrence sur r que s'il existe une valuation qui satisfait Φ et est à distance de Hamming $\leq r$ de f, **Local(**Φ, f, r) renvoie Vrai.

En effet, soit g cette valuation. Si $r = 0$, $f = g$ par définition donc la propriété est vraie.

Sinon, si f ne satisfait pas Φ, la première clause non satisfaite par V contient une variable sur laquelle f diffère de g.

En modifiant la valuation de cette variable, on obtient f' telle que $h(f', g) = h(f, g) - 1 \leq r - 1$. On peut alors utiliser l'hypothèse de récurrence : l'un au moins des trois appels récursifs renvoie Vrai, et par conséquent **Local(**Φ, f, r) renvoie Vrai.

2. Pour terminer la preuve de l'algorithme, nous pouvons choisir $r = n/2$ et partir des valuations $(0, \ldots, 0)$ et $(1, \ldots, 1)$. Si Φ est satisfiable, il existe une valuation qui est à distance au plus $n/2$, soit de $(0, \ldots, 0)$, soit de $(1, \ldots, 1)$. La recherche locale s'exécute alors en temps $\widetilde{\mathcal{O}}\left(3^{n/2}\right)$.

Notons bien qu'il ne suffit pas de partir d'une seule valuation : en effet quel que soit le n-uplet de départ il existe toujours un autre n-uplet à distance n de lui, on aurait donc besoin d'une profondeur n dans l'algorithme pour l'atteindre, et donc d'un temps 3^n. Cela dit on peut supposer qu'en moyenne, notre solution n'est pas si éloignée de notre point de départ. C'est justement l'idée de la question suivante.

Question 6. 1. Soit $V(r)$ le nombre de n-uplets de $\{0,1\}^n$ à distance au plus r d'un n-uplet donné. Soit $t \in \{0,1\}^n$ fixé. Montrer qu'en choisissant $n2^n/V(r)$ n-uplets uniformément au hasard dans $\{0,1\}^n$, la probabilité qu'aucun ne soit à distance au plus r de t (i.e., proche de t d'une distance r) est bornée par e^{-n}.

2. En déduire un meilleur algorithme pour 3-SAT (avec une faible probabilité d'erreur).

Solution : 1. En sélectionnant un n-uplet uniformément au hasard, la probabilité que t soit à distance au plus r de ce n-uplet est $\geq V(r)/2^n$. Donc la probabilité que t ne soit proche d'aucun des n-uplets choisis est au plus :

$$(1 - V(r)/2^n)^{n2^n/V(r)} \leq \left((1 - V(r)/2^n)^{(2^n/V(r))}\right)^n \leq e^{-n} \ .$$

2. En appliquant **Local** avec une distance r et en partant des $n2^n/V(r)$ n-uplets choisis, on a un temps : $\widetilde{\mathcal{O}}\left(\frac{2^n 3^r}{V(r)}\right)$, qu'il faut donc optimiser.

Pour cela on a besoin d'une estimation asymptotique de $V(r)$. On a : $V(r) = \sum_{i=0}^{r} \binom{n}{i} \geq \binom{n}{r}$ (une bonne approximation à facteur polynomial près). En posant $r = \alpha n$ pour une certaine constante α et en utilisant la formule de Stirling, on a :

$$V(r) \geq \frac{n!}{(\alpha n)!((1-\alpha)n!)} \simeq \frac{n^n}{(\alpha n)^{\alpha n}((1-\alpha)n)^{(1-\alpha)n}} \simeq \frac{1}{(\alpha^\alpha (1-\alpha)^{1-\alpha})^n}$$

Donc en posant $r = \alpha n$ on obtient un temps :

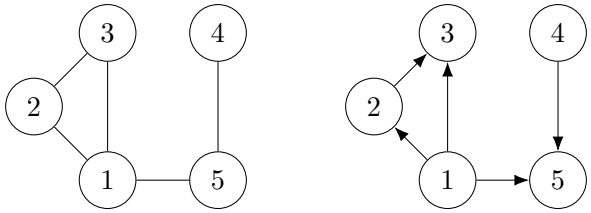
$$\widetilde{\mathcal{O}}\left(\left((3\alpha)^\alpha (1-\alpha)^{1-\alpha}\right)^n\right)$$

Que l'on minimise en prenant $\alpha = 1/4$, ce qui donne une complexité en $\widetilde{\mathcal{O}}\left((3/2)^n\right)$, soit la meilleure de cet exercice.

Graphes transitivement orientables

Ordre partiel. On rappelle qu'un ordre partiel est une relation transitive, réflexive et antisymétrique.
Graphes. Un graphe $G = (V, E)$ est composé d'un ensemble de sommets $V \neq \varnothing$, et d'un ensemble d'arêtes E. Si G est orienté, $E \subseteq \{(v, w) : v \neq w \in V\}$. Si G est non-orienté, $E \subseteq \{\{v, w\} : v \neq w \in V\}$. Dans tout ce sujet, on considère des graphes orientés tels que $(v, w) \in E \Rightarrow (w, v) \notin E$.
Orientation. Une *orientation* d'un graphe non-orienté $G = (V, E)$ est un graphe orienté obtenu en choisissant un orientation pour chaque arête de G. Formellement, $G' = (V, E')$ est une orientation de G si $E = \{\{v, w\} : (v, w) \in E' \vee (w, v) \in E'\}$.
Graphe transitivement orientable. Un graphe orienté $G = (V, E)$ est *transitif* si $(v_1, v_2) \in E$ et $(v_2, v_3) \in E$ implique $(v_1, v_3) \in E$. Un graphe non-orienté est *transitivement orientable* s'il existe une orientation transitive du graphe (c'est-à-dire une orientation qui est un graphe transitif).

Exemple :



Le graphe de gauche est transitivement orientable. Le graphe de droite est une orientation transitive.

Question 1. Donner un ensemble infini de graphes transitivement orientables.

Question 2.

a. Soit $\leq$ un ordre partiel sur un ensemble V, et $E = \{\{v, w\} \subseteq V : ((v \leq w) \vee (w \leq v)) \wedge (v \neq w)\}$. Montrer que le graphe (V, E) est transitivement orientable.

b. Réciproquement, tout graphe transitivement orientable peut-il être obtenu de cette manière à partir d'un ordre partiel ?

Arêtes ouvertes. Pour un graphe non-orienté $G = (V, E)$ donné, on dit qu'une paire d'arêtes $\{v_1, v_2\}, \{v_2, v_3\}$ est *ouverte*, noté $\{v_1, v_2\} \smile \{v_2, v_3\}$, si $\{v_1, v_2\} \in E$ et $\{v_2, v_3\} \in E$ et $\{v_1, v_3\} \notin E$. (On autorise $v_1 = v_3$, donc $\smile$ est réflexive.)
Relation $\smile^*$. Pour $\{v, w\} \in E$ et $\{v', w'\} \in E$, on écrit $\{v, w\} \smile^* \{v', w'\}$ s'il existe $k \geq 1$ arêtes $\{v_1, w_1\}, \ldots, \{v_k, w_k\}$ telles que $\{v_1, w_1\} = \{v, w\}$, $\{v_k, w_k\} = \{v', w'\}$, et pour tout $1 \leq i \leq k - 1$, $\{v_i, w_i\} \smile \{v_{i+1}, w_{i+1}\}$. Noter que $\smile^*$ est une relation d'équivalence (on ne demande pas de vérifier).

Question 3. Montrer que si G contient une suite $(v_i)_{i=1}^k$ de sommets tels que

$$\{v_1, v_2\} \smile \{v_2, v_3\} \smile \{v_3, v_4\} \ldots \smile \{v_{k-1}, v_k\} \smile \{v_k, v_1\} \smile \{v_1, v_2\}$$

avec $k > 1$ impair, alors G n'est pas transitivement orientable. En déduire un exemple de graphe qui n'est pas transitivement orientable.

Composition de graphes. Soit $V_0 = \{1, \ldots, n\}$. Soit $G_0 = (V_0, E_0)$ un graphe orienté à n sommets, et soit n graphes orientés $G_i = (V_i, E_i)$, $1 \leq i \leq n$. La composition $G = G_0[G_1, \ldots, G_n]$ est le graphe obtenu en remplaçant chaque sommet i de G_0 par le graphe G_i, et en ajoutant des arêtes depuis tous les sommets de G_i vers tous ceux de G_j si et seulement si $(i, j) \in E_0$. Formellement, $G = (V, E)$ où :

$$V = V_1 \cup \cdots \cup V_n$$
$$E = E_1 \cup \cdots \cup E_n \cup \bigcup_{(i,j)\in E_0} V_i \times V_j.$$

On définit de manière analogue la composition de graphes non-orientés.
Graphe décomposable. Un graphe $G = (V, E)$, orienté ou non, est dit *décomposable* s'il existe $G_0, G_1, \ldots, G_n$ avec $1 < n < |V|$ tels que $G = G_0[G_1, \ldots, G_n]$. Un graphe G est dit *indécomposable* s'il n'est pas décomposable.

Remarque. La condition $1 < n < |V|$ dans la définition de « décomposable » interdit les décompositions triviales.

Question 4. Montrer que si un graphe non-orienté G est indécomposable, alors il a une seule classe d'équivalence pour $\smile^*$.

Question 5. Montrer que si un graphe non-orienté $G = (V, E)$ avec $E \neq \varnothing$ est transitivement orientable, alors il est indécomposable si et seulement si il existe exactement deux orientations transitives de G.

Graphe désorienté. Si $G = (V, E)$ est un graphe orienté, son graphe désorienté est le graphe non-orienté (V, E') où $E' = \{\{v, w\} : (v, w) \in E \vee (w, v) \in E\}$.
Dimension. La *dimension* d'un graphe orienté transitif $G = (V, E)$ est le plus petit entier d tel qu'il existe d graphes orientés transitifs $G_i = (V, E_i)$ dont le graphe désorienté est le graphe complet, et tels que $E = E_1 \cap \cdots \cap E_d$.

Question 6. Montrer que tout graphe orienté transitif a une dimension finie.

Question 7. Soit $G = (V, E)$ un graphe non-orienté, et $\bar{G} = (V, \bar{E})$ avec $\bar{E} = \{\{v, w\} \subseteq V : \{v, w\} \notin E\}$ son graphe complémentaire. Montrer que G admet une orientation transitive de dimension au plus 2 si et seulement si G et $\bar{G}$ admettent tous les deux une orientation transitive.

Coloriages et graphes de permutation

Pour n un entier, on note $[n] = \{1, \dots, n\}$.
Graphe non-orienté. Un graphe non-orienté $G = (V, E)$ est composé d'un ensemble de sommets V, et d'un ensemble d'arêtes $E \subseteq \{\{v, w\} : v \neq w \in V\}$. On écrit $v -_E w$ pour $\{v, w\} \in E$.
Graphe orienté. Un graphe orienté $G = (V, E)$ est composé d'un ensemble de sommets V, et d'un ensemble d'arêtes $E \subseteq V^2$. On écrit $v \to_E w$ pour $(v, w) \in E$.
Chaîne. Une *chaîne* d'un graphe orienté $G = (V, E)$ est une suite de sommets deux à deux distincts $v_0 \to_E v_1 \to_E \cdots \to_E v_k$. Le nombre de sommets $k + 1$ de la chaîne est appelé sa *longueur*.
Cycle. Un *cycle orienté* d'un graphe orienté $G = (V, E)$ est une chaîne $v_0 \to_E v_1 \to_E \cdots \to_E v_k$ telle que $v_k \to_E v_0$. De même pour un graphe non-orienté $G = (V, E)$, un *cycle non-orienté* est un chemin $v_0 -_E v_1 -_E \cdots -_E v_k -_E v_0$ passant par des sommets deux à deux distincts.
Coloriage. un *coloriage* d'un graphe $G = (V, E)$ (orienté ou non) avec k couleurs est une application $\mathsf{col} : V \to [k]$ telle que pour toute paire de sommets v, w reliés par une arête, $\mathsf{col}(v) \neq \mathsf{col}(w)$.

Question 1.

a. Soit G un graphe *non-orienté* acyclique (c'est-à-dire qui ne contient pas de cycle non-orienté). Montrer qu'il existe un coloriage de G avec deux couleurs.

b. Donner un exemple de graphe *orienté* acyclique (c'est-à-dire qui ne contient pas de cycle orienté) pour lequel il n'existe pas de coloriage avec deux couleurs.

Solution :

a. Le graphe est un arbre. Sans perte de généralité, le graphe est connexe. Une manière de faire : on enracine l'arbre sur un sommet quelconque. Ensuite chaque sommet est colorié suivant la parité de sa hauteur dans l'arbre. De manière équivalente, chaque sommet est colorié suivant la parité de la longueur du chemin (unique) qui le connecte à la racine.

b. La question (a) donne un indice : le graphe doit contenir un cycle non-orienté, mais pas de cycle orienté. On peut prendre le graphe triangle de sommets $\{a, b, c\}$ et arêtes $a \to b \to c$ et $a \to c$.

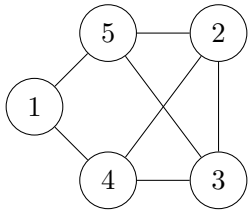
Question 2. Soit G un graphe orienté acyclique. Soit k la longueur de la plus longue chaîne de G. Montrer qu'il existe un coloriage de G avec k couleurs.

Solution : Indication possible : dans la question 1.a. on peut orienter toutes les arêtes de l'arbre vers la racine pour avoir un graphe orienté acyclique. De ce point de vue, la réponse à 1.a. donne la réponse à la question 2 pour un type de graphe particulier. Le candidat peut essayer de généraliser l'idée qu'il a proposée pour 1.a.

On peut donner à chaque sommet comme couleur la longueur de la plus longue chaîne partant de ce sommet. On voit que si deux sommets sont voisins, cette longueur est strictement plus petite pour le descendant. En effet toute chaîne de longueur k partant du descendant donne une chaîne de longueur $k + 1$ partant du parent, qui est valide parce qu'elle ne peut pas repasser par la parent à cause de l'acyclicité.

Isomorphisme de graphe. Deux graphes $G = (V, E)$ et $G' = (V', E')$ sont *isomorphes* s'il existe une bijection $\varphi : V \to V'$ telle que $E' = \{\{\varphi(v), \varphi(w)\} : \{v, w\} \in E\}$.
Graphe de permutation. Soit π une permutation de $[n]$. À π, on associe le graphe $\mathsf{GP}[\pi] = ([n], \{\{i, j\} : i < j \wedge \pi(i) > \pi(j)\})$. Un graphe $G = (V, E)$ est dit *graphe de permutation* s'il existe une permutation π telle que G est isomorphe à $\mathsf{GP}[\pi]$.
Sous-suite. Une *sous-suite* de $(x_1, \dots, x_n)$ est une suite $(x_{i_1}, \dots, x_{i_k})$ telle que $1 \leq i_1 < \cdots < i_k \leq n$.
Clique. Une *clique* d'un graphe non-orienté $G = (V, E)$ est un sous-ensemble de sommets $C \subseteq V$ tel que pour tout $v \neq w \in C$, $v -_E w$. Sa *taille* est son nombre de sommets $|C|$.

Exemple :

Soit π définie par : $\begin{cases} \pi(1) = 3 \\ \pi(2) = 5 \\ \pi(3) = 4 \\ \pi(4) = 1 \\ \pi(5) = 2 \end{cases}$ $\mathsf{GP}[\pi] =$ 

Question 3. Soit π une permutation, et k la longueur de la plus longue sous-suite décroissante de $(\pi(1), \ldots, \pi(k))$.

a. Montrer que la plus grande clique de $\mathsf{GP}[\pi]$ est de taille k.

b. Montrer que le nombre minimal de couleurs nécessaires pour colorier $\mathsf{GP}[\pi]$ est exactement k.
Indication : utiliser la question 2.

Solution :

a. Le point clé est que deux sommets $i < j$ sont liés dans $\mathsf{GP}[\pi]$ si et seulement si $\pi(i) > \pi(j)$. Une sous-suite décroissante de $(\pi(1), \ldots, \pi(n))$ est donc exactement une clique de $\mathsf{GP}[\pi]$.

b. Indication possible : pour utiliser la question 2, orienter judicieusement les arêtes de $\mathsf{GP}[\pi]$.
En fait, il suffit d'orienter chaque arête $\{i, j\}$ vers $\max(i, j)$. Le graphe orienté obtenu est acyclique puisque $i \to j$ implique $i < j$. Par la question 2, on peut donc le colorier avec un nombre de couleurs égal à la longueur de la plus longue chaîne. Or une chaîne est la même chose qu'une sous-suite décroissante dans $(\pi(1), \ldots, \pi(n))$ qui est la même chose qu'une clique (dans le graphe non-orienté). La longueur de la plus longue chaîne est donc égale à la taille de la plus grande clique. Réciproquement, on ne peut pas colorier avec moins de couleurs que la taille de la plus grande clique, puisque tous les sommets de la clique doivent avoir des couleurs deux à deux distinctes.

Question 4. Proposer un algorithme qui prend en entrée une suite de n entiers deux à deux distincts, et renvoie sa plus longue sous-suite décroissante, en temps polynomial en n.

Solution : La contrainte de temps polynomial est là pour interdire les algorithmes triviaux (type : essayer toutes les sous-suites). On n'exige pas un algorithme optimal, ni une complexité précise, tant que le candidat justifie que sa solution est polynomiale.

Un algorithme polynomial raisonnable est le suivant : on calcule et enregistre, pour chaque entier de la suite, la longueur de la plus longue suite décroissante qui finit sur cet entier. Pour ça, on parcourt la suite de gauche à droite. Pour le premier entier, la plus longue suite décroissante qui finit dessus est de longueur 1. Ensuite, pour chaque nouvel entier, on prend le max de la plus longue suite qui finit sur chacun des entiers à sa gauche qui sont plus grands que lui, plus 1. Enfin, pour avoir la plus longue suite, on prend le max de la plus longue suite qui finit sur chaque entier. (Des algorithmes en $O(n \log n)$ existent mais sont hors-sujet.)

