DS1

Ce devoir est composé de deux parties totalement indépendantes. La première partie porte sur les automates et la seconde sur la théorie de jeux et comporte des questions de programmaton à la fois en C et en Ocaml.

1 Longueur discriminante de deux automates

Deux automates $\mathcal{A}$ et $\mathcal{A}'$ sont équivalents s'ils reconnaissent le même langage. S'ils ne sont pas équivalents, alors il existe des mots qui sont reconnus par l'un et pas par l'autre. La longueur minimum des mots qui ont cette propriété est dite discriminante. L'objet de ce problème est d'évaluer, par deux méthodes, un majorant de la longueur discriminante de $\mathcal{A}$ et $\mathcal{A}'$ en fonction des nombres d'états de $\mathcal{A}$ et $\mathcal{A}'$.

Notations et terminologie

Un alphabet Σ est un ensemble fini d'éléments appelés lettres. Un mot sur Σ est une suite finie de lettres de Σ; le mot vide est noté ε. On désigne par Σ^* l'ensemble des mots sur Σ, y compris le mot vide. La longueur d'un mot m, notée $|m|$, est le nombre de lettres qui le composent. Un langage est une partie de Σ^*.

Un automate $\mathcal{A}$ est décrit par une structure $\langle \Sigma, Q, T, I, F >$, où :

- Σ est un alphabet ;
- Q est un ensemble fini et non vide appelé ensemble des états de $\mathcal{A}$;
- $T \subseteq Q \times \Sigma \times Q$ est appelé l'ensemble des transitions ; étant donnée une transition $(p, a, q) \in T$, on dit qu'elle va de l'état p à l'état q et qu'elle est d'étiquette a; on pourra la noter $p \xrightarrow{a} q$;
- $I \subseteq Q$ est appelé ensemble des états initiaux de $\mathcal{A}$;
- $F \subseteq Q$ est appelé ensemble des états finals de $\mathcal{A}$.

On représente graphiquement l'automate $\mathcal{A}$ ainsi :

- un état p est figuré par un cercle marqué en son centre par p; si p appartient à I, cela est figuré par une flèche entrante sans origine ; si un état q appartient à F, cela est figuré par une flèche sortante sans but ;
- une transition $(p, a, q) \in T$ est figurée par une flèche allant de l'état p vers l'état q et étiquetée par la lettre a.

Un calcul c de $\mathcal{A}$ est un chemin de la forme $p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \dots \xrightarrow{a_k} p_k$, avec $p_{i-1} \xrightarrow{a_i} p_i \in T$ pour $1 \leq i \leq k$; p_0 est l'origine du calcul, p_k son extrémité. L'étiquette de c est le mot formé par la suite des étiquettes des transitions successives du chemin.

Un calcul de $\mathcal{A}$ d'origine p, d'extrémité q et d'étiquette m est dit réussi si on a $p \in I$ et $q \in F$. Un mot $m \in \Sigma^*$ est reconnu par $\mathcal{A}$ s'il est l'étiquette d'un calcul réussi. Le langage reconnu par $\mathcal{A}$, noté $\mathrm{L}(\mathcal{A})$, est l'ensemble des mots reconnus par $\mathcal{A}$. Deux automates $\mathcal{A}$ et $\mathcal{A}'$ sont dits équivalents si on a $\mathrm{L}(\mathcal{A}) = \mathrm{L}(\mathcal{A}')$.

L'automate $\mathcal{A}$ est dit déterministe si I ne contient qu'un élément et si, pour tout $(p, a) \in Q \times \Sigma$, il existe au plus un état $q \in Q$ avec $(p, a, q) \in T$. L'automate $\mathcal{A}$ est dit complet si et seulement si, pour tout $p \in Q$ et tout $a \in \Sigma$, il existe $q \in Q$ avec $(p, a, q) \in T$.

PREMIÈRE PARTIE : approche naïve

Q1 Soit $\mathcal{A}$ un automate, déterministe ou non déterministe, avec n états. Montrer que $\mathrm{L}(\mathcal{A})$ est vide si et seulement s'il ne contient aucun mot de longueur inférieure ou égale à $n - 1$.

Q2 Soit $\mathcal{A}$ un automate déterministe complet ayant n états. Donner un automate ayant aussi n états qui reconnait $\overline{\mathrm{L}(\mathcal{A})}$, le complémentaire dans Σ^* de $\mathrm{L}(\mathcal{A})$. Justifier votre réponse.

Q3 Soient $\mathcal{A}$ et $\mathcal{A}'$ deux automates déterministes complets utilisant le même alphabet Σ avec respectivement n et n' états. Donner un automate ayant $n \times n'$ états qui reconnait $\mathrm{L}(\mathcal{A}) \cap \mathrm{L}(\mathcal{A}')$. Justifier votre réponse.

Q4 Soient $\mathcal{A}$ et $\mathcal{A}'$ deux automates déterministes complets utilisant le même alphabet Σ avec respectivement n et n' états. Montrer que si $\mathcal{A}$ et $\mathcal{A}'$ ne sont pas équivalents, il existe un mot de longueur au plus $n \times n' - 1$ qui est reconnu par l'un et non par l'autre, i.e. la longueur discriminante de $\mathcal{A}$ et $\mathcal{A}'$ est inférieure ou égale à $n \times n' - 1$.

SECONDE PARTIE : Approche plus fine

L'objectif de cette partie est de démontrer que la longueur discriminante de deux automates déterministes non équivalents, avec n et n' états respectivement, est inférieure ou égale à $n + n' - 1$.

