

## LICENCE SCIENCES & TECHNOLOGIES, 1<sup>re</sup> ANNÉE

UE INF201

ALGORITHMIQUE ET PROGRAMMATION FONCTIONNELLE

2021-2022

---

### PROJET — LES DAMES CHINOISES

---

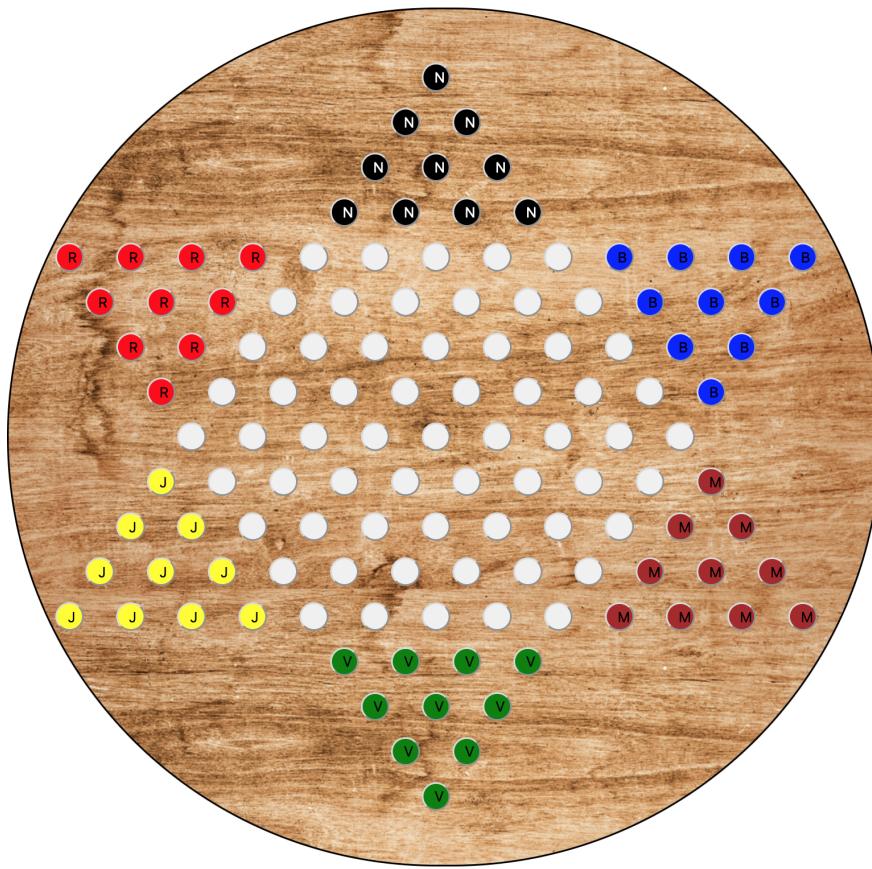


FIG. 1 : Plateau de jeux dans sa configuration initiale pour une partie à six joueurs.

**Résumé** Les **dames chinoises** est, comme son nom ne l'indique pas, un jeu dérivé du jeu Halma inventé par l'Américain George Howard Monks en 1883.

La version classique se joue avec **10 pions** par joueur que l'on déplace sur un plateau comportant **121 emplacements disposés** en étoile à 6 branches. Voir figure 1. Le jeu peut se jouer à **1, 2, 3 ou 6 joueurs**. Dans ce projet nous considérons une variante du jeu avec la règle des sauts distants et symétriques.

**Consignes** Toutes les informations pour le projet se trouvent sur [Caseine](#). Vous y trouverez ce document, une archive `projet.zip` contenant les fichiers du projet, une activité de rendu et un lien vers une page démontrant le projet. Dans ce projet vous modéliserez le jeu de dame chinoise, et écrirez différentes fonctions permettant de vérifier qu'une partie se déroule selon les règles jusqu'à ce que l'un des joueurs gagne. Le projet peut se faire entièrement avec un interpréteur OCaml comme en TP. Cependant nous vous proposons une interface graphique qui une fois complétée de vos fonctions réalisera l'application de jeu sur cette page :

<http://www-verimag.imag.fr/~bassetni/display.html>. Les directives pour créer cette application seront données à part sur Caseine.

**Planning.** Ce projet se fait en groupe de quatre (trois : par dérogation) et se décompose en deux phases. Vous devez rendre selon les modalités fixées par vos enseignants de TD/TP :

- 1) au plus tard le **11 mars à 23h59** : Q1 à Q9 ;
- 2) au plus tard le **29 avril à 23h59** : les autres questions.