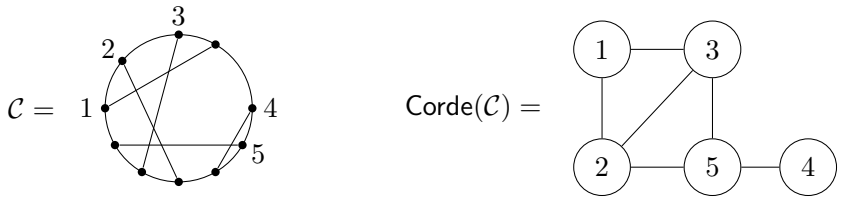
Voisinage. Soit $G = (V, E)$ un graphe non-orienté et $v_0 \in V$ un sommet. Le *voisinage* de v_0 est le sous-graphe de G formé par v_0 et ses voisins. Formellement :

$$\mathsf{voi}(v_0) = (V', \{\{v, w\} : v, w \in V' \wedge v -_E w\}) \quad \text{où } V' = \{v_0\} \cup \{w : v_0 -_E w\}.$$

Corde. Une *corde* d'un cercle est un segment reliant deux points du cercle.
Graphe de cordes. Soit $\mathcal{C}$ un ensemble de cordes d'un même cercle, reliant des points deux à deux distincts. On définit le *graphe de cordes* de $\mathcal{C}$ par :

$$\mathsf{Corde}(\mathcal{C}) = (\mathcal{C}, \{\{c, d\} : c \neq d \in \mathcal{C} \wedge c \cap d \neq \varnothing\}).$$

Exemple :



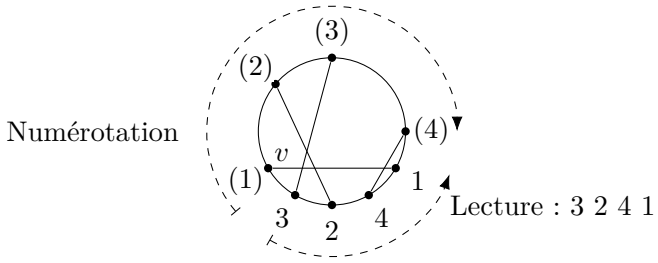
Question 5. Soit $G = \mathsf{Corde}(\mathcal{C})$ un graphe de cordes, et $c \in \mathcal{C}$ un sommet de G. Montrer que $\mathsf{voi}(c)$ est un graphe de permutation.

Indication : les voisins de c sont les cordes intersectant c. On peut voir ces cordes comme les arêtes d'un graphe biparti. Il est possible d'interpréter ce graphe comme décrivant une permutation.

Solution : Indication 2 possible : il faut que la permutation qu'on extrait ait la propriété que deux cordes se croisent si et seulement si la permutation inverse l'ordre de deux éléments.

Indication plus précise : si on voit les extrémités des corde comme des sommets, la corde c « coupe » les sommets appartenant à ses voisins en un graphe biparti. La permutation apparaît visuellement : c'est la permutation qui envoie les sommets d'un côté de c sur les sommets de l'autre côté, en suivant les cordes (pour une numérotation naturelle des extrémités).

On part d'une extrémité arbitraire v de la corde c. On numérote cette corde (1), puis on parcourt le cercle dans le sens horaire, en numérotant les cordes rencontrées dans l'ordre (2) à (k), jusqu'à revenir à la corde c de départ. Ensuite, on repart de v, cette fois dans le sens anti-horaire. En lisant les numéros des cordes rencontrées jusqu'à revenir à (1) (qui n'est pas comptée au départ), on obtient une permutation de $[k]$. Sur l'exemple de l'énoncé, en prenant la corde $c = 5$, on a :



Une manière alternative est de numéroter non pas les cordes, mais les extrémités des cordes, de 1 à k en partant de v, dans chaque direction. On interprète ensuite les cordes comme reliant une extrémité numérotée i d'un côté de c à l'extrémité numérotée $\pi(i)$ de l'autre côté.

Le point clé est que deux cordes vont se croiser si et seulement si les deux entiers correspondants sont inversés par la permutation. C'est particulièrement visible avec la deuxième représentation ci-dessus, puisque i est relié à $\pi(i)$ (et c'est pourquoi l'indication pousse plutôt vers la deuxième représentation). La conclusion est que le graphe de cordes sur les voisins de v est exactement $\mathsf{GP}[\pi]$, pour π la permutation qu'on a lue.

Question 6. Soit $\mathcal{C}$ un ensemble de cordes et $G = \mathsf{Corde}(\mathcal{C}) = (V, E)$ le graphe de cordes correspondant. Étant donné $\mathcal{C}$, proposer un algorithme qui trouve une clique de taille maximale dans G en temps polynomial en $|\mathcal{C}|$.

Indication : utiliser les questions précédentes.

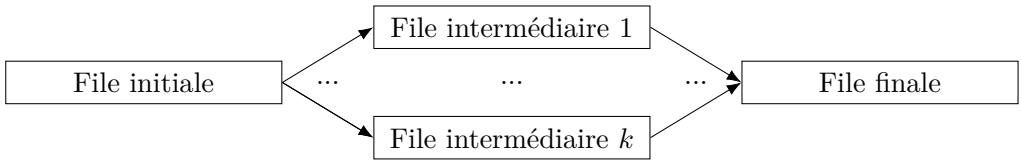
Solution : Indication 2 possible : toute clique du graphe G est aussi une clique dans le voisinage de chacun de ses sommets.

Avec l'indication 2, on voit que pour trouver une clique maximale dans G, il suffit de chercher une clique maximale dans le voisinage de chacun des sommets de G, et de renvoyer la plus grande qu'on a trouvée. Par la question 5, le problème de trouver une clique de taille maximale dans G se réduit donc (polynomialement) au problème de trouver une clique de taille maximale dans un graphe de permutation. Or on a vu dans la question 3 que cela revient à trouver la plus grande sous-suite décroissante dans une suite donnée, ce qu'on sait faire en temps polynomial via la question 4. Une subtilité : dans la question 3, l'équivalence utilise le fait qu'on connaît la permutation, pas juste

le fait que le graphe est un graphe de permutation. Mais c'est bien le cas ici, parce que les cordes qui génèrent le graphe sont données en entrée (c'est explicite dans la question), et qu'on peut retrouver la permutation à partir des cordes (comme on l'a fait dans la question 5).

File. Une *file* est une structure de données abstraite permettant de stocker une suite d'entiers. Elle est munie de deux opérations : enfile(i) ajoute l'entier i en queue de file ; défile() enlève l'élément en tête de file (ordre FIFO), et renvoie sa valeur.
Tri par k files. Un *tri par k files* est un système composé d'une *file initiale*, de $k \geq 1$ *files intermédiaires*, et d'une *file finale*. Au départ, la file initiale contient les entiers $\pi(1), \ldots, \pi(n)$ ($\pi(1)$ en tête de file). Les autres files sont vides. Ensuite, on peut faire à volonté les deux opérations suivantes : (1) défiler la file initiale, et enfiler l'élément sortant dans une des k files intermédiaires au choix ; ou bien (2) défiler une des files intermédiaires au choix, et enfiler l'élément sortant dans la file finale. Le but est d'arriver à une situation où la file finale contient tous les entiers dans l'ordre usuel $1, \ldots, n$ (1 en tête de file).

Schéma :



Question 7. Soit π une permutation. Montrer que le nombre k minimal tel qu'il est possible de trier π par k files est égal au nombre minimal de couleurs nécessaires pour colorier $\mathsf{GP}[\pi]$.

Indication : cette question est indépendante des précédentes.

Solution : Supposons qu'on peut colorier $\mathsf{GP}[\pi]$ avec les couleurs $[k]$. On défile chaque élément de la file initiale de couleur c dans la file intermédiaire numéro c. Les entiers de chaque file intermédiaire sont nécessairement triés, parce que deux entiers dans la même file sont de la même couleur, donc non reliés par une arête dans $\mathsf{GP}[\pi]$, donc étaient dans le bon ordre dans la file initiale, et donc aussi dans leur file intermédiaire commune. Une fois que tous les entiers sont dans les files intermédiaires (en ordre trié dans chaque file, comme on a vu), on défile le plus grand élément disponible vers la file finale, de manière répétée jusqu'à ce que la file finale contienne $1, \ldots, n$.

Réciproquement, si on peut trier π par k queues, on colorie chaque élément avec le numéro de la file intermédiaire dans laquelle il est envoyé. Deux éléments reliés par une arête dans $\mathsf{GP}[\pi]$ sont en ordre inversé dans π, donc ne peuvent pas être envoyés dans la même file, donc sont de couleurs différentes.

ROBDD

Dans ce sujet, nous souhaitons étudier la satisfiabilité de propriétés dans des modèles de Kripke. Pour ce faire, nous souhaitons utiliser des diagrammes de décision binaires (BDD - binary decision diagrams) pour représenter et manipuler des fonctions booléennes.

BDD : un diagramme de décision binaire représentant la fonction $f : \{0,1\}^n \to \{0,1\}$ est un graphe fini, orienté, acyclique ayant une unique racine et tel que chaque noeud est :

- soit un noeud de degré sortant 0, représentant la constante 0 ;
- soit un noeud de degré sortant 0, représentant la constante 1 ;
- soit un noeud de degré 2, noté $N_i(x,l,h)$, tel que si $N_i(x,l,h)$ représente la fonction $f_i(x,x_p\ldots,x_n)$ alors l et h sont deux BDD représentant respectivement $f_i(0,x_p,\ldots,x_n)$ et $f_i(1,x_p,\ldots,x_n)$. La variable x est appelée *étiquette* du noeud tandis que l et h sont appelés ses *fils*. La valeur i identifie de manière unique chaque noeud. Sans perte de généralité, nous supposerons que $i \in \{1,\ldots,n\}$ où n dénote le nombre de noeuds distincts dans le graphe.
- l'unique racine du graphe représente la fonction $f(x_1,\ldots,x_n)$.

Notation : étant donné une fonction $f : \{0,1\}^n \to \{0,1\}$, $f_{|x\mapsto b}$ dénote la fonction f dans laquelle la variable x est assignée à la valeur b.

ρ-OBDD : un BDD G est dit *ordonné* selon un ordre total $<_\rho$ sur les variables, si pour tout chemin $x_1x_2\ldots x_n$ dans G, $x_i <_\rho x_{i+1}$.

ρ-ROBDD : un ρ-OBDD G est dit *réduit* si :

- toutes les feuilles ont des étiquettes distinctes ;
- pour tout noeud $N_i(x,l,h)$, $l \neq h$;
- pour toute paire de noeuds $N_i(x_1,l_1,h_1)$, $N_j(x_2,l_2,h_2)$ tels que $i \neq j$ (i.e., ce sont des noeuds à des positions distinctes dans G), $(x_1,l_1,h_1) \neq (x_2,l_2,h_2)$.

Question 1. (a) Donner un $\rho-$ROBDD représentant la fonction $(a_1 \Leftrightarrow b_1) \wedge (a_2 \Leftrightarrow b_2) \wedge (a_3 \Leftrightarrow b_3)$. Le choix de l'ordre $<_\rho$ sur $\{a_1,a_2,a_3,b_1,b_2,b_3\}$ est au libre choix du candidat.

(b) Le choix de l'ordre a-t-il un impact sur le taille du ROBDD ? Est-il possible de définir une fonction $f : \{0,1\}^n \to \{0,1\}$ et deux ordres $<_{\rho_1}$, $<_{\rho_2}$ tels qu'il existe un ρ_1-ROBDD représentant f de taille $\Theta(2^{n/2})$ et un ρ_2-ROBDD représentant f de taille $\mathcal{O}(n)$?

Question 2. Soit f une fonction booléenne et $<_\rho$ un ordre total sur les variables de f.

(a) Montrer qu'il existe un unique $\rho-$ROBDD représentant f.

(b) En déduire une structure de données qui permettrait de réduire la répétition de calculs identiques lors de la manipulation d'un ou plusieurs $\rho-$ROBDD.

Question 3. Soit $f : \{0,1\}^n \to \{0,1\}$. En exploitant la structure de ROBDD, donner un algorithme qui calcule efficacement le nombre de solutions à l'équation $f(x_1,\ldots,x_n) = 1$.

Soit AP un ensemble fini de propositions atomiques. Un *modèle de Kripke* est un triplet (S, R, L) où S est un ensemble d'états, $R \subseteq S \times S$ est une relation de transition, et $L : S \to \mathcal{P}(AP)$ est une fonction qui étiquette chaque état par un ensemble de propositions atomiques. Par convention nous appellerons $L(s)$ l'ensemble des propositions atomiques *satisfaites* par s, et $AP \setminus L(s)$ l'ensemble des propositions atomiques *non-satisfaites* par s.
Dans la suite, nous supposerons que chaque état s peut être identifié de manière unique par son étiquette, i.e. $L(s)$.

Opérations de base : nous supposerons que nous connaissons des algorithmes efficaces pour calculer les opérations de base suivantes entre $\rho-$ROBDD : $\vee$, $\neg$, $\wedge$.

Question 4. Soit G un $\rho-$ROBDD représentant la fonction f et x une variable de f. Définir $\mathsf{Exist}(\mathsf{G}, \mathsf{x})$ la fonction qui retourne un $\rho-$ROBDD qui représente la fonction $\exists x.f$.

Question 5. Soit $<_\rho$ un ordre total sur AP et (S, R, L) un modèle de Kripke. Nous souhaitons définir un $\rho-$ROBDD représentant un ensemble d'états et un $\rho-$ROBDD représentant la relation de transition.

(a) soit $S' \subseteq S$ définir un $\rho-$ROBDD représentant S' ;

(b) définir un $\rho-$ROBDD représentant R ;

(c) étant donné un ensemble d'état $S' \subseteq S$ définir $Image(S', R)$ le $\rho-$ROBDD représentant l'ensemble des états atteints par R depuis S', i.e. $\{t \mid s \in S' \text{ et } (s, t) \in R\}$.

Question 6. En déduire un algorithme permettant de décider si, étant donné un modèle de Kripke (S, R, L), une proposition $p \in AP$ est satisfaite dans tout état accessible (en un nombre quelconque d'étapes) depuis un état de départ s_0, noté $(S, R, L), s_0 \models p$.

Question 7. Nous souhaitons maintenant vérifier une autre forme de propriétés : étant donné un état initial s_0, existe-t-il un état accessible s_f tel que $p_f \in L(s_f)$ et pour tout état intermédiaire s_{int}, $p_i \in L(s_{int})$?

Donner un algorithme $CheckUntil((S, R, L), s_0, p_i, p_f)$ qui décide cela.

Caractérisation de POLYTIME *de Bellantoni-Cook*

On définit un langage d'expressions :

$$\begin{array}{lcl} N & ::= & 0 \mid \mathbf{S}\ (N) \\ Y & ::= & x \mid 0 \mid \mathbf{S}\ (Y) \\ E & ::= & Y \mid f(E, \ldots, E) \\ L & ::= & f(Y, \ldots, Y) \end{array}$$

où $x \in \mathcal{X}$, un ensemble infini de *variables* et $f \in \mathcal{F}$, un ensemble infini de symboles de *fonctions* d'arités fixées. N représente les entiers en écriture unaire (on s'autorisera dans les exemples à utiliser l'écriture décimale), Y les expressions entières qui peuvent contenir une variable, E les expressions qui peuvent contenir des fonctions, variables et entiers et L les expressions contenant un unique symbole de fonctions.
Une *équation* (l, e) (on écrira $l = e$) est un couple de $L \times E$ tel que si $x \in \mathcal{X}$ apparaît dans e, x apparaît dans l et tel que si x et y sont deux variables qui apparaissent dans l, $x \neq y$.
Un *système* d'équation $S = (l_i, e_i)_{1 \leq i \leq j}$ (avec $j \in \mathbb{N}$) définit une relation de *réduction* $\longrightarrow_S$. Soit une expression e. S'il existe $\sigma : \mathcal{X} \to N$ et $1 \leq k \leq j$ tels qu'une sous-expression e_s de e est égale à $\sigma(l_k)$ (on étend σ aux expressions en remplaçant chaque variable x par $\sigma(x)$), alors $e \longrightarrow_S e'$ (ou simplement $e \longrightarrow e'$, quand il est clair quel S on considère) où e' est obtenue en remplaçant e_s par $\sigma(e_k)$ dans e.
On note $\longrightarrow^*$ la cloture réflexive et transitive de $\longrightarrow_S$ (c'est-à-dire la relation obtenue par zéro, une, ou plusieurs réductions successives). L'ensemble $\{e' | e \longrightarrow_S^* e'\}$ est appelé *les réduits de e par S* et noté $\mathtt{red}_S(e)$ (ou simplement $\mathtt{red}(e)$ quand il est clair quel S on considère).

Question 1. Soit le système S_0 :

$$\begin{array}{lcl} \mathtt{add}(x, 0) & = & x \\ \mathtt{add}(x, \mathbf{S}\ (z)) & = & \mathbf{S}\ (\mathtt{add}(x, z)) \end{array}$$

Calculer $\mathtt{red}_{S_0}(\mathtt{add}(2, 2))$.

On note $\mathtt{u} : \mathbb{N} \to N$, la bijection qui envoie un entier de $\mathbb{N}$ vers son écriture unaire.
Soit un système d'équations S et f un symbole de fonction de ce système. Quand elle est bien définie, on note $\mathbf{I}(f, S) : \mathbb{N}^a \to \mathbb{N}$ la fonction telle que $\mathbf{I}(f, S)(k_1, \ldots, k_a) = v$ quand $f(\mathtt{u}(k_1), \ldots, \mathtt{u}(k_a)) \longrightarrow_S^* \mathtt{u}(v)$.

Question 2. Donner un système qui définit !, la factorielle d'un entier.

Question 3. Préciser quand $\mathbf{I}(f, S)$ est bien définie. Donner des contres-exemples.

Question 4. Donner un critère non-trivial[1] sur S qui assure que pour tout f apparaissant dans S, et tout $(n_1, \ldots, n_a) \in N^a$, $\mathtt{red}_S(f(n_1, \ldots, n_a))$ est un ensemble fini.

Soit P un polynôme à une variable, on dit que *le système S est borné par P* si pour tout $f \in S$ d'arité a, et tout $(n_1, \ldots, n_a) \in N^a$

$$|\mathtt{red}_S(f(n_1, \ldots, n_a))| \leq P(\mathtt{u}^{-1}(n_1) + \cdots + \mathtt{u}^{-1}(n_a))$$

POLYTIME est $\{\varphi : \mathbb{N}^a \to \mathbb{N} \mid \exists (S, f, P)$ tels que S est borné par P et $\mathbf{I}(f, S) = \varphi\}$.

