

SESSION 2022

**AGREGATION
CONCOURS EXTERNE**

Section : INFORMATIQUE

ÉPREUVE SPÉCIFIQUE SELON L'OPTION CHOISIE :

- **ÉTUDE DE CAS INFORMATIQUE**
- **FONDEMENTS DE L'INFORMATIQUE**

Durée : 6 heures

L'usage de tout ouvrage de référence, de tout dictionnaire et de tout matériel électronique (y compris la calculatrice) est rigoureusement interdit.

Si vous repérez ce qui vous semble être une erreur d'énoncé, vous devez le signaler très lisiblement sur votre copie, en proposer la correction et poursuivre l'épreuve en conséquence. De même, si cela vous conduit à formuler une ou plusieurs hypothèses, vous devez la (ou les) mentionner explicitement.

NB : Conformément au principe d'anonymat, votre copie ne doit comporter aucun signe distinctif, tel que nom, signature, origine, etc. Si le travail qui vous est demandé consiste notamment en la rédaction d'un projet ou d'une note, vous devrez impérativement vous abstenir de la signer ou de l'identifier.

Tournez la page S.V.P.

Épreuve spécifique

Étude de cas informatique	1
Fondements de l'informatique	18

INFORMATION AUX CANDIDATS

Vous trouverez ci-après les codes nécessaires vous permettant de compléter les rubriques figurant en en-tête de votre copie.

Ces codes doivent être reportés sur chacune des copies que vous remettrez.

► Etude de cas informatique :

Concours	Section/option	Epreuve	Matière
EAE	6200A	103	9424

► Fondement de l'informatique :

Concours	Section/option	Epreuve	Matière
EAE	6200A	103	9425

Étude de cas informatique

Préliminaires

Le projet proposé s'inspire d'un projet réel. La procédure de conception utilisée dans le sujet s'inspire librement de l'approche agile avec une organisation en plusieurs parties. Chaque partie comprend la définition d'un certain nombre d'objectifs concrets, la réflexion sur les moyens de les atteindre, la préparation d'une procédure de validation permettant de vérifier si les objectifs sont atteints et une discussion critique de l'ensemble du processus.

Attendus. Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction.

Dépendances. Ce sujet contient quatre parties. La première partie est guidée de manière assez précise car elle sert de point de départ à l'ensemble des autres parties. Les autres parties sont largement indépendantes. Il est possible de les aborder dans l'ordre qui vous conviendra le mieux.

Partie I. Problème d'échecs

Le but de ce sujet est de développer la plate-forme **ChessMate** permettant de spécifier et résoudre des problèmes d'échecs tels que le problème des huit dames mais aussi d'autres problèmes tels que les problèmes des mats en N coups. **ChessMate** permettra également de pouvoir jouer une partie d'échecs.

Fonction de résolution du problème des huit dames

Nous allons commencer par le problème des huit dames : "Le but du **problème des huit dames** est de placer huit dames d'un jeu d'échecs sur un échiquier de 8 x 8 cases sans que les dames ne puissent se menacer mutuellement, conformément aux règles du jeu d'échecs (la couleur des pièces étant ignorée). Par conséquent, deux dames ne devraient jamais partager la même rangée, colonne, ou diagonale." (*Problème des huit dames, d'après Wikipedia*)

Le problème des huit dames a 92 solutions distinctes. La figure 1 en présente une. On propose la fonction `check_solution8queens` en Python pour vérifier si une solution au problème des huit dames est correcte. Cette fonction trie les cases où sont positionnées les dames sur l'échiquier et vérifie que les dames ne sont pas sur la même rangée ou colonne. Du moins c'est l'intention du programmeur.

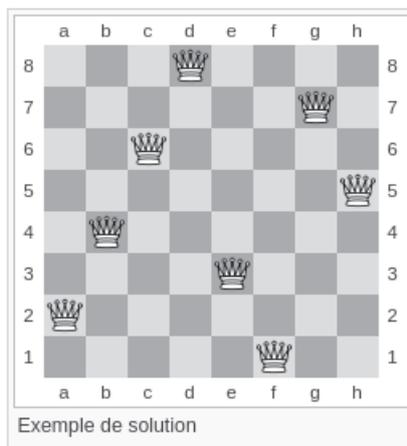


FIGURE 1 – Solution du problème des huit dames

Listing 1 – fonction `check_solution8queens`

```

1  from math import floor
2
3
4  def check_solution8queens(sol):
5      """
6      sol est une solution au probl\eme des 8 dames
7      sol est un ensemble de 8 valeurs
8      exemple: [8, 5, 20, 25, 42, 59, 54, 39]
9      correspondant \a une solution correcte (cf \echiquier ci-dessus)
10     retourne True si la solution est correcte, False sinon
11
12     l'\echiquier est repr\esent\e par une liste de 64 cases
13     la case 0 est le coin bas gauche
14     la case 63 est le coin haut droit
15     56 57 58 59 60 61 62 63
16     48 49 50 51 52 53 54 55
17     40 41 42 43 44 45 46 47
18     32 33 34 35 36 37 38 39
19     24 25 26 27 28 29 30 31
20     16 17 18 19 20 21 22 23
21     8  9  10 11 12 13 14 15
22     0  1  2  3  4  5  6  7
23     """
24
25     sortedSol = sorted(sol)
26     for i in range(len(sortedSol)):
27         square = sortedSol[i]
28         otherSquares = sortedSol[i+1:]
29         for otherSquare in otherSquares:
30             if square % 8 == otherSquare % 8:
31                 return False
32             if (floor(otherSquare/8) - floor(square/8)) == 0:
33                 return False
34
35     return True

```

Question 1. La fonction du listing 1 est en fait fautive car incomplète. Elle retourne True pour certaines solutions qui ne sont en fait pas correctes. Programmer trois tests qui montrent trois défauts différents de la fonction `check_solution8queens(sol)`. Le code de vos tests doit préciser

les données en entrée et le résultat attendu. Commenter en quoi vos trois tests vérifient des propriétés différentes.

L'échiquier avec une classe

Maintenant que nous avons testé la fonction `check_solution8queens(sol)`, on s'intéresse à l'encodage de l'échiquier. L'encodage proposé dans la fonction `check_solution8queens(sol)` n'est pas idéal. En principe, une case est identifiée par une lettre pour la colonne (de 'a' à 'h') et un chiffre pour la ligne (de 1 à 8). Ainsi la case 'a1' correspond à la case 0 de l'encodage proposé à la question 1, la case 'b1' correspond à la case 1, etc.

Pour réaliser cet encodage, nous proposons la classe **Board** (échiquier) qui propose les méthodes suivantes :

- `putQueen(self, square)` qui place une dame sur la case `square`, avec `square` une chaîne de caractères de la forme 'a1'.
- `hasQueen(self, square)` qui retourne `True` si la case `square` est occupée par une reine, `False` sinon.
- `removeQueen(self, square)` qui retire la dame de la case `square`.

Listing 2 – code d'utilisation de la classe Board

```
1 board = Board() # l'\ 'echiquier est vide
2 board.hasQueen('a1') # False
3 board.putQueen('a1')
4 board.hasQueen('a1') # True
5 board.removeQueen('a1')
6 board.hasQueen('a1') # False
```

Listing 3 – code de manipulation des chaînes de caractères

```
1 b8 = 'b8'
2 row = int(b8[1]) # 8
3 column = int(ord(b8[0]) - 96) # 2
4 a1 = chr(97) + str(1) # 'a1'
```

Question 2. Programmer la classe `Board` et faire en sorte que le code du listing 2 puisse être exécuté. Le code du listing 3 vous aide à effectuer les manipulations de chaînes de caractères.

Question 3. Programmer la fonction `queen_legal_moves(square)` qui retourne l'ensemble des cases sur lesquelles une reine peut bouger si elle est placée sur la case `square` et en considérant un échiquier vide. La fonction doit retourner toutes les cases de sa ligne, de sa colonne, ainsi que les deux diagonales qui passent par sa case. Enfin, la case d'origine ne doit pas appartenir à l'ensemble retourné.

Question 4. Modifier la fonction `check_solution8queens(sol)` pour qu'elle prenne en entrée un échiquier (un objet instance de la classe `Board`). La fonction devient alors : `check_solution8queens(board)`. Vous vous appuyerez sur la fonction `queen_legal_moves(square)` pour savoir si l'échiquier passé contient une solution correcte au problème des huit dames.

La dame et les autres pièces

On considère la classe **Queen** qui représente une dame. Dans un premier temps cette classe ne contient qu'une seule méthode : `legal_moves(self, square)`. Cette méthode retourne la liste des cases sur lesquelles une dame peut se déplacer si elle est placée sur la case **square** et si on considère un échiquier vide. Le code de cette méthode est donc le même que la fonction `queen_legal_moves(square)`.

Question 5. La conception de la classe **Queen** fait le choix que la position des pièces est connue par classe **Board** (et non par **Queen**). Discuter de l'intérêt de ce choix de conception (avantages et inconvénients).

Déplacement des pièces

Pour rendre le jeu plus intéressant, on va ajouter d'autres pièces : le roi, la tour et le fou. Quelques explications sur le déplacement de ces pièces aux échecs sont données ci-dessous. Notez que des explications sur les autres pièces du jeu d'échecs (le cavalier ou le pion), ne sont pas données car nous ne les considérerons pas dans la suite du sujet. Nous ne traiterons également pas des règles comme le pat ou la prise en passant.

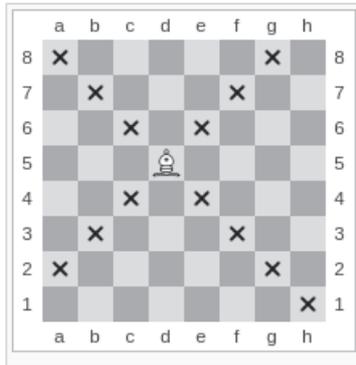
Question 6. Programmer en Python les classes **Queen** (Dame), **King** (Roi), **Rook** (Tour) et **Bishop** (Fou). On ne considère pas le cavalier et le pion. On se contentera d'implémenter les mouvements possibles des pièces, en s'appuyant sur les explications données préalablement. Ces classes proposent la même méthode `legal_moves(self, square)` qui donne la liste des cases sur lesquelles une pièce peut bouger à partir d'une case de départ et en considérant un échiquier vide. Il est attendu à ce que vous utilisiez l'héritage pour définir ces classes et faire en sorte qu'il y ait peu de redondance de code.