Q5 Montrer sur un exemple, avec un alphabet à une seule lettre, qu'il existe des automates déterministes $\mathcal{A}$ et $\mathcal{A}'$ non équivalents et qui reconnaissent les mêmes mots de longueur strictement inférieure à $n + n' - 1$, où n (resp. n') désigne le nombre d'états de $\mathcal{A}$ (resp. de $\mathcal{A}'$).

On introduit un ensemble de définitions et de notations :

Soit $\mathcal{A} = \langle \Sigma, Q, T, I, F \rangle$ un automate déterministe avec n états. On identifie l'ensemble Q des états de $\mathcal{A}$ avec l'ensemble $\{1, 2, \ldots, n\}$ des n premiers entiers naturels non nuls de sorte que chaque état est identifié par un entier compris entre 1 et n. On suppose que l'on a $I = \{\mathbf{1}\}$.

On note $\{\mathrm{e}_1, \mathrm{e}_2, \ldots, \mathrm{e}_n\}$ la base canonique de l'espace vectoriel $\mathbf{R}^n$. Pour un entier i compris entre 1 et n, le vecteur e_i est donc le vecteur de $\mathbf{R}^n$ dont toutes les composantes sont nulles sauf la i-ième qui vaut 1 . On note 0 le vecteur nul de $\mathbf{R}^n$.

Pour chaque lettre a de l'alphabet Σ, on définit une application linéaire φ_a de $\mathbf{R}^n$ dans $\mathbf{R}^n$ par :
pour tout i avec $1 \leq i \leq n$:

- s'il existe $j, 1 \leq j \leq n$, tel que $(i, a, j) \in T$, alors $\varphi_a(\mathrm{e}_i) = \mathrm{e}_j$,
- sinon $\varphi_a(\mathrm{e}_i) = 0$.

L'existence et l'unicité de l'application φ_a découlent du fait que $\mathcal{A}$ est déterministe et de la linéarité de φ_a.

Si $m = a_1 a_2 \ldots a_{k-1} a_k$ est un mot non vide de Σ^*, où $a_1, a_2, \ldots, a_{k-1}, a_k$ sont des lettres de Σ, on pose :

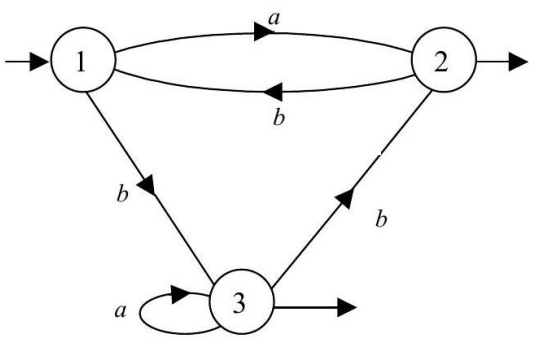
$$\varphi_m = \varphi_{a_k} \circ \varphi_{a_{k-1}} \circ \cdots \circ \varphi_{a_2} \circ \varphi_{a_1}$$

On note φ_ε l'identité de $\mathbf{R}^n$ dans $\mathbf{R}^n$.

On appelle z la somme des vecteurs e_i quand i décrit l'ensemble F des états finals : $z = \sum_{i \in F} e_i$.

Enfin, si u et v sont deux vecteurs de $\mathbf{R}^n$, on note $u.v$ le produit scalaire de u et de v : $u.v = \sum_{i=1}^{n} u_i v_i$.

Exemple :



$$\varphi_a(\mathrm{e}_1) = \mathrm{e}_2, \varphi_a(\mathrm{e}_2) = 0, \varphi_a(\mathrm{e}_3) = \mathrm{e}_3$$

$$\varphi_b(\mathrm{e}_1) = \mathrm{e}_3, \varphi_b(\mathrm{e}_2) = \mathrm{e}_1, \varphi_b(\mathrm{e}_3) = \mathrm{e}_2$$
$$z = (0,1,1).$$

On peut aussi constater les égalités suivantes :
$\varphi_{abb}(\mathrm{e}_1) = \mathrm{e}_3, \varphi_{ba}(\mathrm{e}_3) = 0, \varphi_{ba}(\mathrm{e}_2) = \mathrm{e}_2$.

Q6 Soient $m \in \Sigma^*$ et deux entiers i et j vérifiant $1 \leq i \leq n$ et $1 \leq j \leq n$. Montrer qu'il existe un calcul d'origine i, d'extrémité j et d'étiquette m si et seulement si on a $\varphi_m(\mathrm{e}_i) = \mathrm{e}_j$.

Q7 Soit $m \in \Sigma^*$. Donner les valeurs possibles du produit scalaire $\varphi_m(\mathrm{e}_1).z$ et indiquer une condition nécessaire et suffisante portant sur ce produit scalaire pour que m soit un mot reconnu par $\mathcal{A}$.

Dans la suite du problème, on considère en plus de l'automate déterministe $\mathcal{A}$ un automate déterministe $\mathcal{A}'$. Les notations utilisées pour $\mathcal{A}'$ sont les mêmes que celles utilisées pour $\mathcal{A}$ à cela près que tous les identificateurs sont dotés d'un "prime" (on a donc : $n', \varphi'_a, \varphi'_m, z'$ et les vecteurs de la base canonique de $\mathbf{R}^{n'}$ sont notés e'_i).