1. Le système de la question 2 doit valider le critère.

Question 5. Donner un polynôme qui borne S_1 :

$$\begin{array}{lcl} \texttt{double}(0) & = & 0 \\ \texttt{double}(\mathbf{S}\ (x)) & = & \texttt{add}(\texttt{double}(x), \mathbf{S}\ (\mathbf{S}\ 0)) \end{array}$$

Même question pour S_2 :

$$\begin{array}{lcl} \texttt{double}(0) & = & 0 \\ \texttt{double}(\mathbf{S}\ (x)) & = & \texttt{add}(\mathbf{S}\ (\mathbf{S}\ 0), \texttt{double}(x)) \end{array}$$

Question 6. Exhiber une fonction qui n'est pas dans **POLYTIME**.

> Dans l'équation $f(x_1, \ldots, x_n) = e$, on appelle *appels récursifs* les sous-expressions $f(e_1, \ldots, e_n)$ de e.

Question 7. En discutant des appels récursifs de chaque équation, trouver un critère sur S qui garantit que les fonctions qu'il définit sont dans **POLYTIME**.

Un peu de réécriture

Un *système de réécriture abstrait (ARS)* sur A est une relation binaire $\to$ sur A, i.e. $\to \subseteq A \times A$.
On note $\to^+$ sa clôture transitive et $\to^*$ sa clôture réflexive transitive :

$$\to^+ = \{(x,y) \mid \exists x_1, \ldots, x_n \in A^n \text{ tel que } x \to x_1 \to \ldots \to x_n = y\}$$

$$\to^* = \{(x,x) \mid x \in A\} \cup \to^+$$

Successeurs : pour tout $x \in A$, $Succ(x) = \{y \mid x \to y\}$ l'ensemble des *successeurs directs* de x. Plus généralement, on note $Succ^+(x) = \{y \mid x \to^* y\}$ l'ensemble des *successeurs* de x.
On dit que $\to$ est à *branchements finis* si pour tout $x \in A$, $Succ(x)$ est fini.
Terminaison : un système de réduction $\to$ *termine* s'il n'existe pas de chaine infinie $x_1 \to x_2 \to \ldots$
Confluence : un système de réduction $\to$ est *confluent* si pour tous $x, y_1, y_2 \in A$, tels que $x \to^* y_1$ et $x \to^* y_2$, il existe $z \in A$ tel que $y_1 \to^* z$ et $y_2 \to^* z$.
Confluence locale : un système de réduction $\to$ est *localement confluent* si pour tous $x, y_1, y_2 \in A$, tels que $x \to y_1$ et $x \to y_2$, il existe $z \in A$ tel que $y_1 \to^* z$ et $y_2 \to^* z$.

Question 1. Soit $\to$ un ARS sur A à branchements finis et acyclique, i.e., il n'existe pas de $x \in A$ tel que $x \to^+ x$. Montrer que pour tout $x \in A$, $Succ^+(x)$ est fini si et seulement si $\to$ termine.

Question 2. Soit $\to$ un ARS sur A à branchements finis. Montrer que $\to$ est terminant si et seulement si il existe un plongement monotone dans $(\mathbb{N}, >)$, i.e., il existe une fonction $\varphi : A \to \mathbb{N}$ tel que $x \to y$ implique $\varphi(x) > \varphi(y)$.

Question 3. (Lemme de Newman) Soit $\to$ un ARS sur A terminant et localement confluent.
Montrer que $\to$ est confluent.

On s'intéresse maintenant plus spécifiquement aux systèmes de réécriture de termes.
Terme : Étant donné un ensemble Σ de fonctions et leur arité, noté f/n $(n > 0)$, et un ensemble dénombrable de constantes X, on note $T(\Sigma, X)$ l'ensemble des termes constructibles inductivement, i.e. $t := \begin{cases} x & \text{if } t = x \in X \\ f(t_1, \ldots, t_n) & \text{if } f/n \in \Sigma \text{ et } t_1, \ldots, t_n \in T(\Sigma, X) \end{cases}$.
Position : Soit t un terme, on note $Pos(t)$ l'ensemble de ses positions (des mots sur l'alphabet des entiers) définies inductivement par : $Pos(t) := \begin{cases} \{\varepsilon\} & \text{if } t = x \in \Sigma_0 \\ \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in Pos(t_i)\} & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$.
On notera $t|_p$ le sous-terme de t à la position p. De même, on notera $t[s]_p$ le terme t dans lequel le sous-terme à la position p a été remplacé par s.
Relation de réduction de termes : Étant donné un ensemble fini de règles, $R = \{(l_i, r_i)\}_i$. La relation de réduction $\to_R \subseteq T(\Sigma, X) \times T(\Sigma, X)$ est définie par :

$$s \to_R t \text{ si et seulement si } \exists (l, r) \in R, p \in Pos(s), s|_p = l \text{ et } t = s[r]_p.$$

Exemple : Soit $\Sigma = f/2, g/3, h/1$ et $X = \{x, y, z\}$. Soit $t = f(g(x, h(y), z), h(z))$.
On a $Pos(t) = \{\varepsilon, 1, 2, 11, 12, 13, 121, 21\}$. Les positions de y et $h(z)$ dans t sont respectivement est 121 et 2.

Question 4. Soit R un ensemble fini de règles. Montrer que le problème de terminaison de $\to_R$ est décidable [1].

1. Note : nous avons défini la notion de système de réécriture de termes avec des règles de réécriture closes (i.e., sans variables). Le problème plus général « avec variables » est lui indécidable.

Question 5. Soit R un ensemble fini de règles. On souhaite montrer que la confluence de $\to_R$ est décidable. Pour cela, nous définissons $CP(R) = \{(r_1, l_1[r_2]_p) \mid (l_i, r_i) \in R \text{ for } i = 1, 2 \text{ and } l_1|_p = l_2\}$ l'ensemble des *paires critiques* de R.

1. montrer que $\to_R$ est localement confluent si et seulement si toutes ses paires critiques (u, v) sont joignables, i.e. il existe w tel que $u \to_R^* w$ et $v \to_R^* w$.
2. en déduire que confluence locale de $\to_R$ (et donc la confluence) est décidable.

Évaluation d'expressions

Certaines questions nécessitent décrire du code OCaml. Les approximations et les erreurs mineures dans la syntaxe ne seront pas prises en compte.
On considère des expressions arithmétiques linéaires

$$e ::= v \in \mathcal{V} \mid rand(z_1 \in \mathbb{Z}, z_2 \in \mathbb{Z}) \mid e_1 + e_2 \mid e_1 - e_2$$

Dans la suite, les variables $v \in \mathcal{V}$ sont des chaînes de caractères.

Question 1. Définir un type OCaml qui

— représente les expressions arithmétiques linéaires, et où,

— chaque noeud interne est annoté par deux valeurs d'un type,

— le type des valeurs d'annotations est un paramètre du type que l'on définit

Solution :
```
type 'a annot_expr =
| Var of string
| Rand of int * int
| Add of 'a * 'a annot_expr * 'a * 'a annot_expr
| Sub of 'a * 'a annot_expr * 'a * 'a annot_expr
```

On suppose dans la suite que l'on possède un module V respectant la signature de module `Val`. À titre d'exemple, le module `Int` est un module de type `Val`.

```
module type Val = sig
  type t

  val rand : int -> int -> t
  val add : t -> t -> t
  val sub : t -> t -> t
end
```

```
module Int : Val = struct
  type t = int
  (* Random.int_in_range renvoie un entier
     compris entre i1 et i2 *)
  let rand i1 i2 = Random.int_in_range i1 i2
  let add i1 i2 = i1 + i2
  let sub i1 i2 = i1 - i2
end
```

On définit un module de table d'association dont les clés sont des chaînes de caractères `module VarMap = Map.Make(String)`, dont une partie de l'interface est donnée ci-dessous.

```
module VarMap :
  sig
    type key = string
    type 'a t = 'a Map.Make(String).t
    val add : key -> 'a -> 'a t -> 'a t
    val find : key -> 'a t -> 'a
    val empty : 'a t
  end
```

Ainsi `VarMap.add "x" 1 VarMap.empty` est la table d'association qui lie à la variable x l'entier 1.

Question 2.

(a) Définir une fonction `eval`, qui annote chaque noeud de l'expression e par son interprétation dans un environnement de type `V.t VarMap.t`.

(b) Sous quelle(s) condition(s) `eval` est-elle bien définie ?

Solution : On peut admettre qu'il existe un type `expr`, mais on peut aussi bien utiliser `unit annot_expr` ou `'a annot_expr` en entrée.

```
let rec eval (env: V.t VarMap.t) (e: expr) : V.t * (V.t annot_expr) =
match e with
| Var v ->
    let va = VarMap.find v env in
    va, Var v
| Rand(z1, z2) ->
    let va = V.rand z1 z2 in
    va, Rand (z1, z2)
| Add(_, e1, _, e2) ->
    let v1, e1 = eval e1 env in
    let v2, e2 = eval e2 env in
    let va = V.add v1 v2 in
    va, Add (v1, e1, v2, e2)
| Sub(_, e1, _, e2) ->
    let v1, e1 = eval e1 env in
    let v2, e2 = eval e2 env in
    let va = V.sub v1 v2 in
    va, Sub (v1, e1, v2, e2)
```

Il faut que le domaine de l'environnement contienne toutes les variables de l'expression. La preuve n'est pas demandée.

Question 3. On note $\mathcal{I}$ l'ensemble des intervalles

$$\mathcal{I} = \{[a, b] \,|\, (a, b) \in \mathbb{Z}^2, a \leq b\} \cup \{\bot\}$$

Définir un module `Intervalle` de type `Val` permettant d'interpréter les expressions en intervalles.

Solution :

```
module Intervalle : Val = struct
    type t =
    | Bottom
    | NotBottom of int * int

    let add i1 i2 = match i1, i2 with
    | Bottom, _ | _, Bottom -> Bottom
    | NotBottom(a, b), NotBottom(c, d) ->
        NotBottom(a + c, b + d)

    let sub i1 i2 = match i1, i2 with
    | Bottom, _ | _, Bottom -> Bottom
    | NotBottom(a, b), NotBottom(c, d) ->
        NotBottom(a - d, b - c)
end
```

Ici, on peut noter qu'il y a une différence entre l'objet mathématique (les entiers mathématiques $a, b \in \mathbb{Z}$) et l'implémentation sur des entiers machines (de précision limitée). On pourrait adapter cette modélisation de différentes manières :

1. Utiliser des entiers de précision arbitraire (comme c'est le cas en Python)
2. À minima s'assurer qu'il n'y a pas de dépassement d'entier, et renvoyer l'intervalle [`Int.min_int`, `Int.max_int`] sinon.

Dans la suite, on s'affranchit de cette différence.

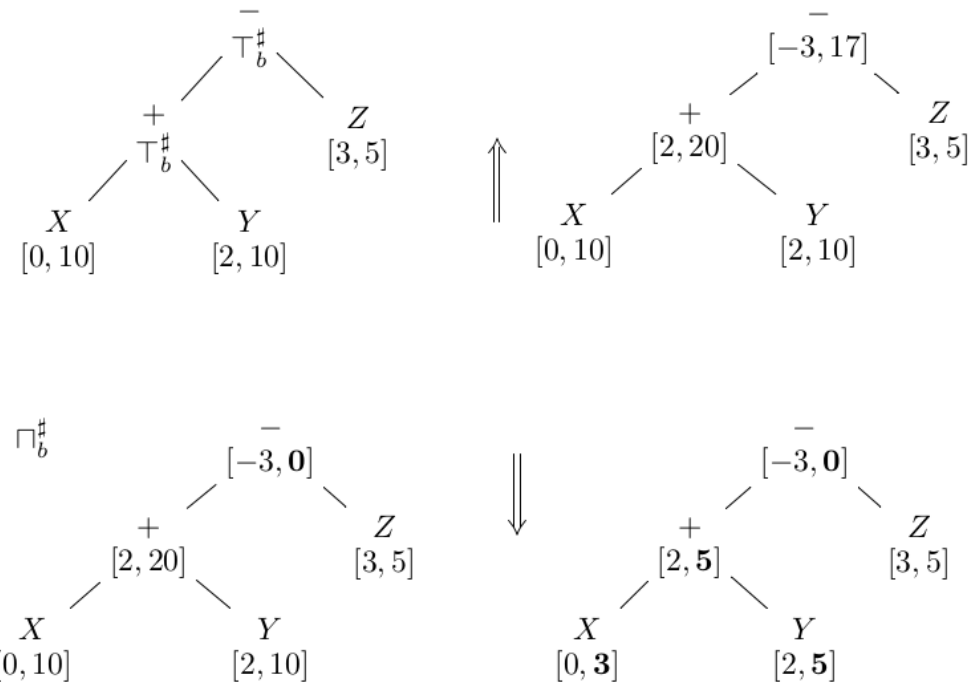
Question 4.

(a) On pose $\Sigma(x) = [0, 10], \Sigma(y) = [2, 10], \Sigma(z) = [3, 5]$. Illustrer l'évaluation de $(x + y) - z$ dans les intervalles.

(b) On suppose que $x + y - z \leq 0$. Que peut-on déduire sur $\Sigma(x), \Sigma(y), \Sigma(z)$?

Solution : Cet exemple est emprunté à "Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation" d'Antoine Miné. La figure ci-dessous reproduit la Figure 4.8 de l'ouvrage ; la première ligne correspond à (a), la seconde à (b). Attention, l'illustration donne les annotations sur les enfants des noeuds, contrairement à l'implémentation.

Pour (b), on commence par raffiner le noeud parent (on intersecte juste avec $[-\infty, 0]$). Il faut ensuite propager en arrière cette information : Sachant que $e_1 - e_2 = r$, avec pour valeurs initiales $e_1 = [2, 20], e_2 = [3, 5], r = [-3, 0]$, on peut déduire que $e_1 = r + e_2 = [0, 5]$. En intersectant avec $e_1 = [2, 20]$, on obtient $e_1 = [2, 5]$. On fait de même pour les autres noeuds, jusqu'à descendre aux feuilles des variables.



Question 5. Soit σ : `Int.t VarMap.t` (où `Int` est le module des entiers OCaml) et Σ : `Intervalle.t VarMap.t`. Quel lien peut-on établir entre `eval e` σ et `eval e` Σ ?

Solution : On introduit $\sqsubseteq$, telle que $\sigma \sqsubseteq \Sigma \Leftrightarrow \forall v \in dom(\sigma), \sigma(v) \in \Sigma(v)$.

Si $\sigma \sqsubseteq \Sigma$, $z = \texttt{eval e } \sigma$, $i = \texttt{eval e } \Sigma$, alors $z \in i$.

Le cas réduit à `Var` et `Add` est prouvé ci-dessous.

Soit σ, Σ tels que $\sigma \sqsubseteq \Sigma$.

Soit $IH(e) = \texttt{eval e } \sigma \in \texttt{eval e } \Sigma$. On prouve $IH(e)$ par induction structurelle sur e.

- Cas de base $e = \texttt{Var}v$. Immédiat par la définition de $\sigma \sqsubseteq \Sigma$.
- Soit $e = \texttt{Add}(e_1, e_2)$, avec $IH(e_1)$ et $IH(e_2)$. $IH(e_j)$ donne $z_j = \texttt{eval e } \sigma \in [l_j, h_j] = \texttt{eval e } \Sigma$. $l_j \leq z_j \leq h_j$, donc $l_1 + l_2 \leq z_1 + z_2 \leq h_1 + h_2$, d'où $z_1 + z_2 \in [l_1, h_1] + [l_2, h_2]$, CQFD.

Question 6. Généraliser le processus de la question (4b) en étendant le module `Intervalle` et en définissant `deduce_negative` telle que `deduce_negative e env` raffine l'environnement des intervalles en supposant que `e` ≤ 0.

Solution :

```
module Intervalle : Val = struct
    (* ... *)

    let inter t1 t2 = match t1, t2 with
    | Bottom, _ | _, Bottom -> Bottom
    | NotBottom(a, b), NotBottom(c, d) ->
      let l = max a c in
      let r = min b d in
      if l <= r then NotBottom(l, r) else Bottom

    (* deduce_add i1 i2 r = (i1', i2') où (i1', i2') sont des
       raffinements de i1, i2 tels que i1' + i2' = r *)
    let deduce_add i1 i2 r =
        inter i1 (sub r i2), inter i2 (sub r i1)

    let deduce_sub i1 i2 r =
        inter i1 (add r i2), inter i2 (sub i1 r)

    (* optionnel *)
    let is_bottom i = i = Bottom
end

let rec refine (e : V.t annot_expr) (env : V.t VarMap.t) (i : V.t) : V.t VarMap.t =
    match e with
    | Var v ->
        let i' = VarMap.find v env in
        let i'' = V.inter i i' in
        if V.is_bottom i'' then
            raise Empty
        else
            VarMap.add v i'' env
    | Rand(a, b) ->
        let i' = V.rand a b in
        let i'' = V.inter i i' in
```

```
        if V.is_bottom i'' then
            raise Empty
        else
            env
    | Add(i1, e1, i2, e2) ->
        let i1', i2' = V.deduce_add i1 i2 i in
        let env = refine e1 env i1' in
        refine e2 env i2'
    | Sub(i1, e1, i2, e2) ->
        let i1', i2' = V.deduce_sub i1 i2 i in
        let env = refine e1 env i1' in
        refine e2 env i2'

let deduce_negative (e: 'a annot_expr) (env: V.t VarMap.t) : V.t VarMap.t =
    let i, annot_e = eval e env in
    refine annot_e env (V.inter i (V.rand Int.min_int 0))
```

Question 7. La fonction `deduce_negative e` est-elle idempotente ?

Solution : Non. Il vaut mieux prendre une inégalité stricte, que l'on peut encoder comme $(x - x) + 1 <= 0$.

En partant de $\sigma_0 = [x \mapsto [-100, 100]]$, on obtient $\sigma_1 = [x \mapsto [-99, 99]]$.

Après `eval e env`, on obtient :

```
Add([-200, 200], Sub([-100, 100], Var "x", [-100, 100], Var "x"),
    [1, 1], Rand(1, 1))
```

On fait ensuite le raffinement, en partant de l'intervalle $[-199, 201] \sqcap [\texttt{min\_int}, 0]$.

Refine sur l'addition appelle `deduce_add` [-200, 200] [1, 1] [-199, 0], l'intervalle de gauche est raffiné de [-200, 200] à [-200, -1].