Question 7. Modifier la classe **Board** et remplacez les méthodes `putQueen`, `hasQueen` et `removeQueen` par des méthodes `put`, `has` et `remove` pour toutes les pièces considérées (Queen,

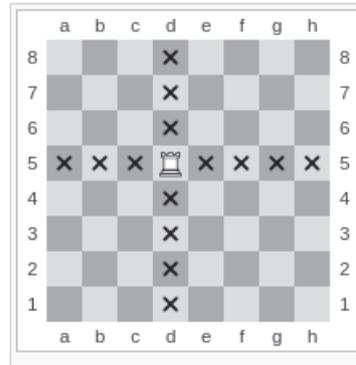
Le fou, la tour et la dame sont des pièces à longue portée (ou pièces de lignes) : elles peuvent se déplacer le long de lignes. Les fous se déplacent toujours sur les cases d'une même couleur, en diagonale. La tour est capable de se mouvoir en ligne droite, verticalement, horizontalement, sur un nombre quelconque de cases inoccupées (voir figure 3). La dame est capable de se mouvoir en ligne droite, verticalement, horizontalement, et diagonalement, sur un nombre quelconque de cases inoccupées (voir figure 3). Le roi se déplace d'une seule case à la fois dans toutes les directions. Les pièces (excepté le pion) capturent une pièce adverse qui se trouve sur leur trajectoire, sans pouvoir aller au-delà. La pièce qui capture prend la place de la pièce capturée, cette dernière étant définitivement retirée de l'échiquier.

FIGURE 2 – Explications sur les déplacements des pièces

Déplacements du fou de cases blanches



Déplacements de la tour



Déplacements de la dame

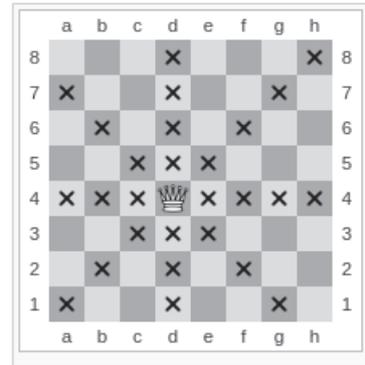


FIGURE 3 – Déplacements Fou, Tour et Dame

King, Rook et Bishop). Vous ajouterez également la méthode *get(self, square)* qui retourne la pièce positionnée à la case square, ou None si aucune pièce n'y est positionnée. Il est attendu que votre code soit le plus générique possible et qu'il y ait le moins de redondance et le plus de réutilisation. Charge à vous de proposer une conception Orientée Objet adaptée et de bien définir les arguments de ces méthodes. Une fois votre classe programmée, le code du listing 4 devra pouvoir être exécuté.

Listing 4 – code d'utilisation de la classe Board

```

1 board1 = Board()
2 board1.put('a1', Queen())
3 board1.put('b3', Rook())
4 board1.has('a2') # False
5 queen = board1.get('a1') # Queen

```

Les problèmes similaires au problèmes des huit dames

On considère la classe **Problem8Queens** qui représente le problème des 8 dames. Cette classe a les méthode suivantes :

- *check_solution8queens(self, board)* qui permet de vérifier que la solution board est valide.
- *propose_solution(self, board)* qui permet à un joueur de proposer une solution au problème. Cette méthode fait appel à la méthode *check_solution8queens(self, board)* pour vérifier la solution proposée. De plus, elle mémorise dans un tableau toutes les solutions proposées par les joueurs, que celles-ci soient valides ou non.
- *statistiques(self)* qui écrit un rapport sur la sortie standard. Ce rapport précise combien de solutions ont été proposées et combien sont valides.

Question 8. Programmer la classe **Problem8Queens**. Vous prendrez soin d'exploiter les méthodes de la classe **Board** et les méthodes des classes représentant les pièces. Une fois votre classe programmée, le code du listing 5 devra pouvoir être exécuté.

Listing 5 – code d'utilisation de la classe Problem8Queens

```
1 board1 = Board()
2 board1.put('a1', Queen())
3 board1.put('b3', Queen())
4 problem = Problem8Queens()
5 problem.propose_solution(board1)
6 problem.statistiques() # affiche sur la sortie standard "1 solution , 0 valide"
```

On souhaite maintenant proposer différents types de problèmes : le problème des huit dames, le problème des huit tours, le problème des 4 fous et des 4 tours, etc.

La classe **ProblemFactory** propose une méthode *create_problem* permettant de créer des problèmes de différents types (8 dames, 8 fous, etc.). A noter qu'il est également possible créer des problèmes avec une combinaison de différents types de pièces (exemple : 4 fous et 4 tours).

Question 9. Décrire la classe **ProblemFactory** (sans la programmer en Python) en prenant soin de bien définir les paramètres de la méthode *create_problem* permettant de créer des problèmes de différents types. Vous expliquerez comment cette méthode permet de créer a minima le problème des 8 dames, le problème des 8 tours et le problème des 4 fous et des 4 tours. Vous décrierez également le lien avec la classe **Problem8Queens** et les modifications qu'il faut y apporter. Enfin vous décrierez les qualités de votre conception en termes de réutilisation de code.

Résolution de problème

On considère qu'un algorithme résout un problème du type problème des 8 dames s'il retourne toutes les solutions à ce problème. De manière générique, on appelle *solve* les fonctions qui implémentent un tel algorithme.

Question 10. Programmer une fonction Python qui : (1) génère une instance de la classe **Board** contenant 8 dames placées aléatoirement, (2) vérifie que cette instance est une solution au problème des 8 dames en appelant la méthode *check_solution8queens(self, board)* d'un objet instance de la classe **Problem8Queens**, et (3) s'arrête lorsqu'une solution valide a été trouvée en imprimant sur la sortie écran le résultat trouvé. Vous discuterez de la qualité de la solution consistant à exploiter cette fonction pour proposer une fonction *solve_random* qui résout le problème des 8 dames.

Question 11. Programmer une fonction Python *solve_walker* qui parcourt l'espace des **board** contenant 8 dames et qui s'arrête dès qu'ont été trouvées x **board** solutions au problème des 8 dames ($0 < x \leq 92$) en imprimant sur la sortie écran chaque solution trouvée. Sachant qu'il existe 92 solutions au problème des 8 dames, on peut dire que cette fonction propose un algorithme qui résout le problème des 8 dames. Est-ce que la connaissance du nombre de solutions (92) est nécessaire ?

Question 12. Programmer une fonction *solve_matrix* qui résout le problème des 8 dames en exploitant les matrices de *permutation*. En effet, comme les dames attaquent le long des lignes, des colonnes et des diagonales, cela équivaut à une matrice de permutation d'ordre 8 dans laquelle la somme de chaque diagonale est au plus égale à 1. On rappelle qu'une matrice de *permutation*

est une matrice carrée qui vérifie les propriétés suivantes : les coefficients sont 0 ou 1 ; il y a un et un seul 1 par ligne ; il y a un et un seul 1 par colonne.

Question 13. À ce stade, nous avons au moins trois implémentations candidates pour calculer l'ensemble des solutions du problème des huit dames : *solve_walker*, *solve_random*, et *solve_matrix*. Ces implémentations ont des qualités différentes mais en toute rigueur, l'ensemble des solutions retourné par les trois implémentations devrait donner le même résultat. Montrer en Python comment cette propriété peut être automatiquement vérifiée avec un programme.

Restructuration et difficulté des problèmes

On souhaite que n'importe quel développeur puisse proposer sa fonction *solve*. L'idée étant de pouvoir disposer de plusieurs fonction *solve* proposées par des développeurs différents.

Question 14. Décrire la restructuration de votre application permettant à un développeur d'intégrer sa fonction *solve* dans votre conception. Vous devez prendre soin de bien décrire toutes les modifications à effectuer sur le code des classes existantes. Vous expliquerez aussi ce que doit faire le développeur pour intégrer sa fonction. Enfin, vous montrerez comment choisir une fonction *solve* qui a été proposée, et comment l'exécuter pour trouver toutes les solutions à un problème.

Question 15. On souhaite générer des problèmes d'échecs difficiles. La notion de difficulté est définie ainsi : un problème d'échecs est difficile si (1) les trois implémentations précédentes en charge de trouver toutes les solutions mettent plus de 10 minutes ; et si (2) le nombre de solutions est inférieur à 10. Le problème des huit dames, par exemple, n'est pas difficile (car le nombre de solutions, 92, est supérieur à 10). Proposer un algorithme pour générer et trouver de tels problèmes difficiles. Expliquer comment cet algorithme peut s'intégrer dans la conception des questions précédentes (classe **ProblemFactory**, fonction *solve*, etc.)

Pour mesurer le temps d'exécution d'un programme en Python, vous vous appuyez sur le module *time*. Le listing 6 illustre le fonctionnement de *time*.

Listing 6 – code d'utilisation du module *time*

```
1 import time
2 start = time.time() # avant l'appel d'un programme
3 # execution du programme
4 # ...
5 end = time.time() # fin du programme
6 duration = end - start # la duree de l'execution en seconde (de type float)
```

Partie d'échecs

On souhaite maintenant étendre notre conception et implémentation précédente pour permettre de jouer une vraie partie d'échecs avec notre application.

Question 16. Modifier votre code pour prendre en compte la couleur des pièces. Vous ne recopiez pas tout votre code et vous décrivez uniquement le code modifié.

Question 17. Dans la classe **Board**, programmer la méthode *move(self, from, to)* qui déplace la pièce de la case **from** vers la case **to**. Cette méthode vérifiera que le déplacement est légal (via la méthode *legal_move* des pièces) et qu'il n'y a pas de pièces entre la case **from** et la case **to** (pour rappel on ne considère pas le cavalier et le pion). La méthode vérifiera aussi que la case d'arrivée ne doit pas contenir une pièce de la même couleur que la pièce déplacée. Enfin, il ne devra pas être possible de déplacer deux fois de suite des pièces de la même couleur. Vous veillerez à ce que le code de la méthode *move* réutilise un maximum de code existant de la classe **Board** et des classes des pièces.

Nous considérons maintenant la classe **Game** qui représente une partie d'échecs. A l'initialisation, une partie contient un **Board** avec les pièces en position initiale. Il sera alors possible de jouer le premier coup des blancs. Puis, alternativement, il sera possible de jouer les coups des noirs et des blancs jusqu'au mat.

Question 18. Proposer une conception générale de la classe **Game** permettant aux joueurs de déplacer les pièces jusqu'au mat. Certaines particularités du jeu d'échecs (prise en passant, pat, fin de partie, etc.) non décrites dans l'extrait (cf figure 2, page 4) ne doivent pas être considérées ici. On ne détectera donc pas le mat. Donner les signatures des méthodes que vous jugez nécessaires (pas le code) pour que les joueurs puissent jouer une partie. Préciser par du texte les propriétés de cette classe (sans donner le code de la méthode `__init__`).

