BANQUE MPI INTER-ENS – SESSION 2023
RAPPORT SUR L'ÉPREUVE PRATIQUE D'ALGORITHMIQUE ET DE PROGRAMMATION DU CONCOURS COMMUN DES ÉCOLES NORMALES SUPÉRIEURES

Écoles concernées : Lyon, Paris-Saclay, Rennes, Ulm

Coefficients (en pourcentage du total d'admission)
- Lyon : 18,3 %
- Paris-Saclay : 13,2 %
- Rennes : 13,9 %
- Ulm : 13,3 %

Jury : Adrien Koutsos, Joseph Lallemand, Gabriel Radanne, Yann Ramusat, Michaël Rao

Contenu de ce document

Dans ce rapport, après avoir rappelé l'organisation de l'épreuve, nous faisons quelques remarques générales sur son déroulement. Enfin, nous revenons plus en détail sur chacun des deux sujets, en insistant sur certains points que nous avons jugé marquants dans les sujets ainsi que les réponses des candidats et candidates.

Le début du rapport, jusqu'à la discussion spécifique aux sujets de cette année, est identique dans les rapports de série MP et MPI.

Organisation de l'épreuve

L'objectif de cette épreuve est d'évaluer la capacité à mettre en œuvre une chaîne complète de résolution d'un problème informatique, à savoir la construction d'algorithmes, le choix de structures de données, leurs implémentations, et l'élaboration d'arguments mathématiques pour justifier ces décisions. Le déroulement de l'épreuve est le suivant : un travail sur machine d'une durée de 3 h 30, immédiatement suivi d'une présentation orale pendant 23 minutes.

Juste avant la distribution des sujets, les candidats et candidates disposent d'une période de 10 minutes pour se familiariser avec l'environnement informatique et poser des questions en cas de difficultés d'ordre pratique.

Un sujet contient typiquement une dizaine de questions écrites et une dizaine de questions orales. Il commence généralement par des questions de programmation simples, ayant pour objet la génération des données d'entrée du problème étudié, qui seront utilisées pour tester les programmes des questions suivantes. Elles sont typiquement générées pseudo-aléatoirement, à partir d'une suite pseudo-aléatoire initialisée par une valeur u_0, différente pour chaque personne, distribuée au début de l'épreuve. Éventuellement certaines données peuvent être lues dans des fichiers fournis.

Nous invitons *fortement* les candidats et candidates à se familiariser à l'avance avec la manière dont ces suites pseudo-aléatoires sont générées et utilisées dans les sujets précédents afin de gagner du temps le jour de l'épreuve.

Les questions écrites demandent de calculer certaines valeurs, typiquement numériques, bien que des chaînes de caractères soient également possibles. Chaque question requiert l'implémentation d'un algorithme et son utilisation sur les entrées générées au début du sujet, pour calculer les valeurs demandées. Une question est typiquement divisée en sous-questions pour des entrées de plus en plus grandes, ce qui permet de tester l'efficacité

de l'algorithme mis en œuvre. Les réponses sont à inscrire sur une fiche-réponse, qui sera remise au jury à l'issue de la partie pratique de l'épreuve.

Une aide précieuse est donnée aux candidats et candidates, sous la forme d'une fiche-réponse type, contenant les valeurs obtenues sur les données générées à partir d'un $\widetilde{u_0}$ donné. Cette fiche leur permet de vérifier l'exactitude des réponses pour une graine différente du u_0 de la leur. Il est très fortement recommandé, comme indiqué dans l'introduction des sujets, de vérifier que le générateur aléatoire se comporte comme attendu avec la graine $\widetilde{u_0}$, pour chaque question. Il serait dommage de traiter le sujet avec un générateur faux, et donc d'obtenir de mauvaises valeurs numériques malgré des algorithmes corrects.

Les questions orales sont de nature plus théorique et sont destinées à être présentées pendant la seconde partie de l'épreuve. Le déroulement de l'oral est le suivant : le candidat ou la candidate présente, le plus efficacement possible, les questions orales préparées pendant la première phase, puis éventuellement, s'il reste du temps, s'ensuit une discussion avec le jury sur les questions non traitées. La présentation orale vise à évaluer la bonne compréhension du sujet et le recul. Le jury s'efforce d'aborder toutes les questions préparées pendant la première étape, et, suivant le temps disponible, des extensions de ces questions, ou des questions qui n'ont pas été traitées par manque de temps. Pour réaliser un bon oral, il est important de prendre le temps de réfléchir aux questions à préparer mentionnées dans le sujet, et de préparer suffisamment de notes au brouillon pour être capable d'exposer clairement et efficacement les solutions, en s'aidant du tableau dans la mesure où il est utile.

La partie écrite de l'épreuve représentait cette année 50 % de la note finale. On observe dans l'ensemble, quoique pas systématiquement, une bonne corrélation entre les résultats obtenus aux deux parties.

Lecture de fichiers ou de l'entrée standard. Depuis 2022, dans certains sujets, une nouveauté a été introduite dans l'épreuve : des données d'entrée sont parfois lues depuis des fichiers, plutôt que générées aléatoirement. Dans ce cas, les fichiers contenant ces données sont fournis aux candidats et candidates au début de l'épreuve, ainsi que du code permettant de les lire. Il peut par exemple être demandé d'utiliser ce code pour récupérer une partie d'un fichier fourni dépendant du u_0, qui sera utilisée comme donnée d'entrée – nous renvoyons le lecteur aux sujets de MPI de cette année qui en sont des exemples.

Conseils et remarques générales

Écriture du programme. Le jury peut être amené à inspecter le code des candidats et candidates afin de lever certaines ambiguïtés lors de la présentation de leurs algorithmes. Cela n'est cependant possible que pour celles et ceux qui avaient soigné la lisibilité de leur code, et seulement si les réponses aux questions étaient facilement identifiables et exécutables. Nous conseillons ainsi aux candidats et candidates de soigner la lisibilité de leur code. On pourra s'inspirer des propositions de corrigés fournies en annexe de ce rapport.

Génération de structures. La plupart des sujets demandent de générer des objets (graphes, arbres, chaînes de caractères ou autre) qui seront manipulés par la suite. Bien que chaque sujet impose son propre modèle de génération de données aléatoires, certaines étapes sont communes entre les années. S'entraîner sur plusieurs sujets en conditions réelles prend un temps considérable (3h30 de préparation par sujet), ainsi nous recommandons plutôt aux candidats et candidates de traiter les premières questions de plusieurs sujets différents afin d'être efficaces sur le début du sujet. Nous leur conseillons également de s'entraîner à traiter un ou deux sujets en entier, et de lire des corrections. Ceci leur permettra d'avoir une idée du genre de subtilités algorithmiques qui les attendent, et de maîtriser les questions qui sont similaires d'un sujet à l'autre.

Par ailleurs, plusieurs personnes mélangent dans leur code le $\widetilde{u_0}$ commun fourni pour tester leur code, et leur u_0 propre. Pour éviter que cela ne cause de problème, nous recommandons fortement d'éviter les copier-coller avec une version du code pour u_0 et une pour $\widetilde{u_0}$, et plutôt de lire le u_0 dans une variable et d'exécuter tout le code avec, *cf.* les corrigés proposés.

Gestion de l'oral. La durée de l'oral étant courte relativement au nombre de questions à traiter, nous conseillons aux candidats et candidates de préparer une réponse précise mais intuitive, plutôt que de se perdre dans une preuve laborieuse au tableau. Si le jury n'est pas convaincu par un argument simple, il sera toujours possible de le convaincre par une preuve plus détaillée sans que cela ne diminue la note finale. Inversement, si le jury est convaincu par un raisonnement intuitif, on dispose alors de plus de temps pour aborder des questions globalement peu traitées et donc susceptibles de rapporter beaucoup de points. Par exemple, on voit parfois des candidats ou candidates se lancer dans d'interminables preuves par induction alors qu'il existe une explication intuitive immédiate. La capacité à exposer un argument formel pour répondre à une question est évaluée dans le cadre de l'épreuve d'informatique fondamentale, tandis que l'objet de l'oral ici est de s'assurer que le candidat ou la candidate fait le lien entre la résolution d'un problème informatique dans un cadre formel inédit et sa mise en pratique. Le jury saura donc apprécier le recul que démontre un argument simple et intuitif par rapport à une suite d'arguments formels désincarnés.