En descendant récursivement, refine sur la soustraction appelle

$$\begin{aligned}
&\texttt{deduce_sub}[-100, 100][-100, 100][-200, -1] \\
&= [-100, 100] \sqcap [-200, -1] + [-100, 100], [-100, 100] \sqcap [-100, 100] - [-200, -1] \\
&= [-100, 100] \sqcap [-300, 99], [-100, 100] \sqcap [-99, 300] \\
&= [-100, 99], [-99, 100]
\end{aligned}$$

Ce qui raffinera les deux bornes de l'intervalle x.

Si l'on réitère le processus, on obtient $\sigma_2 = [x \mapsto [-98, 98]]$.

Question 8. Comment adapter `deduce_negative` pour le cas des entiers ?
Quelle relation établir entre `deduce_negative` sur les entiers et sur les intervalles ?

Solution : Il faut potentiellement lever une exception `Empty` (ou utiliser un type option) pour les cas où les contraintes ne sont pas satisfiables.

Ensuite, si $\sigma \sqsubseteq \Sigma$, et `deduce_negative env` $\sigma = \sigma'$ (ie, que l'exception n'est pas levée) alors :

1. `deduce_negative env` $\Sigma = \Sigma'$
2. $\sigma' \sqsubseteq \Sigma'$

Circuits et formules

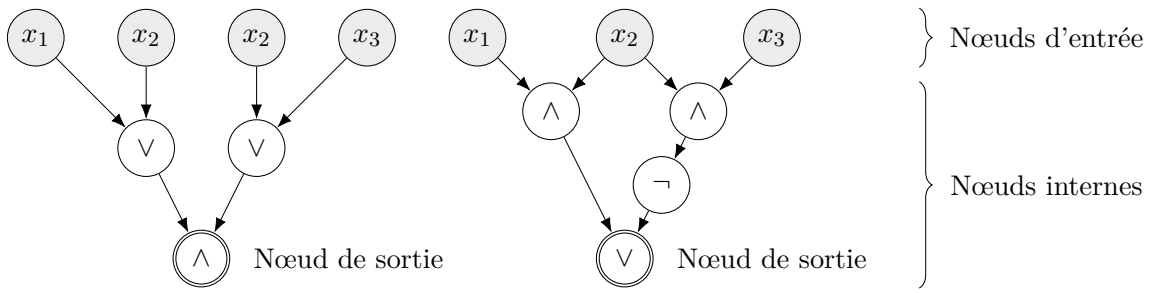
Dans ce sujet, on considère toujours des graphes simples : c'est-à-dire qu'il y a au plus une arête depuis un sommet vers un autre, et jamais d'arête d'un sommet vers lui-même.

Graphes acycliques. Un graphe *orienté* est acyclique s'il ne contient pas de sommets formant un cycle orienté $s_1 \to s_2 \to \cdots \to s_k \to s_1$. Un graphe *non-orienté* est acyclique s'il ne contient pas de sommets formant un cycle non-orienté $s_1 - s_2 - \cdots - s_k - s_1$.
Circuit. Un *circuit* n-aire $C(x_1, x_2, \ldots, x_n)$ est un graphe orienté acyclique avec deux types de nœuds.

1. Les *nœuds d'entrée* ont un degré entrant 0, et pour label soit une variable x_i, soit 0, soit 1.
2. Les *nœuds internes* ont un degré entrant 1 et le label $\neg$, ou un degré entrant 2 et le label $\wedge$ ou $\vee$.

De plus, le graphe contient un *unique* nœud de degré sortant 0, appelé *nœud de sortie*.
Circuit formulaire. Un circuit est dit *formulaire* si pout tout nœud, il existe un unique chemin orienté de ce nœud vers le nœud de sortie. Le *rang* d'un circuit formulaire C, noté $\mathsf{rang}(C)$, est le nombre de nœuds d'entrée de C.
Fonction calculée par un circuit. À un circuit $C(x_1, \ldots, x_n)$, on associe une fonction $\mathsf{F}(C) : \{0,1\}^n \to \{0,1\}$, définie inductivement de la manière naturelle. On dit que le circuit *calcule* cette fonction.

Exemples :



Le circuit de gauche est formulaire de rang 4, et calcule la fonction $(x_1, x_2, x_3) \mapsto (x_1 \vee x_2) \wedge (x_2 \vee x_3)$. Le circuit de droite n'est pas formulaire, et calcule la fonction $(x_1, x_2, x_3) \mapsto (x_1 \wedge x_2) \vee \neg(x_2 \wedge x_3)$.

Question 1. Montrer que pour toute fonction $f : \{0,1\}^n \to \{0,1\}$, il existe un circuit qui calcule cette fonction.

Question 2. On définit la fonction de parité $\mathsf{PAR}_n : \{0,1\}^n \to \{0,1\}$:

$$\mathsf{PAR}_n(x_1, \ldots, x_n) = x_1 + \cdots + x_n \bmod 2.$$

a. Construire un circuit formulaire de rang 4 calculant PAR_2.
b. Construire un circuit formulaire de rang $\mathcal{O}(n^2)$ calculant PAR_n.

Question 3. Soit C un circuit quelconque. Soit $\mathsf{no}(C)$ le graphe non-orienté obtenu en effaçant les orientations des arêtes de C.

a. Montrer que $\mathsf{no}(C)$ est connexe.
Indication : utiliser l'unicité du sommet de degré sortant 0.
b. Montrer que si $\mathsf{no}(C)$ est acyclique, alors C est formulaire. Qu'en est-il de la réciproque ?

Rang d'une fonction. Le *rang* d'une fonction $f : \{0,1\}^n \to \{0,1\}$, noté $\mathsf{rang}(f)$, est le rang minimum d'un circuit formulaire calculant f.
Matrices. Étant donné deux ensembles non-vides A et B, on note $\mathbb{R}^{A\times B}$ l'ensemble des matrices à coefficients dans $\mathbb{R}$, dont les lignes sont indexées par A, et les colonnes par B. Si $M \in \mathbb{R}^{A\times B}$, on note $M_{a,b}$ le coefficient à l'intersection de la ligne a et la colonne b.
Distance de Hamming. Étant donné deux vecteurs $v = (v_1, \ldots, v_n)$ et $w = (w_1, \ldots, w_n)$ dans $\{0,1\}^n$, la distance de Hamming entre v et w est $H(v,w) = |\{i : v_i \neq w_i\}|$.
Produit carré. Soit $A, B \subseteq \{0,1\}^n$. On note $A \boxtimes B \in \mathbb{R}^{A\times B}$, la matrice définie par $(A \boxtimes B)_{a,b} = 1$ si $H(a,b) = 1$, 0 sinon.

Soit f une fonction, et C un circuit formulaire de rang minimal calculant f.

Soit $A \subseteq f^{-1}(0)$, $B \subseteq f^{-1}(1)$ des sous-ensembles non vides. On note $M = A \boxtimes B$.

Question 4. Supposons que C est de la forme $C = C_1 \wedge C_2$. Soit $f_1 = \mathsf{F}(C_1)$, $f_2 = \mathsf{F}(C_2)$.

a. On note $A_1 = f_1^{-1}(0)$, et $A_2 = A \setminus A_1$. Montrer $A_2 \subseteq f_2^{-1}(0)$.

b. Soit $M_1 = A_1 \boxtimes B$, $M_2 = A_2 \boxtimes B$. Montrer $M^\mathsf{T} M = M_1^\mathsf{T} M_1 + M_2^\mathsf{T} M_2$.

Valeurs propres. Si S est une matrice symétrique réelle, on note $\lambda(S)$ sa plus grande valeur propre. On admet que si S et T sont deux matrices symétriques réelles, alors $\lambda(S+T) \leq \lambda(S) + \lambda(T)$.

Question 5. Montrer $\mathsf{rang}(f) \geq \lambda(M^\mathsf{T} M)$.

Indication : procéder par induction sur C.

Question 6. Montrer :

$$\mathsf{rang}(\mathsf{PAR}_n) \geq n^2.$$

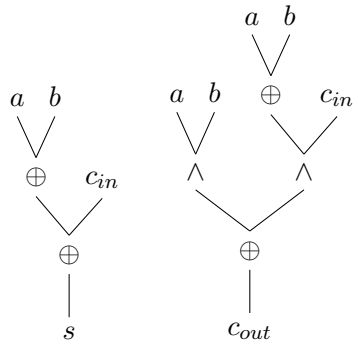
Profondeur des circuits Booléens

Soit $x_0, \ldots, x_{n-1}$ et $y_0, \ldots, y_{m-1}$ des variables Booléennes. Un *circuit Booléen* sur les entrées $x_0, \ldots, x_{n-1}$ et sorties $y_0, \ldots, y_{m-1}$ est une forêt de m arbres. Les feuilles sont étiquetées par des variables de $\{x_0, \ldots, x_{n-1}\}$ ou des constantes de $\{0, 1\}$. Chaque racine est étiquetée par une et une seule variable de $\{y_0, \ldots, y_{m-1}\}$. Les nœuds internes sont étiquetés par des opérations $\neg$ (de degré 1), $\wedge, \vee$ ou $\oplus$ (de degré 2). On rappelle que le ou exclusif (XOR) a la définition : $x \oplus y = (x \wedge \neg y) \vee (\neg x \wedge y)$, et que le $\wedge$ est distributif sur $\oplus$: $(a \oplus b) \wedge c = (a \wedge c) \oplus (b \wedge c)$.

On dit qu'un circuit Booléen $\mathcal{T}$ *calcule* une fonction $f_{\mathcal{T}} : \{0,1\}^n \to \{0,1\}^m$, qui est définie inductivement en évaluant tous les nœuds du circuit à 0 ou 1 : les feuilles sont évaluées selon les valeurs des variables d'entrée et les nœuds internes selon les valeurs de leurs enfants.
À titre d'exemple, le circuit ci-dessous, noté ADD, comporte trois entrées a, b, c_{in} et calcule deux sorties :

$$\begin{cases} s := a \oplus b \oplus c_{in} \\ c_{out} := (a \wedge b) \oplus ((a \oplus b) \wedge c_{in}) \end{cases}$$

La *taille* d'un circuit est son nombre de nœuds. Sa *profondeur* est la hauteur maximale des arbres de la forêt. Une feuille étant de hauteur 0, le circuit ADD est de profondeur 4.



Étant donné deux circuits $\mathcal{T}_1$ et $\mathcal{T}_2$ tels que les sorties de $\mathcal{T}_1$ sont les entrées de $\mathcal{T}_2$, on peut *composer* les circuits $\mathcal{T}_1$ et $\mathcal{T}_2$ en remplaçant chaque occurrence des entrées de $\mathcal{T}_2$ par l'arbre correspondant de $\mathcal{T}_1$. La fonction d'évaluation du circuit ainsi obtenu est alors $f_{\mathcal{T}_2} \circ f_{\mathcal{T}_1}$.

Question 1. Que calcule le circuit ci-dessus ?

Solution : En faisant une table de vérité, on peut remarquer que le circuit est un additionneur à retenue des deux entrées Booléennes a, b. Il y a une retenue d'entrée c_{in} et une retenue de sortie c_{out}.

a	b	c_{in}	$a \wedge b$	$a \oplus b$	$(a \oplus b) \wedge c_{in}$	$s = a \oplus b \oplus c_{in}$	$c_{out} = (a \wedge b) \oplus ((a \oplus b) \wedge c_{in})$
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	1	0	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	0	1	0
1	0	1	0	1	1	0	1
1	1	0	1	0	0	0	1
1	1	1	1	0	0	1	1

En effet, c_{out} vaut 1 si et seulement si $a + b + c_{in} \geq 2$, cas où il y a une retenue. Cette remarque est utile pour les questions suivantes puisque ADD est un composant important des circuits d'addition.

Question 2. On s'intéresse dans cette question à l'addition de deux entiers de n bits écrits en binaire :

$$A := \overline{a_0 a_1 \dots a_{n-1}}^2 \text{ et } B := \overline{b_0 b_1 \dots b_{n-1}}^2$$

Montrer qu'il existe un circuit Booléen de profondeur $\mathcal{O}(n)$ calculant les bits de $(A + B)$.

Solution : L'addition binaire sur n bits se calcule avec une séquence de circuits ADD, qui correspondent à l'addition de deux bits a_i, b_i et de la retenue.

La manière la plus convaincante de procéder est de montrer par induction sur i que le i-ème bit de la sortie se calcule avec un circuit de profondeur $\leq 4i$. En effet, si $\mathcal{T}_i$ est le circuit qui calcule le i-ème bit de la sortie et la retenue correspondante, alors $\mathcal{T}_{i+1}$ peut être obtenu en combinant la sortie de $\mathcal{T}_i$ et le circuit ADD, ce qui n'augmente la profondeur que de 4.

Question 3. Pour tout $0 \leq i \leq n-1$, notons $g_i = a_i \wedge b_i$, $p_i = a_i \oplus b_i$, et c_i la i-ème retenue dans l'addition binaire de A et B (celle qui apparaît à l'addition du i-ème bit, c_0 étant donc la première retenue).

1. Donner une expression de c_i en fonction des g_j et p_j uniquement.
2. En déduire qu'il existe un circuit Booléen de profondeur $\mathcal{O}(\log n)$ pour calculer l'addition binaire.

Solution : 1. On procède par induction sur i. La première retenue c_0 a pour expression $a_0 \wedge b_0 = g_0$ (ce qui correspond à une entrée c_{in} égale à 0 dans le circuit ADD).

Ensuite, la $i+1$-ème retenue c_{i+1} a pour expression :

$$c_{i+1} = g_{i+1} \oplus p_{i+1} \wedge c_i$$

en utilisant le circuit ADD. Par exemple on aura :

$$\begin{cases} c_1 = g_1 \oplus (p_1 \wedge c_0) = g_1 \oplus (p_1 \wedge g_0) \\ c_2 = g_2 \oplus (p_2 \wedge c_1) = g_2 \oplus (p_2 \wedge g_1) \oplus (p_2 \wedge p_1 \wedge g_0) \end{cases} .$$

L'expression générale de c_i se démontre alors par récurrence sur i :

$$c_i = \bigoplus_{j=0}^{i} g_j \wedge \left(\bigwedge_{k=j+1}^{i} p_k \right) .$$

2. Comme on a une expression pour c_i, on peut aussi en déduire une expression pour les bits s_i de la somme :

$$s_i = a_i \oplus b_i \oplus c_i$$

et le dernier bit $s_n = c_{n-1}$ (cas important à ne pas oublier, car la somme de deux entiers sur n bits comporte $n+1$ bits).

Pour montrer que chaque s_i se calcule en profondeur $\mathcal{O}(\log n)$, il suffit donc de le faire pour chaque c_i. On construit l'arbre en deux niveaux : en partant des p_j, g_j, on fait des arbres binaires de AND pour calculer $g_j \wedge (\bigwedge_{k=j+1}^{i} p_k)$. Il y a moins de n termes, donc les arbres ont profondeur $\mathcal{O}(\log n)$. Ensuite, on fait un arbre binaire dans lequel tous les noeuds sont étiquetés par $\oplus$. Comme il y a moins de n termes dans le XOR, cet arbre a profondeur $\mathcal{O}(\log n)$: au total la profondeur est donc toujours $\mathcal{O}(\log n)$.

Question 4. Soit $\varepsilon > 0$ une constante. Est-il possible de calculer la somme $A + B$ en profondeur $\leq (1-\varepsilon)\log n$?

Solution : Non, et prouvons-le par l'absurde.

On va s'intéresser au dernier bit de la somme. (On ne peut pas ici prendre la forêt complète, car elle contient $n+1$ arbres, et cela ne conduirait pas à une contradiction). Supposons qu'il est calculé par un arbre de profondeur $(1-\varepsilon)\log n$. Cet arbre contient $\mathcal{O}\left(2^{(1-\varepsilon)\log n}\right) = \mathcal{O}\left(n^{1-\varepsilon}\right)$ feuilles. Il existe donc au moins un bit parmi les a_i ou b_i qui n'intervient pas dans le calcul. Nous montrons que cela conduit à une contradiction.

Supposons sans perte de généralité que c'est le bit a_i qui n'intervient pas. Nous considérons l'entrée suivante :

— Tous les bits de a sont à 0, sauf a_i, c'est-à-dire $A = 2^i$

— Tous les bits de b sont à 1, c'est-à-dire $B = 2^n - 1$

Si $a_i = 1$, on a $A + B \geq 2^n$, donc le dernier bit de la somme vaut 1. Si $a_i = 0$, il vaut 0. Or, le circuit ne dépendant pas de a_i, il ne peut renvoyer que deux valeurs égales dans ces deux cas : c'est une contradiction. (Autrement dit le calcul n'est pas correct).

Question 5. On s'intéresse dans cette question à l'addition de m entiers de n bits écrits en binaire, notés $A_0, \ldots, A_{m-1}$.

En utilisant le circuit ADD à bon escient, montrer qu'il existe un circuit de profondeur $\mathcal{O}(\log m + \log n)$ calculant les bits de $(A_0 + \ldots + A_{m-1})$.

Solution : C'est une question plus difficile. L'attendu ici n'est pas de détailler un circuit au bit près, mais plutôt d'en donner l'idée générale et d'utiliser la composition à bon escient.

Il faut d'abord réécrire la somme :

$$A_0 + \ldots + A_{m-1} = \sum_{i,j} a_i^j 2^i$$

où a_i^j sont les bits des A_j. Regrouper ensuite selon la puissance de 2 :

$$A_0 + \ldots + A_{m-1} = \sum_i \left(\sum_{j=0}^{m-1} a_i^j 2^i \right) \quad .$$

On considère d'abord ces groupes de m bits. On utilise des circuits ADD pour réduire la taille des sommes : une première couche de ADD réduit des groupes de trois bits de la forme $a_0 2^i + a_1 2^i + a_2 2^i$ pour obtenir $b_0 2^i + b_1 2^{i+1}$.

On réordonne ces bits de sortie en fonction des puissances de 2 correspondantes (tout ceci est fixe dans le circuit). On obtient donc de nouveau une somme de la forme :

$$\sum_i \left(\sum_{j=0}^{2(m-1)/3} a_i^j 2^i \right)$$

mais dans laquelle l'indice i maximal a augmenté de 1, et on n'a plus que $2(m-1)/3$ éléments par somme intermédiaire (un venant du i inférieur, un venant du i actuel).