Si u et v sont des vecteurs respectivement de $\mathbf{R}^n$ et de $\mathbf{R}^{n'}$, on note $(u;v)$ le vecteur w de $\mathbf{R}^{n+n'}$ obtenu en concaténant u et v; plus précisément, le vecteur w est défini par :

$$\begin{cases} \text{si } 1 \leq i \leq n, w_i = u_i \\ \text{si } n+1 \leq i \leq n+n', w_i = v_{i-n} \end{cases}$$

Par exemple, si $u = (0,1,0)$ et $v = (1,2)$, $(u;v)$ est le vecteur $(0,1,0,1,2)$.
On pose :

$$E = (\mathrm{e}_1; -\mathrm{e}'_1) \text{ (bien noter le signe } -)$$
$$Z = (z; z').$$

Pour tout mot m de Σ^*, on définit une application linéaire Φ_m de $\mathbf{R}^{n+n'}$ dans $\mathbf{R}^{n+n'}$ par :

$$\text{pour tout } u \in \mathbf{R}^n \text{ et pour tout } u' \in \mathbf{R}^{n'}, \Phi_m((u;u')) = (\varphi_m(u); \varphi'_m(u'))$$

(on admet la linéarité de Φ_m).

Q8 Soit $m \in \Sigma^*$; indiquer les valeurs possibles du produit scalaire $\Phi_m(E).Z$ et, selon ces valeurs, préciser l'appartenance de m à $\mathrm{L}(\mathcal{A})$ et $\mathrm{L}(\mathcal{A}')$.

Pour $k \in \mathbf{N}$, on note V_k le sous-espace vectoriel de $\mathbf{R}^{n+n'}$ engendré par $\{\Phi_m(E) \,|m \in \Sigma^*, |\, m \,| \leq k\}$; V_0 est donc engendré par le vecteur $(\mathrm{e}_1; -\mathrm{e}'_1)$.

Q9 Montrer que, pour $k \geq 0$, on a $V_k \subseteq V_{k+1}$.

Q10 Soit $m \in \Sigma^*$; on suppose que m s'écrit sous la forme : $m = \mu a$, où $\mu \in \Sigma^*$ et $a \in \Sigma$. Montrer l'égalité $\Phi_m = \Phi_a \circ \Phi_\mu$.

Q11 Soit $w \in V_k$ et $a \in \Sigma$; montrer : $\Phi_a(w) \in V_{k+1}$.

Q12 On suppose qu'il existe $k \geq 0$ tel que $V_k = V_{k+1}$; montrer l'égalité $V_{k+2} = V_{k+1}$.

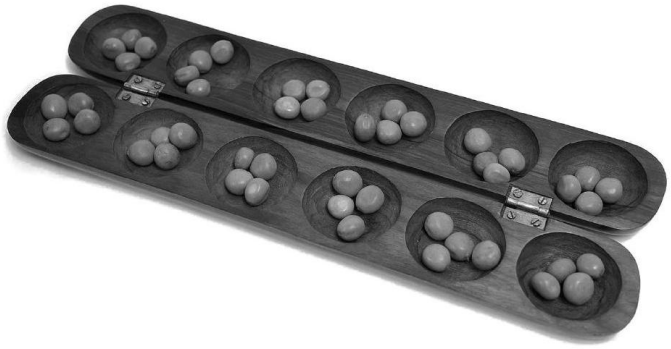
Q13 Montrer qu'il existe un entier $h \leq n + n' - 1$ tel que, pour tout $k \geq h, V_k = V_h$.

Q14 Montrer que si $\mathcal{A}$ et $\mathcal{A}'$ ne sont pas équivalents, il existe un mot de longueur inférieure ou égale à $n + n' - 1$ qui est accepté par l'un et pas par l'autre, i.e. que $n + n' - 1$ est un majorant de la longueur discriminante de $\mathcal{A}$ et $\mathcal{A}'$.

2 Le jeu de l'awalé

Partie 1 : Présentation et règles

Le sujet porte sur l'étude de l'awalé, un jeu de stratégie très ancien qui fait partie des jeux de semailles. En effet, à son origine, il était pratiqué à l'aide de graines qui étaient semées dans deux rangées de 6 trous creusés dans un plateau en bois ou à même le sol. Ce jeu est très répandu en Afrique et à partir du XVII e siècle des indices de sa pratique sont également trouvés en Amérique du Sud et en Asie.



Les règles de ce jeu sont particulièrement simples et s'apprennent rapidement. II en existe plusieurs variantes mais ce sujet n'en exposera qu'une seule. L'awalé se joue à deux. À tour de rôle, les joueurs prennent les graines situées dans un trou de leur rangée pour ensuite les déplacer dans les autres trous. Des graines peuvent ensuite être récoltées pour que les joueurs se constituent une réserve personnelle.

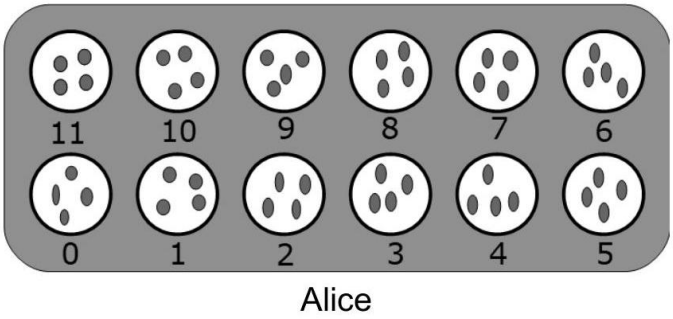
1.1 - But du jeu

L'objectif est de récolter plus de graines que son adversaire. Au départ, le plateau est composé de 2 rangées de 6 trous contenant chacun 4 graines comme le montre la figure 2 . Le jeu s'arrête si l'un des joueurs obtient dans sa réserve personnelle à côté du plateau plus de la moitié des graines en jeu, c'est-à-dire au moins 25 graines ou jusqu'à une situation empêchant le gain de nouvelles graines.