Sur la gestion du tableau, nous invitons les candidats et candidates à éviter l'écueil consistant à écrire tout leur raisonnement au tableau et ainsi perdre beaucoup de temps. L'écueil inverse, de ne pas utiliser du tout le tableau est plus rare, mais il rend parfois le raisonnement difficile à suivre. Il est difficile de donner une règle générale sur l'utilisation du tableau mais il ne faut pas hésiter à faire un dessin ou un exemple au tableau quand une explication s'y prête, ensuite de faire la preuve dessus à l'oral. Dans le cas d'un raisonnement par induction, il peut être intéressant d'écrire l'hypothèse, ou un invariant, sans détailler tout le raisonnement.

Enfin, il est recommandé d'utiliser dans les réponses aux questions d'oral les mêmes notations et terminologie que dans le sujet – nous avons trop souvent vu des candidats ou candidates donner la complexité de leurs algorithmes en fonction d'un "n" non défini, ou n'ayant pas le même sens que dans le sujet. Le jury sera bien entendu capable de suivre un raisonnement qui utilise d'autres termes ou notations que le sujet, mais on risque alors de perdre du temps en explications facilement évitables.

Lecture du sujet. Nous conseillons aux candidats et candidates de lire le sujet en entier, avant de se lancer dans l'écriture de leurs programmes. Ceci peut permettre d'identifier quelles questions sont indépendantes et peuvent être traitées dans le désordre, ainsi que de voir quel genre de problèmes vont être étudiés, ce qui peut orienter le choix des structures de données.

Recherche exhaustive et solutions naïves. Il n'est pas rare que les sujets demandent pour commencer une approche naïve pour résoudre un problème sur de petites valeurs, typiquement un algorithme exhaustif, avant d'orienter les candidats et candidates vers des méthodes plus efficaces. Il est alors généralement inutile de tenter de trop optimiser les algorithmes naïfs demandés – il a semblé au jury que certaines personnes n'ont pas osé résoudre certaines questions par force brute, ayant correctement identifié que cela menait à une complexité exponentielle. C'est dommage, car les valeurs numériques sont alors choisies assez petites pour qu'une telle approche conclue.

Les bornes de complexité d'algorithmes par force brute ont semblé peu claires à certaines personnes : nous avons parfois vu des réponses fantaisistes donnant par exemple une majoration du nombre de chemins dans un graphe linéaire en son nombre de sommets, et

des candidats ou candidates s'étonner qu'un algorithme exponentiel ne permette pas de conclure sur de grandes valeurs.

Signalons enfin qu'un algorithme cherchant à maximiser une valeur par exploration exhaustive n'est pas la même chose qu'un algorithme glouton, comme nous avons vu certains candidats ou candidates le prétendre.

Présentation des algorithmes. Certaines questions orales demandent aux candidats et candidates de présenter leurs algorithmes et d'analyser leur complexité. Nous les encourageons vivement à le faire de façon claire et concise. Contrairement à ce que nous avons trop souvent pu voir, il ne s'agit pas de recopier un programme en Python, OCaml, ou autre au tableau. Il faut s'efforcer de présenter (uniquement) les étapes clefs de l'algorithme, en langage naturel si possible, afin de permettre efficacement l'analyse de complexité par la suite. En particulier, il est essentiel d'en identifier clairement la structure itérative ou récursive.

Quelqu'un qui propose un algorithme correct peut obtenir tous les points à l'oral même sans l'avoir implémenté durant la partie pratique de l'épreuve. Les candidats et candidates ne doivent surtout pas s'interdire d'expliquer un algorithme plus efficace que celui effectivement implémenté, qui leur serait venu à l'esprit par la suite. On peut dans ce cas expliquer d'abord l'algorithme implémenté puis comment l'améliorer, ou bien présenter directement la version optimale.

Complexité des algorithmes. Le jury décerne des points partiels aux algorithmes justes mais à la complexité non optimale. Si l'algorithme proposé est en réalité trop naïf pour traiter les instances proposées dans le sujet, le jury saura apprécier un regard critique, qui exploiterait l'analyse de complexité et un ordre de grandeur sur le nombre d'opérations élémentaires qu'un ordinateur peut effectuer. Nous n'attendons pas une estimation précise du temps de calcul, mais de savoir qu'il est difficile d'obtenir une réponse rapidement si l'algorithme demande 10^{12} tours d'une boucle.

Langages de programmation. En MPI, les sujets demandent explicitement d'utiliser les langages OCaml et C, sans laisser le choix. Les candidats et candidates nous ont semblé moins à l'aise en C qu'en OCaml : il est important de s'entraîner avec les deux langages, qui sont tous deux exigés.

Les sujets de MP, quant à eux, sont prévus et calibrés pour être de difficulté équivalente dans les langages Python et OCaml. Nous notons une tendance générale des candidats et candidates à utiliser plutôt Python que OCaml. Certaines personnes hésitent et passent du temps à chercher le "meilleur" langage pour le sujet. Ceci est une perte de temps. Il est préférable de choisir à l'avance le langage que l'on maîtrise le mieux. Cela permet de s'entraîner pour bien connaître et éviter les problèmes et limitations liées au langage. Certaines années, on a ainsi pu voir des candidats ou candidates se lancent en Python sans se rappeler, par exemple, que dans ce langage les appels récursifs sont limités par défaut à 1000 (*cf.* `sys.setrecursionlimit`) et que les listes sont représentées par des tableaux.

Pour finir, nous voulons aussi conseiller d'étudier les structures de données basiques pour le ou les langages utilisés. La différence en efficacité d'un programme qui utilise une liste là où il aurait fallu un tableau, ou l'inverse, est très visible dans ce type de sujet. Connaître les complexité d'un accès, un parcours, une copie de ces structures est également souvent indispensable à l'analyse de complexité. Cela ne veut pas dire le jury s'attend à avoir tous les détails de l'implémentation lors de l'oral. Au contraire, les candidats et candidates qui obtiennent les meilleures notes savent mentionner les structures utilisées sans pour autant trop y passer de temps.

Exemple. Nous terminons par un exemple, la présentation d'un algorithme de parcours de graphe. Voici les quatre phrases que l'on s'attend typiquement à entendre pour une telle question.

— Un algorithme de parcours de graphe part d'un sommet et suit les arêtes pour visiter les sommets du graphe connectés au sommet original.
— L'ordre de traitement des arêtes est déterminé par le choix du parcours : en largeur, on traite en priorité les arêtes par distance croissante au sommet original, et en profondeur, on traite en priorité les arêtes sortant du dernier sommet visité. Ceci induit la structure de donnée utilisée : une file (FIFO) pour un parcours en largeur, une pile (LIFO) pour un parcours en profondeur.
— Dans les deux cas, chaque arête est ajoutée dans la structure de données exactement une fois, ce qui est assuré par un tableau de booléens déterminant si un sommet a déjà été visité ou non.
— La complexité du parcours est ainsi $O(n+m)$, où n est le nombre de sommets et m le nombre d'arêtes.

Ce dernier résultat est un résultat de cours qui doit être bien intégré : un parcours bien implémenté est linéaire en la taille du graphe. L'argument clef à bien comprendre est que l'on ne parcourt chaque sommet qu'une fois et l'on ne parcourt chaque arête qu'au plus deux fois.

Sujet 1 : Ordre de visite.

Ce sujet se décomposait en deux parties indépendantes, une première en OCaml et une seconde en C. La partie OCaml commençait de façon classique par la génération de structures pseudo-aléatoires, ici, des graphes colorés et leur somme de contrôle associée. La somme de contrôle étant là pour vérifier que les candidats aient bien généré les bons graphes. La notion d'*observation* était ensuite introduite : un chemin et les couleurs des sommets parcourus par ce chemin ; ainsi que la notion de *simulation réciproque* (plus communément appelée *bisimilarité*) : existence d'une stratégie au départ d'un sommet v pour obtenir les mêmes observations qu'au départ d'un sommet u, et réciproquement. Finalement, l'algorithme du *raffinement de partition* était décrit, et une approche itérative permettant de calculer les classes d'équivalences de la relation de bisimilarité était proposée.