On voit qu'en répétant ce processus un nombre $\mathcal{O}(\log m)$ de fois, on aboutit à une somme de la forme :

$$\sum_{i=0}^{n+\mathcal{O}(\log m)} (e_i + f_i) 2^i \quad .$$

Il n'est pas possible, ni nécessaire, de réduire davantage : à ce stade les e_i et f_i sont stockés dans des bits de sortie de notre circuit. On peut donc composer avec une addition de deux entiers à $n + \mathcal{O}(\log m)$ bits, qui a profondeur $\mathcal{O}(\log n)$ d'après la question précédente.

Question 6. En déduire qu'il existe un circuit de profondeur $\mathcal{O}(\log n)$ pour la multiplication de deux entiers de n bits.

Solution : Un produit de deux entiers :

$$(a_0 + 2a_1 + 4a_2 + \ldots)B = a_0B + 2a_1B + \ldots$$

se réduit à une somme de n entiers dont les bits peuvent être calculés en profondeur constante (par de simples AND). On a donc bien un circuit de profondeur $\mathcal{O}(\log n + \log n) = \mathcal{O}(\log n)$.

Annexe : sujets proposés pour la filière MPI

Certains sujets sont accompagnés d'*ébauches* de solution.

Filtrage par motif en OCaml

Ce sujet s'intéresse à modéliser le comportement du filtrage par motif utilisé dans la construction `match ... with ...` d'OCaml.

On s'intéresse à un sous-ensemble d'OCaml. Les constructeurs des types algébriques sont notés $C(e_1, \ldots, e_k)$, où k est l'arité du constructeur C. Dans l'exemple des listes ci-dessous, le premier constructeur de liste (`Empty`) est d'arité 0, le second (`Cons`) est d'arité 2.

```
type 'a list = Empty | Cons of 'a * 'a list
```

Dans la suite, on considère que toutes les valeurs OCaml sont des constructeurs $v ::= C(v_1, \ldots, v_k)$. En particulier, les constantes `true` et `false` sont des valeurs du `type bool = true | false`.

Les motifs sont $m ::= \texttt{v} \mid \_ \mid C(m_1, \ldots, m_k) \mid (m_1|m_2)$, i.e :

- Des identifiants de variables (cas `v`).
- Le motif joker `_`.
- Un constructeur appliqué à k motifs.
- La disjonction de deux motifs.

On introduit une notion de matrice de filtrage, selon la transformation illustrée ci-dessous. Les expressions à droite des motifs (a_i) sont appelées des actions.

```
match (e1, ..., em) with
   | m1,1, ..., m1,m -> a1
   | ...
   | mn,1, ..., mn,m -> an
```

$$\rightarrow \quad \begin{pmatrix} e_1 & \ldots & e_m & & \\ m_{1,1} & \ldots & m_{1,m} & \rightarrow & a_1 \\ \ldots & \ldots & \ldots & \rightarrow & \ldots \\ m_{n,1} & \ldots & m_{n,m} & \rightarrow & a_n \end{pmatrix}$$

Étant donné une expression $e = C(e_1, \ldots, e_k)$, on définit des opérateurs d'accès au constructeur : $\text{constr}(e) = C$ et d'accès au i-ème champ : $\#i(e) = e_i$ (si $1 \leq i \leq k$).
Un *environnement* est une fonction partielle des variables aux valeurs.

Question 1. Définir une relation $m \leq_\sigma v$ établissant la compatibilité entre un motif m et une valeur v, étant donné un environnement σ.

À titre d'exemple, $\texttt{Cons}(1, x) \leq_\sigma \texttt{Cons}(1, \texttt{Cons}(2, \texttt{Empty}))$ lorsque $\sigma(x) = \texttt{Cons}(2, \texttt{Empty})$.

Dans la suite, on note $m \leq v$ si et seulement si $\exists \sigma, m \leq_\sigma v$.

Solution :

- $x \leq_\sigma v \Leftrightarrow \sigma(x) = v$
- $\_ \leq_\sigma v$
- $C(m_1, \ldots, m_n) \leq_\sigma C'(v_1, \ldots, v_o) \Leftrightarrow C = C' \wedge n = o \wedge \forall i, m_i \leq_\sigma v_i$
- $(m_1 \mid m_2) \leq_\sigma v \Leftrightarrow m_1 \leq_\sigma v \vee m_2 \leq_\sigma v$

Question 2. On note $\text{Match}((v_1, \ldots, v_m), M)$ le résultat du filtrage de la matrice M sur le vecteur de valeurs $(v_1, \ldots, v_m)$.

(a) Soit a_i une action et σ un environnement. Sous quelle(s) condition(s) $\text{Match}((v_1, \ldots, v_m), M) = (a_i, \sigma)$?

(b) Y a-t-il d'autres cas de définition de Match ?

Solution : Il faut que $\forall 1 \leq j < i, (v_1, \ldots, v_m) \not\sqsubseteq (m_{j,1}, \ldots, m_{j,m})$, et que $(v_1, \ldots, v_m) \sqsubseteq_\sigma (m_{i,1}, \ldots, m_{i,m})$.
Ne pas oublier le cas où $\forall 1 \leq i \leq n, (v_1, \ldots, v_m) \not\sqsubseteq (m_{i,1}, \ldots, m_{i,m})$, dans ce cas une exception est levée (`Matching_failure` en OCaml).

Question 3.

(a) Décrire informellement comment éliminer le motif joker `_` sans ajouter de cas, sur l'exemple de `len` ci-dessous.

```
let rec len l =
  match l with
  | Empty -> 0
  | Cons(_, tl) -> 1 + (len tl)
```

(b) Définir cette transformation sur les matrices de filtrage.

(c) Donner un critère justifiant de la correction de la transformation.

Solution :

(a) Il faut juste ajouter des variables fraîches, c'est à dire des variables qui ne sont pas les variables libres de l'action à droite de `->` (la notion de variable libre est au programme, mais du côté logique), ou les variables liées par le motif de la ligne.
Dans l'exemple, il faut donc choisir une variable qui n'est ni `l` ni `tl`.

```
let rec len l =
match l with
| Empty -> 0
| Cons(w, tl) -> 1 + (len tl)
```

(b) On note $NJ(M)$ cette transformation. `variables` est définie récursivement pour extraire les variables définies dans un motif.
$NJ(M)_{i,j} = \begin{cases} M_{i,j} \text{ si } M_{i,j} \neq \_ \\ v \text{ tel que } v \notin \texttt{variables\_libres}(a_i) \text{ et } v \notin \{\texttt{variables}(m_{i,k}) \mid k \neq j\} \end{cases}$

(c) Il faut prouver que $\text{Match}((v_1, \ldots, v_m), M) = \text{Match}((v_1, \ldots, v_m), NJ(M))$. On n'exigeait pas une preuve détaillée ici.

Question 4. Soit F une matrice de filtrage. On note $S(c, F)$ l'opérateur qui transforme la matrice de filtrage en supposant que e_1 (la première expression filtrée par F) commence par un constructeur c d'arité a.

(a) Illustrer le résultat de $S(\texttt{Cons}, F)$ sur la matrice de filtrage induite par le code ci-dessous.

```
let rec merge l1 l2 = match l1, l2 with
| Empty, l | l, Empty -> l
| Cons(h1, t1), Cons(h2, t2) ->
  if h1 < h2 then Cons(h1, merge t1 l2)
  else Cons(h2, merge l1 t2)
```

(b) Décrire la transformation opérée par $S(c, F)$.

(c) Donner un critère justifiant de la correction de la transformation.

Solution :

(a) Petit piège, ce n'est pas un cas de | dans les motifs ici, la matrice de filtrage initiale est donc

$$F = \begin{pmatrix} l1 & l2 & & \\ Empty & l & \to & l \\ l & Empty & \to & l \\ Cons(h1,t1) & Cons(h2,t2) & \to & ... \end{pmatrix} \qquad S(Cons,F) = \begin{pmatrix} \#1(l1) & \#2(l1) & l2 & & \\ \_ & \_ & Empty & \to & l \\ h1 & t1 & Cons(h2,t2) & \to & ... \end{pmatrix}$$

(b) N.B : formaliser la transformation sur les indices des matrices n'est pas pratique ici, les candidat.e.s ont été évalués sur leurs choix de présentation pédagogique de cette étape.
On commence par changer $(e_1, \ldots, e_m)$ en $\#1(e_1), \ldots, \#a(e_1), e_2, \ldots, e_m$.
On raisonne par cas sur $m_{i,1}$ pour donner $M' \to A'$. Chaque ligne peut donner au plus deux lignes correspondantes dans la matrice de filtrage de $S(c, F)$.

— Si $m_{i,1} = c(q_1, \ldots, q_a)$, la ligne conservée est étendue à gauche pour donner

$$q_1, \ldots, q_a, m_{i,2}, \ldots, m_{i,m} \to a_i$$

— Si $m_{i,1} = c'(q_1, \ldots q_b)$ avec $c \neq c'$, la ligne est supprimée

— Si $m_{i,1} = v$ (une variable), il n'y a rien à matcher dans l'extension à gauche, et il faut étendre l'action pour lier la variable

$$\_, \ldots, \_, m_{i,2}, \ldots, m_{i,m} \to \texttt{let } v = e_1 \texttt{ in } a_i$$

— Le cas _ est similaire, mais il n'y a même pas besoin de faire de liaison.

— Si $m_{i,1} = q_1 | q_2$, on génère deux lignes :

$$S(c, (q_1, m_{i,2}, \ldots, m_{i,m} \to a_i))$$
$$S(c, (q_2, m_{i,2}, \ldots, m_{i,m} \to a_i))$$

(c)

$$\text{Match}((c(w_1, \ldots, w_a), v_2, \ldots, v_m), F) = \text{Match}((w_1, \ldots, w_a, v_2, \ldots, v_m), S(c, F))$$

Question 5. Donner une fonction C permettant de transformer ces matrices de filtrage vers un langage OCaml modifié, où les filtrages `match ... with ...` sont supprimés au profit de `switch`es sur le constructeur d'une expression :

```
switch constr(e) with
| case ... -> ...
| ...
| default -> ...
```

Solution :

— Si $n = 0$,

$$C(F) = C\begin{pmatrix} e_1 & \ldots & e_m & \end{pmatrix} = \texttt{failwith Matching\_failure}$$

— Si $m = 0$,

$$C(F) = C\begin{pmatrix} \to & a_1 \\ \to & \ldots \\ \to & a_n \end{pmatrix} = a_1$$

— Ce cas peut être inclus dans le suivant, mais avoir un cas particulier permet d'éviter un switch inutile.
Si la première colonne ne contient que des variables ou des _ (pour simplifier, on suppose que ce sont des variables, cf. question 1), ie $m_{\cdot,1} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$:

$$C\begin{pmatrix} e_1 & \dots & e_m & & \\ x_1 & \dots & m_{1,m} & \to & a_1 \\ \dots & \dots & \dots & \to & \dots \\ x_n & \dots & m_{n,m} & \to & a_n \end{pmatrix} = C\begin{pmatrix} e_2 & \dots & e_m & & \\ m_{1,2} & \dots & m_{1,m} & \to & \texttt{let x\_1 = e\_1 in } a_1 \\ \dots & \dots & \dots & \to & \dots \\ m_{n,2} & \dots & m_{n,m} & \to & \texttt{let x\_n = e\_1 in } a_n \end{pmatrix}$$

— sinon, on note Constrs $= \{c \mid m_{i,1} = c(\dots)\}$ l'ensemble des constructeurs apparaissant dans les motifs de la première colonne.

$$C(F) = \begin{array}{l} \texttt{switch constr(}e_1\texttt{) with} \\ \quad \texttt{case } c_1 \texttt{ -> } C(S(c_1, F)) \\ \quad \dots \\ \quad \texttt{case } c_l \texttt{ -> } C(S(c_l, F)) \\ \quad \texttt{default } \texttt{ -> } C(D(F)) \end{array}$$

L'opérateur de défaut est conceptuellement similaire à celui de spécialisation. Les lignes avec constructeurs sont supprimées. Les lignes avec variables ou wildcard sont laissées, et les | sont déroulés. La propriété est que pour tout constructeur c qui n'apparait pas en tête d'un motif dans la première colonne, alors

$$\text{Match}((c(w_1, \dots, w_a), v_2, \dots, v_m), F) = \text{Match}((v_2, \dots, v_m), D(F))$$

Question 6.

(a) Cette transformation a-t-elle un intérêt ?

(b) L'algorithme de transformation peut-il être amélioré ?

Solution :

(a) Oui pour compiler OCaml.

(b) Dans le cadre d'OCaml, il est intéressant de s'intéresser aux gains qu'apportent les informations de type. Cela peut être une passe faite sur le switch, après sa génération.

Un seul constructeur : pas besoin de faire de test via un `switch`

Suppression du otherwise : lorsque tous les constructeurs sont déjà couverts

Chaînes de caractères : ce cas est très à la marge, mais il peut y avoir une compilation vers des arbres de switches sur les caractères.

Question 7. Prouver la terminaison et la correction de l'algorithme en question 5.

Solution : Mesure de terminaison : $\sum_{i,j} \text{taille}(m_{i,j})$

$$\begin{aligned} &\text{taille}(v) = \text{taille}(\_) = 1 \\ &\text{taille}(q_1|q_2) = \text{taille}(q_1) + \text{taille}(q_2) \\ &\text{taille}(C(m_1, \dots, m_n)) = 1 + \sum_i \text{taille}(m_i) \end{aligned}$$

La définition de la sémantique de switch est immédiate. On note

“H(s) : si C(F) = s alors
— si Match$((v_1, \ldots, v_m), F) = (a_i, \sigma)$ alors `let x`$_i$ `= w`$_i$ `in` $a_i =$ `let e`$_i =$ `v`$_i$ `in` s
— sinon les deux cas s'évaluent en `Matching_failure`.”

Le plus simple est de faire la preuve par induction sur F (vu comme l'expression match, sinon il n'y a pas de structure pour faire l'induction).

Composition Monadique

On se place dans un langage récursif, purement fonctionnel, avec des définitions de types inductifs et un système de types polymorphes similaires à ceux d'OCaml. On pourra utiliser indifféremment du pseudocode fonctionnel ou de la syntaxe OCaml.
On suppose l'existence :

- d'une fonction identité `id` polymorphe de type $A \to A$;
- d'un opérateur $(\circ) : (A \to B) \to (B \to C) \to (A \to C)$ de composition de fonctions, noté $\circ$ dans sa version infixe ;
- d'un opérateur $\mapsto$ de définition de fonction anonyme (`fun` en OCaml) ;
- un opérateur de reconnaissance de motifs (`match...with` en OCaml) ; et
- de types de bases usuels, comme `int` le type des entiers, `string` le type des chaînes de caractères, et `unit` le type de l'unique élément ().

Un *constructeur de type* est une entité informatique qui prend un type et renvoie un type.
On définit par exemple le constructeur de type `liste` en définissant le type polymorphe `liste` A (où A est une variable de type). A titre d'exemple, on présente une définition de la fonction longueur de type `liste` $A \to$ `int` en pseudocode :

`liste` $A =$ `Nil` | `Cons` A (`liste` A)
`longueur` : `liste` $A \to$ `int`
  `longueur` `Nil` $= 0$
  `longueur` (`Cons` _ qu) $= 1 +$ (`longueur` qu)

---

et en OCaml :

```
type 'a liste = Nil | Cons of ('a * 'a liste)
let rec longueur (l : 'a liste) : int = match l with
        Nil -> 0
      | Cons (_,qu) -> 1 + (longueur qu)
```

Question 1. Définir un constructeur de type `option` tel qu'un élément de `option` A est soit vide, soit un élément de type `A`.

Un constructeur de types `F` est un *foncteur* quand il existe une fonction polymorphe
`fmap` : $(A \to B) \to (F\ A) \to (F\ B)$
qui vérifie les lois suivantes (pour tout f et g) :
`fmap id` $=$ `id`
`fmap` $(f \circ g) =$ (`fmap` f) $\circ$ (`fmap` g)

Attention : Dans ce sujet, on ne demande pas de preuves que les lois sont respectées.

Question 2. Montrer que `option` et `liste` sont des foncteurs.

Un foncteur `M` est une *monade* quand il existe deux fonctions polymorphes
`return` : $A \to (M\ A)$
`bind` : $(M\ A) \to (A \to (M\ B)) \to (M\ B)$
qui vérifient les lois suivantes, pour tout f, g et x :
`bind` (`return` x) $f = f\ x$
`bind` x `return` $= x$
`bind` (`bind` $x\ f$) $g =$ `bind` x $(y \mapsto$ (`bind` $(f\ y)\ g$))

Question 3. Montrer que `option` et `liste` sont des monades.

Pour tout type S, on définit (`etat` S) A comme étant le type polymorphe $S \to (S, A)$
On suppose l'existence d'un type abstrait `memoire` représentant une table d'association entre `string` et `int` avec les deux fonctions suivantes :
`trouve` : `string` $\to$ `memoire` $\to$ (`option int`)
`majour` : `string` $\to$ `int` $\to$ `memoire` $\to$ `memoire`
La première recherche la valeur associée à une clef, la seconde met-à-jour un couple clef-valeur.

Question 4. Montrer que `etat` S est une monade pour tout S.

En déduire l'écriture avec `bind` d'une *procédure* qui prend en entrée trois clefs, récupère successivement deux valeurs d'une mémoire à partir des deux premières clefs puis associe dans la mémoire la somme des deux valeurs récupérée à la troisième clef.

On donne une définition alternative des monades :
Un foncteur `M` est une *monade* quand il existe deux fonctions polymorphes
`return` : $A \to (M\ A)$
`join` : $M\ (M\ A) \to (M\ A)$
qui vérifient les lois suivantes, pour tout f, g et x :
`join` $\circ$ `return` $=$ `id`
`join` $\circ$ (`fmap return`) $=$ `id`
`join` $\circ$ `join` $=$ `join` $\circ$ (`fmap join`)

Question 5. Montrer que les deux définitions sont équivalentes.

Si M_1 et M_2 sont deux constructeurs de types, on définit (`compose` $M_1\ M_2$) $A = M_1\ (M_2\ A)$.

Question 6. Donner une condition suffisante pour que (`compose` $M_1\ M_2$) soit un foncteur.

Question 7. Donner une condition suffisante pour que (`compose` $M_1\ M_2$) soit une monade.

En déduire une nouvelle écriture, plus simple, de la procédure de la question 4.