1.2 - Déroulement de la partie

Les 2 participants jouent à tour de rôle. Les joueurs sont appelés par la suite Alice et Bob; Alice joue en premier. Les joueurs sont face à face. La rangée de 6 trous située juste devant le joueur est appelée son camp. Voici la configuration de départ du jeu :

Bob

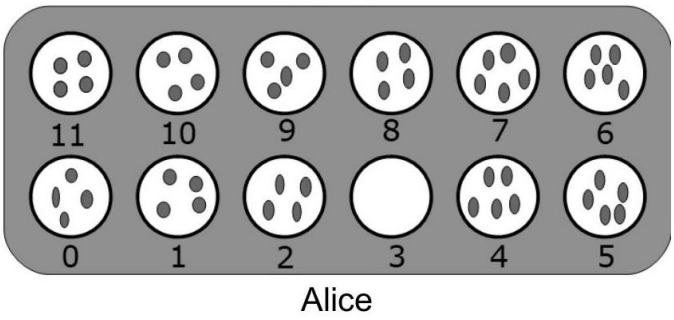


Alice

Chaque coup consiste à choisir une case non vide de son camp, à prendre toutes les graines de cette case en main et à les semer à raison d'une graine par case en suivant le sens direct, c'est-à-dire le sens inverse des aiguilles d'une montre. On ne sème jamais de graine dans la case d'origine choisie. En effet, si la case de départ choisie contient plus de 11 graines, le joueur va semer sur un tour de plateau complet et revenir à la case d'origine des graines; il faut alors sauter cette case pour continuer de semer les graines dans les autres cases. À la fin de son tour de jeu, la case de départ choisie par le joueur est nécessairement vide.

Voici, par exemple, la situation obtenue après qu'Alice ait semé la case d'indice 3 :

Bob



Une règle fondamentale de l'awalé est l'interdiction d'affamer son adversaire (**règle de la famine**). Lorsqu'un joueur n'a plus de graines dans son camp, son adversaire est obligé de jouer un coup qui lui en apporte au moins une.

Par ailleurs, il est interdit de jouer un coup qui ôte, après récolte, toutes les graines du camp adverse.

Une fois qu'un joueur a terminé de semer, il peut récolter les graines du plateau de jeu (en respectant la règle précédente). La récolte consiste à retirer les graines du plateau pour les stocker dans sa réserve personnelle sur la table à côté du plateau de jeu. Le joueur récolte les graines disponibles après son tour de semence, en commençant par la dernière case dans laquelle il a semé et sous les conditions suivantes :

- la case appartient au camp adverse (condition 1);
- cette case contient exactement 2 ou 3 graines (condition 2);
- s'il vient de ramasser les graines de la case, le joueur doit continuer la récolte dans le sens inverse de la semence, si la case respecte les deux premières conditions;
- il est interdit d'affamer son adversaire, on ne peut donc pas prendre toutes les graines du camp adverse. Si la phase de récolte se termine ainsi, alors la récolte est annulée (condition 3 liée à la règle de la famine).

1.3 - Conditions de fin

La partie s'arrête sous certaines conditions :

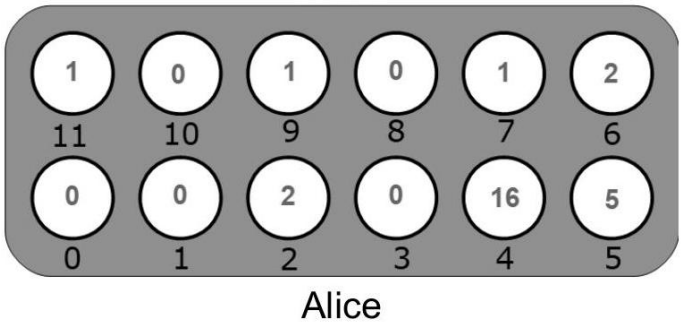
- un joueur obtient au moins 25 graines dans sa réserve;
- 3 graines ou moins restent sur le plateau;
- un joueur est dans l'incapacité de jouer car aucun coup ne permet de respecter les différentes règles.

À la fin de la partie, chaque joueur ramasse les graines de son camp pour les transférer dans sa réserve personnelle. Le décompte des points peut alors se faire. Le joueur ayant obtenu au moins 25 graines est alors déclaré vainqueur. Une situation de nullité est possible si les deux joueurs obtiennent autant de graines chacun à la fin de la partie.

1.4- Compréhension des règles

Exemple de situation - Alice doit jouer :

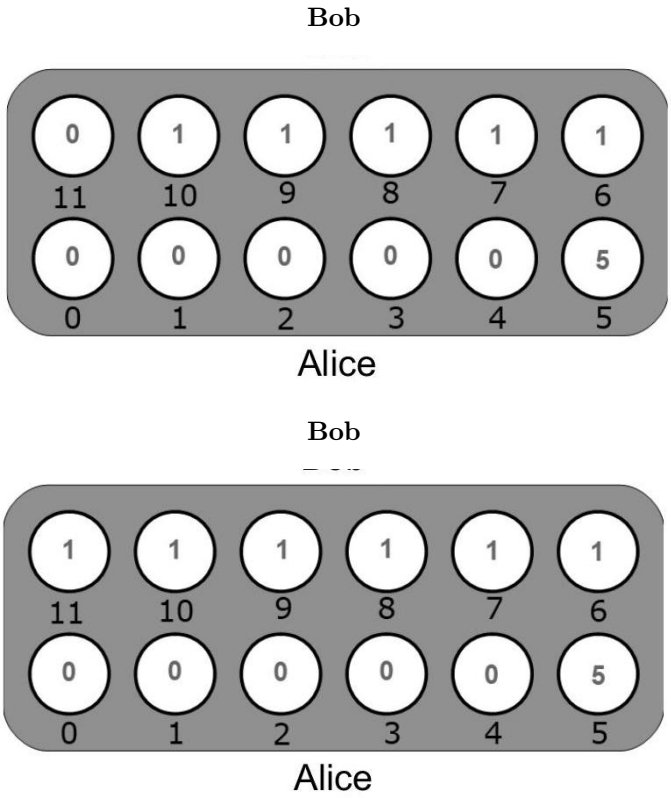
Bob



Q15 Indiquer, en justifiant, les indices des cases qu'Alice peut choisir de jouer.