Lors de la dernière séance de TP, vous devrez effectuer une soutenance de votre projet avec démonstration sur machine devant votre enseignant, éventuellement à distance.

Les détails des modalités de la soutenance et des rendus seront précisés par vos enseignants.

**Les définitions qui vous sont données** Une version embryonnaire de votre projet vous est donnée dans le fichier `rendu_etd.ml`. Ce fichier comprend les définitions des types, quelques squelettes de fonctions à compléter et une fonction d'affichage qui ne fonctionnera pleinement qu'avec l'avancement du projet. Attention vous devez toujours vous conformer aux types de l'énoncé.

## Table des matières

<b>1 Première partie</b>	<b>3</b>
1.1 Le plateau et les coordonées des cases . . . . .	3
1.2 Préparation des coups simples : déplacements unitaires et sauts simples . . . . .	6
<b>2 Deuxième partie</b>	<b>8</b>
2.1 Configuration initiale et rotation du plateau . . . . .	8
2.2 Recherche et suppression de case dans une configuration . . . . .	10
2.3 Jouer un coup unitaire . . . . .	10
2.4 Jouer un coup . . . . .	11
<b>3 Vérifier une partie</b>	<b>12</b>
<b>4 Bonus</b>	<b>12</b>
4.1 Calcul des coups . . . . .	12
4.2 Tous les coups sont permis . . . . .	12

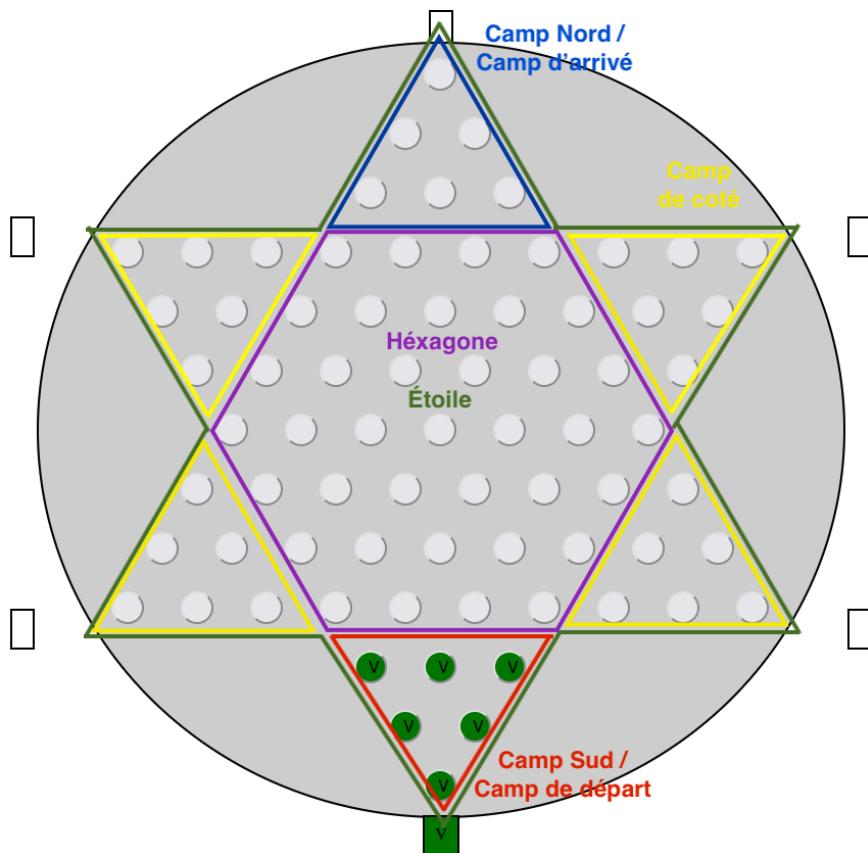


FIG. 2 : Description des zones du plateau, le plateau forme une étoile constitué d'un hexagone central avec un triangle équilatéral sur chaque coté formant les camps de chaque joueur.

## 1 Première partie

### 1.1 Le plateau et les coordonées des cases

Le nombre de joueurs est un diviseur de 6, c'est à dire 6, 3, 2 et même 1 si on veut jouer en solitaire. Les cases du plateaux sont disposées dans une étoile à six branches le long de lignes horizontales, et de lignes obliques orientées par rapport à l'horizontale d'un tiers de tours dans le sens horaire et anti-horaire. Chaque case est ainsi à l'intersection d'une ligne horizontale et de deux lignes obliques. Dans ce projet de programmation, nous pourrons jouer sur des plateaux de différentes tailles. La dimension d'un plateau, noté `dim` par la suite, est un paramètre qui encode la taille du plateau. **Le plateau a  $4 * \text{dim} + 1$  lignes horizontales que nous numérotions de bas en haut de  $-2 * \text{dim}$  à  $2 * \text{dim}$**  et similairement pour les lignes obliques.