Résolution d'inéquations linéaires

Soit $X = \{x_1, \ldots, x_n\}$ un ensemble de variables et $S = \{L_1, \ldots, L_m\}$ un ensemble de fonctions linéaires en les variables de X à coefficients rationnels, assimilé à un système d'inéquations :

$$\forall i, L_i(x_1, \ldots, x_n) \leq 0 \ .$$

Si deux équations peuvent se réécrire sous la forme : $L \leq x$ et $x \leq U$ pour $x \in X$, où les expressions L et U ne contiennent pas x, on définit le « x-résultant » de L et U comme l'inéquation : $L \leq U$ (ou de manière équivalente : $L - U \leq 0$).
Les nombres rationnels manipulés dans cet exercice n'étant a priori pas de taille constante, la complexité en temps des algorithmes sera comptée uniquement en opérations rationnelles (additions, multiplications, divisions, etc.)
Notre objectif est de résoudre le problème suivant de *satisfiabilité des inéquations rationnelles* :

> Soit S un ensemble d'inéquations à au plus deux variables, déterminer si S est satisfiable, i.e., s'il existe une valuation $X \to \mathbb{Q}$ satisfaisant toutes les inéquations de S.

Question 1. Le système suivant :

$$\begin{cases} 2x_1 + x_2 + 1 \leq 0 \\ 2x_2 \leq 0 \\ -x_1 \leq 0 \\ -x_2 \leq 0 \end{cases}$$

est-il satisfiable ?

Solution : Non, car on peut en déduire l'inéquation $x_2 \leq -1$ et $-x_2 \leq 0$ ce qui forme une contradiction. Informellement, il est possible de déduire « $-1 \leq 0$ » uniquement à l'aide de combinaisons linéaires des équations de départ.

Soit S un système d'inéquations linéaires, x une variable de S, et $T \subseteq S$ l'ensemble des inéquations ne contenant pas x. Soit S' l'ensemble des x-résultants de paires d'inéquations dans $S \backslash T$. On définit un nouveau système $FM(S, x) := T \cup S'$. L'opération FM est appelée *l'élimination de Fourier-Motzkin.*

Question 2. Soit S un système d'inéquations linéaires et x une variable de S. Montrer que S est satisfiable si et seulement si $FM(S, x)$ l'est.

Solution : Soit S' l'ensemble des x-résultants.

Une des deux implications est facile : si S admet une solution, alors cette solution satisfait toutes les inégalités dans S, et également les x-résultants. Par conséquent elle satisfait aussi le système $S' \cup T$.

Inversement, soit $y_1, \ldots, y_{n-1}$ une solution de $S' \cup T$, on veut montrer qu'on peut l'étendre en une solution $y, y_1, \ldots, y_{n-1}$ de S. Cela revient à choisir la valeur y telle que $y, y_1, \ldots, y_{n-1}$ satisfera toutes les inéquations de S.

Par hypothèse, toutes les inéquations de T sont satisfaites. Notons les autres de la manière suivante : $L_i \leq x$ et $x \leq U_j$ (en supposant que les ensembles des L_i et U_j sont tous deux non vides). Leurs x-résultants sont toutes les inéquations de la forme $L_i \leq U_j$.

Posons maintenant : $y := \max_i L_i(y_1, \ldots, y_{n-1})$. Alors toutes les inéquations $L_i \leq x$ sont satisfaites par $y, y_1, \ldots, y_{n-1}$ par définition de y. De plus pour tout j, $y \leq U_j(y_1, \ldots, y_{n-1})$. En effet, sinon il existerait i et j tels que $L_i(y_1, \ldots, y_{n-1}) > U_j(y_1, \ldots, y_{n-1})$ ce qui contredirait l'hypothèse.

S'il n'existe pas d'inéquation de la forme $L_i \leq x$ (respectivement, de la forme $x \leq U_j$) alors on choisit $y := \min_j U_j(y_1, \ldots, y_{n-1})$ (respectivement, le même y que précédemment).

Question 3. En déduire un algorithme pour la satisfiabilité des inéquations rationnelles (il n'est pas nécessaire d'en faire une description détaillée). Donner une borne simple sur sa complexité en temps.

Solution : C'est un algorithme récursif qui détermine si le système est satisfiable, et renvoie une solution s'il l'est. Le cas terminal est celui dans lequel un x-résultant fait apparaître une inéquation de la forme $a \leq 0$ où $a > 0$, qui est trivialement fausse. Tant qu'aucune inéquation de cette forme n'apparaît, on sélectionne une nouvelle variable et on applique $FM(S, x)$. À la fin on aura éliminé toutes les variables.

La correction de l'algorithme est évidente grâce à la question précédente.

En partant de m inéquations, on considère toutes celles qui contiennent la variable x avec un coefficient négatif (de la forme $L \leq x$) et un coefficient positif (de la forme $x \leq U$). Si k est le nombre d'inéquations du premier type on a au plus $k(m-k) \leq (m/2)^2$ résultants à calculer. Le plus simple est de borner ceci par m^2, et d'en déduire que la complexité est doublement exponentielle.

Dans la suite de l'exercice, on considère des inéquations à *au plus deux variables*. On s'intéresse à l'algorithme suivant.

Entrée : S un système de m inéquations, contenant n variables au total
1 : **pour** $i = 1$ à $\lceil \log_2 n \rceil + 2$ **faire**
2 : Soit $S' := \cup_{x \in X} FM(S, x)$ (si des inéquations apparaissent en double, on les simplifie)
3 : $S \leftarrow S \cup S'$
4 : **Si** une inéquation de S est insatisfiable, renvoyer "insatisfiable"
5 : **fin pour**
6 : Renvoyer "satisfiable"

Nous admettrons la propriété (P) suivante :

(P) Soit S un système d'inéquations en k variables. Si tout sous-ensemble de S avec $k+1$ inéquations est satisfiable, alors S est satisfiable.

Si $V \subseteq X$, nous noterons également S_V la restriction de S aux inéquations ne contenant que des variables de V.

Question 4. Soit S un système insatisfiable minimal, c'est-à-dire tel que tout sous-système $S' \subsetneq S$ est satisfiable.

1. Montrer qu'aucune variable n'apparaît que dans une seule inéquation.
2. Montrer qu'au plus deux variables apparaissent dans trois inéquations ou plus.

Solution : Soit k_i le nombre de variables de X qui apparaissent dans exactement i inéquations. Soit k le nombre de variables de S.

1. On a $k_1 = 0$. En effet, si x est une variable qui n'apparaît que dans une seule inéquation, alors on peut d'abord résoudre le système en enlevant cette inéquation (par minimalité de S, il devient satisfiable), puis en déduire une valeur admissible pour x, ce qui contredirait le fait que S est insatisfiable. Ce cas est donc impossible.

2. Par la propriété (P), S contient au plus $k+1$ contraintes (en effet, s'il contenait $k+2$ contraintes, on pourrait en déduire sa satisfiabilité). Par hypothèse, il y a deux variables par contrainte au plus. On a donc : $\sum_{i \geq 1}(ik_i) \leq 2k+2$.

De plus $\sum k_i = k$ par définition des k_i et $k_1 = 0$. On a donc :

$$\sum_{i\geq 1}(ik_i) = k_1 + 2k_2 + 3k_3 + \ldots \geq 2k_2 + 3(k - k_2)$$

donc : $2k_2 + 3(k - k_2) \leq 2k + 2 \implies k_2 \geq k - 2$ ce qui donne bien $(k - k_2) \leq 2$. Autrement dit, il y a au plus deux variables qui apparaissent dans trois inéquations ou plus.

Question 5. Soit S un système insatisfiable minimal sur un ensemble de $k > 3$ variables X, et soit $S' := \cup_{x\in X} FM(S, x)$.

1. Montrer qu'il existe un ensemble $X' \subseteq X$ de $\lceil k/2 \rceil - 1$ variables telles qu'aucune paire $\{x_1, x_2\}$ de X' n'apparaît dans une inéquation de S.
2. Montrer que $(S \cup S')_{X\setminus X'}$ est insatisfiable.

Solution : 1. Construisons un graphe non dirigé $G = (X, E)$ où les sommets sont les variables et il existe une arête entre x_1 et x_2 s'ils apparaissent ensemble dans une inéquation. Par le résultat de la question précédente, tous les sommets du graphe, sauf au plus deux, sont de degré inférieur ou égal à 2. Enlevons ces deux sommets.

Il nous reste un graphe dirigé de degré 2 à $k-2$ sommets. Un tel graphe est un ensemble de *chaînes* disjointes (on peut l'affirmer ici sans preuve). On peut donc choisir $\lceil (k-2)/2 \rceil$ sommets dans ce graphe tels que deux sommets ne sont pas côte à côte (chaque chaîne de longueur ℓ nous donne $\lceil \ell/2 \rceil$ sommets).

On prend pour X' l'ensemble de ces $\lceil (k-2)/2 \rceil$ variables.

2. Soit $X'' := X\setminus X'$. Quand on utilise l'élimination FM sur une des variables de X' on obtient une seule nouvelle inéquation, qui ne fait pas intervenir de variable de X', et n'intervient donc pas dans de futures éliminations. Donc le système obtenu en éliminant *toutes* les variables de X' est contenu dans $(S \cup S')_{X''}$.

Or S est insatisfiable, donc d'après la question 2 l'élimination des variables de X' produit un nouveau système insatisfiable. Ce système est contenu dans $(S \cup S')_{X''}$, qui par conséquent est également insatisfiable.

Question 6. Soit $f(x) = \lfloor x/2 \rfloor + 1$. On pose $f^0(n) = n$ et on admet que pour tout entier naturel n, $f^{\lceil \log_2 n \rceil}(n) = 2$.

1. Soit S un système insatisfiable en entrée de l'algorithme. Montrer que pour tout $k \geq 1$, à la sortie de la k-ième itération de la boucle, il existe un sous-ensemble $X_k \subseteq X$ de taille $\leq f^k(n)$ tel que S_{X_k} est insatisfiable.
2. En déduire que l'algorithme est correct.
3. Donner une borne simple sur sa complexité (à facteur polynomial près).

Solution : 1. Soit S insatisfiable en entrée de la boucle. On démontre cette propriété par récurrence sur k ; si $k = 0$ on est à l'entrée de l'algorithme et elle est donc vraie.

Soit $k > 0$ et considérons le système $S_{X_k} \subseteq S$ insatisfiable. Il existe un sous-système $T \subseteq S_{X_k}$ qui est insatisfiable et minimal (on peut le construire itérativement en enlevant des inéquations de S_{X_k}), et contient $\leq f^k(n)$ variables.

On calcule S' l'ensemble des résultants de S, et $S \cup S'$ contient donc $T \cup T'$ où T' est l'ensemble des résultants de T. En utilisant la question 5, on sait qu'il existe un sous-ensemble $X_{k+1} \subseteq X_k$ contenant $\leq f^{k+1}(n)$ variables tel que $(T \cup T')_{X_{k+1}}$ est insatisfiable, mais aussi $(S_{X_k} \cup S'_{X_k})_{X_{k+1}}$ et enfin $(S \cup S')_{X_{k+1}}$; en effet on n'a fait ici que (potentiellement) remettre des inéquations dans le système.

2. Les itérations de f convergent vers 2. Si le système S initial est insatisfiable, on obtient un ensemble de deux variables (x, y) tel que $S_{(x,y)}$ est insatisfiable. Pour conclure, il suffit de remarquer que les deux dernières itérations sur $S_{(x,y)}$ vont contenir la résolution FM par x suivi de y. Par conséquent on obtiendra une contradiction immédiate.

Si le système S est satisfiable, on ne trouvera pas de contradiction et l'algorithme renverra donc le bon résultat.

3. On n'a que $\lceil \log_2 n \rceil + 2$ étapes dans l'algorithme, et à chaque étape le nombre d'inéquations grandit comme : $T_{i+1} \leq T_i + \frac{T_i(T_i-1)}{2} = \frac{T_i(T_i+1)}{2} \leq T_i^2$. Par conséquent $\log_2 T_i = 2^i \log_2 m$ et le nombre d'inéquations à la dernière étape est borné par : $2^{\log_2 m 2^{\lceil \log_2 n \rceil + 2}} = \widetilde{\mathcal{O}}(m^n)$.

On s'autorise à modifier l'algorithme en ajoutant entre l'étape 4 et 5 : $S \leftarrow \mathsf{Simplifie}(S)$, où Simplifie est une opération transformant S en un système équivalent avec moins d'inéquations.

Question 7. On considère maintenant le cas particulier où les coefficients des équations sont $+1, -1$ ou 0. En utilisant un choix opportun pour Simplifie, proposer un algorithme en temps polynomial en n.

Solution : On commence par démontrer par induction que toutes les nouvelles inéquations produites par l'algorithme sont aussi à coefficients dans $\{0, -1, 1\}$. En effet le x-résultant entre deux inéquations à coefficients dans $\{0, -1, 1\}$ est aussi à coefficients dans $\{0, -1, 1\}$.

Pour chaque paire de variables (x, y), il n'y a donc que 3×3 choix possibles pour les coefficients de x et y (sachant que dans le cas $(0, 0)$ on tombe sur une inéquation triviale). Cela fait donc $n(n-1)/2 \times 9$ configurations possibles de la forme $b_1 x_1 + b_2 x_2 \leq a$.

La procédure Simplifie va classer les inéquations en fonction de leur configuration et sélectionne le coefficient a minimal pour chacune d'entre elles : les autres inéquations ainsi enlevées sont évidemment redondantes. Concrètement, à chaque étape de la boucle, Simplifie ramène le nombre d'inéquations en-dessous de $\mathcal{O}\left(n^2\right)$.

Après application de Simplifie, chaque variable x apparaît au plus dans $9n$ équations (c'est une borne très large) : on ne peut donc ajouter qu'au maximum $\mathcal{O}\left(n^2\right)$ x-résultants, soit $\mathcal{O}\left(n^3\right)$ nouvelles équations au total. Simplifie peut être implémentée en temps $\mathcal{O}\left(n^3\right)$ à l'aide d'une table de hachage. Par conséquent l'algorithme demande $\widetilde{\mathcal{O}}\left(n^3\right)$ temps et $\mathcal{O}\left(n^2\right)$ mémoire.

Terminaison de $\lambda_{\texttt{ref}}$

Le λ-calcul, langage formel modélisant la programmation fonctionnelle, est défini par la syntaxe :
$M, N ::= x \mid M\ N \mid \lambda x.M \mid ()$
où x est issu d'un ensemble infini de *variables de termes*.
Ainsi, un terme M est :

- soit une *variable* x,
- soit une *abstraction* $\lambda x.M$ (qu'il faut comprendre comme la fonction qui au paramètre x associe le terme M), on dit dans ce cas que la variable x est *liée* dans M,
- soit une *application* $M\ N$ (qu'il faut comprendre comme le terme - fonction - M appliqué au terme - argument - N)
- soit l'*unité* (), un terme de base.

Ces termes correspondent respectivement aux expressions OCaml suivantes : `x` un nom, `fun x-> e` une fonction anonyme, `e1 e2` une application et (), l'unité.
On note $M\{N/x\}$ le terme obtenu en remplaçant toutes les occurrences (non-liées) de la variable x dans M par le terme N.
Une relation d'*α-conversion* $\equiv_\alpha$ autorise le renommage des variables liées : si $y \notin M$, alors $\lambda x.M \equiv_\alpha \lambda y.(M\{x/y\})$. Dans la suite, on considèrera les termes *modulo α-conversion* (on s'autorisera à renommer les variables liées dans les sous-termes), et on utilisera cette opération pour présenter des termes dans lesquels les variables liées sont distinctes deux à deux, et distinctes des variables non-liées.
Les *contextes d'évaluation* sont définis par : $\mathbf{E} ::= [] \mid M\ \mathbf{E} \mid \mathbf{E}\ M$
Si M est un terme et $\mathbf{E}$ un contexte, on note $\mathbf{E}[M]$ le terme obtenu en remplaçant [] dans $\mathbf{E}$ par M.
La sémantique (le comportement d'un terme) est donnée par une relation de réduction $\longrightarrow$ donnée par :

1. $(\lambda x.M)\ N \longrightarrow M\{N/x\}$ (on applique la fonction qui à x associe M à N, et on récupère le corps de la fonction M dans lequel l'argument N remplace le paramètre x.)
2. si $M \longrightarrow M'$ alors $\mathbf{E}[M] \longrightarrow \mathbf{E}[M']$ (on peut réduire dans un contexte).

Comme en OCaml, l'application est parenthésée à gauche, ainsi $M_1\ M_2\ M_3$ désigne $(M_1\ M_2)\ M_3$ et on écrit $\lambda xy.M$ pour $\lambda x.(\lambda y.M)$
$\longrightarrow^*$ désigne la clôture réflexive et transitive de $\longrightarrow$. Les *réduits* d'un terme M sont les éléments de l'ensemble $\{M' \mid M \longrightarrow^* M'\}$.

Question 1. Soit $\delta = \lambda x.(x\ x)$ et $I = \lambda x.x$. Expliciter les réduits du terme $A = (\lambda x.I)\ (\delta\ \delta)$.

On donne un système de *types simples* au λ-calcul. Les types sont donnés par :
$S, T ::= S \to T \mid \texttt{unit}$
Ainsi un type est soit le type fonctionnel $S \to T$ des fonctions de S dans T soit `unit` le type de base.
Une *hypothèse* $x : T$ associe une variable de terme à un type. Un *contexte de typage* Γ est un ensemble d'hypothèses et on note $\Gamma, x : T$ le contexte $\Gamma \cup \{x : T\}$ quand x n'est pas dans Γ. Un *jugement* de typage $\Gamma \vdash M : T$ indique que le terme M a le type T dans le contexte Γ (on dit qu'un terme est *typable* s'il existe Γ, T tel que $\Gamma \vdash M : T$).

Les règles de typage permettant de déduire un jugement sont données par :

1. $\Gamma \vdash () : \texttt{unit}$
2. $\Gamma, x : T \vdash x : T$
3. si $\Gamma, x : T_1 \vdash M : T_2$ alors $\Gamma \vdash \lambda x.M : T_1 \to T_2$
4. si $\Gamma \vdash M : T_1 \to T_2$ et $\Gamma \vdash N : T_1$, alors $\Gamma \vdash M\ N : T_2$

On note $\lambda_{\texttt{ST}}$ l'ensemble des termes typables.

Question 2. Montrer que si M est typable et $M = \mathbf{E}[N]$, alors N est typable, puis expliquer pourquoi A n'est pas typable.