La partie en C fournissait un jeu de données de taille modérée, contenant 16 graphes. Le plus grand graphe avait de l'ordre de 17 000 sommets et 27 000 arêtes. Le code C implémentant la lecture et le stockage en mémoire par des listes d'adjacence était fourni, ainsi que des directives de compilation dans un makefile. Après de brèves questions simples comme le calcul du nombre d'arbres et de forêts parmi ces graphes, des approches basées sur des parcours en profondeur de complexités conceptuelles croissantes étaient demandées afin de calculer le nombre de ponts et de points d'articulation dans les graphes. Il est à noter que nous avons décrit dans le sujet des caractérisations qui, si elles étaient suivies à la lettre, permettaient d'obtenir facilement les algorithmes optimaux attendus. Cette partie se terminait par une question simple, demandant de calculer des plus longs chemins dans des graphes dirigés acycliques.

Pour la partie écrite, les questions Q1 à Q3 étaient de simples exercices de programmation et ont été correctement traitées par une grande majorité de candidats. La question Q4 demandant d'implémenter le raffinement de partition a été plutôt bien traitée dans l'ensemble. Malheureusement, la Q5 clôturant la partie OCaml (que nous estimions pourtant simple) n'a été traitée correctement que par un seul candidat, et partiellement par un autre. Nous déplorons ici que les candidats ont eu, dans une grande majorité, beaucoup de mal à s'approprier les notions théoriques introduites dans ce sujet, alors que la difficulté pour implémenter les algorithmes attendus n'était pas particulièrement élevée. Les questions suivantes Q6 à Q10 concernaient la partie C. La question la plus simple demandant de calculer le nombre d'arbres dans le jeu de données n'a été traitée que par la moitié des candidats, et la question suivante portant sur le calcul du nombre de ponts n'a été traitée que par une petite minorité d'entre eux. Aucune des questions suivantes n'a été traitée par les

candidats, malgré le fait que certaines (comme la Q9) était très accessibles et relativement faciles à implémenter.

Pour la partie orale, il est devenu clairement évident au vu des réponses dès la seconde question que les définitions du sujet ont été très difficiles à s'approprier par les candidats et que la préparation des questions orales n'avait, dans l'ensemble, pas été faite rigoureusement. Par exemple en QO2, il était demandé de montrer, ou d'infirmer à l'aide d'un contre-exemple, que pour deux sommets, avoir les mêmes ensembles d'observations était équivalent à être bisimilaires. Nous avons eu des candidats prétendant avoir une preuve, mais n'ayant en réalité prouvé qu'une seule des deux directions demandées. Nous avons dû guider des candidats pour trouver un contre-exemple à cette affirmation, et malgré l'aide, certains d'entre eux avaient du mal à l'expliquer, même une fois qu'il était donné.

La question suivante demandait d'expliciter comment se servir du raffinement de partition pour calculer la relation d'équivalence attendue. Très peu de candidats ont pu trouver la solution à cette question alors que celle-ci était presque suggérée dans le sujet : il suffisait d'utiliser les voisinages entrants et non sortants comme cela avait été fait pour tester l'implémentation du raffinement de partition dans la question écrite 4. Ceci explique l'absence quasi totale de réponses à la question écrite 5.

Du côté du C, nous avons été très étonnés d'avoir eu un nombre non négligeable de candidats ne maîtrisant pas l'analyse de complexité d'un parcours en profondeur. Cette notion est néanmoins au programme et chaque rapport annuel des jurys de cette épreuve en fait état. De plus, très peu de candidats ont eu l'idée de préparer durant les 3h30 des questions orales portant sur des exercices qu'ils n'avaient pas eu le temps d'aborder dans la partie écrite. Il ne faut pas sous-estimer l'importance des questions orales ; en effet, rien n'empêche un candidat de gagner des points en répondant correctement à une question orale relative à une question écrite n'ayant pu être traitée dans les temps.

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Tous les points	94	92	84	61	2	41	4	0	2	0
Réponses partielles	98	98	94	82	4	49	4	0	2	0

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8	QO9	QO10
Tous les points	63	8	6	30	63	6	0	4	6	0
Réponses partielles	96	73	45	73	98	51	18	22	8	2

TABLE 1. Pourcentages de réponses correctes et partielles à chaque question du sujet 1.

SUJET 2 : RECHERCHE EFFICACE DE SÉQUENCES ADN

Ce sujet se focalisait sur la conception de *structures de données*, et plus spécifiquement trois exemples classiques spécialisés pour la recherche de *motif* – un ensemble de mots – dans un texte. Le sujet commençait avec les Tries, ou arbre de préfixes, en OCaml, avant de proposer deux parties indépendantes. D'un coté, l'implémentation des arbres de Patricia en C privilégiait une bonne exploitation de concepts bas niveau (pointeurs et tableaux en C). De l'autre l'implémentation de l'algorithme Aho-Corasick en OCaml, qui construit un automate reconnaissant un motif, demandait une bonne maîtrise de l'algorithmique et l'implémentation. Nous rappelons aux candidat · es qu'il est fortement recommandé de lire le sujet en avance et de choisir la partie indépendante avec laquelle ils sont le plus à l'aise.

Dans tout les cas, l'évaluation portait sur des motifs de taille variables et des chaînes de tailles croissantes, culminant avec des chaines de 10 millions de caractères. Les tailles des

entrées étaient particulièrement étudiées pour forcer une bonne conception des structures de données, et des parcours de chaînes efficaces. Contrairement à la plupart des sujets, les entrées étaient déjà générées et devait simplement être lue dans les fichiers fournies. Les candidat · es ne semblent pas avoir rencontré de problème sur ce point, qui sera probablement réutilisé à l'avenir.

La première section, sur les Tries, à été bien abordée par la plupart des candidat · es, avec des implémentations d'une efficacité variable, mais permettant généralement de traiter les trois premières entrées proposées pour chaque question. Q1, Q2 et QO1 étaient des questions d'échauffement qui n'ont pas posé problème. Q3 et Q4 testaient la correction de l'insertion et l'efficacité du trie obtenue, et ont été bien implémentées. QO2 demandaient une analyse de la structure de donnée obtenue, qui a rarement été bien comprise par les candidat · es, qui ont généralement eu une grande confusion sur l'espace mémoire utilisé par les types (mutuellement) récursifs en OCaml. Q5 et QO3 étaient plus délicate. En particulier il est chaudement conseillé aux candidat · es de regarder les tailles des entrées proposées quand elles sont indiquées dans le sujet : une entrée de taille 1000000 est un indice qu'il faudra trouver un algorithme linéaire. On peut également noter plusieurs faiblesses dans l'évaluation des complexités de programmes réels par la plupart des candidat · es : la pile d'appel, en particulier dans le cas de récursivité non terminale, est souvent ignorée dans l'analyse d'espace. De plus, les copies sont souvent oubliées, que cela soit `String.sub` (souvent traité comme $O(1)$), ou les copies de listes (souvent ignorées). Finalement, QO4 a été globalement réussie par les candidat · es.

Les candidat · es ont eu plus des difficultés dans la section 2, en grande partie liées à la programmation en C. La conception du type de donnée en C (pré Q6) à été très bien réalisée par les candidat · es, ce qui a été apprécié, mais son explication (en QO5) était souvent brouillonne. Q6, quand elle était traitée, était bien faite. L'implémentation des algorithmes suivants en Q7, Q8 et QO6 a été plus laborieuse, et rarement comprises par les candidat · es. Nous encourageons les candidat · es à travailler davantage l'implémentation d'algorithmes simples en C.