Q16 Pour chacun des choix de cases possibles, renseigner la situation possible du plateau de jeu après le coup d'Alice (c'est-à-dire après avoir semé et récolté les graines). Indiquer également le gain éventuel pour chaque coup possible.

On considère les deux situations de jeu indiquées par la figure suivante. C'est au tour d'Alice de jouer.



Q17 Pour chacune de ces deux situations de jeu, donner et justifier la situation du plateau après le tour de jeu d'Alice.

Partie 2 : Programmation de la structure du jeu

Nous allons programmer les fonctions principales permettant de disposer d'une interface du jeu de l'awalé.

2.1 Représentation du jeu

Nous allons représenter le jeu en C par la structure suivante :

```
struct jeu {
    int scorej1;
    int scorej2;
    int* plateauj1;
    int* plateauj2;
    int tour;
};

typdef struct jeu jeu;
```

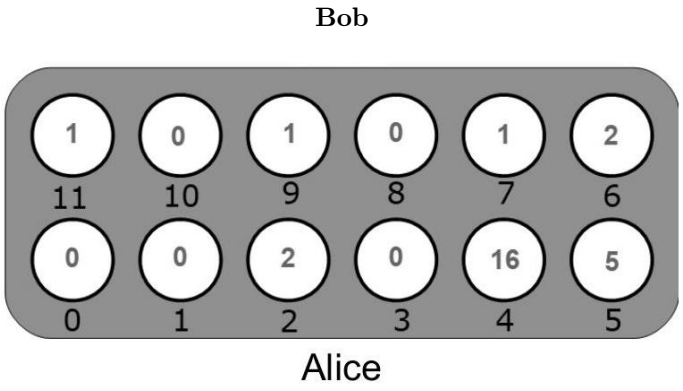
Si on dispose d'un jeu `j` alors les variables `j.scorej1` et `j.scorej2` représentent respectivement le nombre de points des joueurs 1 et 2, ces deux valeurs sont initialisées à 0.

La variable `j.tour` représente le nombre de tours effectués dans le jeu et commence donc à 0.

C'est le joueur 1 qui commencera la partie.

Les tableaux `j.plateauj1` et `j.plateauj2` représentent les camps de chacun des deux joueurs, ce sont donc des tableaux d'entiers de taille 6. Chaque case du tableau contient le nombre de graines contenues dans la case du jeu.

L'indexation des tableaux correpond à la lecture de gauche à droite des cases du point de vue du joueur. Ainsi, si Alice est le joueur 1 et Bob le joueur 2 alors la situation



est représentée par :

```
j.plateauj1 = {0,0,2,0,16,5} //camp d'Alice
j.plateauj2 = {2,1,0,1,0,1} //camp de Bob de son point de vue
```

Q18 Ecrire une fonction `jeu* initialisation(void)` qui renvoie un jeu initialisé selon les règles présentées. Ecrire une fonction `void libere(jeu* j)` qui libère toute la mémoire allouée pour `j`.

Q19 Donner la parité de `j.tour` lorsque c'est au tour du joueur 1 de jouer.
Ecrire une fonction `bool tour_joueur1(jeu* j)` qui renvoie `true` si c'est le tour du joueur 1 et `false` sinon.

Q20 Donner le maximum de graines que peut contenir une case. Déterminer alors le nombre de bits nécessaires pour coder les entiers représentants le nombre de graines par case.

Dans la suite du sujet, nous aurons besoin d'une fonction permettant de copier le jeu.

Q21 Ecrire une fonction `jeu* copie(jeu* j)` qui effectue une telle copie profonde.

Le déroulement d'un jeu d'awalé entre deux joueurs a la structure suivante :

```
int awale_jcj(void){
    jeu* j = initialisation();
    bool jeu_continue = true;
    int* tab = malloc(6*sizeof(int));
    while(jeu_continue){
        int case_choisie;
        printf("Choisir une case : \n");
        scanf("%d", &case_choisie);
        jeu_continue = tour_jeu(j,case_choisie,tab);
    }
    free(tab);
    return (gagnant(j));
}
```

Nous allons considérer que le joueur ne commet pas de faute de frappe et rentre toujours un indice de case valide.

2.2 - Programmation d'un tour de jeu

Une fois la case de début de tour choisie par le joueur, un tour de jeu a la structure suivante :

- tester si la case où l'on prend les graines est valide ou non (les conditions de validité sont explicitées ensuite) ;
- si le choix est valide, semer les graines puis récolter des graines. On incrémente alors le nombre de tours et on ajoute au score du joueur le nombre de graines récoltées ;
- tester si la partie est finie ou non (les conditions sont décrites dans la première partie).

Q22 Ecrire une fonction `int deplacer_graines(jeu* j, int casenb)` qui prend en arguments un jeu `j` et un numéro de case `casenb` (nécessairement compris entre 0 et 5) non vide qui correspond à la case où le joueur courant prend les graines. Cette fonction modifie le plateau de jeu de `j` en vidant la case où sont prises les graines et en les semant dans le sens direct (inverse des aiguilles d'une montre). Elle renvoie l'indice (compris entre 0 et 11) correspondant à la case où la dernière graine a été semée. L'indexation entre 0 et 11 correspond au point du vue du joueur courant. Ainsi, si c'est au joueur 2 de jouer alors les cases 0 à 5 sont celles de son camp alors que les cases 6 à 11 sont celles du camp adverse. La fonction pourra consister en deux parties symétriques selon que ce soit le tour du joueur 1 ou 2 de jouer.

