BANQUE MP 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 : 16,9 %
- Paris-Saclay : 13,2 %
- Ulm : 13,3 %

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

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 2 : Des marches dans des graphes.

Ce sujet abordait différents problèmes de calcul de marche de poids optimal dans des graphes. Le sujet commence de façon classique par la génération de graphe et leurs sommes de contrôle. La somme de contrôle est là pour vérifier que la procédure de génération des graphes des candidat · es est correcte. Ensuite, le sujet étudie plusieurs problèmes de marche dans des graphes, suivant diverses classes de stratégies. La première partie s'intéressait à la marche de concert de nombreux agents suivant une stratégie sans mémoire commune. Dans la seconde partie, un seul agent était considéré. Enfin, la dernière question étudiait des marches infinies.

Pour la partie écrite, les questions Q1 à Q5 étaient de simples exercices de programmation et ont été correctement traitées par presque tout · es les candidat · es. Les questions Q6 et Q7 demandaient de faire se déplacer simultanément un grand nombre d'agents sur des graphes. Bien que la majorité des candidat · es ait proposé une solution, réussir à résoudre les grands cas demandaient plusieurs optimisations pour factoriser les déplacements des agents, ce que peu de candidat · es ont réussi. Dans la question Q8, il s'agissait de faire se déplacer un seul agent suivant une stratégie sans mémoire pour un très grand nombre de pas. Résoudre totalement cette question totalement demandait d'exploiter le fait qu'un tel

agent finit nécessairement dans une boucle, permettant d'accélérer le calcul de sa marche. Peu de candidat · es l'ont remarqué. Dans les questions suivants, les candidat · es devaient trouver des stratégies optimales au départ d'un sommet donné, mais cette fois avec mémoire. La première approche, proposée dans la question Q9, demandait une exploration exhaustive. Bien que standard, un faible nombre de candidat · es ont abordé cette question, probablement par manque de temps. La question Q10 n'était presque pas guidée, et pouvait se résoudre efficacement par programmation dynamique, ce que seuls les meilleurs candidat · es ont vu. Enfin, la question Q11 se réduisait à chercher des cycles de poids moyen optimal. Une approche exhaustive était possible mais insuffisante pour résoudre les plus grand cas, qui demandait une approche plus efficace, aussi par programmation dynamique. Un seul candidat · e a réussi à totalement résoudre cette question.

Pour la partie orale, nous nous contentons de discuter quelques questions notables. La question QO2 a été bien traité dans l'ensemble, bien que le jury a été étonné que certains candidat · es ne sachent pas motiver leur choix de représentation des graphes par des listes ou matrices d'adjacences. Lors de l'analyse de complexité de la question QO4, quelques candidat · es ont considéré — probablement par inattention — pouvoir calculer l'union de deux listes en $\mathcal{O}(1)$. Pour obtenir une bonne complexité dans la question QO5, il était nécessaire d'utiliser une structure de données supportant des unions en $\mathcal{O}(1)$, par exemple en utilisant un arbre binaire (même non équilibré). Surprenamment, très peu de candidat · es l'ont vu. Dans leur réponse à la question QO8, tout les candidat · es n'ont pas pensé à majorer le nombre de chemins de longueur L au départ d'un sommet par d^L ou d est le degré maximal du graphe.

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Tous les points	97	100	97	94	97	38	25	31	25	13	3
Réponses partielles	100	100	100	100	100	84	81	84	31	25	13

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8	QO9	QO10
Tous les points	94	78	75	69	13	34	63	13	9	3
Réponses partielles	100	100	100	94	94	84	69	59	22	9

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

Des marches dans des graphes

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.

1 Préliminaires

1.1 Notations

On rappelle que pour deux entiers naturels a et b, $(a \mod b)$ désigne le reste de la division entière de a par b, c'est-à-dire l'unique entier r avec $0 \leq r < b$ tel que $a = k \times b + r$ pour $k \in \mathbb{N}$.

Ce sujet portant sur des graphes pondérés orientés, chaque utilisation du mot graphe doit être entendue comme graphe pondéré orienté. Un graphe est donc, pour ce sujet, un triplet $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$ où $\mathcal{N}$ est l'ensemble fini des sommets ou nœuds du graphe ; $\mathcal{E} \subset \mathcal{N}^2$ est l'ensemble des arcs, chaque arc étant une paire de sommets ; et $\mathsf{poids} : \mathcal{N} \to \mathbb{N}$ est une fonction associant à chaque sommet du graphe un poids positif. Notez que cette définition interdit les arcs multiples (deux sommets ne peuvent être reliés que zéro ou une fois), mais qu'il est possible d'avoir un arc d'un sommet s à lui-même, par l'arc (s, s). Un arc (s, t) de $\mathcal{E}$ est appelé un arc sortant de s, et le sommet t est un successeur de s. On notera $\mathsf{succ}_{\mathcal{G}}(s)$ l'ensemble des successeurs de s dans $\mathcal{G}$:

$$\mathsf{succ}_{\mathcal{G}}(s) = \{t \mid (s, t) \in \mathcal{E}\}$$

Par souci de simplicité, dans ce sujet, $\mathcal{N}$ est toujours constitué des nombres de 0 à $n - 1$ avec n le nombre de sommets.

1.2 Génération de nombres pseudo-aléatoires

Étant donné u_0, on définit la récurrence :

$$\forall t \in \mathbb{N},\ \ u_{t+1} = (909\,091 \times u_t) \mod 1\,010\,101\,039$$

L'entier u_0 vous est donné, et doit être recopié sur votre fiche réponse avec vos résultats. Une fiche réponse type vous est donnée en exemple, et contient tous les résultats attendus pour une valeur de u_0 différente de la vôtre (notée $\widetilde{u_0}$). Il vous est conseillé de tester vos algorithmes avec cet $\widetilde{u_0}$ et de comparer avec la fiche de résultats fournie. Pour chaque calcul demandé, avec le bon choix d'algorithme le calcul ne devrait demander qu'au plus de l'ordre de quelques secondes, jamais plus d'une minute.

On aura souvent besoin de nombreuses valeurs consécutives de la suite u_n. Il est donc conseillé que votre implémentation calcule le tableaux de tous les u_n jusqu'à un certain rang.