Nous définissons donc le type `dimension` par type `dimension=int`.

Au début de la partie, les pions d'un joueur se situent dans son camp qui est le triangle devant lui qui comprend les `dim` lignes de pions comportant en allant de l'extérieur vers le centre 1, 2, ... et `dim` pions.

Nous avons déjà vu une configuration initiale à six joueurs et dimension 4 en Figure 1. En Figure 1 il y a une configuration initiale pour le jeu à un joueur et dimension 3. Des configurations initiales pour des jeu à deux et trois joueurs sont représentées Figure 3.

On dira que le joueur le plus au Sud est le protagoniste, son but est de déplacer ses pions du camp le plus au Sud vers le camp, en face, des cases les plus au Nord. Les joueurs jouant à tour de rôle, on tournera le plateau de telle sorte que le joueur qui doit jouer soit le protagoniste, son camp

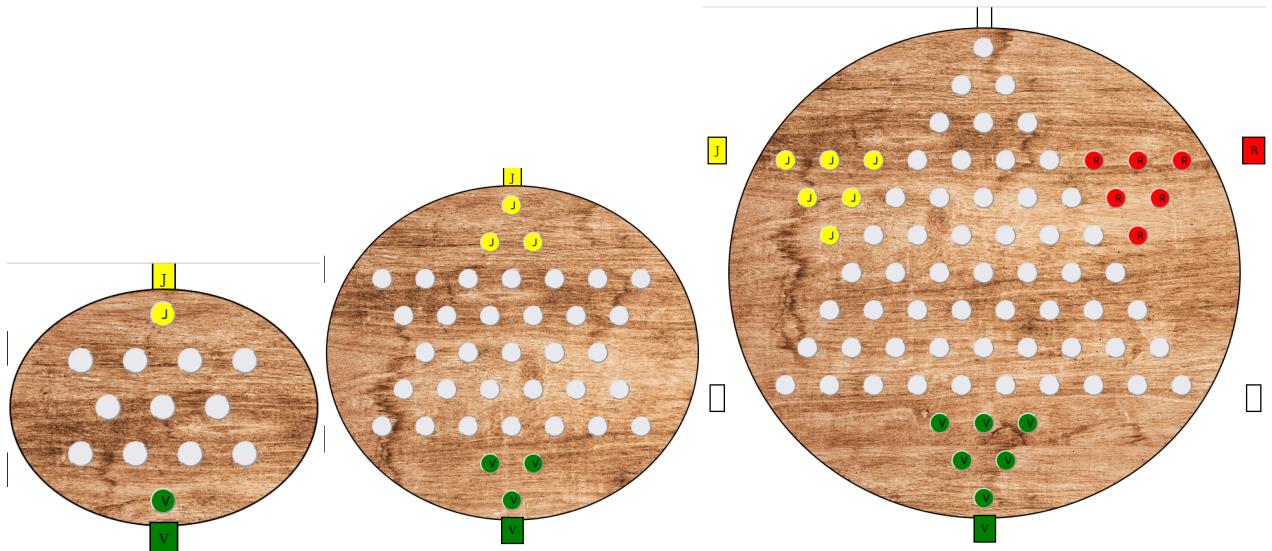


FIG. 3 : Différentes configurations initiales. Gauche : 2 joueurs, dim=1 ; Milieu : 2 joueurs, dim=2 ; Droite : 3 joueurs, dim=3

étant donc au Sud avec pour but le camp Nord.

Une case est définie par trois coordonnées  $(i, j, k)$ , la case au centre du plateau de jeu a pour coordonnées  $(0, 0, 0)$ . La figure 4 illustre ce système de coordonnées. Les coordonnées représentent :

- le numéro de la ligne horizontale
- le numéro de la ligne horizontale lorsqu'on a tourné le plateau d'un tiers de tour dans le sens antihoraire (dans la Figure 1, les rouges prennent la place des verts au sud).
- le numéro de la ligne horizontale lorsqu'on a tourné le plateau d'un tiers de tour dans le sens horaire (dans la Figure 1, les bleus prennent la place des verts au sud).