Question 19. Illustrer les 4 premiers coups échangés entre deux joueurs (2 coups chacun) à l'aide d'un diagramme décrivant la séquence des échanges de messages entre les objets de l'application. L'objectif est d'illustrer graphiquement combien d'objets participent à ces échanges, comment ces objets sont créés et reliés entre eux, et dans quel ordre se font les échanges. Votre diagramme devra alors faire apparaître graphiquement : tous les objets impliqués, tous les messages échangés, et l'ordre de ces messages

Le roque

On veut maintenant supporter et implémenter le *roque*. Des explications, issues et adaptées de Wikipedia, sont données dans l'encadré ci-dessous.

Question 20. Implémenter une fonction en Python qui implémente le mouvement du roque en vérifiant préalablement si le roque est possible. On fera l'hypothèse que le joueur indiquera la case de départ du roi et la case d'arrivée (le déplacement de la tour étant déduit). Expliquer comment vous avez organisé votre code notamment pour vérifier la possibilité du roque.

Le roque est un déplacement spécial du roi et d'une des tours au jeu d'échecs. Le roque permet, en un seul coup, de mettre le roi à l'abri tout en centralisant une tour, ce qui permet par la même occasion de mobiliser rapidement cette dernière. Il s'agit du seul coup légal permettant de déplacer deux pièces, sans respecter le déplacement classique du roi et de la tour de surcroît. On distingue le petit roque et le grand roque. Dans le cas du petit roque, le roi blanc se déplace de deux cases en direction de la tour, et la tour se place immédiatement à gauche du roi. La figure 5 donne une illustration.

Le grand roque s'effectue de la même manière que le petit roque : le roi avance de deux cases en direction de la tour (en a1 ou en a8), et cette dernière se place de l'autre côté du roi. La figure 6 illustre le principe.

Une particularité importante du roque est que plusieurs conditions doivent être respectées :

- toutes les cases qui séparent le roi de la tour doivent être vides. Par conséquent, il n'est pas permis de prendre une pièce adverse en roquant ;
- ni le roi, ni la tour ne doivent avoir quitté leur position initiale. Par conséquent, le roi et la tour doivent être dans la première rangée du joueur ;
- chaque camp ne peut roquer qu'une seule fois par partie ; en revanche, le roi peut déjà avoir subi des échecs, s'il s'est soustrait à ceux-ci sans se déplacer.
- aucune des cases (de départ, de passage ou d'arrivée) par lesquelles transite le roi lors du roque ne doit être contrôlée par une pièce adverse. Par conséquent, le roque n'est pas possible si le roi est en échec.

FIGURE 4 – Explications du roque

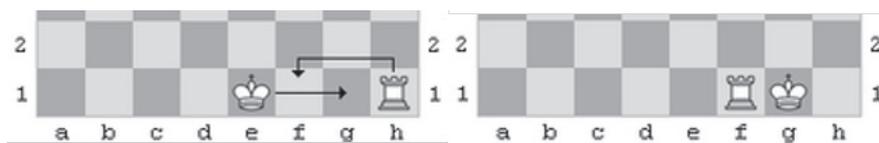


FIGURE 5 – Petit roque (avec les pièces blanches)

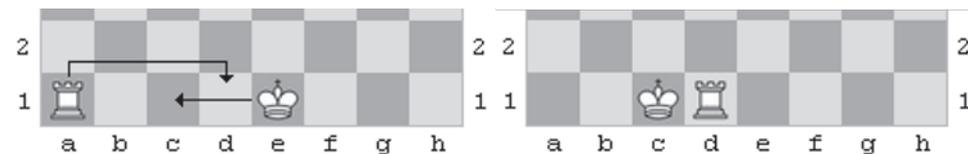


FIGURE 6 – Grand roque (avec les pièces blanches)

Problème des mats en N coups

L'application **ChessMate** ne porte pas bien son nom puisque, en l'état, les problèmes d'échecs et mat ne sont pas supportés. On se propose d'y remédier.

La classe **MatInNMoves** représente un problème de type mat en N coups. Un tel problème est défini par une position initiale des pièces sur l'échiquier, une couleur (blanc ou noir) pour laquelle le problème est posé, et un nombre de coups maximum pour que cette couleur mette l'autre couleur en position de mat (c'est-à-dire dans l'impossibilité d'empêcher la prise de son roi).

Question 21. Proposer les méthodes de cette classe (pas le code) permettant à un utilisateur d'essayer de résoudre un tel problème. A noter que dans le cadre spécifique des mats en N coups, c'est la même entité qui manipule à la fois le joueur avec les pièces blanches et avec les pièces noires. Vous ferez en sorte que la classe **MatInNMoves** réutilise un maximum la classe **Game**, et vous expliquerez la façon dont la vérification du problème se fait.

Réflexion sur la conception

Question 22. Il existe différentes façons de représenter l'échiquier dans la classe **Board**. Cela peut se faire notamment à l'aide d'un tableau comme c'est le cas dans le code donné du listing 1. Proposer une autre façon de représenter l'échiquier et discuter de ses avantages et de ses inconvénients (en espace et en temps).

Question 23. On souhaite généraliser le problème des huit dames (et uniquement ce problème) et faire en sorte que l'échiquier soit de $N \times N$ cases (N étant un nombre positif donné lors de la création du **Board**). Discuter de l'impact de cette généralisation sur la conception réalisée jusqu'à présent. Vous préciserez quels sont les impacts sur les classes et sur leurs méthodes.

Partie II. Web

On souhaite réaliser une interface graphique HTML, CSS et JavaScript permettant de jouer à des problèmes échiquéens.

L'application Web est composée de deux pages. La première page permet au joueur de sélectionner le problème. L'interface voulue est présentée par la figure 7.

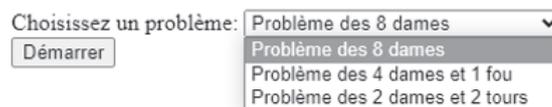


FIGURE 7 – Première page de l'application Web

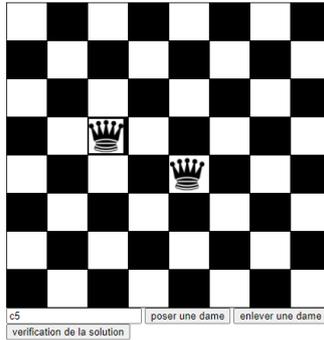


FIGURE 8 – Deuxième page de l'application Web pour le problème des 8 dames

La deuxième page permet de jouer (résoudre le problème proposé). La figure 8 présente l'écran pour tenter de résoudre le problème des 8 dames. On y voit l'échiquier avec ses 64 cases. Une première rangée de boutons permet de saisir les coordonnées d'une case (sur l'exemple : 'c5' et 'e4'), de déposer ou d'enlever une dame. Enfin, un dernier bouton permet de demander la vérification de la solution qui sera réalisée au niveau du serveur web. Le code HTML de la deuxième page web vous est partiellement donné pour le problème des 8 dames (voir listing 7).

Listing 7 – page HTML du problème des 8 dames

```

1 <html>
2 <head>
3   <title>Chess Board</title>
4   <style>
5     table {
6       border-collapse: collapse;
7       border: 1px solid black;
8     }
9     td {
10      width: 50px;
11      height: 50px;
12      border: 1px solid black;
13    }
14    img {
15      width: 45px;
16      height: 45px;
17    }
18    .black {
19      background-color: black;
20    }
21    .white {
22      background-color: white;
23    }
24  </style>
25 </head>
26 <body>
27   <table id="board">
28   </table>
29   <div>
30     <input type="text" id="square" value="e4" />
31     <input type="button" id="putQueen" value="poser une dame" />
32     <input type="button" id="removeQueen" value="enlever une dame" />
33   </div>
34   <div>
35     <input type="button" id="solve" value="verification de la solution" />
36   </div>
37 </body>
38 </html>

```

Question 24. Programmer la première page. Celle-ci contient un formulaire permettant au joueur de choisir le type de problème qu'il veut résoudre. Ce formulaire soumet une requête HTTP GET vers `/choose_problem`. Il contient un seul champ nommé `problem`. Vous proposerez une programmation en HTML pur de l'ensemble de cette page (i.e., sans JavaScript ou CSS).

Question 25. Dans la deuxième page, programmer un script JavaScript permettant d'afficher un échiquier à l'ouverture de la page. L'échiquier est un tableau composé de 8 lignes (balises TR) et de 64 cellules en tout (balises TD). Vous ferez en sorte que chaque cellule utilise le style proposé pour la couleur des cases. De plus, chaque cellule doit avoir un identifiant unique. Votre code sera inséré dans la page HTML de la deuxième page web à l'aide de la balise proposée dans le listing 8. Les premières lignes du code JavaScript sont déjà écrites, il vous reste à écrire le code pour afficher l'échiquier.

Listing 8 – script pour afficher l'échiquier

```
1 <script>
2   let board = document.getElementById("board");
3   for (let i = 8; i > 0; i--) {
4     //TODO
5   }
6 </script>
```

Question 26. Dans la deuxième page, programmer un script JavaScript permettant de déposer une dame sur la case choisie. Il faudra alors ajouter une image (balise IMG) à la cellule choisie avec l'attribut `src` de la balise IMG étant égal à `queen.jpg`. La première ligne du code JavaScript est déjà écrite (voir listing 9), il vous reste à compléter ce code.

Listing 9 – script pour déposer une dame

```
1 <script>
2   let putQueenButton = document.getElementById("putQueen");
3   //TODO
4 </script>
```

Question 27. Programmer un script JavaScript permettant de retirer une dame de la case choisie. Vous ferez en sorte de minimiser la redondance avec le code de la question précédente. La première ligne du code JavaScript est déjà écrite dans le listing 10, il vous reste à compléter ce code.

Listing 10 – script pour supprimer une dame

```
1 <script>
2   let removeQueenButton = document.getElementById("removeQueen");
3   //TODO
4 </script>
```

La vérification d'une solution s'effectue en envoyant une requête HTTP vers un serveur web (note : vous n'avez pas à proposer d'implémentation du serveur web). La requête de vérification est une requête GET vers `/solve8queens` et qui a un paramètre dont le nom est `solution` et dont la valeur est une chaîne de caractères décrivant les positions des dames en séparant chaque position par le caractère `'-'`. Par exemple, si l'adresse du serveur est `http://www.serveur-chess.fr`, envoyer la requête GET `http://www.serveur-chess.fr/solve8queens?solution=e1-b2-c3-a4-h5-d7-f8-h6` permet de demander la vérification de la solution : (`'e1'`, `'b2'`, `'c3'`, `'a4'`, `'h5'`, `'d7'`, `'f8'`, `'h6'`).

Si la réponse à la requête est positive (code de retour : 200), le script doit ouvrir une alerte précisant que la solution est valide. Sinon, une alerte précisera que la solution n'est pas valide. On rappelle que la fonction JavaScript "alert(message)" permet de lancer une alerte à l'utilisateur.

Question 28. Programmer ce script JavaScript permettant de soumettre une solution au problème des 8 dames. La première ligne du code JavaScript est déjà écrite dans le listing 11, il vous reste à compléter ce code.