Question 1 *Calculer les valeurs suivantes :*

a) $u_1 \mod 1000$

b) $u_{12} \mod 1000$

c) $u_{1\,234} \mod 1000$

d) $u_{7\,654\,321} \mod 1000$

```
let next_u (x : int) : int = 909_091 * x mod 1_010_101_039

(* Tableau des n premiers éléments de la suite, dans l'ordre
   [array_un = \[u_0, ..., u_{n-1}\]] *)
let array_un (n : int) : int array =
  let a = Array.make n 0 in

  let rec fill (i : int) (ui : int) =
    if i = n then ()
    else begin a.(i) <- ui; fill (i + 1) (next_u ui) end
  in

  fill 0 u0;
  a
```

1.3 Extraction d'une sous-liste croissante

Soit L une liste $(a_1, \ldots, a_n)$ d'entiers positifs. On note $\mathsf{incr\text{-}list}(L)$ la sous-liste de L obtenue en ne gardant un a_i que si il est plus grand que tous les éléments précédents dans la liste. Par exemple, $\mathsf{incr\text{-}list}(1, 4, 2, 3, 6, 5) = (1, 4, 6)$, puisque 2 et 3 ont été exclus car plus petits que 4 (qui les précède), et de même 5 a été exclu car plus petit que 6.
Formellement, si $L = (a_1, \ldots, a_n)$ alors $\mathsf{incr\text{-}list}(L) = (b_1, \ldots, b_k)$ telle que $(b_1, \ldots, b_k)$ est une liste d'entiers strictement croissante ($b_i < b_j$ quand $i < j$) et :

$$\{b_1, \ldots, b_k\} = \{a_i \mid \forall j < i,\, a_j < a_i\}$$

Question 2 *Soit v_n la suite telle que $\forall n \in \mathbb{N}$, $v_n = u_n \mod 9$. On pose L_n la liste $(v_0, \ldots, v_{n-1})$. Implémenter la fonction $\mathsf{incr\text{-}list}(\cdot)$, et utiliser votre implémentation pour calculer la somme des entiers de $\mathsf{incr\text{-}list}(L_n)$, c'est à dire $\left(\sum_{s \in \mathsf{incr\text{-}list}(L_n)} s\right)$ pour les valeurs de n suivants :*

a) $n = 3$ ***b)*** $n = 6$ ***c)*** $n = 123$ ***d)*** $n = 1\,234$

Pour tester votre code, vous pouvez vous aider du fait que pour $\widetilde{u_0}$ on a $L_6 = (5, 5, 4, 3, 7, 2)$ et donc $\mathsf{incr\text{-}list}(L_6) = (5, 7)$.

```
let extract_sublist (l : int list) : int list =
  let rec extr (max : int) (l : int list) (acc : int list) =
    match l with
    | [] -> List.rev acc
    | h :: t ->
      if h > max then extr h t (h :: acc)
      else extr max t acc
  in
  extr (-1) l []

(* build list `L_n` *)
let ln : int -> int list =
  let un = array_un 7654322 in
  fun n -> List.init n (fun i -> un.(i) mod 9)
```

Question à développer pendant l'oral 1 *Donner la complexité en temps de votre algorithme.*

Question (très) simple pour commencer.
Soit n la longueur de la liste. Une implémentation naïve garde un élément u_i en testant tous les u_j pour $j < i$, et est donc en $\mathcal{O}(n^2)$. On peut faire mieux, en parcourant la liste de gauche à droite, et en gardant à tout instant le plus grand élément déjà rencontré, ce qui donne une implémentation en $\mathcal{O}(n)$.

1.4 Génération de graphe

Étant donné trois entiers strictement positifs n, M et p, on note $G(n, M, p)$ le graphe $(\mathcal{N}, \mathcal{E}, \mathsf{poids})$ où $\mathcal{N} = \{0, \dots, n-1\}$ et pour chaque sommet $s \in \mathcal{N}$:

— La liste des arcs sortants potentiels du sommet s est la liste :

$$(s, t_0), \dots, (s, t_{M-1}) \qquad \text{où } \forall 0 \leq i < M,\ t_i = u_{s \times M + i} \mod n.$$

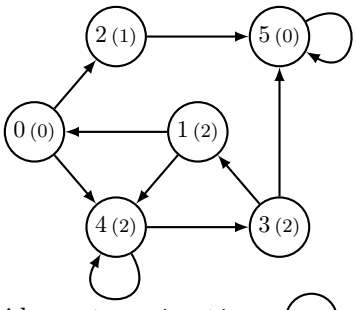
La liste des arcs sortants de s est obtenue en ne gardant dans $(s, t_0), \dots, (s, t_{M-1})$ que les arcs vers des sommets croissants. Plus précisément,

$$\mathsf{succ}_{\mathcal{G}}(s) = \big\{ t_j \mid t_j \in \mathsf{incr\text{-}list}(t_0, \dots, t_{M-1}) \big\}.$$

On remarquera que tous les sommets ont au plus M successeurs. De plus, puisque (s, t_0) est toujours un arc sortant de s, tous les sommets ont au moins 1 successeur.

— Le poids $\mathsf{poids}(s)$ du sommet s est $(u_{n \times M + s} \mod p)$.

Un exemple de graphe est donné Figure 1.



Légende : Un sommet s de poids p est représenté par  .

FIGURE 1 – Le graphe $G(6, 3, 3)$ pour $\widetilde{u_0}$.

On appelle degré sortant d'un sommet s le nombre de successeurs de s, c'est à dire $|\mathsf{succ}_{\mathcal{G}}(s)|$.

Question 3 *Implémenter une fonction calculant le graphe $G(\cdot, \cdot, \cdot)$, et l'utiliser pour calculer la somme $\sum_{s \in \mathcal{N}} |\mathsf{succ}_{\mathcal{G}}(s)|$ des degrés sortants des sommets des graphes $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$ suivants :*

a) $G(6, 3, 10)$ ***b)*** $G(123, 7, 10)$ ***c)*** $G(1\,234, 10, 10)$ ***d)*** $G(10\,001, 22, 10)$

```
(* Un graphe dirigé [(n,d,poids)] :
   - [n] est le nombre de sommets, numérotés de [0] à [n-1].
   - [d] est le tableau des listes de transitions pour chaque sommet
     c-à-d que si [d.(i) = \[x1,...,xq\]] alors on a les transitions
     [i → x1], ..., [i → xq]
   - [p(i)] est le poids du [i]-ème sommet *)
type graph = int * int list Array.t * (int -> int)

let nodes (g : graph) : int = let n, _, _ = g in n

(* graphe à [n] sommets, [m] arcs, pour le paramètre [p] *)
let graph ~(n : int) ~(m : int) ~(p : int) : graph =
  let ln = array_un (n * (m + 1)) in

  let e =
    Array.init n
      (fun i ->
         List.init m (fun j -> ln.( i * m + j) mod n) |>
         extract_sublist
      )
  in
  (* poids, avec un tableau pré-calculé *)
  let p : int -> int =
    let w = Array.init n (fun i -> ln.( (n * m + i)) mod p) in
    fun i -> w.(i)
  in
  (n, e, p)

let sum_degrees (g : graph) : int =
  let _,e,_ = g in
  Array.fold_left (fun sum l -> sum + List.length l) 0 e
```

Question à développer pendant l'oral 2 *Décrire la structure de données que vous avez choisie pour représenter les graphes.*
On exprimera l'ordre de grandeur de l'espace mémoire utilisé par votre représentation en fonction du nombre de sommets n et du nombre d'arcs m des graphes.

On représente le graphe comme un tableau de n cases, où la i^e case contient la liste des arcs sortants de i. Il y a au plus $n \times M$ arcs dans le graphe, donc cette représentation est en $\mathcal{O}(n \times M)$.
La représentation des poids est un simple tableau de n entiers, et est donc en $\mathcal{O}(n)$. Au total, on a donc une représentation en $\mathcal{O}(n \times M)$.
Si on note m le nombre d'arcs, notre représentation est en $\mathcal{O}(n + m)$.
Utiliser des listes d'adjacence est préférable ici, car M est toujours beaucoup plus petit que n dans ce sujet, et qu'on n'a jamais besoin de vérifier rapidement si un arc (i, j) est présent dans le graphe.