Il est à noter que l'équation  $i + j + k = 0$  est satisfaite pour toutes les cases de la grille et nous définissons donc le type `case` par :

```
type case = int * int * int;; (*restreint au triplet tels (i, j, k) tels que i+j+k=0*)
```

Passons en revue les autres types définis dans le fichier de rendu. Le type couleur représente les couleurs des joueurs :

```
type couleur = Vert | Jaune | Rouge | Noir | Bleu | Marron
| Code of string (*une chaine restreinte a 3 caracteres*)
| Libre;;
```

Le constructeur `Code` vous permet d'entrer vos propres noms de joueur par exemple si vous voulez que "Ken" affronte "Ryu". La couleur Libre est une couleur en plus pour coder l'absence de joueur (dans une case ou pour le gagnant d'une partie).

Le fait qu'un pion d'une couleur `col` se situe sur une case `c` est codé par un couple  $(c, col)$  que l'on appelle une case colorée. Nous utiliserons donc le type `case_coloree = case * couleur;;`

Une configuration du jeu est donnée par un triplet formé d'une liste de cases colorées, une liste de joueurs et une dimension (un entier).

```
type configuration = case_coloree list * couleur list * dimension;;
```

La liste de case colorée donne l'emplacement des pions et leurs couleurs. On veillera à ce que pour chaque case `c` il y ait au plus un pion sur cette case, c'est-à-dire il y a au plus une couleur `col` tel que le couple  $(c, col)$  est dans la liste ; l'absence de pion sur la case `c` sera codé par l'absence

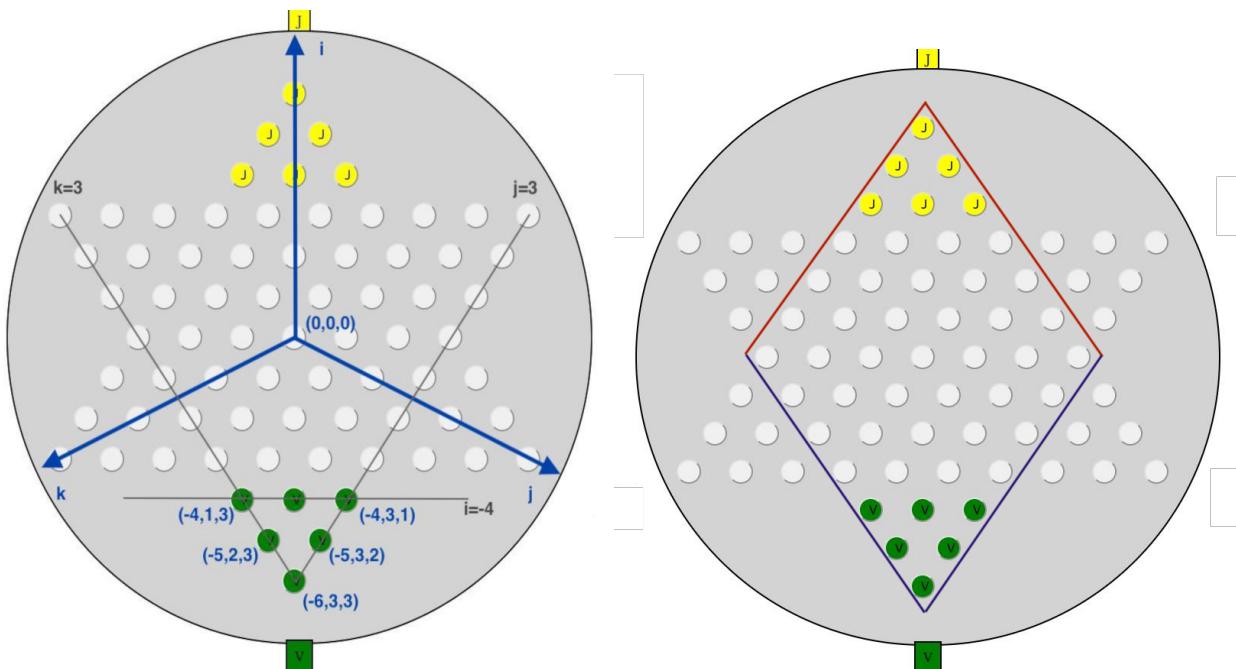


FIG. 4 : Figure de gauche : Système de coordonnées. Les axes sont marqués en bleu foncé, les coordonnées de l'origine, et des pions verts sont indiquées également en bleu. Les lignes gris-clair représentent les droites  $i = -4$ ,  $j = 3$  et  $k = 3$ . Figure de droite : le Losange Nord-Sud, ce losange contient toutes les cases dans lesquelles un coup peut se terminer.

de couple  $(c, \text{col})$  dans la liste et non pas avec  $(c, \text{Libre})$ . La liste de joueur permet de savoir à qui est le tour (tête de liste) et quel sera le tour des suivants (en suivant l'ordre de la liste). Enfin même si elle ne change pas au cours de la partie la dimension est donnée dans la configuration car nous devons pouvoir accéder facilement à celle-ci et pouvoir en changer si nous souhaitons faire une partie sur un plateau de taille différente.