Q23 Ecrire une fonction `bool case_ramassable(jeu* j, int casenb)` qui renverra `true` ssi le joueur dont c'est le tour a le droit de ramasser les graines de la case proposée. On ne testera pas la condition de la famine pour simplifier le problème. Pour rappel, le joueur peut ramasser le contenu de la case si :

- la case appartient au camp adverse.
- la case contient 2 ou 3 graines.

Q24 Ecrire une fonction `int ramasser_graines(jeu* j, int casenb)` qui prend en arguments un jeu `j` et un indice de case `casenb`, on suppose que l'étape de semage a déjà eu lieu et que `casenb` correspond à l'indice où a été semée la dernière graine, cette fonction va procéder au ramassage des graines. La fonction attendue doit être récursive et doit modifier le plateau en place. Cette fonction renvoie le résultat de la récolte c'est-à-dire le gain du joueur courant à l'issue du tour.

Il faut vérifier à chaque tour de jeu si le choix d'une case est autorisé ou non. Si c'est un joueur humain qui joue, son choix peut se porter sur une case "interdite", c'est-à-dire dont il ne peut pas prendre les graines. Si c'est un joueur virtuel, celui-ci doit pouvoir faire la liste des cases "acceptables". Une case est "acceptable" si :

- condition 1 : elle est du côté du joueur dont c'est le tour ;
- condition 2 : elle est non vide ;
- condition 3 : à la fin du tour de jeu, les cases de l'adversaire ne sont pas complètement vides (condition de famine).

Q25 Ecrire une fonction `bool test_famine(jeu* j, int casenb)` qui vérifie si la case `casenb` vérifie la condition 3. Cette fonction ne doit pas modifier les arguments passés en entrée ni engendrer de fuite mémoire.

Q26 Ecrire une fonction `bool test_case (jeu* j, int casenb)` qui vérifie que la case passée en argument vérifie bien les trois conditions connaissant l'état actuel du jeu.

On veut maintenant pouvoir déterminer l'ensemble des cases pour lesquelles il est possible de jouer. Pour cela, on utilisera un tableau de booléens `tab` de taille 6 tel que `tab[i]` vaut `true` ssi le joueur courant peut jouer la case d'indice `i` du tableau correpondant à son camp.

Q27 Ecrire une fonction `bool cases_possibles(jeu* j,bool* tab)` qui prend en entrée le jeu et un tel tableau déjà alloué et le complète afin de savoir quels sont les indices de cases qu'il est effectivement possible de choisir. La fonction renverra `true` ssi il existe au moins un coup possible.

Après un tour, il faut vérifier si le jeu est terminé ou si les joueurs peuvent continuer à jouer. Le jeu se termine si l'une des conditions suivantes est vérifiée :

- un des joueurs possède plus de la moitié des graines (soit au moins 25) ;
- le nombre de tours joués est supérieur ou égal à 100 (pour éviter un jeu infini lorsqu'il y a peu de graines) ;
- il reste 3 graines ou moins sur le plateau ;
- le joueur qui va jouer ne possède plus de case jouable.

Q28 Ecrire une fonction `bool tour_suivant(jeu* j,int* tab)` qui renvoie `true` ssi le jeu peut continuer. Cette fonction prend en argument, en plus du jeu, un tableau alloué permettant de stocker les coups possibles.

```
bool tour_jeu(jeu* j, int casenb, int* tab){
    if (test_case(j,casenb)){
        int indice_fin = /*compléter*/
        int ajout_score = /*compléter*/
        if (tour_joueur1(j)){
            /*compléter*/
        }
        else{
            /*compléter*/
        }
        j->tour++;
        return (tour_suivant(j,tab));
    }
    else{
        printf("La case choisie n'est pas valable\n");
        return true;
    }
}
```

Q29 L'ébauche de la fonction `bool tour_jeu(jeu* j, int casenb, int* tab)` est donnée ci-dessus. Si le test de la case n'est pas valide, la fonction affiche un message et renvoie `true` pour permettre au joueur de proposer une nouvelle case. Sinon, elle renvoie `true` si le jeu peut continuer et `false` si le jeu ne peut pas continuer. Compléter le code proposé.

Q30 Ecrire la fonction `int gagnant(jeu* j)` qui procède au ramassage des graines de chacun des camps, les ajoute aux scores de chacun des joueurs et renvoie 0 s'il y a égalité, 1 si le joueur 1 est gagnant et 2 si c'est le joueur 2 qui a gagné.

Partie 3 : Programmation d'une Intelligence artificielle

Cette partie aborde la programmation de l'Intelligence Artificielle (IA) si l'on désire jouer contre l'ordinateur. La structure du déroulement du jeu reste la même que précédemment, seul le choix de la case de jeu est différent. Il s'agit ici de programmer l'IA de manière à ce qu'elle choisisse la meilleure case pour elle. Nous allons pour cela utiliser un algorithme MinMax ou plus précisément sa version appelée Negamax.

3.1 - Arbre des configurations

On peut décrire l'ensemble des configurations possibles du plateau par un graphe orienté acyclique où chaque nœud correspond à un état du jeu. Chaque arête orientée correspond à un tour de jeu.