À partir de maintenant, on ne comptera pas le temps de calcul des graphes, ni leur espace mémoire, dans les évaluations de complexités d'algorithmes.

Hachage d'un graphe On définit une fonction de hachage sur les graphes comme suit : pour tout graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$,

$$\mathsf{hash}(\mathcal{G}) = \Big(\sum_{s \in \mathcal{N}} \sum_{t \in \mathsf{succ}_{\mathcal{G}}(s)} (s+1) \times (t + \mathsf{poids}(t)^2) \Big) \mod 1000.$$

Question 4 *Calculer la valeur de $\mathsf{hash}(\mathcal{G})$ pour les graphes $\mathcal{G}$ suivants :*

a) $G(6, 3, 10)$ ***b)*** $G(123, 7, 10)$ ***c)*** $G(1\,234, 10, 10)$ ***d)*** $G(10\,001, 22, 10)$

```
let rec ( *^ ) : int -> int -> int =
  fun i j -> if j = 0 then 1 else i * (i *^ (j - 1))

let hash_graph (g : graph) : int =
  let n,e,p = g in
  let sum = ref 0 in
  Array.iteri (fun i l ->
      sum :=
        List.fold_left (fun sum j ->
            sum + (i + 1) * (j + (p j) *^ 2)
          ) !sum l
    ) e;

  !sum mod 1000
```

2 Marches sur les graphes

Une stratégie sans mémoire π pour un graphe $(\mathcal{N}, \mathcal{E}, \mathsf{poids})$ est une fonction associant à chaque sommet s l'un de ses successeurs :

$$\pi : \mathcal{N} \to \mathcal{N} \quad \text{telle que} \quad \forall s \in \mathcal{N}, \ (s, \pi(s)) \in \mathcal{E}.$$

Notez qu'il n'existe pas de telle fonction si le graphe possède un sommet sans successeur. Comme les sommets des graphes de ce sujet ont tous au moins un successeur, cela ne sera pas un problème.

Étant donné un graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$ avec $\mathcal{N} = \{0, \dots, n-1\}$, on note $\pi_0(\mathcal{G})$ la stratégie sans mémoire choisissant dans chaque sommet son plus petit successeur :

$$\forall s \in \mathcal{N}, \ \ \pi_0(\mathcal{G})(s) = \min\{t \in \mathcal{N} \mid (s,t) \in \mathcal{E}\}.$$

2.1 Marches simples

Étant donné un graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$, une marche (ou chemin) ρ sur $\mathcal{G}$ est une liste de sommets $s_0, \dots, s_L$ reliés par des arcs, c'est à dire tel que $(s_i, s_{i+1}) \in \mathcal{E}$ pour tout $0 \leq i < L$. On appelle L la longueur de la marche ρ : il s'agit du nombre d'arcs dans ρ.

Étant donné un sommet initial s et un entier L, une stratégie π définit une marche de longueur L en partant du sommet s et en se déplaçant dans le graphe selon la stratégie π. Plus précisément, on note $\mathsf{marche}(\mathcal{G}, L, \pi, s)$ la marche $s_0, \dots, s_L$ telle que $s_0 = s$ et $s_{i+1} = \pi(s_i)$ pour tout $0 \leq i < L$.

Question 5 *Implémenter une fonction calculant la marche* $\mathsf{marche}(\cdot,\cdot,\cdot,\cdot)$.
Pour un graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$ *où* $\mathcal{N} = \{0, \ldots, n-1\}$, *évaluer la somme modulo* 1000 *des sommets de la marche de longueur* L *au départ du sommet* $n-1$ *dans* $\mathcal{G}$ *suivant la stratégie* $\pi_0(\mathcal{G})$. *C'est-à-dire que si* $\mathsf{marche}(\mathcal{G}, L, \pi_0(\mathcal{G}), n-1) = s_0, \ldots, s_L$, *alors on cherche la quantité :*

$$\left(\sum_{0 \leq i \leq L} s_i\right) \mod 1000$$

pour les graphes $\mathcal{G}$ *et longueurs* L *suivants :*

a) $G(6, 3, 10), L = 4$

b) $G(123, 7, 10), L = 20$

c) $G(1\,234, 10, 10), L = 100$

d) $G(10\,001, 22, 10), L = 5\,000$

```
type strat = int -> int

(* dans le sommet [i], la stratégie [pi0 i] choisit toujours
   le sommet de degré minimal dans le voisinage de [i] *)
let pi0 (g : graph) : strat =
  let n,d,_ = g in
  let strat =
    Array.init n
      (fun i ->
         List.fold_left (fun m j -> min m j) (List.hd d.(i)) d.(i)
      )
  in
  fun i -> strat.(i)

(* une marche *)
type run = int list

(* marche à l'envers de longueur [l] suivant [strat] dans [g] au
   départ de [s0].  [s0 → ... → sl] où [forall 0 <= i < l, s_{i+1} = strat
   s_i] *)
let run (strat : strat) (g : graph) (s0 : int) (l : int) : run =
  let rec exec (s0 : int) (l : int) acc =
    if l = 0 then (s0 :: acc) else exec (strat s0) (l - 1) (s0 :: acc)
  in
  exec s0 l []

let control_run (g : graph) ~(l : int) : int =
  let init = nodes g - 1 in
  let strat = pi0 g in
  let run = run strat g init l in
  ( List.fold_left (+) 0 run ) mod 1_000
```

Question à développer pendant l'oral 3 *Donner la complexité en temps et en mémoire de votre algorithme, en fonction du nombre de sommets* n, *du nombre d'arcs* m, *et de la longueur de la marche* L.

Le temps de calcul de $\pi_0(\mathcal{G})$ est en $\mathcal{O}(n \times M)$ ou $\mathcal{O}(n+m)$.
Le temps de calcul de la marche est en $\mathcal{O}(L)$.
La représentation mémoire de $\pi_0(\mathcal{G})$ est en $\mathcal{O}(n)$, et de la marche en $\mathcal{O}(L)$. Si on calcule le poids de la marche à la volée, alors on a un coût mémoire en $\mathcal{O}(1)$ (auquel il faut ajouter la représentation mémoire de $\pi_0(\mathcal{G})$).

2.2 Marches de concert avec seuil

Cette section est indépendante de la section 3.

Soit un graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$ et une stratégie π de $\mathcal{G}$. Dans cette section, nous allons étudier des marches de concert dans le graphe : à chaque instant, un certain nombre d'agents seront répartis sur les sommets du graphe, et ceux-ci se déplaceront de concert et simultanément sur le graphe en suivant la même stratégie. De plus, ces déplacements seront soumis à un effet de seuil : pour qu'un groupe d'agents présents dans un sommet se déplace, il sera nécessaire qu'ils soient strictement plus nombreux que le poids du sommet.