On dispose d'un ensemble infini *d'adresses* $\mathcal{A}$. Une mémoire σ est une fonction qui a chaque élément d'un sous-ensemble fini de $\mathcal{A}$ (appelé son *support*) associe un λ-terme.
On définit un λ-calcul appelé $\lambda_{\texttt{ref}}$ contenant une valeur () de type `unit` et trois opérateurs qui permettent de manipuler une mémoire, inspirés de leurs homologues en OCaml :

1. une opération `ref` de référencement qui prend un λ-terme, le stocke en mémoire à une nouvelle adresse α et renvoie α.
2. une opération `deref` (! en OCaml) de déréférencement qui prend une adresse et renvoie le terme contenu à cette adresse en mémoire.
3. une opération `assig` (:= en OCaml) d'assignation qui prend une adresse α et un terme M, modifie la mémoire pour remplacer la valeur en α par M, et renvoie `unit`.

Question 3. Donner des règles de typage pour les termes de $\lambda_{\texttt{ref}}$ et les mémoires.

Question 4. Donner une sémantique pour $\lambda_{\texttt{ref}}$ explicitant comment réduire un couple composé d'un terme typable et d'une mémoire.

Les réductions qui consomment un opérateur `ref`, `deref` ou `assig` sont appelée *impures*, les autres *pures*.

Question 5. Définir une fonction *d'élagage* $\mathbf{p}$ qui associe à chaque terme typable de $\lambda_{\texttt{ref}}$ un terme typable de $\lambda_{\texttt{ST}}$ telle que si $(M, \sigma) \longrightarrow (M', \sigma)$ avec une réduction pure, alors $\mathbf{p}(M) \longrightarrow \mathbf{p}(M')$.

Un terme M est *terminant* quand il n'existe pas de suite infinie $(M_n)_{n\in\mathbb{N}}$ telle que
$M_0 = M$ et $\forall n \in \mathbb{N}, M_n \longrightarrow M_{n+1}$.
On admettra le résultat suivant :
Si M est un terme de $\lambda_{\texttt{ST}}$, alors M est terminant.

Question 6. Montrer que toute chaîne de réduction infinie dans $\lambda_{\texttt{ref}}$ contient une infinité de réductions impures. Proposer un terme de $\lambda_{\texttt{ref}}$ typable et non-terminant.

On divise la mémoire en *régions* identifiée par des entiers naturels. Les opérateurs $\texttt{ref}_n$, $\texttt{deref}_n$ et $\texttt{assign}_n$ sont maintenant indexés par la région qu'ils manipulent.

Question 7. Modifier le système de types pour qu'il associe à un terme typable son *effet* c'est à dire la région la plus haute qu'une réduction de ce terme peut manipuler (en effectuant une réduction d'un des trois opérateurs impurs).

Question 8. En contraignant les types par les régions, délimiter un sous-ensemble de $\lambda_{\texttt{ref}}$ terminant.

Déduction de messages

Nous souhaitons nous intéresser au problème suivant appelé problème de déduction :

entrée un ensemble fini de termes clos T et un terme clos u

sortie est-ce que u est déductible depuis T, noté $T \vdash u$?

Terme et sous-terme Nous nous intéressons aux termes construits inductivement à partir du symbole binaire $f(\cdot,\cdot)$, d'un ensemble infini dénombrable de constantes $\mathcal{C}$, et d'un ensemble infini dénombrable de variables $\mathcal{V}$.
Un terme est donc généré par la grammaire : $t, t_1, t_2 := v \in \mathcal{C} \mid x \in \mathcal{V} \mid f(t_1, t_2)$.
Si un terme ne contient pas de variable, alors ce terme est dit *clos*.
Étant donné un terme t nous notons $st(t)$ l'ensemble des *sous-termes* de t, i.e., le plus petit ensemble S tel que $t \in S$, et si $f(t_1, t_2) \in S$ alors $t_1, \ldots, t_n \in S$.

Règle d'inférence une règle d'inférence est une règle de déduction de la forme :

$$\frac{T \vdash t_1 \quad \ldots \quad T \vdash t_n}{T \vdash t}\ \textsc{Regle}$$

où T est un ensemble fini de termes et $t_1, \ldots, t_n, t$ sont des termes.
Un *système d'inférence* $\mathcal{I}$ est un ensemble fini de règles d'inférence.

Preuve Une *preuve* (ou *arbre de preuve*) Π de $T \vdash u$ dans $\mathcal{I}$ est un arbre tel que :

- chaque feuille est étiquetée avec un terme $v \in T$;
- pour chaque noeud ayant pour étiquette v_0 et enfants $v_1, \ldots, v_n$ il existe une règle d'inférence dans $\mathcal{I}$ ayant pour conclusion v_0 et hypothèses $v_1, \ldots, v_n$ (à instanciation près des variables) ;
- la racine de l'arbre est étiquetée par u.

La taille d'une preuve Π, notée $size(\Pi)$, est son nombre de noeuds. $Termes(\Pi)$ dénote l'ensemble des étiquettes, i.e., termes, apparaissant dans Π.

Lorsque $T \vdash u$ nous disons que u est déductible à partir de l'ensemble de termes T.

$$\frac{\text{si } u \in T}{T \vdash u}\ \textsc{Ax}$$

$$\frac{T \vdash x \quad T \vdash y}{T \vdash f(x,y)}\ \textsc{App-f} \qquad \frac{T \vdash f(x,y) \quad T \vdash y}{T \vdash x}\ \textsc{Red-f}$$

FIGURE 1 – Système d'inférence $\mathcal{I}_0$

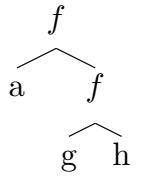


FIGURE 2 – Représentation sous forme d'arbre du terme $f(a, f(g, h))$

Question 1. Soit $T = \{f(f(a, k_1), k_2), k_2, f(k_1, k_2)\}$. Donner l'arbre de preuve de $T \vdash a$ dans $\mathcal{I}_0$.

Solution :

$$\dfrac{\dfrac{\dfrac{}{T \vdash f(f(a,k_1),k_2)}\ \textsc{Ax} \quad \dfrac{}{T \vdash k_2}\ \textsc{Ax}}{T \vdash f(a,k_1)}\ \textsc{Red-f} \quad \dfrac{\dfrac{}{T \vdash f(k_1,k_2)}\ \textsc{Ax} \quad \dfrac{}{k_2}\ Ax}{T \vdash k_1}\ \textsc{Red-f}}{T \vdash a}\ \textsc{Ref-f}$$

Question 2. Soit T un ensemble de termes clos. Montrer que pour tout $T \vdash u$ dans $\mathcal{I}_0$, un arbre de preuve de taille minimale Π de $T \vdash u$ contient seulement des termes issus de $st(T \cup \{u\})$, i.e., $Termes(\Pi) \subseteq st(T \cup \{u\})$.

Montrer de plus que si Π est réduit à une feuille ou termine par une règle Ax ou Red-f alors il contient uniquement des termes issus de $st(T)$, i.e. $Termes(\Pi) \subseteq st(T)$.

Solution : On fait une preuve par induction sur l'arbre de preuve.

Si l'arbre de preuve est réduit à la règle Ax alors la preuve est immédiate pour le cas général et le cas particulier.

Sinon, l'arbre de preuve se termine par une autre règle. Nous pouvons faire une étude de cas :

— App-f : par hypothèse d'induction, la propriété que nous souhaitons prouver est vraie.

— Red-f : par minimalité de l'arbre de preuve, nous savons que les sous-arbres Π_1 et Π_2 ne terminent pas par une règle App-f. En effet, si tel était le cas, alors nous pourrions construire un arbre plus petit en omettant les deux dernières étapes. Par conséquent, Π ne contient que des termes issus de $st(T)$.

Question 3. En déduire que le problème de déduction dans $\mathcal{I}_0$ est décidable en temps polynomial.

Nous considérerons que la taille du problème est : $size(T, u) = |st(u)| + \Sigma_{t \in T}|st(t)|$.

Solution :

— On calcule tous les sous-termes de $T \cup \{u\}$

— Tant qu'on n'a pas atteint un point fixe, on sature T avec les termes déductibles en une étape (si le terme déduit n'est pas dans $st(T) \cup \{u\}$, alors on ne l'ajoute pas). Le nombre maximal d'itérations est $|st(T) \cup \{u\}|$. On remarquera que $|st(T)|$ est au plus quadratique en la taille du problème car chaque sous terme d'un terme de T est plus petit que ce même terme.

— si u est dans l'ensemble saturé alors on retourne « oui », sinon on retourne « non ».

On définit le problème HORN-SAT :

entrée une formule Φ étant une conjonction finie de clauses de Horn

sortie est-ce que Φ satisfiable ?

Une clause de Horn est une formule du calcul propositionnel qui contient au plus un littéral positif. Une clause de Horn peut donc avoir trois formes :

— un littéral positif et aucun négatif : $C = (true \Rightarrow x)$

— un littéral positif et au moins un littéral négatif : $C = (x_1 \wedge \ldots \wedge x_n \Rightarrow x)$

— aucun littéral positif : $C = (x_1 \wedge \ldots \wedge x_n \Rightarrow false)$.

On admettra que HORN-SAT est P-complet, c'est-à-dire (intuitivement) que tout problème de décision dans P admet une réduction linéaire à HORN-SAT.

Question 4. Montrer que le problème de déduction dans $\mathcal{I}_0$ est P-complet.

Solution : Le problème de déduction est clairement dans P avec la question précédente (la taille du problème est le nombre de sous-termes).

L'idée est la suivante :

— nous allons associer à chaque variable propositionnelle x, une constante $v_x \in \mathcal{C}$

— nous allons remarquer que le terme t est déductible depuis $f(\ldots f(f(t, t_1), t_2), \ldots, t_n)$ si et seulement si $t_1, \ldots, t_n$ le sont aussi.

Pour montrer que le problème de déduction est P-complet, on peut utiliser HORN-SAT. Soit $\Phi = C_1 \wedge \ldots \wedge C_n$ une conjonction finie de clauses de Horn. Nous construisons :

1. Pour tout $x \in \mathcal{V}$, on associe une constante $v_x \in \mathcal{C}$. On définit également une constante $v_\perp$.
2. pour tout i, on définit

$$t_i = \begin{cases} f(\ldots f(f(v_x, v_{x_1}), v_{x_2}), \ldots, v_{x_n}) & \text{si } C_i = (x_1 \wedge \ldots \wedge x_n \Rightarrow x) \\ f(\ldots f(f(v_\perp, v_{x_1}), v_{x_2}), \ldots, v_{x_n}) & \text{si } C_i = (x_1 \wedge \ldots \wedge x_n \Rightarrow false) \\ v_x \text{ si } C_i = (true \Rightarrow x) \end{cases}$$

3. Soit $T = \{t_1, \ldots, t_n\}$. Par construction, $T \vdash v_x$ implique $x = true$ (pour tout v_x apparaissant dans Φ) dans toute valuation satisfaisant Φ.
   Par conséquent, $T \vdash v_\perp$ implique Φ est non-satisfiable. Réciproquement, si $T \nvdash v_\perp$ alors nous définissons la valuation $\{x \to 1 \mid T \vdash v_x\} \cup \{x \to 0 \mid T \nvdash v_x\}$ qui satisfait Φ.

Nous souhaiterions maintenant nous intéresser au même problème mais en ajoutant le ou-exclusif. Un terme est donc maintenant généré par la grammaire :

$$t, t_1, t_2 := v \in \mathcal{C} \mid x \in \mathcal{V} \mid f(t_1, t_2) \mid t_1 \oplus t_2.$$

Nous ne prouverons pas ici que le problème de déduction est encore décidable en temps polynomial. Nous nous intéresserons à prouver une étape de la preuve : étant donné un ensemble de termes T et un terme t, est-ce que $T \vdash t$ en utilisant uniquement les règles GX et Ax' ?

$$\frac{T \vdash u_1 \quad \ldots \quad T \vdash u_n}{T \vdash u_1 \oplus \ldots \oplus u_n}\ \text{GX} \qquad \frac{\text{si } u =_{AC} v \quad \text{et } v \in T}{T \vdash u}\ \text{Ax'}$$

On note $=_{AC}$ la plus petite relation telle que :

(refl.) $x = x$ (sym.) $(x = y) \Rightarrow (y = x)$ (trans.) $(x = y) \wedge (y = z) \Rightarrow (x = z)$

(comm.) $x \oplus y = x \oplus y$ (assoc.) $x \oplus (y \oplus z) = (x \oplus y) \oplus z$

(congr.) $(x_1 = y_1) \wedge (x_2 = y_2) \Rightarrow f(x_1, x_2) = f(y_1, y_2)$

Question 5. Soit u et v deux termes clos. Donner un algorithme en temps polynomial qui décide si $u =_{AC} v$.

Solution : Deux remarques ici :

— On peut se restreindre à une terme de la forme $u_1 \oplus ... \oplus u_n$ avec u_i ne contenant pas de $\oplus$.

— On peut simplifier le multi-ensemble obtenu en fonction du nombre d'occurrences de chaque facteur. On garde le facteur si son nombre d'occurrences est impair, on l'enlève s'il est pair. Le facteur 0 peut être retiré du multi-ensemble. On note cette procédure de simplification $Simplify(\cdot)$.

Étant donné que l'on raisonne modulo AC, on peut remarquer que sans perte de généralité, on peut aplatir tous les $\oplus$, i.e., considérer $\oplus$ comme un opérateur n-aire.

Étant donné que u et v sont clos, il suffit de calculer le multi-ensemble des facteurs de u et v en se donnant pour définition : $Facteurs(t) = \begin{cases} \bigcup_{i=1}^{n} Facteurs(t_i) & \text{si } t = t_1 \oplus \ldots \oplus t_n \\ \{t\} & \text{sinon} \end{cases}$

L'algorithme est comme suit :

$$Egal(u, v) = \begin{cases} \perp & \text{si } u = f(t_1, t_2) \text{ and } v \neq f(t'_1, t'_2) \text{ (ou inversement)} \\ Egal(t_1, t'_1) \wedge Egal(t_2, t'_2) & \text{si } u = f(t_1, t_2) \text{ and } v = f(t'_1, t'_2) \\ Simplify(Facteurs(u)) == Simplify(Facteurs(v)) & \text{sinon} \end{cases}$$

Question 6. Soit T un ensemble de termes clos. Soit t un terme clos.

Montrer que $T \vdash t$ dans $\{\text{GX}, \text{Ax'}\}$ est décidable en temps polynomial.

Solution : On peut commencer par remarquer que sans perte de généralité, on peut supposer que notre arbre est de hauteur 2 : une unique application de GX et ensuite uniquement des applications de Ax'. En effet, si on a deux étages, on peut juste les « aplatir » et on obtient toujours un arbre de preuve valide.

On définit $Facteurs(t)$ et $Simplify(S)$ comme dans la réponse à la question précédente.

Voici l'algorithme :

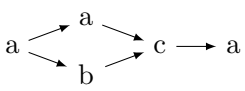
1. On calcule $S_t = Simplify(Facteurs(t))$ et $S_T = Simplify(Facteurs(T))$ (temps polynomial)
2. Si $S_t \not\subseteq S_T$ alors t n'est pas déductible à partir de T. En effet, il existe un sous-terme de t que nous ne pourrons jamais construire à partir de T (temps polynomial)
3. Sinon on représente S_t comme un vecteur de taille $|S_T|$ avec pour valeurs 0/1. Le vecteur a un 1 en position i si le i-ème facteur de S_T est aussi présent dans S_t. Sinon, on affecte la valeur 0. Notons ce vecteur V^t.
4. On peut définir les mêmes vecteurs pour les différents termes de T pris individuellement, notons les V_i^T
5. la déductibilité de t est maintenant réduite à l'existence d'une combinaison linéaire dans $\mathbb{F}_2^p$ des vecteurs V_i^T telle que $\Sigma_i \alpha_i V_i^T = V^t$. Cette étape se calcule en temps polynomial à l'aide d'un pivot de Gauss.

Mots partiels et Théorème de Dilworth

Mots partiels

Soit Σ un ensemble fini de lettres, et Σ^* l'ensemble des mots de Σ, qui sont des séquences finies de lettres. Nous généralisons la notion de mots à des structures partiellement ordonnées.
Soit $G = (E, R)$ un graphe dirigé fini : E est un ensemble fini et R un sous ensemble de $E \times E$. Un chemin de G est une séquence $(x_1, y_1), \ldots, (x_n, y_n)$ d'éléments de R telle que $y_i = x_{i+1}$. Le graphe G est dit *acyclique* s'il n'existe pas de chemin tel que $y_n = x_1$.
Un mot partiel sur Σ est un triplet (E, R, μ) avec (E, R) un graphe fini acyclique et $\mu \colon E \to \Sigma$. La taille d'un mot partiel est le nombre d'éléments de E.
Soit $p = (E, R, \mu)$ un mot partiel de taille n. Un mot $u_1 \cdots u_n \in \Sigma^*$ généralise p s'il existe $\psi \colon E \to \{0, \ldots, n-1\}$, injective, tel que pour tout $(x, y) \in R$, $\psi(x) < \psi(y)$ et si pour tout $e \in E$, on a $u_{\psi(e)} = \mu(e)$. Dans la suite on note Total(p) l'ensemble des mots qui généralisent p.

Question 1. Que vaut Total(p) pour p dessiné ci-dessous, où chaque sommet est résumé par son image par μ.



Question 2. Donnez une borne supérieure sur la taille de Total(p) en fonction de la taille de Σ.

Soit L un langage de Σ^*. Un mot partiel p sur Σ *appartient à* Partiel(L) s'il existe $v \in$ Total(p) qui appartient à L.

Question 3. Montrez que si L est dans PTIME, alors Partiel(L) appartient à NP.

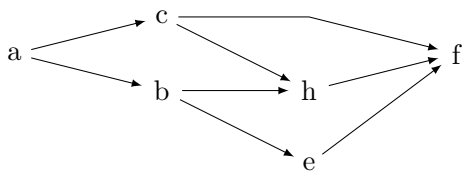
(Unary) 3-partition est un problème qui prend en entrée une suite finie de $3n$ entiers écrits **en unaire** $x_1, \ldots, x_{3n}$ et vérifie s'il existe une partition en n triplets dont les sommes deux à deux sont égales. On admet dans la suite que ce problème est NP-complet.

Question 4. Montrez par une réduction à 3-partition qu'il existe un langage L PTIME tel que Partiel(L) est NP-complet.

Question 5. Montrez que Partiel($\Sigma^* a b^* a \Sigma^*$) est vérifiable en temps polynomial.

Soit $p = (E, R, \mu)$ un mot partiel sur Σ. Une décomposition en chemins de p est un ensemble X de chemins tel que tout sommet de E appartient à exactement un chemin.