Les coups seront décrits plus tard il en existe de deux sortes des déplacement unitaire (constructeur `Du`) et des sauts multiples (constructeur `Sm`) :

```
type coup = Du of case * case | Sm of case list;;
```

Il sera utile de tester si une case est bien dans une certaine partie du plateau, ou même tout simplement dans l'étoile. Le but des questions suivantes est de vous familiariser avec les coordonnées des cases et d'écrire les premières fonctions de tests.

**Q1.** Pour les conditions ci-après, décrire les cases qui satisfont ces conditions :

1.  $i < -\dim$
2.  $i > \dim$
3.  $j < -\dim$
4.  $(i, j, k) = (2\dim, -\dim, -\dim)$
5.  $(i, j, k) = (-\dim - 1, 1, \dim)$
6.  $i \geq -\dim \wedge j \geq -\dim \wedge k \geq -\dim$

**Q2.** Donner une formule booléenne qui est vraie si et seulement si une case est dans le losange Nord-Sud (voir figure (4)).

En déduire une implémentation de la fonction `est_dans_losange`: `case -> dimension ->bool` telle que (`est_dans_losange c dim`) est vraie ssi `c` est une case du losange Nord-Sud du plateau de dimension `dim`.

**Q3.** Donner une formule booléenne qui est vraie si et seulement si une case est dans l'étoile. Remarquez que plusieurs définitions de l'étoile sont possibles : union de trois grands losanges, union de deux triangles, union de l'hexagone centrale et des camps de chaque joueur. Vous choisirez et expliquerez une définition correcte.

En déduire une implémentation de la fonction `est_dans_etoile`: `case->dimension->bool` telle que (`est_dans_etoile c dim`) est vraie ssi `c` est une case de l'étoile du plateau de dimension `dim`.

Maintenant la fonction d'affichage doit afficher correctement les configurations donnée dans le fichier de rendu (il faut réinterpréter toutes les fonctions entre `est_dans_etoile` et la fonction `affiche`). Vous essaierez aussi vos propres exemples.

**Q4.** Implémenter une fonction `tourner_case`: `int -> case -> case` telle que `tourner_case m c` est la case `c` après avoir fait tourner le plateau de `m` sixième de tour dans le sens antihoraire (dans la Figure 1, une case jaune se retrouve dans le camp vert après un sixième de tour.)

Nous définissons le type `vecteur` comme synonyme du type `case` car les triplets d'entiers  $(i, j, k)$  tels que  $i + j + k = 0$  servent aussi bien comme `case` de la grille que comme `vecteur` permettant des translations.

**Q5.** Implémenter la fonction `translate` : `case -> vecteur -> case` telle que (`translate c v`) est la case obtenue par translation de vecteur `v` à partir de `c`, c'est à dire  $(translate c v) = (c_1 + v_1, c_2 + v_2, c_3 + v_3)$  en notant  $(c_1, c_2, c_3) = c$  et  $(v_1, v_2, v_3) = v$ .

Ainsi dans l'exemple de la Figure 4 le pion vert le plus à gauche est en  $(-4, 1, 3)$ . Si on lui ajoute le vecteur  $(0, 2, -2)$ , on arrive dans la case  $(-4, 3, 1)$  c'est à dire la case du pion vert le plus à droite. Si on ajoute encore le vecteur  $(1, 0, -1)$ , on arrive dans la case libre  $(-3, 3, 0)$ .

**Q6.** Implémenter la fonction `diff_case` : `case -> case -> vecteur` telles que `diff_case c1 c2` calcule la différence de chacune des coordonnées pour donner un vecteur de translation. Par exemple, dans la figure 5, la différence entre la case du pion jaune et celle du pion vert est  $diff_case (0, 0, 0) (0, -2, 2) = (0, 2, -2)$ . Cela correspond au vecteur à appliquer au pion vert pour le translater dans la case du pion jaune.