Plus précisément, on considère un ensemble fini d'agents $\mathcal{A} = \{0, \dots, n-1\}$ (il y a donc autant d'agents que de sommets). Ces agents sont répartis sur les sommets du graphe $\mathcal{G}$: il est possible qu'il y ait plusieurs agents sur le même sommet, et qu'il y ait des sommets sans agents. Initialement, chaque sommet du graphe contient exactement un agent : l'agent i est dans le sommet i. Tous les agents se déplacent sur le graphe en suivant la stratégie $\pi_0(\mathcal{G})$. Les agents essaient de se déplacer par groupe (tous les agents dans le sommet s se déplacent ensemble vers le sommet $\pi_0(\mathcal{G})(s)$), et les déplacements sont simultanés (tous les groupes d'agents se déplacent en même temps). Cependant, ces déplacements ne peuvent pas toujours avoir lieu : pour qu'un groupe d'agents $a_1, \dots, a_k$ dans un sommet s puisse se déplacer en $\pi_0(\mathcal{G})(s)$, il est nécessaire qu'ils soient strictement plus nombreux que le poids du sommet s, c'est à dire que $k > \mathsf{poids}(s)$; si $k \leq \mathsf{poids}(s)$, les agents $a_1, \dots, a_k$ restent dans le sommet s, en attendant que suffisamment d'autres agents les rejoignent. Une étape de déplacement de *tous* les groupes d'agents le pouvant est appelé un pas, et une séquence de pas décrit une marche de concert.
On souhaite effectuer une marche de concert pour un grand nombre de pas, et sur des graphes relativement importants. Pour être plus efficace, nous utiliserons la notion de sommet vivant : un sommet vivant est un sommet dans lequel il y a au moins un agent, et plus d'agents que le poids du sommet. Autrement dit, les sommets vivants sont ceux depuis lesquels les groupes d'agents vont se déplacer au prochain pas. La Figure 2 décrit un pas d'une marche de concert sur le graphe $G(6,3,3)$, ainsi que les sommets vivants lors de cette marche. Par exemple, il y a initialement deux sommets vivants, et il ne reste qu'un sommet vivant après deux pas.
Nous proposons de calculer la marche de concert en maintenant deux structures de données : la première contenant l'état du graphe, c'est-à-dire les agents présents dans chaque sommet ; et la seconde contenant l'ensemble des sommets vivants.

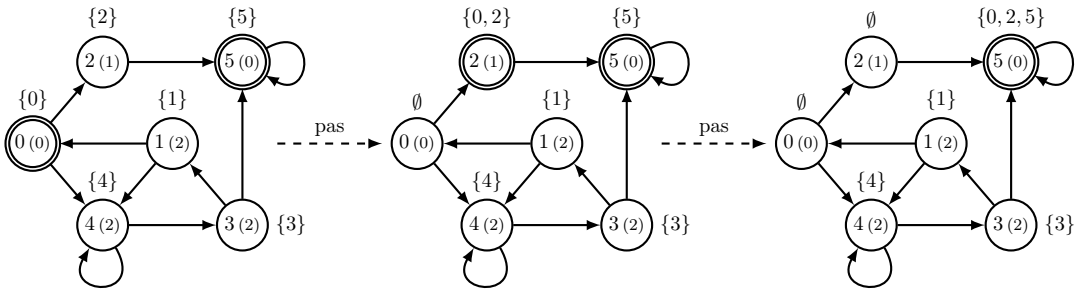
Question 6 *Implémenter l'algorithme réalisant une marche de concert à l'aide de l'approche proposée ci-dessus. Utiliser cette implémentation pour calculer le nombre de sommets vivants après L pas de la marche de concert dans les graphes suivants :*

a) $G(6,3,3), L = 4$

b) $G(1\,234, 10, 3), L = 100$

c) $G(10\,001, 22, 10), L = 50\,000$

d) $G(100\,001, 40, 10), L = 100\,000$



Légende : Les agents présents dans un sommet sont indiqués au dessus ou à côté de celui-ci. Les sommets vivants sont marqués d'un double cercle. Par exemple, avant le premier pas (donc dans le graphe le plus à gauche), les sommets 0 et 5 sont vivants.

FIGURE 2 – Deux pas d'une marche de concert dans le graphe $G(6, 3, 3)$ pour $\widetilde{u_0}$.

```
(* Pour un graphe à [n] sommets, [state] est un tableau de [n] éléments.
   La [i]-ème case de [state] contient une paire [(k,agents)] où :
   - [agents] est la liste des agents présents dans le sommet [i]
   - [k = List.length agents] *)
type state = (int * int list) array
```

```
(* retourne la paire de :
   - la liste des paires de sommets et des agents dans ces sommet
   - la liste des sommets vivants
   après [max_step] pas de la marche de concert sur le graphe [g]. *)
let concurrent_executions
    (strat : strat) (g : graph) (max_step : int) : (int * int list) list *
    ↪  int list
  =
  let n, _, p = g in
  let st : state = Array.init n (fun i -> (1, [i])) in

  (* Tout le monde avance d'un pas si possible.
     La liste [alive] contient la liste des sommets dans lesquels
     des agents sont présents : cette liste mise à jour est
     retournée par la fonction. *)
  let all_step1 (alive : int list) : int list =
    (* liste des agents ayant fait un pas *)
    let moved : (int * int * int list) list =
      List.concat_map (fun i ->
          let k, agents = st.(i) in
          (* on enlève les agents du sommet [i] *)
          let () = st.(i) <- (0,[]) in
          if k = 0 then [] (* personne à déplacer *)
          else
            let j = strat i in (* prochain sommet *)
            [j, k, agents]     (* déplacement ajouté à [moved] *)
        ) alive
    in

    (* applique les déplacements de [moved] dans l'état, en calculant le
       nouveau [alive] *)
    List.fold_left (fun alive (j,k,agents) ->
        (* agents déjà en [j] *)
        let k', agents' = st.(j) in
        let new_k = k + k' in
        let new_agents =
          (* optimisation : on concatène la plus petite des deux listes. *)
          if k < k' then agents @ agents' else agents' @ agents
        in
        st.(j) <- (new_k, new_agents);
        if new_k > p j then j :: alive else alive
      ) [] moved
  in

  (* Tout le monde avance de [step] pas si possible. *)
  let rec all_stepn (alive : int list) (step : int) : int list =
    if step = 0 then alive else all_stepn (all_step1 alive) (step - 1)
  in
  let init_alive : int list =
    let rec init (i : int) acc =
      if i < 0 then List.rev acc else
        init (i - 1) (if p i < 1 then i :: acc else acc)
    in
    init (n - 1) []
  in
  let alive = all_stepn init_alive max_step in
  let final_conf =
    Array.mapi (fun i (_, agents) -> i, agents) st |>
    Array.to_list
  in
  (* on élimine les doublons *)
  final_conf, List.sort_uniq Stdlib.compare alive
```

```
let nb_alive (g : graph) ~(l : int) : int =
  let strat = pi0 g in
  let conf, alive = concurrent_executions strat g l in
  List.length alive
```

Question 7 *On note $C(\mathcal{G}, L, s)$ l'ensemble des agents dans le sommet s après L pas de la marche de concert dans le graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$. Calculer la quantité :*

$$\left(\sum_{s \in \mathcal{N}} \sum_{a \in C(\mathcal{G},L,s)} s \cdot a\right) \ mod \ 1000$$

pour les graphes et nombres de pas suivants :

a) $G(6, 3, 3), L = 4$

b) $G(1\,234, 10, 3), L = 100$

c) $G(10\,001, 22, 10), L = 50\,000$

d) $G(100\,001, 40, 10), L = 100\,000$

```
let sum_agents (g : graph) ~(l : int) : int =
  let strat = pi0 g in
  let conf, _ = concurrent_executions strat g l in
  let sum =
    List.fold_left
      (fun sum (s,agents) ->
         sum + s * List.fold_left (+) 0 agents
      )
      0 conf
  in
  sum mod 1_000
```

Dans les deux questions à préparer à l'oral suivantes, on exprimera les coûts en temps et en mémoire en fonction, entre autres, du nombre de groupes d'agents et du nombre de groupes d'agents vivant au i-ème pas (pour i allant de 0 à L).
Remarque : on ne demande pas de trouver une formule caractérisant le nombre de groupes d'agents et de groupes d'agents vivants après i pas.