Listing 11 – script pour envoyer une solution

```
1 <script>
2   let solveButton = document.getElementById("solve");
3   //TODO
4 </script>
```

Question 29. Discuter des avantages et inconvénients de réaliser certaines validations de la solution proposée avant d'envoyer une demande de validation au serveur (on considère que le calcul de la vérification est trop long pour se faire intégralement sur le navigateur).

Partie III. Base de données

Pour améliorer l'application **ChessMate** il a été convenu de mettre à disposition une base de données qui contient différents problèmes d'échecs.

Chaque problème a un titre et une description. De plus, certains problèmes sont accompagnés d'une solution déjà trouvée et établie que les utilisateurs pourront consulter. D'autres problèmes, ouverts ('open' en anglais), n'ont pas de solution associée. Enfin, pour certains problèmes, un programme de vérification ('checking_procedure' en anglais) est éventuellement associé et permet de vérifier si la solution proposée au problème est correcte.

Pour commencer la construction de cette base de données, il a été convenu d'utiliser le format **YAML**, qui est un format de représentation de données clé/valeur et arborescent. Un exemple de données en cours de construction est donné ci-dessous :

Listing 12 – exemple de la base

```
1 chessmate:
2   - description: "Le but du probl\`eme des huit dames est de placer..."
3     version: "1.0.0"
4     title: "8 queens problem"
5     type: "all_solutions"
6     contact:
7       name: "Elisabeth Harmon"
8       email: "XXX@agreg-info.org"
9     problem: "QQQQQQQQ"
10    checking_procedure: "https://chessmate.org/8queens"
11    solutions: [[8, 5, 20, 25, 42, 59, 54, 39], [0, 14, 20, 31, 33, 43, 53, 58]]
12
13
14   - description: "Votre objectif est de placer 4 dames et 1 fou..."
15     title: "4 Queens + Bishop problem"
16     type: "one_solution"
17     open: true
18     contact:
19       name: "Judit Polkarov"
```

```

20     email: "YYY@agreg-info.org"
21     problem: "QQQB"
22     checking_procedure: "https://chessmate.org/4QueensBishop"
23
24     - description: "Votre objectif est de placer 2 dames et 2 tours sur un plateau 6x
25       par 6..."
26       version: "0.1"
27       title: "2 Queens + 2 Rooks problem"
28       type: "one_solution"
29       open: true
30       contact:
31         name: "Judit Polkarov"
32         email: "YYY@agreg-info.org"
33         problem: "QRRR"
34         chessboard: 36
35         checking_procedure: "https://chessmate.org/2QueensRooks36"
36
37     - description: "Mate in 2"
38       version: "0.1"
39       title: "Mate in 2"
40       type: "check_mate"
41       moves: 3
42       turn: "white"
43       contact:
44         name: "Garry Agreg"
45         email: "XXYY@agreg-info.org"
46         problem: "r1b2r1k/ppp3p1/2n2q1N/7Q/2BPp3/2P5/P4PPP/R1B3K1 w -- 1 16"
47         solutions: "Nf7+ Kg8 Qh8#"

```

L'exemple YAML fourni spécifie 4 problèmes. Les trois premiers problèmes sont des "puzzles", avec respectivement le problème des huit dames, le problème des 4 dames et du fou, et le problème des 2 dames et des deux tours sur un échiquier 6*6.

Le quatrième est un problème d'échecs et mat : il faut trouver une série de 3 coups ('moves' en anglais) en commençant par un coup des pièces blanches (cf `turn` et `white`).

La description des puzzles ou les problèmes de mat partagent un certain nombre de caractéristiques, mais exhibent également des informations spécifiques. On peut catégoriser ainsi les informations caractérisant les problèmes :

- *ProblemDescription* : description du problème, version du problème, titre du problème, type de problème (échecs et mat, puzzle), si le problème est ouvert, etc. Un problème a nécessairement une `description` qui est une chaîne de caractères, possiblement une `version` qui est une chaîne de caractères, un titre via `title`. La notion de `turn` n'a aucun sens pour les problèmes de type puzzle et donc n'est pas spécifiée pour les 3 problèmes de l'exemple. Quand `chessboard` n'est pas spécifié, on considère par défaut que l'échiquier est 8*8 et contient 64 cases. Pour les "puzzles", on distingue deux types de problèmes : le cas où il faut trouver une seule solution (`one_solution`), et le cas où il faut trouver toutes les solutions (`all_solutions`). Toujours pour les puzzles, un problème est considéré comme ouvert (`open`) si une ou des solutions ne sont pas connues et donc renseignées. Pour les "échecs et mat", la valeur de `type` est nécessairement égale à `check_mate` ;
- *Author* : des informations sur l'auteur du problème avec son nom (`name`) et son email (`email`) ;

- *Solution* : une ou des solutions si elles existent, spécifié dans 'solutions' avec une chaîne de caractères et une syntaxe bien particulière selon que ce soit un "puzzle" ou un "échecs et mat".

On souhaite maintenant remplacer l'utilisation de **YAML** par un système de gestion de bases de données (SGBD) relationnel.

Autrement dit, nous souhaitons migrer le document YAML dans un SGBD relationnel.

Question 30. Proposer un modèle relationnel avec au moins trois tables : "ProblemDescription", "Author", "Solution". Pour chaque table, donner les noms de colonne ainsi que leurs types.

En plus de la description des tables, vous préciserez vos choix en répondant aux quatre questions ci-dessous :

Question 31. Comment avez-vous représenté l'attribut "type" ?

Question 32. Comment avez-vous représenté l'optionnalité de la 'version' d'un problème ?

Question 33. Comment avez-vous organisé les informations spécifiques à un "échecs et mat" et à un "puzzle" ?

Question 34. Dans ce schéma relationnel, quelles sont les clés primaires de chaque table ? Qu'existe-t-il comme clés étrangères ?

Pour illustrer le fonctionnement de votre base de données :

Question 35. On veut ajouter le problème échec et mat "Mate in 2" de l'exemple YAML ci-dessus. Donner le(s) ordre(s) d'ajout nécessaire(s) dans les tables.

En vous appuyant sur vos tables, écrire en SQL :

Question 36. une requête retournant tous les problèmes "ouverts" (correspondant à open dans le YAML).

Question 37. une requête retournant des problèmes, de type puzzle, avec uniquement 8 * 8 cases.

Question 38. une requête retournant le nombre de problèmes qui ont des solutions ou qui sont des problèmes de type "échecs et mat".

On souhaite identifier tous les problèmes de type "puzzle" qui mettent en jeu au moins un fou ('Bishop' en anglais).

Question 39. Est-ce possible de le faire uniquement avec une requête SQL ? Justifier.

Pour y parvenir on se propose de le faire en deux temps :

- via une requête SQL qui retourne tous les problèmes de type "puzzle" (excluant de fait les échecs et mat) et en particulier l'information `problem` qui décrit le problème avec une syntaxe spécifique. Par exemple, pour le problème numéro 2 de l'exemple YAML, `problem` : "QQQQB" spécifie qu'il y a 4 dames à positionner (Q pour Queen) et 1 fou à positionner (B pour Bishop) ;
- via un programme Python qui analysera le contenu de 'problem' pour ne garder que les problèmes avec au moins un fou.

Question 40. En vous appuyant sur l'API Python `sqlite3` dont le principe de fonctionnement est décrit ci-dessous, programmer une telle solution en Python. Deux modifications de Listing 13 sont à réaliser : (1) adapter la ligne 4 pour écrire la requête SQL ; (2) adapter la ligne 6, possiblement en écrivant plusieurs lignes de code avant ou après, pour analyser le contenu et ne garder que les problèmes qui utilisent au moins un fou.

Les lignes 1, 2 et 3 permettent de se connecter à la base de donnée et n'ont pas d'intérêt majeur. La ligne 4 permet d'itérer sur chaque ligne (`row`) de la requête. La ligne 5 permet d'obtenir un dictionnaire avec clé/valeur de la ligne, la clé étant le nom de la colonne (dans cet exemple fictif : `date`, `type`, `price`, `city`). Ainsi, ligne 6, il est possible de traiter ce dictionnaire et d'analyser le contenu de la ligne (dans cet exemple fictif, c'est une simple impression écran avec `print`).

Listing 13 – API Python `sqlite3`

```
1 import sqlite3
2 con = sqlite3.connect('example.db') # nom de la base
3 cur = con.cursor()
4 for row in cur.execute('SELECT * FROM stocks ORDER BY price'): # requete SQL
5     r = dict(row) # dictionnaire cle/valeur
6     print(r)
```

Le résultat obtenu est :

Listing 14 – Résultat de la requête SQL

```
1 {'date': '2006-01-05', 'type': 'BUY', 'price': 35.14, 'city': 'Rennes'}
2 {'date': '2006-01-08', 'type': 'SELL', 'price': 5.42, 'city': 'Marseille'}
3 {'date': '2016-11-05', 'type': 'BUY', 'price': 18.94, 'city': 'Bordeaux'}
```

Question 41. Discuter des avantages et inconvénients de votre solution, basée sur une requête SQL puis un traitement programmatique, en termes de (1) maintenance et de (2) performance.

Partie IV. Architecture

Question 42. L'analyse des requêtes et de l'audience de **ChessMate** montre que le problème des 8 dames est très populaire et que de nombreux joueurs essayent de résoudre ce problème et proposent régulièrement des solutions. L'unique machine serveur qui vérifie ces solutions est

donc trop petite et ne supporte plus la charge. Proposer une infrastructure réseau permettant de déployer plusieurs serveurs. Cette infrastructure doit répartir les requêtes des joueurs (requêtes HTTP) sur les différents serveurs. Vous expliquerez de façon algorithmique et pratique comment ces requêtes sont réparties (comment orienter les requêtes vers les serveurs) et comment la charge est distribuée entre les serveurs (de manière équitable ? avec des effets de seuil ?).

Question 43. La solution proposée par la question précédente est intéressante mais elle cible essentiellement les pics de charge. En effet les serveurs déployés sont très souvent inexploités, la nuit par exemple lorsqu'aucun joueur ne propose de solutions. Proposer une infrastructure avec des serveurs virtuels. Vous expliquerez de façon algorithmique et pratique comment les serveurs virtuels se partagent la charge et comment ils sont démarrés et éteints. Vous préciserez combien de serveurs virtuels sont déployés (1) au déploiement de l'application et (2) lors d'un pic de charge.

Question 44. Dans l'application **ChessMate**, il est possible vérifier si une solution est bonne. Il est également possible de vérifier si *toutes* les solutions proposées sont bonnes. Plutôt que d'effectuer la vérification en séquentiel, sur une seule machine, solution après solution, on propose de distribuer la vérification des solutions sur plusieurs machines. Si une solution parmi toutes celles proposées n'est pas valide, alors la vérification s'arrête et retourne la solution erronée. Proposer une architecture permettant de distribuer une telle vérification de plusieurs solutions.