Très peu de candidat · es ont abordé la section 3, qui démarrait pourtant avec des questions aisées. Q07 étaient une question de culture sur les automates, qui a été bien réussie. Q9 étaient une traduction directe du pseudo-code indiqué, avec une analyse de complexité simple en QO8. QO9 demandant de trouver un contre-exemple. Les candidats qui s'y sont essayé ont bien réussi jusqu'ici. Q10 et QO10 suivaient la description fournie, mais demandaient une implémentation plus délicate. Enfin, Q11 et QO11 étaient nettement plus dures, demandant d'inventer un parcours approprié (ici, en largeur) pour ajuster les transitions d'échecs. QO12 était une question d'ouverture finale pour obtenir une automate efficace.

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Tous les points	97	74	89	69	49	20	20	6	23	0	0
Réponses partielles	97	94	94	86	77	23	20	9	23	0	0

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8	QO9	QO10	QO11	QO12
Tous les points	23	57	26	54	34	9	29	29	9	9	0	0
Réponses partielles	100	100	94	94	77	43	46	31	11	9	0	0

TABLE 2. Pourcentages de réponses correctes et partielles à chaque question du sujet 2.

Ordre de visite

Épreuve pratique d'algorithmique et de programmation
Concours commun des Écoles normales supérieures

Durée de l'épreuve : 3 heures 30 minutes

Juin/Juillet 2023

ATTENTION !

N'oubliez en aucun cas de recopier votre u_0
à l'emplacement prévu sur votre fiche réponse

Important.

Il vous a été donné un numéro u_0 qui servira d'entrée à vos programmes. Les réponses attendues sont généralement courtes et doivent être données sur la fiche réponse fournie à la fin du sujet. À la fin du sujet, vous trouverez en fait deux fiches réponses. La première est un exemple des réponses attendues pour un $\widetilde{u_0}$ particulier (précisé sur cette même fiche et que nous notons avec un tilde pour éviter toute confusion !). Cette fiche est destinée à vous aider à vérifier le résultat de vos programmes en les testant avec $\widetilde{u_0}$ au lieu de u_0. Vous indiquerez vos réponses (correspondant à votre u_0) sur la seconde et vous la remettrez à l'examinateur à la fin de l'épreuve.

En ce qui concerne la partie orale de l'examen, lorsque la description d'un algorithme est demandée, vous devez présenter son fonctionnement de façon schématique, courte et précise. Vous ne devez en aucun cas recopier le code de vos procédures !
Quand on demande la complexité en temps ou en mémoire d'un algorithme en fonction d'un paramètre n, on demande l'ordre de grandeur en fonction du paramètre, par exemple : $O(n^2)$, $O(n \log n)$,...
Il est recommandé de commencer par lancer vos programmes sur de petites valeurs des paramètres et de ***tester vos programmes sur des petits exemples que vous aurez résolus préalablement à la main ou bien à l'aide de la fiche réponse type fournie en annexe.*** Enfin, il est recommandé de lire l'intégralité du sujet avant de commencer afin d'effectuer les bons choix de structures de données dès le début.

Les Parties 1 et 2 (OCaml) sont indépendantes des Parties 3 et 4 (C).

Il est précisé au début de chaque partie le langage à utiliser pour l'implémentation. Cette consigne doit impérativement être suivie. Il est à noter que le jury inspectera le code fourni et reproduira l'obtention des résultats à l'aide de votre code.

Ainsi, il est impératif de nous fournir sur votre clé usb votre fichier OCaml, ainsi que l'ensemble de votre code C que l'on compilera avec les commandes indiquées en début de Partie 3.

1 Génération pseudo-aléatoire de structures en OCaml

Vous devez utiliser OCaml tout au long de cette partie.

Étant donné u_0, on définit par récurrence

$$u_{t+1} := 19\ 999\ 991 u_t \pmod{19\ 999\ 999}$$

pour tout $t \in \mathbb{N}$, c'est-à-dire u_{t+1} est le reste de la division euclidienne par 19 999 999 du produit entre 19 999 991 et u_t.

Question 1 *Calculer les valeurs de u_t pour les valeurs de paramètres suivantes :*

a) $u_{50} \pmod{1\ 000}$ ***b)*** $u_{100} \pmod{1\ 000}$

c) $u_{1\ 000} \pmod{1\ 000}$ ***d)*** $u_{5\ 000} \pmod{1\ 000}$

1.1 Graphes orientés

Un graphe orienté fini $G = (V, A)$ est défini par :
- $V \subseteq \mathbb{N}$, son ensemble (fini) de sommets ;
- un ensemble $A \subseteq V^2 \setminus \{(v, v) \mid v \in V\}$ d'arcs consistant en des paires de sommets de V.

On notera que l'on interdit explicitement les boucles, c'est-à-dire des arcs joignant un sommet à lui-même.

Dans un graphe orienté, pour un sommet $x \in V$, on note $N^+(x)$ le voisinage sortant de x, consistant en l'ensemble des sommets atteignables en au plus un arc depuis x :

$$N^+(x) \coloneqq \{y \mid (x, y) \in A\}.$$

De même, le voisinage entrant pour un sommet $x \in V$ sera défini de la façon suivante :

$$N^-(x) \coloneqq \{y \mid (y, x) \in A\}.$$

Par extension, pour un ensemble $U \subseteq V$ de sommets de G, on définira le voisinage sortant de U :

$$N^+(U) \coloneqq \{y \in N^+(x) \mid x \in U\}$$

et entrant de U :

$$N^-(U) \coloneqq \{y \in N^-(x) \mid x \in U\}.$$

1.2 Génération pseudo-aléatoire de graphes orientés

On définit pour tout $n, m \in \mathbb{N}^*$ avec $m < 1\ 000$, le graphe orienté $G(n, m) = (\{0, \ldots, n-1\}, A)$ avec A défini de la façon suivante :

$$A \coloneqq \{(x, y) \mid x \neq y \wedge (u_{n+m+7x+11y} \pmod{1\ 000} < m)\}.$$

Question 2 *Calculer le nombre d'arcs des graphes suivants :*

a) $G(100, 100)$ ***b)*** $G(100, 500)$ ***c)*** $G(1\ 000, 100)$ ***d)*** $G(1\ 000, 500)$

1.3 Ajout de colorations

Étant donné un graphe orienté $G = (V, A)$, une coloration de ce graphe est la donnée d'une fonction $\lambda : V \to \{0, \ldots, p-1\}$. Un graphe coloré est un triplet $C = (V, A, \lambda)$.

À partir d'un graphe $G(n, m)$ défini précédemment, on définit un graphe coloré $C(n, m, p)$ en y ajoutant la fonction λ suivante :

$$v \in V \mapsto u_{5v} \pmod p.$$

La somme de contrôle d'un graphe colorié $C = (V, A, \lambda)$ est définie comme : $|A| + \sum_{v \in V} \lambda(v)$.

Question 3 *Calculer la somme de contrôle des graphes coloriés suivants :*

a) $C(777, 222, 5)$ ***b)*** $C(777, 500, 25)$ ***c)*** $C(1\ 234, 222, 25)$ ***d)*** $C(1\ 234, 500, 75)$

Question à développer pendant l'oral 1 *Décrire la structure de données que vous avez choisie pour représenter les graphes. Votre réponse comportera une estimation de l'espace utilisé en fonction des paramètres n, m et p. Vous devez particulièrement insister sur l'impact du paramètre m dans votre choix.*

Le graphe peut être représenté sous forme de listes d'adjacences.
En effet, pour $m = 500$ on gagne un facteur 2 en espace sur la représentation par matrices d'adjacence.
Pour $m = 222$, environ un facteur 5.
Ceci peut ne pas paraître obligatoire étant donné que ça ne rend pas pour autant le graphe épars.

2 Observations sur des graphes colorés en OCaml

Vous devez utiliser OCaml tout au long de cette partie.

Étant donné un graphe orienté coloré $C = (V, A, \lambda)$, un chemin de C de longueur k est une suite $(v_0, \ldots, v_k)$ de sommets de V telle que pour tout $0 \leq i < k$, $(v_i, v_{i+1}) \in A$. v_0 est appelé la source de ce chemin.

Une observation sur C de longueur k est une suite $(\lambda(v_0), \ldots, \lambda(v_k))$ de couleurs de C quand $(v_0, \ldots, v_k)$ est un chemin de C de longueur k.