Question à développer pendant l'oral 4 *Décrire la structure de données que vous avez utilisée pour : i) stocker l'état du graphe ; et ii) stocker l'ensemble des sommets vivants.*
Donner leur espace mémoire, et le coût en temps pour mettre à jour ces deux structures de données lors de l'exécution d'un pas de la marche de concert.

On utilise :
- un tableau A de n cases où la i^{e} case contient la liste des agents présents dans le sommet i ;
- et la liste V des sommets vivants.

Soit V' la liste qui contiendra les sommets vivants après le pas de la marche de concert. On utilisera aussi une liste M dans laquelle on stockera (temporairement) des ordres de déplacement : M est une liste de paires $(j, [a_1, \dots, a_k])$ où $[a_1, \dots, a_k]$ sont des agents devant se déplacer en j.
Pour effectuer un pas de la marche de concert, il suffit d'itérer sur la liste des sommets vivants (donc $|\mathsf{V}|$ fois), et pour chaque sommet vivant :

(1) de faire avancer tous les agents dans $\mathsf{A}[i]$ d'un pas vers j, où $j = \pi_0(\mathcal{G})(i)$: on vide $\mathsf{A}[i]$, et on stocke l'ordre de déplacement $(j, \mathsf{A}[i])$ dans M (en $\mathcal{O}(1)$).
(2) d'ajouter j à la liste V' des sommets vivants après le pas de la marche de concert (en $\mathcal{O}(1)$).

Après cela, on applique tous les ordres de déplacement dans M : où chaque ordre $(j, [a_1, \dots, a_k])$ coûte $\mathcal{O}(k)$ (car on utilise des listes).

L'application des ordres de déplacement peut être optimisé, par exemple :

(i) en stockant dans A la paire de la liste des agents présents et de la longueur de la liste (idem dans M, on garde en mémoire le nombre d'agents), de façon à ce qu'appliquer l'ordre de déplacement $(j, k, [a_1, \dots, a_k])$ dans $\mathsf{A}[j] = (k', [b_1, \dots, b_{k'}])$ coûte la plus petite des deux listes, c'est à dire $\min(k, k')$. Cela donne un très bon coût amorti, voir question suivante.
(ii) En utilisant une structure de donnée plus complexe pour représenter les ensembles d'agents, qui supporte l'union en $\mathcal{O}(1)$. Par exemples, on pourrait utiliser des arbres.

Au total, la complexité en temps d'un pas est en $\mathcal{O}(|\mathsf{V}| \times d)$, où d est le nombre maximal d'agents dans un sommet vivant (avec l'optim. (ii), on est en $\mathcal{O}(|\mathsf{V}|)$).
La complexité en espace est en $\mathcal{O}(n)$.

Question à développer pendant l'oral 5 *Donner la complexité en temps et mémoire de l'évaluation de L pas de la marche de concert.*

Le nombre maximal d d'agents dans un sommet vivant au i^{e} pas peut être de l'ordre de n. Donc, sans optimisations, on est en $\mathcal{O}(\sum_{i \leq L} |\mathsf{V}_i| \times n)$.
L'optimisation (ii) donne une bien meilleure complexité, en $\mathcal{O}(\sum_{i \leq L} |\mathsf{V}_i|)$.
Une analyse fine de la complexité amortie de (i) montre que celle-ci est efficace. En effet, dans ce cas le coût d'application d'un ordre de déplacement vers un sommet vide est en $\mathcal{O}(1)$: seules les fusions d'agents sont coûteuses. Il y a au plus n fusions d'agents lors d'une marche complète, et une fusion coûte au plus $\mathcal{O}(n)$. On a donc une complexité totale en $\mathcal{O}((\sum_{i \leq L} |\mathsf{V}_i|) + n^2)$.

En bornant chaque $|\mathsf{V}_i|$ par n (ce qui est très grossier, puisqu'après un certain temps, $|\mathsf{V}_i|$ peut être bien plus petit que n), on obtient une complexité en : $\mathcal{O}(L \times n^2)$ sans optimisation, $\mathcal{O}(L \times n + n^2)$ avec (i), et $\mathcal{O}(L \times n)$ avec (ii).

3 Marche et stratégie optimale

Dans cette section, nous allons associer à chaque marche une valeur (entière), puis nous chercherons à déterminer la stratégie optimale à horizon fini L, c'est-à-dire la stratégie dont la marche associée (en partant du sommet 0) est de valeur maximale parmi toutes celles de longueur L.

3.1 Valeur d'une marche

On considère une notion de valeur d'une marche paramétrée par un entier $\alpha \in \mathbb{N}$. Étant donné une marche ρ sur un graphe $\mathcal{G}$, la valeur de ρ pour le paramètre α, que l'on note $\mathsf{valeur}_\alpha(\rho)$, est la somme du poids des sommets par lesquels passe ρ, en ne comptant pas plus de α fois le poids de chaque sommet. Plus précisément :

$$\mathsf{valeur}_\alpha(\rho) = \sum_{s \in \mathcal{N}} \mathsf{poids}(s) \cdot \min(\alpha, \mathsf{count}(s, \rho))$$

où $\mathsf{count}(s, \rho)$ compte le nombre de passage de ρ dans un sommet s.

Question 8 *Implémenter la fonction $\mathit{valeur}_\alpha(\cdot)$, et l'utiliser pour calculer*

$$\mathit{valeur}_\alpha(\rho) \mod 999$$

pour la marche ρ de longueur L partant du sommet 0 et suivant la stratégie $\pi_0(\mathcal{G})$, pour les graphes $\mathcal{G}$, paramètres α et longueurs de marche L suivants :

- ***a)*** $G(6, 3, 10), \alpha = 2, L = 6$
- ***b)*** $G(123, 7, 10), \alpha = 20, L = 100$
- ***c)*** $G(1\,234, 10, 10), \alpha = 1000, L = 100\,000$
- ***d)*** $G(10\,001, 22, 10), \alpha = 10\,000, L = 10\,000\,000$
- ***e)*** $G(100\,001, 40, 10), \alpha = 100, L = 1\,000\,000\,000\,000$
- ***f)*** $G(200\,002, 50, 10), \alpha = 100, L = 1\,000\,000\,000\,000$

Question à développer pendant l'oral 6 *Détailler votre algorithme et donner sa complexité en temps et en mémoire.*