Question 45. L'approche distribuée de la question précédente peut potentiellement fournir des bénéfices en termes d'accélération du calcul et ainsi réaliser une vérification plus rapide. Cependant, cette approche a aussi des défauts car elle induit également des coûts computationnels et réseaux. Identifier tous les facteurs qui peuvent soit influencer sur les bénéfices soit sur les coûts définis ci-dessus soit sur les deux à la fois. Discuter des compromis à trouver.

Fondements de l'informatique

Les questions de programmation doivent être traitées en langage OCaml. On pourra utiliser toutes les fonctions des modules `Array` et `List`, ainsi que les fonctions de la bibliothèque standard (celles qui s'écrivent sans nom de module, comme `max`, `incr` ainsi que les opérateurs comme `@`). L'utilisation d'autres modules est interdite.

Dans l'écriture d'une fonction, on pourra faire appel à des fonctions définies dans les questions précédentes, même si ces dernières n'ont pas été traitées. On pourra également définir des fonctions auxiliaires, mais il faudra alors préciser leurs rôles ainsi que le type et les significations de leurs arguments. Si les paramètres d'une fonction à coder sont supposés vérifier certaines hypothèses, il ne sera pas utile dans l'écriture de cette fonction de tester si les hypothèses sont bien satisfaites. Lorsque les choix d'implémentation ne découlent pas directement des spécifications de l'énoncé, il est conseillé de les expliquer.

On identifiera une même grandeur écrite dans deux polices de caractères différentes, en italique du point de vue mathématique (par exemple n) et en Computer Modern à chasse fixe du point de vue informatique (par exemple `n`).

Sans précision supplémentaire, lorsqu'une question demande la complexité d'une fonction, il s'agira de la complexité temporelle dans le pire des cas. La complexité sera exprimée sous la forme $\mathcal{O}(f(n))$ où n est la taille de l'argument de l'algorithme, et f une expression la plus simple possible. Les calculs de complexité seront justifiés succinctement sauf mention spécifique dans la question.

Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction.

1 Compilation d'expressions vers une machine à pile

1.1 Une machine à pile

On s'intéresse à une machine à pile de type «calculatrice en notation polonaise inverse». Cette machine dispose de 4 instructions qui opèrent sur une pile contenant des valeurs entières :

<code>CONST(n)</code>	empile l'entier n
<code>VAR(X)</code>	empile la valeur de la variable X
<code>ADD</code>	dépile deux entiers n_1 et n_2 , empile leur somme $n_1 + n_2$
<code>MUL</code>	dépile deux entiers n_1 et n_2 , empile leur produit $n_1 \times n_2$

Un *code* pour la machine à pile est une liste d'instructions. L'exécution d'un code consiste à exécuter chaque instruction de la liste en séquence. Un *environnement*, donné en paramètre à la machine, associe une valeur à chaque variable. La machine s'arrête lorsque la dernière instruction a été exécutée. Le résultat de l'exécution est l'entier au sommet de la pile.

Exemple : l'exécution du code `CONST(1); VAR(X); ADD` dans un environnement où X vaut 2, et à partir d'une pile vide, effectue les étapes suivantes :

	Instruction exécutée	État de la pile après exécution de l'instruction
Étape 1	<code>CONST(1)</code>	1
Étape 2	<code>VAR(X)</code>	2 · 1 (2 au sommet, 1 en dessous)
Étape 3	<code>ADD</code>	3

Le résultat de l'exécution est l'entier 3.

Pour simplifier, on supposera tout au long du sujet que l'exécution d'un code machine ne produit jamais d'erreurs : ni débordements arithmétiques, ni tentative de dépiler un entier depuis une pile vide.

Définition : On définit la *consommation en espace* d'un code comme la hauteur maximale atteinte par la pile pendant l'exécution du code, en partant de la pile vide.

Exemple : le code de l'exemple ci-dessus a une consommation en espace égale à 2 puisque juste avant l'exécution de l'instruction `ADD`, la pile contient deux valeurs, et en tout autre point la pile est vide ou contient une seule valeur.

Question 1 Dans un environnement où X vaut 2 et Y vaut 5, déterminer le résultat de l'exécution du code `VAR(X); CONST(1); ADD; VAR(Y); MUL`. Préciser la consommation en espace de ce code.

Une instruction est représentée en OCaml par le type `instr` suivant :

```
type instr = CONST of int | VAR of string | ADD | MUL
```

Un code est alors représenté par une liste d'éléments de type `instr`. L'environnement est représenté par un objet de type `string -> int`, c'est-à-dire une fonction qui prend en argument une chaîne de caractère correspondant à une variable et renvoie un entier.

Question 2 Écrire une fonction `exec : instr list -> (string -> int) -> int` qui prend en argument un code et un environnement et renvoie le résultat de l'exécution du code à partir d'une pile vide.

Question 3 Écrire une fonction `conso_espace : instr list -> int` qui calcule la consommation en espace d'un code donné en argument.

1.2 Compilation simple des expressions arithmétiques

On s'intéresse aux expressions arithmétiques formées à partir de constantes entières $0, 1, 2, \dots, n, \dots$ et de variables symboliques X, Y, Z, \dots en utilisant les opérateurs $+$ (somme) et \times (produit).

Ces expressions sont représentées par leurs arbres de syntaxe abstraite : des arbres binaires avec, à chaque feuille, une constante ou une variable et, à chaque nœud, un opérateur $+$ ou \times . Par exemple, l'arbre A représenté en figure 1 correspond au produit de X plus 1 et de Y :

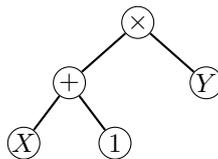


FIGURE 1 – L'expression A

La sémantique de ce langage d'expressions arithmétiques est donnée par les équations suivantes, qui définissent la valeur entière $\llbracket e \rrbracket_\rho$ d'une expression e dans un environnement ρ .

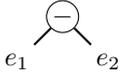
$$\begin{array}{ll} \llbracket \textcircled{n} \rrbracket_\rho = n & \llbracket \textcircled{X} \rrbracket_\rho = \rho(X) \\ \llbracket \textcircled{+} \begin{array}{l} e_1 \\ e_2 \end{array} \rrbracket_\rho = \llbracket e_1 \rrbracket_\rho + \llbracket e_2 \rrbracket_\rho & \llbracket \textcircled{\times} \begin{array}{l} e_1 \\ e_2 \end{array} \rrbracket_\rho = \llbracket e_1 \rrbracket_\rho \times \llbracket e_2 \rrbracket_\rho \end{array}$$

On rappelle l'algorithme de compilation classique \mathcal{C} qui, à une expression e représentée par son arbre, associe un code $\mathcal{C}(e)$ de la machine à pile :

$$\begin{array}{ll} \mathcal{C}(\textcircled{n}) = \text{CONST}(n) & \mathcal{C}(\textcircled{X}) = \text{VAR}(X) \\ \mathcal{C}(\textcircled{+} \begin{array}{l} e_1 \\ e_2 \end{array}) = \mathcal{C}(e_1); \mathcal{C}(e_2); \text{ADD} & \mathcal{C}(\textcircled{\times} \begin{array}{l} e_1 \\ e_2 \end{array}) = \mathcal{C}(e_1); \mathcal{C}(e_2); \text{MUL} \end{array}$$

Question 4 Quel code est produit par \mathcal{C} pour l'expression A de la figure 1 ?

Question 5 Soit e une expression, P la pile de la machine, et ρ un environnement. On fait exécuter par la machine le code compilé $\mathcal{C}(e)$ avec ρ comme environnement et P comme pile initiale. Démontrer par récurrence sur la structure de e que la pile finale, lorsque la machine s'arrête, est $\llbracket e \rrbracket_\rho \cdot P$, c'est à dire la pile contenant la valeur $\llbracket e \rrbracket_\rho$ au sommet et les mêmes valeurs que P en dessous.

Question 6 Dans cette question et la question 16 uniquement, on ajoute à notre langage d'expressions la soustraction $e_1 - e_2$, représentée par un arbre de la forme , ainsi que le calcul de l'opposé $-e$,

représenté par un arbre de la forme . En parallèle, on ajoute à notre machine à pile une instruction OPP qui change le signe de l'entier au sommet de la pile. On veut étendre l'algorithme de compilation \mathcal{C} pour traiter la soustraction et l'opposé. Donner les équations qui définissent $\mathcal{C}(\text{arbre } \ominus)$ et $\mathcal{C}(\text{arbre } \ominus)$, et les justifier informellement.

1.3 Consommation en espace du code compilé

Question 7 On considère les deux familles d'expressions G_n et D_n pour $n \geq 0$ définies par les récurrences suivantes :

$$G_0 = \textcircled{0} \qquad D_0 = \textcircled{0}$$

$$G_n = \begin{array}{c} \textcircled{+} \\ \swarrow \quad \searrow \\ G_{n-1} \quad \textcircled{n} \end{array} \quad \text{si } n > 0 \qquad D_n = \begin{array}{c} \textcircled{+} \\ \swarrow \quad \searrow \\ \textcircled{n} \quad D_{n-1} \end{array} \quad \text{si } n > 0$$

Montrer que pour tout $n \geq 1$, le code compilé $\mathcal{C}(G_n)$ a une consommation en espace égale à 2, et que pour tout $n \geq 1$ le code compilé $\mathcal{C}(D_n)$ a une consommation en espace égale à $n + 1$.

Question 8 Pour une expression e quelconque, on note $\mathcal{T}(e)$ la consommation en espace du code compilé $\mathcal{C}(e)$. Donner et justifier les relations de récurrence permettant de calculer $\mathcal{T}(\textcircled{n})$, $\mathcal{T}(\textcircled{X})$, $\mathcal{T}(\text{arbre } \oplus)$ et

$\mathcal{T}(\text{arbre } \otimes)$ en fonction de $\mathcal{T}(e_1)$ et $\mathcal{T}(e_2)$.

1.4 Compilation qui minimise la consommation en espace

L'informaticien russe Andreï Erchov propose un algorithme de compilation en 1957 : l'idée centrale est, dans le cas d'une somme ou d'un produit de deux expressions e_1 et e_2 , de traiter en premier la sous-expression e_i qui a le plus grand nombre de Strahler $\mathcal{S}(e_i)$, le nombre de Strahler d'une expression étant défini par la récurrence suivante :

$$\mathcal{S}(\textcircled{n}) = \mathcal{S}(\textcircled{X}) = 1$$

$$\mathcal{S}(\text{arbre } \oplus) = \mathcal{S}(\text{arbre } \otimes) = \begin{cases} 1 + \mathcal{S}(e_1) & \text{si } \mathcal{S}(e_1) = \mathcal{S}(e_2) \\ \max(\mathcal{S}(e_1), \mathcal{S}(e_2)) & \text{si } \mathcal{S}(e_1) \neq \mathcal{S}(e_2) \end{cases}$$

Ces nombres ont été introduits par le géographe américain Arthur N. Strahler en 1952 pour l'étude des réseaux hydrographiques des cours d'eau.