L'ensemble des observations possibles depuis un sommet v du graphe est défini comme :

$$O(v) \coloneqq \{(\lambda(v_0), \ldots, \lambda(v_k)) \mid k \geq 0,\ (v_0, \ldots, v_k) \text{ est un chemin de source } v \text{ (i.e., } v = v_0) \text{ dans } C\}.$$

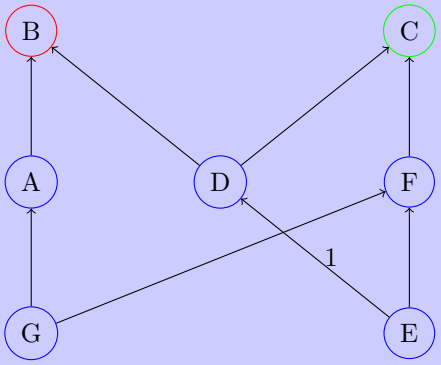
Une simulation réciproque sur C est une relation binaire R sur V telle que $(v, w) \in R$ si et seulement si :
- $\lambda(v) = \lambda(w)$;
- $\forall v' \in N^+(v),\ \exists w' \in N^+(w),\ (v', w') \in R$;
- $\forall w' \in N^+(w),\ \exists v' \in N^+(v),\ (v', w') \in R$.

Deux sommets v et w sont indistinguables, noté $v \simeq w$, si et seulement si il existe une simulation réciproque R telle que $(v, w) \in R$.

Notre objectif dans cette première partie de l'épreuve est de partitionner les sommets de V suivant le critère d'indistinguabilité.

Question à développer pendant l'oral 2 *Est-ce que $v \simeq w$ est équivalent à $O(v) = O(w)$? Justifier rapidement ou donner un contre-exemple.*

$v \simeq w$ n'est pas équivalent à $O(v) = O(w)$. La figure suivante présente un contre-exemple :



L'ensemble des observations depuis les sommets G et E sont identiques : $\{(B, B, R), (B, B, V)\}$. Néanmoins suite à la transition 1 du sommet E vers le sommet D, il n'est pas possible de trouver une transition depuis G vers un sommet simulant D.

2.1 Raffinement de partition

Soit un ensemble E, une partition $\mathcal{L} = (\mathcal{I}_1, \ldots, \mathcal{I}_l)$ est une liste ordonnée de sous-ensembles disjoints (appelés classes) de E dont l'union est E.

Raffiner une partition $\mathcal{L}$ vis-à-vis d'un ensemble $S \subseteq E$ consiste à diviser chaque classe $\mathcal{I}_\alpha \in \mathcal{L}$ en deux : $\mathcal{I}_\alpha \cap S$ et $\mathcal{I}_\alpha \setminus S$, et ne garder ces ensembles que s'ils sont non vides.

Algorithme 1 Raffinement de partition

Entrée: une partition $\mathcal{L} = (\mathcal{I}_1, \ldots, \mathcal{I}_l)$ d'un ensemble E et un sous-ensemble $S \subseteq E$.
Sortie: une partition raffinée $\mathcal{L}' = (\mathcal{I'}_1, \ldots, \mathcal{I'}_m)$.

```
pour chaque classe I_α faire
    soit J les éléments de I_α qui sont dans S
    retirer J de I_α
    si I_α est vide alors
        remplacer I_α par J
    sinon si J n'est pas vide alors
        ajouter J avant I_α
    fin si
fin pour
renvoyer L
```

On note $\mathsf{R}_E(\mathcal{L}, S)$ l'action de raffiner une partition $\mathcal{L}$ de E par un sous-ensemble S de E à l'aide de l'algorithme 1.

Après avoir raffiné une partition $\mathcal{L}$ par rapport à un ensemble S, plus aucune classe ne chevauche S : pour toute classe $\mathcal{I}_\alpha \in \mathcal{L}'$, $S \cap \mathcal{I}_\alpha = \mathcal{I}_\alpha$ ou $S \cap \mathcal{I}_\alpha = \emptyset$.

Implémentation et test du raffinement de partition. Soit $G(n, m)$ un graphe orienté pseudo-aléatoire et soit $r \geq 0$.
Pour $0 \leq i \leq r$, on note v_i le sommet $u_{n+m+i} \mod n$. Nous allons séquentiellement raffiner V par $N^+(v_i)$, on note $\mathcal{L}_r$ la partition ainsi obtenue :

$$\mathcal{L}_r \coloneqq \mathsf{R}_V(\ldots \mathsf{R}_V(\mathsf{R}_V(V, N^+(v_0)), N^+(v_1)) \ldots, N^+(v_r))$$

Noter que l'on a implicitement identifié V dans la formule précédente avec l'unique partition de V ayant V pour seule classe.

Question 4 *Calculer $|\mathcal{L}_r|$ pour les couples de graphes et de r suivants :*

a) $G(111, 222), 5$ ***b)*** $G(111, 500), 100$ ***c)*** $G(1\ 234, 222), 5$ ***d)*** $G(1\ 234, 500), 100$

2.2 Calcul itératif de la relation $\simeq$

Nous allons maintenant calculer itérativement la relation $\simeq$.

Deux sommets v et w sont indistinguables pour des observations de longueur 0, noté $v \simeq_0 w$, si et seulement si :

— $\lambda(v) = \lambda(w)$.

Deux sommets v et w sont indistinguables pour des observations de longueur au plus $k \geq 1$, noté $v \simeq_k w$, si et seulement si :

— $\lambda(v) = \lambda(w)$;
— $\forall v' \in N^+(v)\ \exists w' \in N^+(w),\ v' \simeq_{k-1} w'$;
— $\forall w' \in N^+(w)\ \exists v' \in N^+(v),\ v' \simeq_{k-1} w'$.

Il est clair que pour tout k, on peut identifier une classe d'équivalence de $\simeq_k$ avec un sous-ensemble de V. On peut alors représenter l'ensemble des classes d'équivalences de la relation $\simeq_k$ par une partition de V.

Nous voulons maintenant nous aider du raffinement de partition pour calculer $\simeq_{k+1}$ à partir de $\simeq_k$.

Question à développer pendant l'oral 3 *Présenter comment s'aider du raffinement de partition pour passer d'une représentation de $\simeq_k$ à une représentation de $\simeq_{k+1}$. Aucune preuve formelle n'est attendue.*

Soit $\mathcal{L}_k = (\mathcal{I}_1, \ldots, \mathcal{I}_l)$ une partition représentant $\simeq_k$, $\mathcal{L}_{k+1}$ une partition représentant $\simeq_{k+1}$ peut être obtenue de la façon suivante :

$$\mathcal{L}_{k+1} \coloneqq \mathsf{R}_V(\ldots \mathsf{R}_V(\mathsf{R}_V(\mathcal{L}_k, N^-(\mathcal{I}_1)), N^-(\mathcal{I}_2)) \ldots, N^-(\mathcal{I}_\ell))$$

Question 5 *Soit $C(n, m, p)$ un graphe orienté colorié précédemment défini. Calculer le nombre de classes d'équivalence pour $\simeq$ sur :*

a) $C(3\ 000, 1, 4)$ ***b)*** $C(3\ 001, 2, 4)$ ***c)*** $C(3\ 002, 900, 200)$ ***d)*** $C(3\ 002, 900, 224)$

Question à développer pendant l'oral 4 *Donner le pseudo-code de l'algorithme que vous avez utilisé pour calculer $\simeq$ sous la forme d'une partition. Justifier de la terminaison de votre algorithme.*

```python
def raffiner(partition, ensemble):
    nouvelle_partition = list()
    for classe in partition:
        c1 = classe.intersection(ensemble)
        c2 = classe.difference(ensemble)
        if c1 : nouvelle_partition.append(c1)
        if c2 : nouvelle_partition.append(c2)
    return nouvelle_partition

def calcul_equivalence(n, m, p):
    graphe, coloration = construire_graphe_colore(n, m, p, inverser = True)
    partition = list()
    for c in range(p): # construire les classes initiales, une pour chaque couleur
        classe = set(filter(lambda x : coloration[x] == c, range(n)))
        if classe:
            partition.append(classe)
    while True:
        partition_tmp = partition.copy()
        for classe in partition:
            ens = None
            # construire ens = N^-1(classe)
            tmp = [set(graphe[x]) for x in classe] # avoir inversé les arcs est
            ↪  utile ici
            ens = set.union(*tmp)
            partition_tmp = raffiner(partition_tmp, ens)
        if len(partition_tmp) == len(partition): # si point-fixe atteint
            break
        partition = partition_tmp # swap
    return partition
```

3 Algorithmique des graphes en C

Vous devez utiliser C tout au long de cette partie.