Une marche suivant une stratégie sans mémoire suffisamment longue finit toujours dans un cycle. Un cycle élémentaire est de longueur au plus n, et après α répétitions, il est inutile de continuer à exécuter la marche. Donc il suffit de simuler la marche pour au plus $(n+1) \times \alpha$ pas (n pas pour atteindre le cycle, α répétitions du cycle).
De plus, il est inutile de simuler la marche pour plus de L pas.
Complexité pour n sommets et m arcs :
- temps : $\mathcal{O}(\min(L, n \times \alpha))$
- espace : $\mathcal{O}(1)$

3.2 Stratégie avec mémoire optimale

Nous allons maintenant chercher à déterminer la stratégie optimale parmi l'ensemble des marches d'une certaine longueur L.
Pour l'instant, nous n'avons considéré que des stratégies sans mémoire. Dans cette section, nous allons nous intéresser à des stratégies plus complexes, les stratégies avec mémoire. Une stratégie avec mémoire π_h pour un graphe $(\mathcal{N}, \mathcal{E}, \mathsf{poids})$ est une fonction qui choisit dans quel sommet se

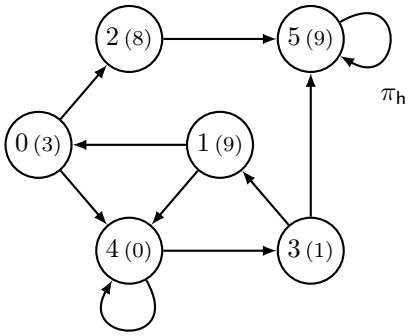
```
let value_min (a : int) (count : int array) (g : graph) : int =
  let n, _, p = g in
  let sum = ref 0 in
  (* pour chaque sommet [i] de [g], ajoute [min a (count(i))] *)
  Array.iteri (fun i c ->
      sum := !sum + p i * min a c
    ) count;

  !sum

(* version optimisée, qui profite de quand [a] est petit. *)
let run_value_opt ~(a : int) (g : graph) (strat : strat) ~(l : int) : int =
  let n,_,p = g in
  let count = Array.make n 0 in (* compteur du nombre de passages dans chaque
  ↪  état *)

  let s = ref 0 in              (* sommet courant, pas encore comptabilisé *)
  let cycle : int option ref = ref None in       (* début d'un cycle *)
  let i = ref 0 in              (* compteur de la longueur du run *)

  (* on cherche un cycle *)
  while (!i <= l && !cycle = None) do
    if count.(!s) > 0 then cycle := Some !s;     (* on a trouvé un cycle, on
    ↪  sort *)
    count.(!s) <- count.(!s) + 1;
    s := strat !s;
    incr i;
  done;

  let cycle : int = Option.get !cycle in
  let cpt_cycle = ref 1 in

  (* on fait au plus [a] tour dans le cycle *)
  while (!i <= l && !cpt_cycle < a) do
    count.(!s) <- count.(!s) + 1;
    s := strat !s;
    incr i;
    if !s = cycle then incr cpt_cycle; (* on a fait un cycle de plus *)
  done;

  (* sanity check, on vérifie qu'on a bien compté tous les sommets du cycles
  ↪  *)
  for _ = 0 to n do
    s := strat !s;
  done;

  value_min a count g
```

FIGURE 3 – Code OCaml correction question 8.

déplacer en fonction de l'historique des sommets déjà empruntés. Plus précisément, une stratégie avec mémoire π_{h} est une fonction associant à toute marche non-vide $\rho = s_0, \dots, s_l$ de $\mathcal{G}$ un successeur de s_l :

$$(s_l, \pi_{\mathsf{h}}(s_0, \dots, s_l)) \in \mathcal{E} \quad \text{pour toute marche } s_0, \dots, s_l \text{ de } \mathcal{G}$$

On note $\mathsf{marche}_{\mathsf{h}}(\mathcal{G}, L, \pi_{\mathsf{h}}, s)$ la marche de longueur L partant d'un sommet s et suivant la stratégie avec mémoire π_{h} dans le graphe $\mathcal{G}$. Un exemple de stratégie avec mémoire et de marche est donné Figure 4.



$$\pi_{\mathsf{h}}(s_0, \dots, s_l, 0) = 4 \qquad \pi_{\mathsf{h}}(s_0, \dots, s_l, 1) = 4$$

$$\pi_{\mathsf{h}}(s_0, \dots, s_l, 2) = 5 \qquad \pi_{\mathsf{h}}(s_0, \dots, s_l, 3) = 1 \qquad \pi_{\mathsf{h}}(4) = 4$$

$$\pi_{\mathsf{h}}(s_0, \dots, s_l, 4) = \begin{cases} 4 & \text{si } s_{l-1} \neq 4 \text{ (avec } l > 0) \\ 3 & \text{sinon} \end{cases}$$

$$\pi_{\mathsf{h}}(s_0, \dots, s_l, 5) = 5$$

$$\mathsf{marche}_{\mathsf{h}}(\mathcal{G}, 7, \pi_{\mathsf{h}}, 0) = (0, 4, 4, 3, 1, 4, 4, 3)$$

À droite, une stratégie avec mémoire π_{h} restant dans le sommet 4 deux fois de suite, et la marche de longueur 7 suivant π_{h} et partant de 0.

FIGURE 4 – Exemple de stratégie avec mémoire et de marche sur le graphe $G(6, 3, 10)$ pour $\widetilde{u_0}$.

Une stratégie π^o_{h} est optimale pour $\mathsf{valeur}_\alpha(\cdot)$ à horizon fini $L \in \mathbb{N}$ si la marche associée est de valeur maximale parmi toutes les marches de longueur L au départ du sommet 0, c'est-à-dire :

$$\pi^o_{\mathsf{h}} = \underset{\pi_{\mathsf{h}}}{\operatorname{argmax}} \Big(\mathsf{valeur}_\alpha(\mathsf{marche}_{\mathsf{h}}(\mathcal{G}, L, \pi_{\mathsf{h}}, 0)) \Big).$$

Question à développer pendant l'oral 7 *Donner un graphe $\mathcal{G}$ et un horizon $L \in \mathbb{N}$ tels qu'il existe une stratégie avec mémoire sur $\mathcal{G}$ de valeur strictement meilleure que toute stratégie sans mémoire sur $\mathcal{G}$ (à horizon L, au départ de 0).*

Le graphe $0, 1, 2$ de poids respectifs 1, 10, et 0, avec les transitions $0 \to 0$, $0 \to 1$, $1 \to 2$, et $2 \to 2$.
Alors, pour $L = 5$ par exemple, la meilleure stratégie au départ de 0 est d'attendre en 0 aussi longtemps que possible (ici, 4 pas), avant de passer en 1 pour obtenir le gain de 10 (qui ne peut être obtenu qu'une fois).

Question 9 *En énumérant toutes les marches de longueur L, calculer la valeur selon $\mathsf{valeur}_\alpha(\cdot)$ de la stratégie avec mémoire optimale au départ du sommet 0 pour les graphes $\mathcal{G}$, horizons L et*

paramètres α suivants :

a) $G(6,3,10), \alpha = 1, L = 6$

b) $G(6,3,10), \alpha = 6, L = 6$

c) $G(1\,234,10,10), \alpha = 2, L = 9$

d) $G(10\,001,22,10), \alpha = 2, L = 4$

Question à développer pendant l'oral 8 *Évaluer la complexité en temps et en mémoire de l'exécution de votre algorithme.*

Complexité pour n sommets, au plus M arcs sortant par sommet, et longueur de marche L.
On énumère toutes les marches de longueur L au départ de 0 dans le graphe. Il y a au plus M^L telles marches.
On n'a besoin de stocker que le chemin courant qu'on est en train d'énumérer, d'au plus L sommets.