## 1.2 Préparation des coups simples : déplacements unitaires et sauts simples

On dit de deux cases qu'elles sont voisines si pour aller de l'une à l'autre on change deux coordonnées de 1. Ainsi les cases voisines de  $(1, 0, -1)$  sont  $(1, 1, -2), (2, 0, -2), (2, -1, -1), (0, 0, 0), (0, 1, -1)$ .

**Q7.** Implémenter une fonction `sont_cases_voisines`: `case ->case -> bool` qui teste si deux cases sont voisines.

Les triangles situés en périphérie du jeu (autrement dit en dehors de l'hexagone central) sont les camps des joueurs. Un joueur a au départ tous ses pions dans son camp devant lui et a pour but de les déplacer dans le camp diamétriquement opposé. Les autres camps seront appelés camps de côté.

Les joueurs jouent à tour de rôle dans le sens des aiguilles d'une montre. À son tour chaque joueur déplace un pion de son choix sur le plateau en respectant les règles suivantes :

- le pion joué doit être de la couleur du joueur à qui c'est le tour
- le pion une fois joué doit être sur une case préalablement libre du plateau, qui n'est pas dans un camp de côté. programme voisins
- le déplacement est d'un des deux types suivants :
  - déplacement unitaire : la case d'arrivée est une des six cases voisines de la case de départ
  - saut multiple : la case d'arrivée est obtenue après un ou plusieurs sauts simples au-dessus de pions pivots.

**Sauts simples et pion pivot.** Si un pion est dans une case  $c_1$  et fait un saut jusqu'à la case  $c_2$ , il faut que :

- la case  $c_1$  contienne un pion de la couleur du joueur;
- la case  $c_2$  soit une case libre valide du plateau (dans l'étoile); Voir l'équation de l'étoile pa
- il existe une case exactement à mi-distance entre  $c_1$  et  $c_2$  qui contient un pion de n'importe quelle couleur. Cette case est appelée le pivot  $p$ ;
- toutes les cases (il peut ne pas y en avoir) entre  $c_1$  et  $p$  et entre  $p$  et  $c_2$  sont vides.

Par exemple dans la figure 5 le saut du pion Vert vers la case sélectionnée à gauche utilisant le pion jaune comme pivot est valide. Le pion qui sert de pivot n'est pas affecté par le saut (on ne retire pas de pion comme aux Dames classiques). Les deux pions, celui qui se déplace et le pivot, peuvent être de la même couleur.

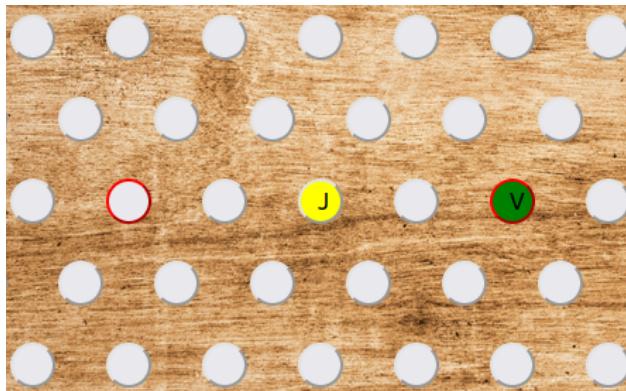


FIG. 5 : Calcul du pivot, le pion Jaune a pour coordonnées  $(0, 0, 0)$ , le pion Vert a pour coordonnées  $(0, 2, -2)$  et la case sélectionnée à gauche a pour coordonnées  $(-1, 0, 0)$

Le type `case option = None | Some of case` est un type prédéfini en Ocaml que nous utilisons dans la question suivante.

- Q8.** Implémenter la fonction `calcul_pivot : case -> case -> case option` qui étant données deux cases alignées (elles ont une de leurs coordonnées égale) retourne `Some c` où  $c$  est la case à mi-chemin entre les deux si elle existe. Si une telle case n'existe pas (si le nombre de cases entre les deux cases est paire ou les cases ne sont pas alignées) retourne `None`. Dans la figure 5 si on applique cette fonction à la case sélectionnée à gauche et au pion Vert, la fonction retourne la case du pion Jaune.

**Q9.** Implémenter la fonction `vec_et_dist` : `case -> case -> vecteur*int` telle que `vec_et_dist c1 c2` retourne  $(v, d)$  où  $v$  est le vecteur de déplacement unitaire et  $d$  la distance entre  $c1$  et  $c2$ . C'est à dire que  $v$  correspond au vecteur de translation d'un déplacement unitaire (les coordonnées valent 1, -1 et 0 dans n'importe quel ordre) et que si on applique  $d$  fois cette translation en  $c1$  on arrive dans la case  $c2$ . Dans la figure 5 si on applique cette fonction au case Verte et Jaune on obtient :  $\text{vec\_et\_dist } (0, 2, -2) (0, 0, 0) = ((0, 1, -1), 2)$ .