Question 6. Donnez une décomposition en chemins avec le moins de chemins possibles pour le mot partiel suivant :



Question 7. Soit L un langage régulier (reconnu par un automate déterministe). Montrez que Partiel(L) est vérifiable en temps $O(n^w)$ avec n la taille du mot partiel et w la taille de sa plus petite décomposition en chemins.

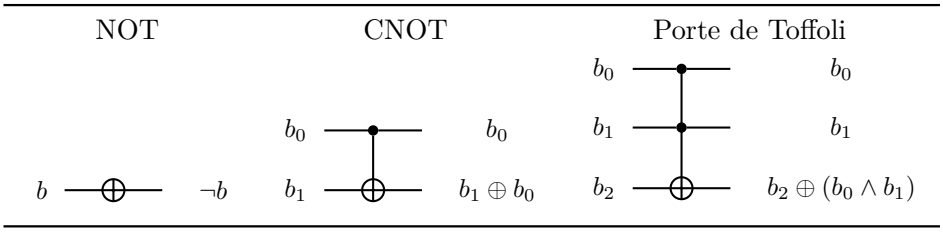
Soit $p = (E, R, \mu)$ un mot partiel sur Σ. Une anti-chaîne de p est un ensemble X tel qu'il n'existe pas de chemin entre deux éléments de X. On appelle largeur de p la taille de la plus grosse anti-chaîne de p.
On admet le théorème suivant :

Théorème 1 (Dilworth). Soit p un mot partiel. La largeur de p est égale à la taille de sa plus petite décomposition en chemins. Cette dernière est calculable en temps polynomial.

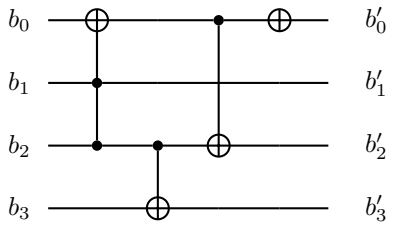
Question 8. En déduire que pour les mots partiels de largeur fixée, le problème Partiel(L) pour L un langage régulier est calculable en temps polynomial. Peut-on en déduire que le problème est calculable en temps polynomial, sans la contrainte que la largeur est fixée ?

Implémentation des circuits réversibles

On rappelle que l'opération de OU exclusif (XOR) est définie par : $a \oplus b = (a \vee b) \wedge (\neg a \vee \neg b)$, et qu'on a la propriété de distributivité suivante : $(a \oplus b) \wedge c = (a \wedge c) \oplus (b \wedge c)$. On définit les *portes logiques réversibles* suivantes.



Soit $n \in \mathbb{N}$. Un *circuit logique réversible sur n bits* est une séquence de portes logiques réversibles appliquées sur n entrées booléennes, qui seront numérotées de 0 à $n-1$. À toute porte logique du circuit correspond une fonction de $\{0,1\}^n$ vers $\{0,1\}^n$ qui modifie la valeur des entrées booléennes auxquelles elle est appliquée, selon sa définition donnée ci-dessus. Ici b_0 (porte CNOT) et respectivement b_0, b_1 (porte de Toffoli) sont nommés les *bits de contrôle*.
À tout circuit réversible $\mathcal{C}$ correspond une fonction $f_{\mathcal{C}} : \{0,1\}^n \to \{0,1\}^n$, obtenue en composant les fonctions de chaque porte logique, dans l'ordre de lecture de gauche à droite. On dit que $\mathcal{C}$ *implémente* la fonction $f_{\mathcal{C}}$. Ainsi, le circuit suivant :



implémente la fonction :

$$(b_0, b_1, b_2, b_3) \mapsto (\neg(b_0 \oplus (b_1 \wedge b_2)), b_1, b_2 \oplus b_0 \oplus (b_1 \wedge b_2), b_3 \oplus b_2)$$

Deux circuits réversibles $\mathcal{C}_1$, $\mathcal{C}_2$ sur n bits peuvent être *composés* en écrivant les portes de $\mathcal{C}_2$ suivies des portes de $\mathcal{C}_1$; on implémente alors la fonction $f_{\mathcal{C}_2} \circ f_{\mathcal{C}_1}$.

Question 1. 1. Montrer que tout circuit réversible $\mathcal{C}$ constitué d'une seule porte (NOT, CNOT ou Toffoli) implémente une permutation. Quel est son inverse ?

2. Comment implémenter l'inverse d'un circuit réversible quelconque ?

Solution : 1. Les circuits sont réversibles, c'est dans le titre. Une manière simple de résoudre cette question est donc d'exhiber l'inverse du circuit : pour un circuit $\mathcal{C}$ constitué d'une seule porte, $f_{\mathcal{C}\mathcal{C}} = f_{\mathcal{C}} \circ f_{\mathcal{C}}$ est l'identité. Par conséquent $f_{\mathcal{C}}$ est son propre inverse.

2. Comme les portes individuelles sont involutives, on peut inverser le circuit en écrivant ses portes dans l'ordre inverse.

Une *porte de Toffoli à k contrôles* implémente la fonction :

$$(b_0, \ldots, b_{k-1}, b_k) \mapsto (b_0, \ldots, b_{k-1}, b_k \oplus \bigwedge_{i=0}^{k-1} b_i)$$

Question 2. Montrer que dans un circuit sur n bits, une porte de Toffoli à $n-2$ contrôles peut être implémentée à l'aide de 2 portes de Toffoli à $n-3$ contrôles, et deux portes de Toffoli.

Solution : Cette question a posé problème à beaucoup de candidat(e)s.

L'idée principale est de séparer le AND des $n-2$ contrôles en un premier AND de $n-3$ contrôles, suivi d'une autre opération AND calculée avec une porte de Toffoli. Il faut ensuite « corriger » les valeurs contenues dans les bits de sortie afin d'implémenter exactement la porte voulue, en utilisant la distributivité du AND sur le XOR.

Soient $x_0, \ldots, x_{n-3}$ les contrôles, x_{n-2} le bit résultat, x_{n-1} le dernier bit du circuit.

On effectue les opérations suivantes :

$$\begin{cases} x_{n-1} \leftarrow x_{n-1} \oplus (x_0 \wedge \ldots \wedge x_{n-4}) \\ x_{n-2} \leftarrow x_{n-2} \oplus (x_{n-1} \wedge x_{n-3}) \\ x_{n-1} \leftarrow x_{n-1} \oplus (x_0 \wedge \ldots \wedge x_{n-4}) \\ x_{n-2} \leftarrow x_{n-2} \oplus (x_{n-1} \wedge x_{n-3}) \end{cases} \tag{1}$$

On vérifie que cette implémentation donne le résultat escompté. En effet :

1. Après les deux premières opérations, le bit à position $n-2$ contient $(x_{n-1} \oplus (x_0 \wedge \ldots \wedge x_{n-4})) \wedge x_{n-3} \oplus x_{n-2}$. (Ensuite, développer).
2. Après la troisième opération, le bit à position $n-1$ contient de nouveau x_{n-1}
3. La dernière opération permet d'enlever le terme $x_{n-1} \wedge x_{n-3}$ dans le bit à position $n-2$, ce qui donne le résultat.

Dans la suite de cet exercice, on va démontrer le théorème suivant :

Pour tout $n \geq 7$, une permutation Π de $\{0,1\}^n$ est paire si et seulement s'il existe un circuit réversible $\mathcal{C}$ sur n bits tel que $\Pi = f_{\mathcal{C}}$.

On rappelle qu'une permutation est paire (de signature 1) si et seulement si toutes ses décompositions en transpositions ont un nombre de transpositions pair.

Question 3. Trouver des contre-exemples simples pour $n = 2$ et $n = 3$.

Solution : Une seule porte suffit. Dans le cas $n = 2$ on considère la porte CNOT définie plus haut : la permutation envoie (00) vers (00), (01) vers (01), (10) vers (11) et (11) vers (10). C'est donc une transposition, qui est impaire.

Dans le cas $n = 3$ on considère une porte de Toffoli. Toutes les entrées sont laissées invariantes hormis (110) et (111) qui sont interverties. La permutation est donc également une transposition.

Question 4. Soit $\mathcal{C}$ un circuit réversible sur $n \geq 4$ bits ne comportant qu'une seule porte logique. Soit S l'ensemble des cycles de $f_{\mathcal{C}}$.

1. Montrer qu'il existe une partition $S = S_0 \cup S_1$ et une bijection $S_0 \to S_1$ préservant la longueur des cycles.
2. En déduire une implication du théorème.

Solution : 1. Considérons le bit sur lequel la porte ne s'applique pas, sans perte de généralité plaçons-le à la position 0. Considérons un cycle de $f_{\mathcal{C}}$ de longueur ℓ : $(t_0\ t_1 \dots t_{\ell-1})$ où $t_0, \dots, t_{\ell-1}$ sont des n-uplets.

Comme $f_{\mathcal{C}}$ n'agit pas sur le bit à position 0, tous les t_i ont la même valeur pour ce bit. On peut donc partitionner les cycles selon que le bit à position 0 vaut 0 (S_0) ou 1 (S_1).

Pour tout t_i ci-dessus, soit t'_i la valeur obtenue en modifiant le bit 0. Alors $f_{\mathcal{C}}(t_i) = t_{i+1} \implies f_{\mathcal{C}}(t'_i) = t'_{i+1}$, car $f_{\mathcal{C}}$ laisse le bit 0 invariant. Par conséquent $(t'_0 t'_1 \dots t'_{\ell-1})$ est un cycle de même longueur. Ce qui définit la bijection.

2. En décomposant les cycles de S_0 et S_1 on obtient le même nombre de transpositions, donc $f_{\mathcal{C}}$ est paire. Une composition de permutations paires est paire, donc par induction triviale les permutations issues des circuits réversibles sont paires.

Question 5.
1. Soit $n \geq 4$. Soit $a, b \in \{0,1\}^n, a \neq b$. Montrer qu'il existe un circuit $\mathcal{C}$ n'utilisant que des portes NOT et CNOT tel que $f_{\mathcal{C}}(a) = (1, 1, \dots, 1)$ et $f_{\mathcal{C}}(b) = (0, 1, \dots, 1)$.
2. Soit $c, d \in \{0,1\}^{n-1}, c \neq d$. Montrer qu'il existe un circuit $\mathcal{C}$ tel que $f_{\mathcal{C}}$ est la paire de transpositions $((0,c)\ (0,d))\ \ ((1,c)\ (1,d))$.
3. En déduire que si Π est une permutation de $\{0,1\}^n$ laissant au moins un bit invariant, il existe un circuit réversible $\mathcal{C}$ tel que $\Pi = f_{\mathcal{C}}$.

Solution : 1. L'objectif du circuit est d'« envoyer » a sur $(1, 1 \dots, 1)$ (tous les bits à 1) et b sur $(0, 1 \dots, 1)$ (tous les bits à 1 sauf le premier). Notre seule hypothèse est que $a \neq b$. Il existe différentes manières, plus ou moins efficaces, d'implémenter une telle opération. Nous ne présentons ici qu'une possibilité.

Écrivons les bits de a et de b : $a_0 \cdots a_{n-1}$ et $b_0 \cdots b_{n-1}$. Sans perte de généralité supposons que $a_0 = 1$ et $b_0 = 0$ (ce pourrait être l'inverse, ou ce pourrait être une position différente, mais dans ce cas on modifierait simplement les indices dans ce qui suit).

1. On applique des CNOTs dont le bit de contrôle est le numéro 0 et la cible les bits qui sont non nuls dans a : ainsi, si l'entrée du circuit est a, on obtient bien $(1, \dots, 1)$ à ce stade. Si l'entrée est b, on obtient b.
2. On applique un NOT sur le bit numéro 0. Si l'entrée est b, le premier bit devient 1.
3. On applique des CNOTs dont le bit de contrôle est le numéro 0 et la cible les bits qui sont non nuls dans b.
4. On applique un NOT sur le bit numéro 0.

Concrètement, on a utilisé le bit 0 comme contrôle intermédiaire pour transformer la sortie : en partant de a, l'étape 3 ne s'applique pas et on obtient $(1, \dots, 1)$. En partant de b, l'étape 1 ne s'applique pas et on obtient $(0, 1, \dots, 1)$.

2. On va utiliser le circuit $\mathcal{C}'$ précédemment construit. D'abord, on envoie c et d sur $(1, 1, \dots, 1)$ et $(0, 1, \dots, 1)$ respectivement (indépendamment du premier bit), ce qui signifie qu'on envoie :

$$\begin{cases} (0,c) \to (0, 1, 1, \dots, 1) \\ (1,c) \to (1, 1, 1, \dots, 1) \\ (0,d) \to (0, 0, 1, \dots, 1) \\ (1,d) \to (1, 0, 1, \dots, 1) \end{cases} \tag{2}$$

On applique ensuite une Toffoli contrôlée par les $n-2$ derniers bits (c'est possible car on l'a construite à la question 2) où la cible est le bit 1. Cela intervertit les paires $(0, 1, 1, \dots, 1)$ et $(0, 0, 1, \dots, 1)$, et aussi $(1, 1, 1, \dots, 1)$ et $(1, 0, 1, \dots, 1)$. On applique ensuite l'inverse de $\mathcal{C}'$.

Le circuit envoie donc $(0, c)$ sur $(0, d)$ et $(1, c)$ sur $(1, d)$, et inversement. Il est important de vérifier que les autres entrées sont inchangées : c'est le cas car en arrivant au niveau de la Toffoli on aura autre chose que $(1, \ldots, 1)$ pour les contrôles (il ne se passera donc rien).

3. Si une permutation laisse un bit invariant, on peut suivre le raisonnement de la Question 4 pour séparer ses cycles en deux ensembles disjoints, et ensuite ses transpositions en paires disjointes. On obtient des paires de transpositions telles que vues en Question 5.2, et donc une implémentation.

Question 6. Montrer que pour $n \geq 6$, toute permutation paire de $\{0,1\}^n$ peut s'écrire comme une composition de paires de transpositions disjointes, c'est-à-dire :

$$\Pi = (x_0\ y_0)(z_0\ t_0)(x_1\ y_1)(z_1\ t_1)\ldots(x_k\ y_k)(z_k\ t_k)$$

où pour tout i, x_i, y_i, z_i, t_i sont distincts deux à deux (mais on peut avoir par exemple $x_0 = y_1$).

Solution : C'est une question préparatoire pour la question suivante. Il existe une version plus compliquée de ce résultat, qui optimise le nombre de transpositions ; ce n'est pas l'objectif ici.

Il suffit de décomposer la permutation en produit de transpositions :

$$\Pi = (x_0\ y_0)\cdots(x_{2k-1}\ y_{2k-1})$$

(Qui sont en nombre pair, par hypothèse). Entre chaque paire de transpositions $(x_{2i}\ y_{2i})(x_{2i+1}\ y_{2i+1})$ on insère $(z_i\ t_i)^2$ (qui est l'identité), où on choisit z_i et t_i différents de $x_i, y_i, x_{i+1}, y_{i+1}$. C'est possible car $n \geq 6$.

Question 7. Soit $n \geq 7$ et soit x, y, z, t des éléments de $\{0,1\}^n$ distincts deux à deux. Montrer qu'il existe un circuit réversible $\mathcal{C}$ implémentant une permutation P telle que :

$$P(x) = (0,0,1,\ldots,1),\quad P(y) = (0,1,1,\ldots,1),\quad P(z) = (1,0,1,\ldots,1),\quad P(t) = (1,1,1,\ldots,1)\ .$$

En déduire un circuit réversible qui implémente la paire de transpositions $(x\ y)(z\ t)$.

Solution : Nous allons commencer par implémenter une permutation Q qui va nous aider, en envoyant x, y, z, t sur des sorties $Q(x), Q(y), Q(z), Q(t)$ telles que le bit à position $n-1$ vaut 1. Pour ce faire, il suffit de sélectionner un sous-ensemble de $\leq n-1$ positions telles que les restrictions de x, y, z, t à ces positions sont toujours distinctes deux à deux.

Existe-t-il un tel ensemble ? Oui car $n \geq 7$, grâce à un raisonnement par l'absurde et au principe des tiroirs (c'est ici que le choix $n \geq 7$ devient pertinent). Il n'y a que $\binom{4}{2} = 6$ paires possibles parmi (x, y, z, t), et il y a $\binom{7}{6} = 7$ choix de 6 positions parmi les 7. Par conséquent, si chaque choix de 6 positions provoque une collision sur une paire, il existe une paire pour laquelle *deux* choix collisionnent. Or, ce sont deux choix *différents* de 6 positions parmi 7, donc leur union contient toutes les positions : tous les bits sont égaux (ce qui est impossible).

L'échange de bits dans le circuit est implémentable (également grâce à la Question 5 ; plus simplement le SWAP de deux bits s'implémente à l'aide de 3 CNOTs). Sans perte de généralité, réécrivons donc x, y, z, t sous la forme : $(b_x, x'), (b_y, y'), (b_z, z'), (b_t, t')$ où x', y', z', t' sont distincts deux à deux. On utilise la question 5 pour implémenter une permutation Q qui laisse le premier bit invariant, et fixe le dernier bit à 1 :

$$\begin{cases} (b_x, x') \to (b_x, x'', 1) \\ (b_y, y') \to (b_y, y'', 1) \\ (b_z, z') \to (b_z, z'', 1) \\ (b_t, t') \to (b_t, t'', 1) \end{cases} \tag{3}$$

On remarque que $(b_x, x''), (b_y, y''), (b_z, z''), (b_t, t'')$ sont des valeurs distinctes deux à deux (sinon, on n'aurait pas implémenté une permutation). On utilise la Question 5 une nouvelle fois pour les envoyer respectivement sur

$(0,0,1,\ldots,1),(0,1,1,\ldots,1),(1,0,1,\ldots,1),(1,1,1,\ldots,1)$, en laissant le dernier bit invariant (qui de toute façon vaut 1 dans tous les cas).

La composition de toutes ces permutations nous donne P.

Pour terminer la question (et donc le théorème), on applique une porte de Toffoli à $n-2$ contrôles, dont la sortie est le bit numéro 1 et les contrôles les $n-2$ derniers bits. Cette porte échange $(0,0,1,\ldots,1),(0,1,1,\ldots,1)$ et $(1,0,1,\ldots,1),(1,1,1,\ldots,1)$.

En appliquant ensuite l'inverse de P, on obtient bien $(x\ y)(z\ t)$.