Cette partie sera traitée en Ocaml et on utilisera le type suivant pour représenter un jeu :

```
type jeu = {
   mutable scorej1 : int;
   mutable scorej2 : int;
   mutable tour : int;
   tableauj1 : int array;
   tableauj2 : int array;
}
```

On rappelle le rôle des fonctions utiles créées en C dans les parties précédentes, on supposera ici disposer de ces fonctions en Ocaml, langage avec lequel on demandera de traiter les questions de programmation de cette partie :

- `copie : jeu -> jeu` : réalise la copie profonde du jeu passé en argument ;
- `deplacer_graines: jeu -> int ->int` : réalise le déplacement des graines sur le plateau depuis la case choisie et renvoie la case (entier) où la dernière graine a été déposée ;
- `ramasser_graines : jeu -> int -> int` : réalise le ramassage des graines sur le plateau depuis la case où la dernière graine a été déposée et renvoie le nombre de graines récoltées (nombre de points gagnés) ;
- `tour_suivant : jeu -> bool` : teste si la configuration de jeu permet de continuer, renvoie `true` si c'est le cas et `false` sinon ;

- `test_case : jeu-> int -> bool` : teste si le joueur dont c'est le tour a le droit de jouer la case choisie, renvoie `true` si c'est le cas et `false` sinon ;
- `cases_possibles : jeu -> int list` : renvoie la liste des cases jouables par le joueur dont c'est le tour.
- `tour_joueur1 : jeu-> bool` renvoie `true` ssi c'est au tour du joueur 1 de jouer.
- `gagnant : jeu-> int` renvoie 1 si le jeu courant a gagné, 2 si c'est son adversaire 0 s'il y a égalité. **Attention, cette fonction n'a pas exactement les mêmes spécifications que celles de la partie précédente.**

Q31 Ecrire une fonction `gain : jeu-> int -> int*jeu` qui prend en entrée un état du jeu et un indice de case et renvoie un couple contenant la valeur du gain du joueur actif, c'est-à-dire le nombre de graines gagnées et un **nouveau** jeu contenant l'état du jeu après le coup. On considère que l'indice de la case passée en entrée est valide.

3.2 - Algorithme MinMax

Le jeu d'awalé se prête particulièrement bien à une variante de l'algorithme MinMax appelée Negamax car la façon d'évaluer la valeur de jeu est symétrique par rapport à 0 entre le joueur et l'adversaire. Ainsi, au lieu de différencier le cas "Joueur" qui maximise et "Adversaire" qui minimise, il suffit à chaque noeud p dont les enfants sont notés p_i de remarquer que :

$$\min(NegaMax(p_i)) = \max(-NegaMax(p_i))$$

Ainsi, tout le monde maximise sa propre valeur de NegaMax.

Pour l'awalé, la valeur de jeu est assez évidente : il s'agit de la différence de graines gagnées par chaque camp, positive si le joueur courant en ramasse plus et négative si c'est l'adversaire. Pour la calculer, on procéde de cette manière :

- si le noeud est une feuille, on connaît alors qui est le gagnant. On va donc tester qui est le gagnant et :
  - si c'est le joueur actif, on renvoie une valeur de jeu très grande (500) qui ne pourra être dépassée que par une autre configuration gagnante ;
  - si c'est l'adversaire, on renvoie une valeur de jeu très petite (-500) qui sera forcément dépassée par une autre configuration non perdante ;
  - si la partie est nulle on donne une valeur de 0 ;
- si l'on a atteint la profondeur maximale fixée, il n'y a pas de gain supplémentaire, donc cette valeur est nulle $NegaAwale(p) = 0$;
- sinon, pour chaque noeud enfant p_i , on calcule le gain g_i (nombre de graines ramassées) pour passer du noeud p au noeud p_i, puis on lui retranche la valeur de jeu calculée au noeud p_i. La valeur de jeu renvoyée par $NegaAwale(p)$ correspond alors au maximum des différences, soit $NegaAwale(p) = \max_i(g_i - NegaAwale(p_i))$.

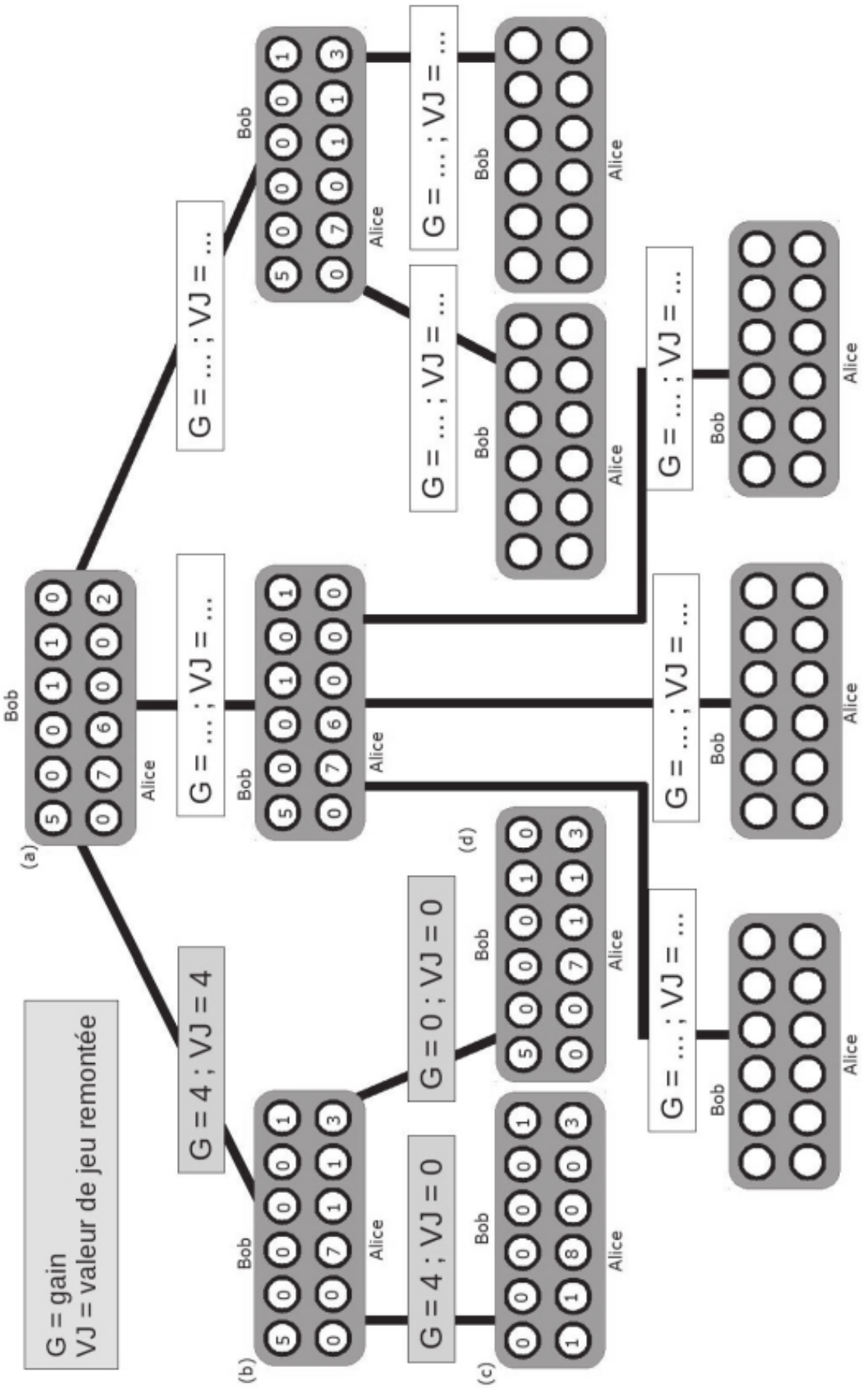
Pour la question suivante, on choisit une profondeur de 2. On donne un DAG indiquant la valeur du gain G obtenue en passant d'un noeud supérieur à un noeud inférieur, puis la valeur de jeu VJ du noeud inférieur, égale à $NegaAwale(p)$. On suppose que, pour le sommet (a), Bob vient tout juste de jouer.