3.1 Code de base fourni et explication du jeu de données

Vous trouverez dans un dossier nommé `base` les fichiers suivants : `makefile`, `graphes.txt`, `main.c`, `graphe.c`, `graphe.h`, `utils.c` et `utils.h`.

Le fichier `makefile` contient les informations à destination de la commande `make` afin de permettre la compilation de votre code. N'y touchez que si vous savez ce que vous faites.

Pour compiler, il suffit d'invoquer la commande `make` depuis le répertoire `base`, par exemple :

```
cd base
make
./main
```

Le fichier `graphes.txt` est en lecture seule, et contient l'ensemble des graphes sur lesquels nous allons travailler (ci-après, appelé le jeu de données). Vous devez donc calculer vos réponses à partir du fichier `graphes.txt`.

Vous n'avez pas besoin de consulter son contenu car les entrées/sorties et le stockage en mémoire des graphes est déjà implémenté. Pour tester votre code et produire le résultat, veillez simplement à calculer vos valeurs à partir du nombre de graphes demandés.

Le fichier `main.c` contient la partie principale du code et c'est dans ce fichier que vous êtes encouragés à écrire vos solutions. Vous pourrez aussi avoir besoin de modifier la structure de votre graphe à partir du fichier `graphe.h`. Enfin, `graphe.c`, `utils.c` et `utils.h` contiennent les primitives de base que nous avons implémentées pour vous.

Les graphes chargés à partir du fichier sont stockés sous forme de liste d'adjacence, plus précisément :

- Chaque structure de graphe `struct graphe` contient un champ `size_t taille`[1] indiquant le nombre de sommets dans ce graphe ; puis un pointeur vers le premier élément d'un tableau de longueur `taille` de `struct sommet`, ordonné par ordre croissant d'identifiants.
  N. B. : Les identifiants sont indicés de 0 à `taille` − 1.
- Chaque `struct sommet` contient des informations relatives au sommet : son identifiant `size_t identifiant` et un pointeur sur le premier élément d'une liste simplement chaînée de `struct arete` représentant sa liste d'adjacence, elle aussi ordonnée suivant les identifiants des sommets ; vous êtes libre d'y ajouter des champs supplémentaires.
- Chaque `struct arete` contient l'identifiant du voisin dans un champ `size_t idvoisin` et un pointeur vers l'élément suivant dans la liste simplement chaînée des voisins.

Les valeurs d'exemple des réponses attendues ont été calculées uniquement à partir des $X = 4$ premiers graphes du jeu de données.

Définitions et suite de l'épreuve. Un graphe non-orienté fini $G = (V, E)$ est défini par :

- $V \subseteq \mathbb{N}$, son ensemble (fini) de sommets ;
- un ensemble $E \subseteq \{e \in 2^V \mid |e| = 2\}$ d'arêtes consistant en des ensembles (de cardinal strictement 2) de sommets de V.

Dans un graphe non-orienté, pour un sommet $x \in V$, on note $N(x)$ le voisinage de x consistant en l'ensemble des sommets reliés à x, c'est-à-dire $N(x) \coloneqq \{y \mid \{x, y\} \in E\}$.

Dans un graphe non-orienté, on appellera un cycle une suite d'arêtes consécutives distinctes (chaîne) dont les deux sommets extrémités sont identiques. Si la chaîne est élémentaire, c'est-à-dire ne passe pas deux fois par un même sommet (les extrémités pouvant néanmoins être identiques), alors on parle de cycle élémentaire. Un graphe est dit acyclique si il ne contient aucun cycle élémentaire.

Tous les graphes du jeu de données sont non-orientés et sans boucles.

3.2 Arbres et forêts

Un arbre est un graphe acyclique connexe, une forêt est un graphe acyclique.

Question 6 *Soit a le nombre d'arbres parmi les graphes du jeu de données. Soit f le nombre de forêts parmi les graphes du jeu de données. Calculer les valeurs suivantes :*

a) a ***b)*** f

Question à développer pendant l'oral 5 *Présenter le pseudo-code de l'algorithme pour les arbres, et faire l'étude de sa complexité temporelle.*

3.3 Ponts

Un pont dans un graphe est une arête dont la suppression entraîne une augmentation du nombre de composantes connexes. C'est à dire qu'il existe deux sommets du graphe qui étaient précédemment joignables, mais qui ne le sont plus quand on supprime cette arête. Notre but est de proposer un algorithme permettant de compter le nombre de ponts dans un graphe.

Pour ce faire, nous allons utiliser une approche basée sur le parcours en profondeur d'un graphe. Nous appellerons "ordre de visite" l'ordre obtenu en numérotant les sommets visités par un parcours en profondeur avec le rang auquel ils ont été visités. L'algorithme 2 présente la construction d'un tel ordre dans un tableau global nommé `rang`.

1. Le type de données `size_t` en C est un type d'entiers non signés utilisé entre autres pour stocker la taille de tableaux.

Algorithme 2 Parcours en profondeur (DFSRANG) – Construction du rang

Entrée: un graphe non-orienté $G = (V, E)$, un sommet r et un entier i.
Sortie: le rang du dernier sommet visité à partir de r par le parcours en profondeur.

```
rang[r] ← i
marquer r comme visité
pour chaque voisin v de r faire
    si v n'a pas déjà été visité alors
        i ← DFSRANG(G, v, i + 1)
    fin si
fin pour
renvoyer i
```

Pour identifier les ponts dans un graphe, nous allons nous baser sur l'observation suivante. Disons que l'on est au cours de l'exécution d'un parcours en profondeur sur un graphe donné, et que l'on est en train d'itérer sur les voisins d'un sommet r. L'arête actuelle (r, v) est un pont si et seulement si, en excluant cette arête, ni le sommet v, ni les sommets visités à partir de v par le parcours en profondeur n'ont d'arêtes vers le sommet r ou un sommet visité avant r. En effet, cette condition signifie qu'il n'existe pas d'autre possibilité que l'arête (r, v) pour aller de r à v.

Au sein d'une même composante connexe, être visité avant r signifie avoir un rang plus petit que celui de r, et être visité à partir de v signifie avoir un rang compris entre $\text{rang}(v) + 1$ et le rang retourné par le parcours en profondeur sur v.

Question à développer pendant l'oral 6 *Présenter le pseudo-code l'algorithme en découlant, et faire l'étude de sa complexité temporelle.*

Algorithme 3 Calcul du nombre de ponts dans une composante connexe (DFSPONTS)

Entrée: un graphe non-orienté $G = (V, E)$, un sommet r, son parent p et un entier i.
Sortie: le rang du dernier sommet visité à partir de r par le parcours en profondeur.

```
1:  visite[r] ← Vrai
2:  rang[r] ← i
3:  minrang[r] ← i
4:  pour chaque voisin v ≠ p ∈ N(r) faire
5:      si visite[v] = Faux alors
6:          i ← DFSPONTS(G, v, r, i + 1)
7:          minrang[r] ← MIN(minrang[r], minrang[v])
8:          si ordre[r] < minrang[v] alors
9:              nbponts ← nbponts + 1 // (r,v) est un pont
10:         fin si
11:     sinon // arête retour
12:         minrang[r] ← MIN(minrang[r], rang[v])
13:     fin si
14: fin pour
15: renvoyer i
```

Algorithme 4 Calcul du nombre de ponts dans un graphe

Entrée: un graphe non-orienté $G = (V, E)$.
Sortie: le nombre de ponts dans G.

```
1:  nbponts ← 0
2:  visite[v] ← Faux, ∀v ∈ V
3:  rang[r] ← ⊥, ∀v ∈ V
4:  minrang[r] ← ∞, ∀v ∈ V
5:  pour chaque sommet r ∈ V faire
6:      si visite[r] = Faux alors
7:          DFSPONTS(G, r, r, 0)
8:      fin si
9:  fin pour
10: renvoyer nbponts
```

La complexité temporelle de l'algorithme 4 est en $\mathcal{O}(|V| + |E|)$. En effet :

- L'ensemble des initialisations effectuées par l'algorithme 4 et sa boucle en ligne 5 se font en $\mathcal{O}(|V|)$.
- DFSPONTS est appelé au plus une fois sur chaque sommet : tout appel sur un sommet r est conditionné au fait que visite$[r]$ = Faux et la ligne 1 de l'algorithme 3 met visite$[r]$ à Vrai.
- Itérer sur les voisins de v en ligne 4 de l'algorithme 3 se fait en temps $\mathcal{O}(N(v))$ car on utilise une représentation du graphe par listes d'adjacence. Le temps total, agrégé sur l'ensemble des appels, passé en ligne 4 par l'algorithme 3 est donc en $\mathcal{O}(|E|)$.
- Chaque arête est considérée au plus deux fois (la seconde fois pour éliminer le parent en ligne 4 car on est non-dirigé).