L'algorithme de compilation d'Erchov \mathcal{E} considéré est alors :

$$\mathcal{E}(\textcircled{n}) = \text{CONST}(n) \qquad \mathcal{E}(\textcircled{X}) = \text{VAR}(X)$$

$$\mathcal{E}(\text{arbre } \oplus) = \begin{cases} \mathcal{E}(e_1); \mathcal{E}(e_2); \text{ADD} & \text{si } \mathcal{S}(e_1) \geq \mathcal{S}(e_2) \\ \mathcal{E}(e_2); \mathcal{E}(e_1); \text{ADD} & \text{si } \mathcal{S}(e_1) < \mathcal{S}(e_2) \end{cases} \quad \mathcal{E}(\text{arbre } \otimes) = \begin{cases} \mathcal{E}(e_1); \mathcal{E}(e_2); \text{MUL} & \text{si } \mathcal{S}(e_1) \geq \mathcal{S}(e_2) \\ \mathcal{E}(e_2); \mathcal{E}(e_1); \text{MUL} & \text{si } \mathcal{S}(e_1) < \mathcal{S}(e_2) \end{cases}$$

Question 9 Montrer l'analogue de la question 5 pour l'algorithme de compilation d'Erchov : si on fait exécuter par la machine le code compilé $\mathcal{E}(e)$ avec ρ comme environnement et P comme pile initiale, la machine s'arrête avec la pile finale $\llbracket e \rrbracket_\rho \cdot P$. On ne demande pas de refaire une démonstration complète, mais juste d'expliquer ce qui change par rapport à la démonstration de la question 5.

Question 10 Calculer les nombres de Strahler $\mathcal{S}(G_n)$ et $\mathcal{S}(D_n)$ pour les expressions G_n et D_n définies à la question 7.

Question 11 En déduire la forme des codes $\mathcal{E}(G_n)$ et $\mathcal{E}(D_n)$ produits par l'algorithme d'Erchov. Quelle est la consommation en espace de ces codes ?

Question 12 Démontrer, pour toute expression e , que la consommation en espace du code compilé $\mathcal{E}(e)$ est égale à $\mathcal{S}(e)$. En d'autres termes, le nombre de Strahler est la consommation en espace du code produit par l'algorithme d'Erchov.

Question 13 On définit la taille $\|e\|$ d'une expression e comme le nombre de nœuds et de feuilles dans son arbre de syntaxe abstraite. Démontrer l'inégalité suivante :

$$\mathcal{S}(e) \leq \log_2(\|e\| + 1)$$

Question 14 En déduire qu'une machine disposant de 4 emplacements de pile peut évaluer n'importe quelle expression contenant au plus 7 additions ou multiplications.

Question 15 Expliquer comment construire une expression e de taille minimale dont le code compilé $\mathcal{E}(e)$ ne peut pas s'exécuter sur une machine disposant de 4 emplacements de pile. Quelle est la taille de cette expression ? Pourquoi cette taille est-elle minimale parmi toutes les expressions e satisfaisant cette propriété ?

Question 16 Comme dans la question 6, on ajoute à notre langage d'expressions l'opposé $\overset{\ominus}{\uparrow}_e$ et la différence

$\overset{\ominus}{\downarrow}_{e_1 e_2}$. Toujours comme dans la question 6, on ajoute aussi l'instruction OPP à notre machine à pile.

Étendre la définition des nombres de Strahler et l'algorithme d'Erchov aux deux nouvelles formes d'expressions.

On donnera les équations qui définissent les nombres $\mathcal{S}(\overset{\ominus}{\downarrow}_{e_1 e_2})$ et $\mathcal{S}(\overset{\ominus}{\uparrow}_e)$, ainsi que les codes $\mathcal{E}(\overset{\ominus}{\downarrow}_{e_1 e_2})$

et $\mathcal{E}(\overset{\ominus}{\uparrow}_e)$.

On représente une expression en OCaml par le type `expr` suivant :

```
type expr =
  | Const of int
  | Var of string
  | Somme of expr * expr
  | Produit of expr * expr
```

Question 17 Écrire une fonction `strahler : expr -> int` qui calcule le nombre de Strahler d'une expression donnée en argument. Cette fonction devra avoir une complexité linéaire en la taille de son argument, et on demande de prouver cette complexité.

Question 18 Écrire une fonction `erchov : expr -> instr list` qui prend en argument une instruction et calcule le code associé selon l'algorithme d'Erchov.

Question 19 Déterminer la complexité temporelle de la fonction `erchov`.

Question 20 Comment modifier la fonction `erchov` pour que sa complexité temporelle soit linéaire en la taille de son argument (si ce n'est pas déjà le cas) ? On ne demande pas de réécrire le code mais de décrire les modifications en français.

2 Compilation d'expressions vers une machine à registres

2.1 Une machine à registres

À la place de la machine à pile de la partie 1, nous allons maintenant cibler une machine à registres. Cette machine dispose de K registres nommés R_1, \dots, R_K , chaque registre pouvant contenir une valeur entière. La machine dispose de 4 instructions :

$\text{MOV}(R_i, n)$	met l'entier n dans le registre R_i
$\text{LOAD}(R_i, X)$	charge la valeur de la variable X depuis la mémoire et la met dans le registre R_i
$\text{ADD}(R_i, R_j, R_k)$	met dans R_i la somme des valeurs des registres R_j et R_k
$\text{MUL}(R_i, R_j, R_k)$	met dans R_i le produit des valeurs des registres R_j et R_k

Un code pour la machine à registres est une liste d'instructions, qui sont exécutées en séquence.

Exemple : l'exécution du code $\text{LOAD}(R_1, X); \text{MOV}(R_2, 1); \text{ADD}(R_1, R_1, R_2)$ a pour effet de mettre la valeur de l'expression $X + 1$ dans le registre R_1 , et l'entier 1 dans le registre R_2 . Les autres registres ne sont pas modifiés.

2.2 Compilation des expressions arithmétiques

On peut adapter les algorithmes de la partie 1 en traitant les K registres de la machine comme une pile de hauteur au plus K . Pour représenter une pile de hauteur $k \leq K$, on stocke le sommet de la pile dans le registre R_k , l'élément suivant de la pile dans le registre R_{k-1} , jusqu'à la base de la pile qui est stockée dans R_1 .

En appliquant cette technique à l'algorithme de compilation simple de la section 1.2, on obtient l'algorithme de compilation \mathcal{C} suivant, qui prend en arguments une expression e et un numéro de registre d :

$$\begin{aligned} \mathcal{C}(\textcircled{n}, d) &= \text{MOV}(R_d, n) \\ \mathcal{C}(\textcircled{X}, d) &= \text{LOAD}(R_d, X) \\ \mathcal{C}\left(\begin{array}{c} \oplus \\ \swarrow \quad \searrow \\ e_1 \quad e_2 \end{array}, d\right) &= \mathcal{C}(e_1, d); \mathcal{C}(e_2, d+1); \text{ADD}(R_d, R_d, R_{d+1}) \\ \mathcal{C}\left(\begin{array}{c} \otimes \\ \swarrow \quad \searrow \\ e_1 \quad e_2 \end{array}, d\right) &= \mathcal{C}(e_1, d); \mathcal{C}(e_2, d+1); \text{MUL}(R_d, R_d, R_{d+1}) \end{aligned}$$

Question 21 Quel code est produit par $\mathcal{C}(A, 1)$ pour l'expression A représentée figure 1 ?

Question 22 Pour e une expression et d un entier, expliquer informellement l'effet du code produit par $\mathcal{C}(e, d)$ sur les registres de la machine : une fois exécuté ce code, quelle valeur contient le registre R_d ? et le registre R_i pour $i < d$?

Question 23 Donner un exemple d'expression e telle que le code compilé $\mathcal{C}(e, 1)$ est incorrect car utilisant trop de registres : le code utilise le registre R_{K+1} qui n'existe pas.

Question 24 En s'inspirant de l'algorithme d'Erchov, définir un algorithme de compilation \mathcal{E} prenant en entrée e et d et produisant du code de la machine à registre qui :

1. calcule la valeur de l'expression e et met cette valeur dans le registre R_d ;
2. utilise au plus $\mathcal{S}(e)$ registres différents.

En déduire qu'on peut compiler correctement toutes les expressions dont le nombre de Strahler est au plus K .

2.3 Optimalité en nombre de registres utilisés

Dans cette section, nous allons montrer la réciproque de la question précédente : tout code de la machine à registres qui évalue correctement une expression e a besoin d'au moins $\mathcal{S}(e)$ registres différents. Par conséquent, l'algorithme de la question précédente est optimal en nombre de registres utilisés. Ce résultat a été publié par Ravi Sethi et Jeffrey D. Ullman en 1970.

Définition : Soient c un code de la machine à registres (une liste d'instructions), e une expression, et R un registre. On dit que c calcule la valeur de e dans R si les conditions suivantes sont remplies :

- c est de la forme $c_1; I; c_2$ où c_1 et c_2 sont des listes d'instructions et I une seule instruction ;
- aucune instruction de la liste c_2 n'écrit dans le registre R ;
- si e est une constante n , l'instruction I est $\text{MOV}(R, n)$;
- si e est une variable X , l'instruction I est $\text{LOAD}(R, X)$;
- si e est une somme $\begin{array}{c} \oplus \\ \swarrow \quad \searrow \\ e_1 \quad e_2 \end{array}$, l'instruction I est de la forme $\text{ADD}(R, R', R'')$, et de plus le code c_1 calcule la valeur de e_1 dans R' et la valeur de e_2 dans R'' ;

- si e est un produit $\begin{array}{c} \otimes \\ \swarrow \searrow \\ e_1 \quad e_2 \end{array}$, l'instruction I est de la forme $MUL(R, R', R'')$, et de plus le code c_1 calcule la valeur de e_1 dans R' et la valeur de e_2 dans R'' .

Exemple : le code $c = MOV(R_2, 1); LOAD(R_3, X); ADD(R_2, R_3, R_2)$ calcule la valeur de la variable X dans le registre R_3 et la valeur de la somme $\begin{array}{c} \oplus \\ \swarrow \searrow \\ X \quad 1 \end{array}$ dans le registre R_2 . Mais il ne calcule pas la valeur de la constante 1 dans R_2 .

Définition : On dit qu'un code c qui calcule la valeur d'une expression e utilise au moins k registres s'il existe un point dans le code c où k registres différents contiennent des résultats intermédiaires du calcul de e , c'est-à-dire les valeurs de k sous-expressions de e .