— temps : $\mathcal{O}(M^L)$
— espace : $\mathcal{O}(L)$

3.3 Stratégie avec mémoire optimale : cas $\alpha > L$

Dans le cas où $\alpha > L$, la valeur $\mathsf{valeur}_\alpha(\rho)$ d'une marche $\rho = s_0, \ldots, s_L$ est la somme des poids des sommets de ρ, et peut être écrite :

$$\mathsf{valeur}_\alpha(s_0, \ldots, s_L) = \sum_{0 \leq i \leq L} \mathsf{poids}(s_i)$$

Puisque la valeur précise de α n'a plus d'importance quand $\alpha > L$, on pose $\mathsf{valeur}(\rho) = \mathsf{valeur}_{L+1}(\rho)$ où L est la longueur de ρ.

Question 10 *Calculer la valeur selon $\mathsf{valeur}(\cdot)$ de la stratégie avec mémoire optimale au départ du sommet 0 pour les graphes $\mathcal{G}$ et horizons L suivants :*

a) $G(6,3,10), L = 6$

b) $G(6,3,10), L = 200$

c) $G(1\,234,10,10), L = 40$

d) $G(10\,001,22,10), L = 20$

Question à développer pendant l'oral 9 *Détailler votre algorithme et donner sa complexité en temps et en mémoire.*

Par programmation dynamique. On calcule le tableau $a[l][i]$ qui contient la valeur de meilleure marche de longueur l au départ de i.
$a[l+1][i]$ se calcule en itérant sur tous les j tels que $i \to j$, donc en temps $\mathcal{O}(M)$.
On a seulement besoin de stocker la l^{e} ligne du tableau a pour calculer la $l+1^{\text{e}}$ ligne.
Complexité pour n sommets et au plus M arcs sortants par sommet :

— temps : $\mathcal{O}(L \times n \times M)$
— espace : $\mathcal{O}(n)$ (si on ne stocke pas les chemins)

```
(* la valeur d'un run [rho] dans [g] avec le coefficient [a]
   [\sum_i p i * min(occ[i],a)] *)
let run_value_min (a : int) (g : graph) (rho : run) : int =
  let n, e, p = g in
  let count = Array.make n 0 in

  (* compte combien de fois la marche [rho] est passée par
     chaque état *)
  List.iter (fun i -> count.(i) <- count.(i) + 1) rho;

  value_min a count g

(* Meilleur run pour [run_value] dans [g] à horizon fini [l] au départ
   de [0].
   Énumère toutes les marches.
   Pré-requis : [forall r, run_value g r > 0] *)
let best_run_enum ~run_value ~(l : int) (g : graph) : int * run =
  let n, e, p = g in

  (* Retourne la meilleure marche entre :
     - la meilleure marche de longueur [l] étendant [List.rev part_run]
     - la marche dans [acc]

     Invariants :
     - [i = List.hd part_run] (donc [part_run] est non-vide)
     - si [acc = (v,r)], alors [v = run_value g r]

     Tail-recursive *)
  let rec best_run
      (part_run : run) (i : int) (l : int) (acc : (int * run))
    : (int * run)
    =
    if l = 0 then begin
      let run = List.rev part_run in
      let value = run_value g run in
      let value', run' = acc in
      if value > value' then (value, run) else (value',run')
    end
    else
      List.fold_left (fun acc j ->
          best_run (j :: part_run) j (l - 1) acc
        ) acc (e.(i))
  in

  if l = 0 then 0, [] else
    (* meilleur run de longueur [l] étendant [0]. *)
    best_run [0] 0 l (-1, [])

let best_run_min ~a = best_run_enum ~run_value:(run_value_min a)
```

FIGURE 5 – Code OCaml correction question 9.

```
(* la valeur d'un run [rho] dans [g]
   [\sum_j p (rho j)] *)
let run_value_sum (g : graph) (rho : run) : int =
  let _, _, p = g in
  List.fold_left (fun sum j -> sum + p j) 0 rho

(* Meilleur run pour [run_value_sum] dans [g] à horizon fini [l]
   au départ de [0].
   Programmation dynamique. *)
let best_run_sum (g : graph) ~(l : int) : int * run =
  let n, e, p = g in

  (* meilleures marches de longueur [1] *)
  let a1 = Array.init n (fun i -> p i, [i]) in
  let a2 = Array.make n (-1, []) in

  (* [src] et [dst] sont des tableaux de longueur [n] :
     - [src.(i)] contient la meilleure marche (et sa valeur),
       de longueur [l] au départ de [i].
     - [dst.(i)] doit être remplit de même, mais pour les marches
       de longueur [l + 1]. *)
  let doit1 ~(src : (int * run) array) ~(dst : (int * run) array) =
    (* pour chaque sommet [i] *)
    Array.iteri (fun i _ ->
        (* itère sur les sommets [j] tel que [i → j], en cherchant la
           meilleur marche *)
        let value, run =
          List.fold_left (fun (value,run) j ->
              let valuej,runj = src.(j) in
              if valuej > value then (valuej,runj) else (value,run)
            ) (-1,[]) e.(i)
        in
        dst.(i) <- p i + value, i :: run
      ) dst
  in

  (* Similaire à [doit1], mais avance de [l] pas :
     - [src] doit contenir les meilleures marche de longueur [l]
     - retourne le tableau avec les meilleures marche de longueur [l + h]. *)
  let rec doit ~src ~dst (l : int) : (int * run) array =
    if l = 0 then src else
      let () = doit1 ~src ~dst in
      doit ~src:dst ~dst:src (l - 1) (* inverse les tableaux *)
  in

  if l = 0 then 0, [] else begin
    let res = doit ~src:a1 ~dst:a2 l in
    res.(0)
  end
```

FIGURE 6 – Code OCaml correction question 10.

3.4 Stratégie optimale pour des marches infinies

Nous considérons maintenant des marches infinies sur un graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$. Une marche infinie est une liste de sommets $(s_i)_{i\in\mathbb{N}}$ reliés par des arcs, c'est-à-dire telle que $(s_i, s_{i+1}) \in \mathcal{E}$ pour tout $i \in \mathbb{N}$. Plus précisément, nous allons nous intéresser aux marches ultimement cycliques. Une marche infinie est ultimement cyclique si elle est de la forme :

$$\underbrace{s_0, \dots, s_L}_{\text{préfixe fini}}, \underbrace{t_0, \dots, t_K}_{\text{cycle}}, \underbrace{t_0, \dots, t_K}_{\text{cycle}}, \dots$$

c'est-à-dire si la marche commence par un *préfixe fini* $s_0, \dots, s_L$ et se termine par un *cycle* $t_0, \dots, t_K$ se répétant un nombre infini de fois.
La valeur d'une telle marche infinie $\mathsf{valeur}((s_i)_{i\in\mathbb{N}})$ ne peut pas être la somme des poids des sommets s_i, puisque cette somme risque d'être divergente. À la place, nous considérerons que la valeur d'une marche infinie ultimement cyclique est la valeur moyenne des poids des sommets du cycle final, c'est-à-dire que si la marche $(s_i)_{i\in\mathbb{N}}$ finit par répéter le cycle $t_0, \dots, t_K$, alors

$$\mathsf{valeur}_\infty((s_i)_{i\in\mathbb{N}}) = \frac{\sum_{0\le i\le K} \mathsf{poids}(t_i)}{K+1}$$

Noter que cette valeur est bien défini, car elle ne dépend pas du cycle choisi.