## 2 Deuxième partie

### 2.1 Configuration initiale et rotation du plateau

Dans cette partie, nous verrons comment implémenter la configuration initiale, les déplacements unitaires et les changements de tours. Nous verrons dans la partie suivante comment traiter les sauts multiples et le déroulement d'une partie.

D'abord nous implémentons quelques fonctions génériques sur les listes.

**Q10.** Donner une réalisation récursive des fonctions suivantes :

#### SPÉCIFICATION 1 — Permutation circulaire d'une liste

**PROFIL**  $\text{tourner\_liste} : 'a list \rightarrow 'a list$

**SÉMANTIQUE** ( $\text{tourner\_liste} l$ ) est la liste obtenue à partir de  $l$  en déplaçant le premier élément à la fin de la liste.

**EX. ET PROP.** (i)  $(\text{tourner\_liste}[e_1; e_2; \dots; e_n]) = [e_2; \dots; e_n; e_1]$

(ii)  $(\text{tourner\_liste}[Vert; Jaune; Rouge]) = [Jaune; Rouge; Vert]$

#### SPÉCIFICATION 2 — Dernier élément d'une liste

**PROFIL**  $\text{der\_liste} : 'a list \rightarrow 'a$

**SÉMANTIQUE** ( $\text{der\_liste} l$ ) est le dernier élément de la liste  $l$

**EX. ET PROP.** (i)  $(\text{der\_liste}[e_1; e_2; \dots; e_n]) = e_n$

(ii)  $(\text{tourner\_liste}[Vert; Jaune; Rouge]) = Rouge$

**Q11.** Donner une réalisation récursive de la fonction `remplir_segment` dont la spécification est :

#### SPÉCIFICATION 3 — Remplissage d'un segment

**PROFIL**  $\text{remplir\_segment} : int \rightarrow case \rightarrow case list$

**SÉMANTIQUE** ( $\text{remplir\_segment} m (i, j, k)$ ) est la liste de cases du segment horizontal dont la case la plus à gauche est  $(i, j, k)$  et comprenant  $m$  cases c'est à dire  $(\text{remplir\_segment} m (i, j, k)) = [(i, j, k), (i, j + 1, k - 1), \dots, (i, j + m - 1, k - m + 1)]$

**EX. ET PROP.** (i)  $(\text{remplir\_segment} 1 (0, 0, 0)) = [(0, 0, 0)]$

(ii)  $(\text{remplir\_segment} 3 (-4, 1, 3)) = [(-4, 1, 3); (-4, 2, 2); (-4, 3, 1)]$

On donnera d'abord des équations récursives définissant cette fonction formalisant l'idée que pour remplir un segment il suffit de remplir la première case du segment et le segment à partir de la case suivante.

**Q12.** Donner une réalisation récursive de la fonction `remplir_triangle_bas` dont la spécification est :

**SPÉCIFICATION 4 — Remplissage d'un triangle pointe en bas**

**PROFIL**  $remplir\_triangle\_bas : int \rightarrow case \rightarrow case list$

**SÉMANTIQUE**  $remplir\_triangle\_basm (i, j, k)$  est la liste de cases du triangle dont les sommets sont  $(i, j, k)$ ,  $(i, j + m - 1, k - m + 1)$  et  $(i - m + 1, j + m - 1, k)$ .

**EX. ET PROP.** (i)  $remplir\_triangle\_bas 1 (0, 0, 0) = [(0, 0, 0)]$

(ii)  $remplir\_triangle\_bas 3 (-3, 4, -1) =$

$\{(-3, 4, -1); (-3, 5, -2); (-3, 6, -3); (-2, 4, -2); (-2, 5, -3); (-1, 4, -3)\}$

On donnera d'abord des équations récursives définissant cette fonction formalisant l'idée qu'un triangle est composé d'un segment du haut et du triangle sous ce segment.

**Q13.** Donner une réalisation récursive de la fonction `remplir_triangle_haut` dont la spécification est :

**SPÉCIFICATION 5 — Remplissage d'un triangle pointe en haut**

**PROFIL**  $remplir\_triangle\_haut : int \rightarrow case \rightarrow case list$

**SÉMANTIQUE**  $remplir\_triangle\_hautm (i, j, k)$  est la liste de cases du triangle dont les sommets sont  $(i, j, k)$ ,  $(i, j + m - 1, k - m + 1)$  et  $(i + m - 1, j, k - m + 1)$ .