À titre d'exemple, la branche de gauche, où Alice joue sa deuxième case, a été remplie :

- en appliquant les règles de jeu, on obtient facilement que le gain de (a) vers (b) vaut 4, le gain de (b) vers (c) vaut 4 et le gain de (b) vers (d) vaut 0 ;
- les valeurs de jeu de (c) et (d) sont nulles car on a atteint la profondeur de 2 ;
- enfin la valeur de jeu de (b) est égale au max(4 - 0; 0 - 0) = 4.

Q32 Compléter le reste de l'arbre. Donner la case à jouer avec cette profondeur de recherche pour optimiser le gain d'Alice.

Arbre des jeux possibles à compléter



Q33 Ecrire une fonction `max_val : (int*int) list-> int*int` qui prend en entrée une liste de couples (correspondant ici aux couples (indice de case, valeur de jeu)) et qui renvoie le couple qui maximise la valeur de jeu.

Q34 Ecrire une fonction `negaawale : jeu -> int -> int*int` telle que `negaawale j p_max` renvoie la valeur du jeu `j` passé en entrée estimée par l'algorithme Negamax en utilisant `p_max` comme profondeur maximale.

3.3 - Bibliothèque d'ouverture

Afin d'améliorer l'efficacité de l'IA, il est possible pour les premiers coups d'effectuer une recherche dans une base de données relationnelle. En effet, l'exploration en profondeur de l'ensemble des coups possibles est très coûteuse et on préfère s'appuyer sur l'historique des parties pour déterminer les configurations du plateau qui seront les plus avantageuses.

La base de données contient des informations sur chaque joueur, ainsi que les parties effectuées entre les joueurs.

Table Joueur

id_Joueur	nom	prenom	niveau	naissance
18571	Martin	Jean	2048	23/02/1958
18572	Dupond	Marie	2103	03/01/1972
18573	Develion	Théo	1857	05/10/2004

Tale Partie

id_Partie	id_joueur1	id_joueur2	resultat	jour	jeu
1	1547	1568	0.5	08/01/2001	'egai...'
2	1204	3	0	12/07/1998	'egaj...
3	4	2	1	15/07/2018	'egbi...'

La table Joueur contient les attributs suivants :

- id_Joueur : identifiant d'un joueur (entier, clé primaire);
- nom : nom du joueur (chaîne de caractères);
- prenom : prénom du joueur (chaîne de caractères);
- niveau : niveau maximal atteint par le joueur au cours de sa carrière (entier);
- naissance : date de naissance du joueur (date).

La table Partie contient les attributs suivants :

- id_Partie : identifiant de la partie (entier, clé primaire);
- id_joueur1 : identifiant du joueur débutant la partie (entier);
- id_joueur2 : identifiant du second joueur (entier);
- resultat : 1 est une victoire du joueur1, 0.5 une égalité et 0 une victoire du joueur2 (flottant);
- jour : date du jour de la partie (date);
- jeu : liste des coups successifs de la partie, stockée sous forme d'une chaîne de caractères. 'egai...' signifie que le joueur1 a joué la 5^e case (d'indice 4) représentée par la lettre 'e', puis le joueur2 a joué la 7^e case (d'indice 6) représentée par la lettre 'g' et ainsi de suite.

Q35 Écrire une requête SQL permettant d'extraire les identifiants des joueurs ayant un niveau strictement supérieur au score 1900.

Q36 Écrire une requête SQL permettant d'afficher le nom et le prénom des 3 joueurs ayant le niveau le plus élevé.

Q37 Écrire une requête SQL permettant de déterminer les joueurs ayant plus de cent victoires lorsqu'ils commencent la partie. La requête doit renvoyer le nom, le prénom et le nombre de victoires de ces joueurs classés par ordre décroissant du nombre de victoires.