Question 7 *On note p le nombre total de ponts pour l'ensemble des graphes du jeu de données. Implémenter la méthode ci-dessus afin de calculer la valeur suivante :*

a) p

3.4 Points d'articulation

Un point d'articulation est un sommet dont la suppression entraîne une augmentation du nombre de composantes connexes. C'est-à-dire qu'il existe deux sommets du graphe qui étaient précédemment joignables, mais qui ne le sont plus quand on supprime ce point d'articulation. Notre but est de proposer un algorithme permettant de compter le nombre de points d'articulation dans un graphe.

Nous allons à nouveau utiliser une approche basée sur le parcours en profondeur d'un graphe. Nous appellerons l'arbre d'un parcours en profondeur l'arbre dans lequel chaque sommet visité par un parcours en profondeur a pour parent le sommet depuis lequel il a été visité. L'algorithme 5 présente la construction d'un tel arbre, représenté dans un tableau global nommé parent, associant à chaque sommet l'identifiant de son parent.

Algorithme 5 Parcours en profondeur (DFSPARENTS) – Construction de l'arbre

Entrée: un graphe non-orienté $G = (V, E)$, un sommet r et le parent p.
Sortie: le sous-arbre du parcours en largeur, enraciné à r, représenté dans parent.

```
parent[r] ← p
marquer r comme visité
pour chaque voisin v de r faire
    si v n'a pas déjà été visité alors
        DFSPARENTS(G, v, r)
    fin si
fin pour
```

Pour identifier les points d'articulation dans un graphe, nous allons nous baser sur l'observation suivante. Dans l'arbre du parcours en profondeur, un sommet r est un point d'articulation si et seulement si l'une des deux conditions suivantes est vraie :

— r est la racine de l'arbre du parcours en profondeur et il a au moins deux fils dans cet arbre ;
— r n'est pas la racine de l'arbre du parcours en profondeur et il a un fils v tel qu'aucun sommet dans le sous-arbre du parcours en profondeur enraciné à v n'ait une arête vers un ancêtre de r.

Question à développer pendant l'oral 7 *Présenter le pseudo-code l'algorithme en découlant, et faire l'étude de sa complexité temporelle.*

Algorithme 6 Calcul du nombre d'articulations dans une composante connexe (DFSARTICULATIONS)

Entrée: un graphe non-orienté $G = (V, E)$, un sommet r, son parent p et un entier i.
Sortie: le rang du dernier sommet visité à partir de r par le parcours en profondeur.

```
visite[r] ← Vrai
rang[r] ← i
minrang[r] ← i
nbfils ← 0
estArticulation ← Faux // pour éviter de compter deux fois ce sommet
pour chaque voisin v ≠ p ∈ N(r) faire
    si visite[v] = Faux alors
        i ← DFSARTICULATIONS(G, v, r, i + 1)
        minrang[r] ← MIN(minrang[r], minrang[v])
        si r ≠ p et rang[r] ≤ minrang[v] et estArticulation = Faux alors
            nbarticulations ← nbarticulations + 1 // r est un point d'articulation ; second cas
            estArticulation ← Vrai
        fin si
        nbfils ← nbfils + 1
    sinon // arête retour
        minrang[r] ← MIN(minrang[r], rang[v])
    fin si
fin pour
si r = p et nbfils > 1 alors
    nbarticulations ← nbarticulations + 1 // r est un point d'articulation ; premier cas
fin si
renvoyer i
```

Algorithme 7 Calcul du nombre de points d'articulation dans un graphe

Entrée: un graphe non-orienté $G = (V, E)$.
Sortie: le nombre de ponts dans G.

```
nbarticulations ← 0
visite[v] ← Faux, ∀v ∈ V
rang[r] ← ⊥, ∀v ∈ V
minrang[r] ← ∞, ∀v ∈ V
pour chaque sommet r ∈ V faire
    si visite[r] = Faux alors
        DFSARTICULATIONS(G, r, r, 0)
    fin si
fin pour
renvoyer nbarticulations
```

La complexité temporelle de l'algorithme 7 est en $\mathcal{O}(|V| + |E|)$. En effet :

— L'ensemble des initialisations effectuées par l'algorithme 7 et sa boucle en ligne 5 se font en $\mathcal{O}(|V|)$.
— DFSARTICULATIONS est appelé au plus une fois sur chaque sommet : tout appel sur un sommet r est conditionné au fait que visite$[r]$ = Faux et la ligne 1 de l'algorithme 6 met visite$[r]$ à Vrai.
— Itérer sur les voisins de v en ligne 6 de l'algorithme 6 se fait en temps $\mathcal{O}(N(v))$ car on utilise une représentation du graphe par listes d'adjacence. Le temps total, agrégé sur l'ensemble des appels, passé en ligne 6 par l'algorithme 6 est donc en $\mathcal{O}(|E|)$.
— Chaque arête est considérée au plus deux fois (la seconde fois pour éliminer le parent en ligne 6 car on est non-dirigé).

Pour la correction, car ce n'est pas forcément dit explicitement dans les indications du sujet, on utilise le fait que pour détecter un ancêtre de r (cas 2), il suffit de vérifier que le rang est plus petit ou égal à celui de r.

Question 8 *On définit a le nombre de points d'articulation pour l'ensemble des graphes du jeu de données. Implémenter la méthode ci-dessus afin de calculer la valeur suivante :*

a) a

3.5 Plus longs chemins

La projection dirigée $D_G^< = (V, E_G^<)$ d'un graphe non-orienté $G = (V, E)$, par rapport à une relation d'ordre totale et stricte $<$ sur V, est définie par :

— la donnée V de son ensemble de sommets ;
— d'un ensemble $E_G^< \coloneqq \{(u, v) \in E^2 \mid \{u, v\} \in E, u < v\}$ d'arcs consistant en des paires de sommets de V.

On considère maintenant la projection dirigée des graphes du jeu de données, par rapport à la relation ordonnant les sommets par identifiants croissants.

Question 9 *Pour chaque graphe du jeu de données, on calcule la longueur maximale d'un chemin. Puis on somme toutes ces valeurs pour obtenir l. Calculer la valeur suivante :*

a) l

Vous serez évalué sur votre capacité à proposer, et implémenter, une solution la plus efficace possible.