On pose $H(n, M, p)$ le graphe obtenu à partir de $G(n, M, p)$ en remplaçant tout arc d'un sommet i vers lui-même par un arc de i vers $(i + 1 \mod n)$. Ces graphes sont sans boucles. Un exemple de tel graphe est donné Figure 7.

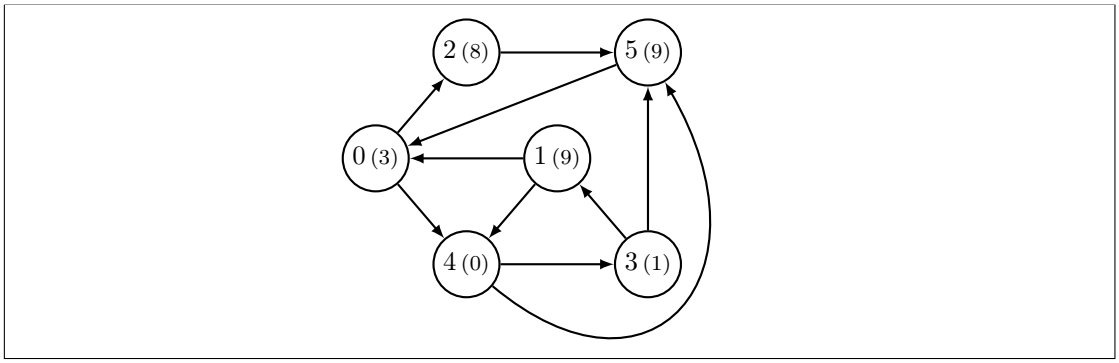


FIGURE 7 – Le graphe sans boucles $H(6, 3, 10)$ pour $\widetilde{u_0}$.

Dans la question suivante, on donnera la valeur d'une marche infinie avec 2 chiffres après la virgule.

Question 11 *Calculer la valeur selon $\mathsf{valeur}_\infty(\cdot)$ de la marche infinie ultimement cyclique (au départ de n'importe quel sommet) de valeur maximale dans les graphes $\mathcal{G}$ (sans boucles) suivants :*

a) $H(6, 3, 10)$ ***b)*** $H(10, 3, 10)$ ***c)*** $H(40, 4, 10)$ ***d)*** $H(99, 7, 10)$

Question à développer pendant l'oral 10 *Détailler votre algorithme et donner sa complexité en temps et en mémoire.*

```
let best_omega_run_any (g : graph) : float =
  let n, e, p = g in
  (* [a.(i).(j)]: value of the best path of length exactly [l] from
     [i] to [j], excluding [j]'s weight *)
  let a  = Array.init n (fun j -> Array.make n ([],min_int)) in
  (* copy, to store results at step [l+1] *)
  let a' = Array.init n (fun j -> Array.make n ([],min_int)) in
  (* initialise [a] for [l=1] *)
  for i = 0 to n-1 do
    List.iter (fun j -> a.(i).(j) <- ([i; j],p i)) e.(i)
  done;
  let best_cycle = ref (-1., []) in

  for l = 2 to n do
    (* compute in [a'] the values of the best path of length [l+1] *)
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        let bestp, best =
          List.fold_left (fun (bestp, best) i0 ->
              let i0_j, val_i0_j = a.(i0).(j) in
              let val_i_i0_j = p i + val_i0_j in
              if best < val_i_i0_j && val_i0_j <> min_int then
                (i :: i0_j, val_i_i0_j)
              else
                (bestp, best)
            ) ([],min_int) e.(i)
        in
        a'.(i).(j) <- (bestp, best)
      done;
    done;
    (* copy [a'] into [a] *)
    for i=0 to n-1 do for j=0 to n-1 do a.(i).(j) <- a'.(i).(j) done; done;
    for i = 0 to n-1 do
      let i_i, val_i_i = a.(i).(i) in
      let val_cycle = float_of_int val_i_i /. float_of_int l in
      if val_cycle > fst !best_cycle && val_i_i <> min_int then
        best_cycle := val_cycle, i_i
    done;
  done;

  let val_cycle, _ = !best_cycle in
  val_cycle

let move_self_loops (g : graph) : graph =
  let n, e, p = g in
  let new_e =
    Array.mapi (fun i l -> List.filter (fun j -> j <> i) l) e
  in

  Array.iteri (fun i l ->
      if List.mem i l then
        let next_i = (i+1) mod n in new_e.(i) <- next_i :: new_e.(i)
    ) e;

  (n, new_e, p)
```



```
let graph_nl ~n ~m ~p = move_self_loops (graph ~n ~m ~p)
```

FIGURE 8 – Code OCaml correction question 11.

Par programmation dynamique aussi. Le tableau $A[l][i][j]$ contient la valeur de la meilleure marche de longueur l de i à j, en excluant le poids de j. On utilise une valeur spéciale $\perp$ s'il n'y a pas de telle marche.
Alors $A[l+1][i][j]$ se calcule à partir des $A[l][i_0][j]$ pour $i \to i_0$, donc en $\mathcal{O}(M)$.
Il est suffisant de regarder les cycles élémentaires, donc de calculer $A[l]$ pour l allant jusqu'à n.
On a donc trois boucles imbriquées, chacune itérée n fois.
À la fin, on itère sur tout les $A[l][i][i]$ pour l de 1 à n, en cherchant le meilleur cycle.

Le tableau A est de taille $\mathcal{O}(n^3)$. Comme il suffit de se rappeler de $A[l]$ pour calculer $A[l+1]$, et en calculant le meilleur cycle au fur et à mesure, on peut descendre à $\mathcal{O}(n^2)$.

Complexité pour n sommets et au plus M arcs sortants par sommet :
- temps : $\mathcal{O}(n^3 \times M)$
- espace : $\mathcal{O}(n^2)$ (si on ne stocke pas les chemins)

Fiche réponse type : Des marches dans des graphes

$\widetilde{u_0}$: 104

Question 1

a) 464

b) 281

c) 919

d) 390

Question 2

a) 5

b) 12

c) 20

d) 20

Question 3

a) 10

b) 316

c) 3583

d) 36938

Question 4

a) 582

b) 629

c) 820

d) 92

Question 5

a) 25

b) 284

c) 382

d) 704

Question 6

a) 1

b) 63

c) 8

d) 57

Question 7

a) 61

b) 729

c) 651

d) 453

Question 8

a) 29

b) 108

c) 310

d) 41

e) 313

f) 461

Question 9

a) 30

b) 56

c) 78

d) 37

Question 10

a) 56

b) 1802

c) 363

d) 169

Question 11

a) 6.67

b) 4.75

c) 7.00

d) 7.57