Exemple : le code c de l'exemple précédent utilise au moins deux registres, car au point qui suit immédiatement l'instruction $LOAD$, les registres R_2 et R_3 contiennent des résultats intermédiaires du calcul de $\begin{array}{c} \oplus \\ \swarrow \searrow \\ X \quad 1 \end{array}$, à savoir les valeurs de 1 et de X .

Définition : On dit qu'une expression e nécessite au moins k registres si tout code machine c qui calcule la valeur de e (dans un registre quelconque) utilise au moins k registres.

Question 25 En utilisant les résultats de la question 22, montrer que le code $\mathcal{C}(e, d)$ produit par l'algorithme simple de la section 2.2 calcule la valeur de e dans R_d (au sens de la définition ci-dessus).

Question 26 Soient e_1 et e_2 deux expressions telles que e_1 ou e_2 nécessite au moins k registres. Montrer que les expressions $\begin{array}{c} \oplus \\ \swarrow \searrow \\ e_1 \quad e_2 \end{array}$ et $\begin{array}{c} \otimes \\ \swarrow \searrow \\ e_1 \quad e_2 \end{array}$ nécessitent au moins k registres.

Question 27 Soient e_1 et e_2 deux expressions telles que e_1 et e_2 nécessitent au moins k registres. On suppose que les valeurs de e_1 et de ses sous-expressions sont différentes des valeurs de e_2 et de ses sous-expressions, de sorte qu'aucun résultat intermédiaire ne peut être réutilisé. Montrer que les expressions $\begin{array}{c} \oplus \\ \swarrow \searrow \\ e_1 \quad e_2 \end{array}$ et $\begin{array}{c} \otimes \\ \swarrow \searrow \\ e_1 \quad e_2 \end{array}$ nécessitent au moins $k + 1$ registres.

Question 28 Dédurre des deux questions précédentes que toute expression e sans partage de sous-expressions nécessite au moins $\mathcal{S}(e)$ registres.

3 Compilation d'expressions avec partage

3.1 Introduction du problème

Lors de la compilation d'une expression, une même sous-expression peut apparaître plusieurs fois. Par exemple, dans l'expression e_0 suivante :

$$(((3 \times 5) + 2) + (3 \times 5)) \times ((3 \times 5) + 2)$$

la sous-expression 3×5 apparaît trois fois, et la sous-expression $(3 \times 5) + 2$ apparaît deux fois. En s'autorisant à ne calculer qu'une seule fois certaines de ces sous-expressions, on peut améliorer l'efficacité du calcul, en temps (nombre d'évaluations de sous-expressions) et en espace (nombre maximal de registres utilisés simultanément). On représente donc dorénavant une expression non plus par un arbre, mais par un graphe orienté sans cycle.

Question 29 En utilisant le graphe H_0 , montrer qu'il existe une suite d'instructions sur une machine à registres qui compile e_0 en utilisant strictement moins de registres et strictement moins d'instructions que la compilation de l'arbre \mathcal{A}_0 .

Définition : On définit un *graphe orienté* $G = (S, A)$ comme un couple composé d'un ensemble fini non vide de *sommets* S et un ensemble d'*arêtes* $A \subset S \times S$.

Pour $(s, t) \in A$, on dit que t est un *successeur* de s et que s est un *prédécesseur* de t . Pour $(s, t) \in S^2$, un *chemin* de s à t est une suite finie $c = (s_0, s_1, \dots, s_k)$ d'éléments de S , où $s_0 = s$, $s_k = t$, et pour $i \in \{0, 1, \dots, k - 1\}$, s_{i+1} est un successeur de s_i . On dit alors que k est la *longueur* du chemin c . Si de plus $k \geq 1$ et $s = t$, on dit que c est un *cycle*.

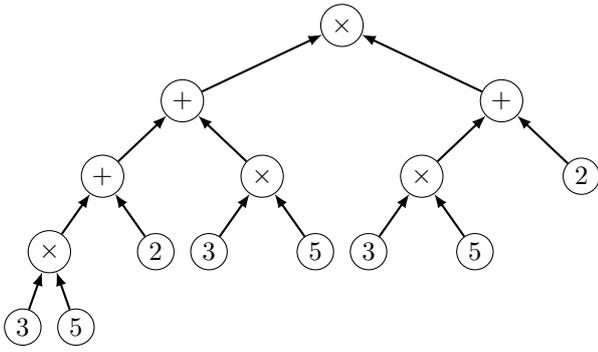


FIGURE 2 – e_0 sous forme d'un arbre \mathcal{A}_0

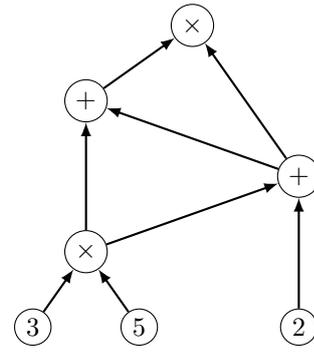


FIGURE 3 – e_0 sous forme d'un graphe H_0

On appelle *degré sortant* d'un sommet $s \in S$ son nombre de successeurs et *degré entrant* de s son nombre de prédécesseurs. On dit que s est un *puits* s'il est de degré sortant nul, et une *source* s'il est de degré entrant nul.

Pour la suite de ce problème, on ne considérera que des graphes orientés sans cycle.

On cherche à établir une heuristique pour déterminer l'ordre dans lequel doivent être calculés les sommets d'un graphe correspondant à une expression. Dans la mesure où cet ordre ne dépend pas des opérations, on représente un graphe en OCaml sans se soucier de l'étiquette des sommets.

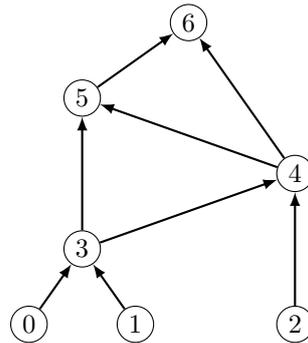


FIGURE 4 – Représentation de H_0 sans étiquette, en numérotant les sommets

Le type OCaml utilisé est le suivant :

```
type graphe = int list array
```

Dans le détail, pour $n \geq 0$, un graphe $G = (S, A)$ à n sommets sera représenté par un tableau g de taille n , où on assimilera S à $\{0, 1, \dots, n-1\}$, et tel que pour $s \in S$, $g.(s)$ est une liste qui contient tous les successeurs du sommet s (dans un ordre arbitraire et sans doublon). Par exemple, le graphe H_0 précédent peut être créé par la commande :

```
let h0 = [| [3]; [3]; [4]; [5; 4]; [5; 6]; [6]; [] |]
```

On remarque que lors de la compilation, il est intéressant de savoir quels sont les successeurs d'un sommet s donné (quels sont les sommets dont le calcul dépend de s ? À quel moment le registre qui stocke la valeur de s peut-il être libéré?), ainsi que les prédécesseurs d'un sommet s (quels sont les sommets dont le calcul est nécessaire pour calculer s ?). Pour un graphe $G = (S, A)$, on appelle *graphe transposé* de G , noté tG , le graphe où on a « retourné » toutes les arêtes, c'est-à-dire le graphe ${}^tG = (S, A')$ où $A' = \{(s, t), (t, s) \in A\}$.

Question 30 Écrire une fonction `transpose : graphe -> graphe` qui prend en argument un graphe $G = (S, A)$ et renvoie le graphe tG . La complexité de cette fonction devra être en $\mathcal{O}(|S| + |A|)$ et on demande de prouver cette complexité.

Définition : Pour un graphe $G = (S, A)$ à n sommets, un *ordre topologique* $(s_0, s_1, \dots, s_{n-1})$ est une suite de sommets de taille n et sans doublon telle que pour $i, j \in \{0, 1, \dots, n-1\}^2$, $(s_i, s_j) \in A \Rightarrow i < j$. Autrement dit, un sommet apparaît toujours dans l'ordre avant ses successeurs.

Pour un graphe G à n sommets, on représente en OCaml un ordre topologique $(s_0, s_1, \dots, s_{n-1})$ par un tableau de taille n contenant les valeurs s_0, s_1, \dots, s_{n-1} . Par exemple, le tableau suivant représente un ordre topologique de H_0 :

```
let topo = [|2; 1; 0; 3; 4; 5; 6|]
```

Question 31 On considère la fonction `est_ordre : graphe -> int array -> bool` suivante, qui prend en argument un graphe `g` correspondant à un graphe à n sommets et un tableau `topo` de taille n contenant tous les entiers de $\{0, 1, \dots, n-1\}$ sans doublon.

```
let est_ordre g topo =
  let n = Array.length topo in
  let rang = Array.make n (-1) in
  for i = 0 to n - 1 do
    rang.(topo.(i)) <- i
  done;
  let rec verif_aretes s lst =
    match lst with
    | [] -> s = n - 1 || verif_aretes (s + 1) g.(s + 1)
    | t :: q -> rang.(s) < rang.(t) && verif_aretes s q in
  verif_aretes 0 g.(0)
```

On cherche à montrer que cette fonction renvoie `true` si `topo` est un ordre topologique valide pour `g` et `false` sinon. Expliquer et caractériser par une assertion logique ce que contient le tableau `rang` à la fin de la boucle `for`. Donner une précondition nécessaire et suffisante pour que l'appel `verif_aretes s lst` renvoie `true`. Conclure que la fonction `est_ordre` a le comportement attendu.

Pour un ordre topologique donné, on compile un graphe de la manière suivante :

- les sommets du graphe sont calculés dans l'ordre topologique ;
- lorsqu'un sommet s_j est calculé :
 - pour chaque prédécesseur s_i de s_j ($i < j$), si s_j est le dernier successeur de s_i apparaissant dans l'ordre topologique, le registre stockant s_i peut être libéré au moment du calcul ;
 - on utilise alors un registre (libéré si possible, ou nouveau sinon) pour stocker le résultat de s_j ;
 - si s_j est un puits, on libère ensuite ce registre.

En particulier, chaque sommet ne sera calculé qu'une seule fois.

Question 32 Donner le nombre de registres utilisés pour calculer H_0 selon l'ordre $(2, 1, 0, 3, 4, 5, 6)$. Donner un ordre topologique utilisant strictement moins de registres.

On cherche à déterminer une heuristique permettant de calculer un ordre topologique utilisant peu de registres.

Définition : On redéfinit le *nombre de Strahler* d'un sommet de la manière suivante :

- si s est une source, alors $\mathcal{S}(s) = 1$;
- si s possède un unique prédécesseur t , alors $\mathcal{S}(s) = \mathcal{S}(t)$;
- si s possède deux prédécesseurs t et u , alors $\mathcal{S}(s) = \begin{cases} 1 + \mathcal{S}(t) & \text{si } \mathcal{S}(t) = \mathcal{S}(u) \\ \max(\mathcal{S}(t), \mathcal{S}(u)) & \text{si } \mathcal{S}(t) \neq \mathcal{S}(u) \end{cases}$

Pour les deux questions suivantes, on supposera que les graphes donnés en arguments correspondent à des expressions écrites avec des opérateurs binaires, c'est-à-dire possédant un unique puits et dont tous les sommets ont un degré entrant inférieur ou égal à 2.