Question à développer pendant l'oral 8 *Présenter le pseudo-code de votre algorithme pour calculer la longueur maximale d'un chemin dans un des graphes ainsi obtenus, donner sa complexité temporelle et prouver sa correction. Expliciter comment vous avez adapté votre algorithme au vu de la structure de données utilisée dans le code de base fourni.*

Algorithme 8 Calcul du plus long chemin dans un DAG

Entrée: un graphe acyclique dirigé $G = (V, E)$ avec $V := \{0, \ldots, |V|\}$, tel que l'ordre sur les entiers de V soit un ordre topologique du graphe.
Sortie: la longueur du plus long chemin dans G.

```
1: resultat ← 0
2: mlongueur[v] ← 0, ∀v ∈ V
3: pour chaque sommet r ∈ V par ordre d'identifiant décroissant faire
4:     pour chaque voisin v ∈ N(r) (i.e. tel que v > r) faire
5:         mlongueur[r] ← MAX(mlongueur[r], mlongueur[v] + 1)
6:     fin pour
7:     resultat ← MAX(resultat, mlongueur[r])
8: fin pour
9: renvoyer resultat
```

La complexité temporelle de l'algorithme 8 est en $\mathcal{O}(|V| + |E|)$. En effet :

- L'ensemble des initialisations effectuées par l'algorithme 8 se fait en $\mathcal{O}(|V|)$.
- La boucle en ligne 3 s'effectue au plus $\mathcal{O}(|V|)$ fois.
- Itérer sur les voisins de r en ligne 4 de l'algorithme 8 se fait en temps $\mathcal{O}(N(v))$ car on utilise une représentation du graphe par listes d'adjacence. Le temps total, agrégé sur l'ensemble des appels, passé en ligne 4 par l'algorithme 8 est donc en $\mathcal{O}(|E|)$.

L'algorithme 8 explicite en lui-même comment bénéficier de la structure de données utilisée dans le sujet. Pour la correction, il est attendu que les candidats formalisent la propriété de sous-structure optimale. Soit $r \in V$ considérons $c = rv_1 \cdots v_k$ un chemin de longueur maximale dans G ayant pour source r, ce chemin est nécessairement de la forme $r < v_1 < \cdots < v_k$, par hypothèse sur le graphe d'entrée ($<$ est topologique). On peut prouver aisément que le chemin $v_1 \cdots v_k$ est un chemin de longueur maximale dans G parmi les chemins ayant v_1 pour source. En effet, si $v_1 v'_2 \cdots v'_{k'}$ est un chemin dans G strictement plus long que $v_1 \cdots v_k$, alors $rv_1 v'_2 \cdots v'_{k'}$ est aussi un chemin dans G et est strictement plus long que $rv_1 \cdots v_k$, ce qui contredit la maximalité de c.
On obtient ainsi, en notant $\mathsf{mlongueur}[r]$ la longueur maximale dans G d'un chemin de source r, l'équation dynamique suivante :

$$\mathsf{mlongueur}[r] = \underset{v \in N(r)}{\mathrm{MAX}} \left(\mathsf{mlongueur}[v] + 1\right).$$

Finalement, la longueur maximale d'un chemin dans ce graphe est la valeur maximale trouvée dans mlongueur.

4 Détection de graphes cordaux en C

Vous devez utiliser C tout au long de cette partie.

Soit $G = (V, E)$ un graphe non-orienté et soit $V' \subseteq V$ un sous-ensemble de V. On définit le sous-graphe induit par V' sur G, dénoté $G(V')$, comme étant le graphe $(V', E \cap 2^{V'})$, c'est-à-dire G restreint aux sommets de V' et à l'ensemble des arêtes de E reliant les sommets de V' entre eux.

Une clique dans G est un sous-ensemble de sommets de V deux à deux reliés entre eux par E.

Un graphe est dit cordal quand pour tous ses cycles élémentaires de quatre sommets ou plus, il existe deux sommets non consécutifs du cycle reliés par une arête. Une définition équivalente consiste à s'assurer de l'existence d'un ordre d'élimination simplicial, c'est-à-dire un ordre total $\sigma = (v_1, \ldots, v_n)$ sur les sommets de G tel que pour tout $v_i \in V$, le voisinage de v_i dans $G(v_{i+1}, \ldots, v_n)$ – le sous-graphe induit par $\{v_{i+1}, \ldots, v_n\}$ sur G – est une clique. On admettra qu'un graphe est cordal si et seulement si il admet un ordre d'élimination simplicial.

Question à développer pendant l'oral 9 *Présenter succinctement la méthode que vous avez utilisée pour construire les permutations d'une liste d'entiers, et prouver sa correction.*

```python
def permutations(lst):
    if len(lst) == 0:
        return []
    if len(lst) == 1:
        return [lst]

    l = []
    for i in range(len(lst)):
       pivot = lst[i]
       restes = lst[:i] + lst[i+1:]
       for p in permutations(restes):
           l.append([pivot] + p)
    return l

sommets = list('123')
for p in permutations(sommets):
    print(p)
```

Question à développer pendant l'oral 10 *Étant donnée une permutation des sommets de V présentée sous la forme d'une liste d'entiers, et compte tenu de la représentation du graphe utilisée dans ce sujet :*

- *Quelle est la complexité en temps et en espace pour vérifier que cette permutation est bien un ordre simplicial ?*
- *Peut-on obtenir une meilleure complexité temporelle ? Quelle serait alors la complexité en espace ? Il ne vous est pas demandé d'implémenter cette solution alternative.*

Algorithme 9 Vérification d'un ordre simplicial

Entrée: un graphe acyclique dirigé $G = (V, E)$ avec $V \coloneqq \{0, \dots, |V|\}$ et L une permutation des entiers de V sous forme d'une liste chaînée.
Sortie: Vrai si L est un ordre simplicial, Faux sinon.

```
1:  vu[v] ← Faux, ∀v ∈ V
2:  pour chaque sommet r ∈ L faire
3:      vu[r] ← Vrai
4:      pour chaque voisin v ∈ N(r) tel que vu[v] = Faux faire
5:          pour chaque voisin v' ∈ N(r) tel que v' ≠ v et vu[v'] = Faux faire
6:              si v' ∉ N(v) alors
7:                  renvoyer Faux
8:              fin si
9:          fin pour
10:     fin pour
11: fin pour
12: renvoyer Vrai
```

La complexité temporelle de l'algorithme 9 est en $\mathcal{O}(|V|^4)$ dans le pire cas et $\mathcal{O}(|V|)$ en espace. En effet :

- L'initialisation de vu effectuée par l'algorithme 9 se fait en temps et en espace $\mathcal{O}(|V|)$.
- La boucle en ligne 2 s'effectue $\mathcal{O}(|V|)$ fois.
- Itérer sur les voisins de r en ligne 4 se fait en temps $\mathcal{O}(N(r))$ ($\mathcal{O}(|V|)$ dans le pire cas).
- Itérer sur les voisins de r en ligne 5 se fait en temps $\mathcal{O}(N(r))$ ($\mathcal{O}(|V|)$ dans le pire cas).
- Itérer sur les voisins de v en ligne 6 se fait en temps $\mathcal{O}(N(v))$ car on utilise une représentation du graphe par listes d'adjacence ($\mathcal{O}(|V|)$ dans le pire cas).

On peut baisser la complexité temporelle de cet algorithme en $\mathcal{O}(|V|^3)$, en transformant la représentation du graphe en une matrice d'adjacence (ce qui est faisable en $\mathcal{O}(|V|^2)$ – initialisations comprises). En effet, ceci permet de vérifier la condition de la ligne 6 en temps constant.

Le désavantage de cette approche étant qu'il faut maintenant un espace en $\mathcal{O}(|V|^2)$ pour stocker la matrice d'adjacence.

Question 10 *Soit c le nombre de graphes cordaux parmi les $Y_0 = 10$ premiers graphes du jeu de données. Implémenter cette caractérisation afin de calculer la valeur suivante :*

a) c

Fiche réponse type : Ordre de visite

$\widetilde{u_0}$: XXu0XX

Question 1

a) XX1.aXX

b) XX1.bXX

c) XX1.cXX

d) XX1.dXX

Question 2

a) XX2.aXX

b) XX2.bXX

c) XX2.cXX

d) XX2.dXX

Question 3

a) XX3.aXX

b) XX3.bXX

c) XX3.cXX

d) XX3.dXX

Question 4

a) XX4.aXX

b) XX4.bXX

c) XX4.cXX

d) XX4.dXX

Question 5

a) XX5.aXX

b) XX5.bXX

c) XX5.cXX

d) XX5.dXX

Question 6

a) XX6.aXX

b) XX6.bXX

Question 7

a) XX7.aXX

Question 8

a) XX8.aXX

Question 9

a) XX9.aXX

Question 10

a) XX10.aXX