Question 33 Écrire une fonction `strahler_graphe : graphe -> int array` qui prend en argument un graphe G et renvoie un tableau `st` tel que pour tout sommet s , `st.(s)` est égal à $\mathcal{S}(s)$. La complexité de cette fonction devra être en $\mathcal{O}(|S| + |A|)$ mais on ne demande pas de le prouver.

L'heuristique proposée pour trouver un ordre topologique particulier, appelé *ordre de Strahler*, pour un sommet s d'un graphe G est la suivante :

- si s est une source, l'ordre de Strahler est (s) ;
- si s possède un unique prédécesseur t , alors l'ordre de Strahler de s est l'ordre de Strahler de t suivi de s ;
- si s possède deux prédécesseurs t et u tels que $\mathcal{S}(t) \geq \mathcal{S}(u)$, alors l'ordre de Strahler de s est l'ordre de Strahler de t suivi de l'ordre de Strahler de u , auquel on a enlevé les sommets apparaissant dans l'ordre de Strahler de t , suivi de s .

L'ordre de Strahler d'un graphe G est alors l'ordre de Strahler de son unique puits.

Question 34 Écrire une fonction `ordre_strahler : graphe -> int array` qui prend en argument un graphe G et renvoie l'ordre de Strahler de G .

Question 35 En supposant qu'on commence toujours par le sommet de plus petit indice lorsqu'on a le choix entre deux sommets, donner l'ordre de Strahler déterminé par l'heuristique précédente sur le graphe de la figure 5 et le nombre de registres utilisés par cet ordre. Montrer qu'il existe un ordre topologique utilisant strictement moins de registres que l'ordre précédent.

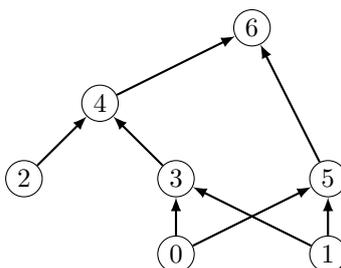


FIGURE 5

3.2 Jeu du marquage

Pour un graphe $G = (S, A)$ de \mathcal{G} , on définit le *jeu du marquage* sur G comme un jeu à un joueur. Les mouvements autorisés dans ce jeu sont les suivants :

- enlever un marqueur d'un sommet ;
- si tous les prédécesseurs d'un sommet s ont un marqueur, marquer s ;
- si tous les prédécesseurs d'un sommet s ont un marqueur, déplacer un marqueur de l'un de ces prédécesseurs vers s .

Le jeu commence sans aucun sommet marqué, et l'objectif du jeu est que chaque puits G soit marqué au moins une fois (pas nécessairement simultanément). On remarque qu'un cas particulier du mouvement (ii) est qu'une source peut être marquée à tout moment.

L'intérêt de ce jeu est qu'il représente la compilation d'un graphe orienté sans cycle : la libération d'un registre correspond au mouvement (i), et le calcul d'une expression en fonction de ses sous-expressions directes au mouvement (ii) si un nouveau registre est utilisé pour stocker le résultat, ou au mouvement (iii) si on utilise le registre de l'une des sous-expressions.

Définition : Une *stratégie* pour un marquage de G est une suite de mouvements permettant de gagner le jeu. Le *temps* de cette stratégie est le nombre de mouvements (ii) et (iii) utilisés, et l'*espace* de cette stratégie est le nombre maximal de marqueurs utilisés simultanément. On notera $M_{\min}(G)$ l'espace minimal d'une stratégie d'un marquage de G , c'est-à-dire le nombre minimal de marqueurs nécessaires pour marquer tous les puits de G .

Question 36 Déterminer $M_{\min}(H_0)$, H_0 étant le graphe représenté en figure 4.

Question 37 Montrer que tout graphe à n sommets possède une stratégie en temps n et en espace n .

Définition : Pour $k > 0$, on définit le *graphe pyramide* P_k par :

- P_1 est le graphe à un seul sommet, sans arête ;
- pour $k > 1$, si P_k est défini, et que $\{s_1, s_2, \dots, s_k\}$ sont les sources de P_k , alors on définit P_{k+1} en rajoutant $k + 1$ sommets $\{t_0, \dots, t_k\}$ et les $2k$ arêtes (t_{i-1}, s_i) et (t_i, s_i) , pour $i \in \{1, 2, \dots, k\}$.

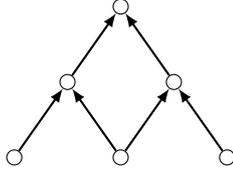


FIGURE 6 – Le graphe P_3

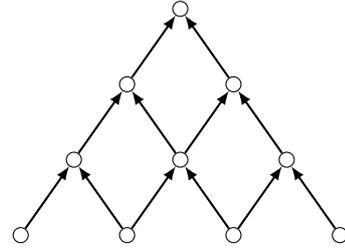


FIGURE 7 – Le graphe P_4

Une représentation des graphes P_3 et P_4 est donnée en figures 6 et 7.

Question 38 Montrer que pour $k > 0$, il existe une stratégie en espace k pour le graphe P_k . Quel est le temps minimal pour une telle stratégie ?

On cherche à montrer dans les deux questions suivantes qu'une telle stratégie est optimale.

On appelle p le puits de P_k . On considère une stratégie en temps T , c'est-à-dire utilisant T mouvements (m_1, \dots, m_T) , et en espace M . On définit l'ensemble suivant :

$$J = \{j \in \{1, 2, \dots, T\} \mid \text{après } m_j, \text{ tous les chemins d'une source de } P_k \text{ à } p \text{ contiennent un sommet marqué}\}$$

Question 39 Montrer que J est non vide. On note j_0 son minimum. Montrer que le mouvement m_{j_0} a marqué une source s_0 de P_k .

Question 40 Montrer que M est supérieur ou égal à la longueur du chemin de s_0 à p . En déduire que $M_{\min}(P_k) = k$ pour $k > 0$.

3.3 Une borne supérieure en espace

L'étude des graphes pyramides a permis de montrer que pour $n > 0$, il existe des graphes à n sommets qui ne possèdent pas de stratégie en espace inférieur à $\sqrt{2n}$. Il s'agit d'une borne inférieure de l'espace minimal requis pour une stratégie de certains graphes à n sommets. En 1976, W. Paul, R. Tarjan et J. Celoni ont amélioré cette borne inférieure en construisant, pour tout $n > 0$, un graphe à n sommets dont toute stratégie nécessite un espace de l'ordre de $\frac{n}{\log n}$.

Dans cette partie, on cherche à montrer que sous certaines hypothèses de degrés entrants, cette borne est optimale. On considère pour la suite uniquement des graphes orientés sans cycles dont les degrés entrants de tous les sommets sont inférieurs ou égaux à 2, en d'autres termes, tels que chaque sommet a zéro, un ou deux prédécesseurs.

On veut montrer que tout graphe à n sommets possède une stratégie en espace $\mathcal{O}\left(\frac{n}{\log n}\right)$.

On note, pour $m > 0$, $a(m)$ le nombre d'arêtes minimal d'un graphe G tel que $M_{\min}(G) = m$. Dans un souci de simplification de la preuve, jusqu'à la question 48 incluse, on supposera que m est un multiple de 4.

Question 41 Montrer que a est une fonction croissante.

On considère un graphe $G = (S, A)$ tel que $M_{\min}(G) = m$, on note T_1 l'ensemble des sommets qui peuvent être marqués en utilisant $\frac{m}{2}$ marqueurs ou moins, et $T_2 = S \setminus T_1$. Par définition, tous les sommets de T_2 nécessitent strictement plus que $\frac{m}{2}$ marqueurs. On note de plus $B_1 = A \cap (T_1 \times T_1)$, $B_2 = A \cap (T_2 \times T_2)$, et $B = A \setminus (B_1 \cup B_2)$. Enfin, on note $H_1 = (T_1, B_1)$ et $H_2 = (T_2, B_2)$.

Question 42 Montrer que pour toute arête $(s, t) \in B$, $s \in T_1$ et $t \in T_2$.

Question 43 Montrer que pour toute stratégie pour un jeu restreint au graphe H_1 , il existe un sommet de T_1 qui nécessite $\frac{m}{2} - 1$ marqueurs ou plus pour être marqué.

On pourra raisonner par l'absurde en considérant un sommet $s \in T_2$ dont tous les prédécesseurs dans le graphe G sont dans T_1 .

Question 44 On cherche à montrer que pour une stratégie pour un jeu restreint au graphe H_2 , il existe un sommet qui nécessite $\frac{m}{2} - 1$ marqueurs ou plus. Pour ce faire, on raisonne par l'absurde, et on suppose pour cette question que tous les sommets peuvent être marqués dans H_2 en utilisant $\frac{m}{2} - 2$ marqueurs ou moins. Montrer que dans un jeu dans G , tous les sommets de S peuvent être marqués en utilisant $m - 1$ marqueurs ou moins et conclure.

On suppose pour les trois questions suivantes que $|B| < \frac{m}{4}$.

Question 45 Montrer que dans un jeu dans le graphe H_1 , tous les prédécesseurs dans T_1 d'un sommet de T_2 peuvent être marqués simultanément en utilisant $\frac{m}{2} + \frac{m}{4} - 1 \leq \frac{3m}{4}$ marqueurs ou moins.

Question 46 En déduire qu'une stratégie dans le graphe H_2 nécessite $\frac{3m}{4}$ marqueurs ou plus.

Question 47 Montrer qu'il existe un ensemble $C \subset B_2$, de cardinal $\frac{m}{4}$ ou plus, tel qu'une stratégie sur le graphe $(T_2, B_2 \setminus C)$ nécessite $\frac{m}{2}$ marqueurs ou plus.

On revient dans le cas général sans hypothèse sur $|B|$.

Question 48 Montrer que dans tous les cas, $a(m) \geq 2a\left(\frac{m}{2} - 1\right) + \frac{m}{4}$.

On suppose que le résultat de la question 48 est vrai même lorsque m n'est pas un multiple de 4.

Question 49 Montrer que pour tout $m > 0$, $a(m) \geq \frac{m}{8} \log_2 m$. En déduire qu'un graphe qui nécessite m marqueurs ou plus pour être marqué possède $\frac{m}{16} \log_2 m$ sommets ou plus. Conclure que, pour n assez grand, tout graphe à n sommets possède une stratégie en espace au plus $\frac{32n}{\log_2 n}$.