**EX. ET PROP.** (i)  $remplir\_triangle\_haut 1 (0, 0, 0) = [(0, 0, 0)]$

(ii)  $remplir\_triangle\_haut 3 (-4, 1, 3) =$

$\{(-4, 1, 3); (-4, 2, 2); (-4, 3, 1); (-5, 2, 3); (-5, 3, 2); (-6, 3, 3)\}$

On donnera d'abord des équations récursives définissant cette fonction formalisant l'idée qu'un triangle est composé d'un segment du bas et du triangle au-dessus de ce segment.

**Q14.** Implémenter la fonction :

**SPÉCIFICATION 6 — Coloriage d'une liste de cases avec une couleur**

**PROFIL**  $colorie : couleur \rightarrow case list \rightarrow case\_coloree list$

**SÉMANTIQUE**  $colorie coul lc$  est la liste formée à partir des cases de  $lc$  en leur ajoutant la couleur  $coul$ .

**Q15.** Implémenter une fonction `tourner_config`: configuration  $\rightarrow$  configuration telle que `tourner_config conf` est la configuration `conf` après passage au joueur suivant. On tourne donc la liste des couleurs ainsi que les cases en utilisant la fonction `tourner_case` avec les bons paramètres. Remarquer que le nombre de sixième de tours à tourner est  $6/N$  où  $N$  est le nombre de joueurs. Dans la Figure 1, le camp jaune se retrouve à la place du camp vert après l'application de `tourner_config`.

**Q16.** Implémenter la fonction `remplir_init` : liste\_joueur  $\rightarrow$  dimension  $\rightarrow$  configuration qui à partir d'une liste de joueurs et la dimension du plateau construit la configuration initiale.

Tester votre fonction en utilisant la fonction d'affichage `affiche`.

## 2.2 Recherche et suppression de case dans une configuration

**Q17.**

On donne dans le fichier de rendu, la fonctions suivante :

SPÉCIFICATION 7 — Associe avec valeur par défaut	
PROFIL	<i>associe</i> : ' $a \rightarrow' a *' b list \rightarrow' b \rightarrow' b$
SÉMANTIQUE	( <i>associe a l defaut</i> ) vaut <i>b</i> si ( <i>a, b</i> ) est dans la liste et <i>defaut</i> sinon. Si plusieurs couple avec <i>a</i> existe le premier est utilisé.
EX. ET PROP.	<ul style="list-style-type: none"> <li>(i) (<i>associe (0,0,0) [((-1,0,1),Jaune);((0,0,0),Vert)] Libre</i>) = <i>Vert</i></li> <li>(ii) (<i>associe (1,-1,0) [((-1,0,1),Jaune);((0,0,0),Vert)] Libre</i>) = <i>Libre</i></li> </ul>

Utiliser *associe* pour implémenter une fonction *quelle\_couleur*: *case*  $\rightarrow$  *configuration*  $\rightarrow$  *couleur* qui étant donnée une case et une configuration retourne *Libre* si la case est libre, sinon la couleur du pion qui occupe cette case.

**Q18.**

Implémenter la fonction *supprime\_dans\_config*: *configuration*  $\rightarrow$  *case*  $\rightarrow$  *configuration* tel que *supprime\_dans\_config conf c* est la configuration *conf* dans laquelle on a supprimé, dans la liste de cases colorées, la case colorée correspondant à la case *c*.

## 2.3 Jouer un coup unitaire

On fera les trois questions suivantes **uniquement pour les déplacement unitaires**. Le cas des saut multiples sera traité plus tard et pour l'instant on utilisera *failwith "Saut multiples non implementes"* à la place du code manquant.

**Q19.**

Implémenter la fonction *est\_coup\_valide*: *configuration*  $\rightarrow$  *coup*  $\rightarrow$  *bool* tel que *est\_coup\_valide conf (Du(c1,c2))* est vraissi le coup (*Du(c1,c2)*) est valide dans la configuration *conf*. C'est à dire que les cases *c1* et *c2* sont voisines que *c1* contient un pion du joueur auquelle c'est le tour de jouer, que *c2* est libre et que *c2* est dans le losange.

**Q20.**

Implémenter la fonction *appliquer\_coup*: *configuration*  $\rightarrow$  *coup*  $\rightarrow$  *configuration* qui applique le coup (ici, un déplacement unitaire) en supposant qu'il soit valide.

**Q21.**

Modifier la fonction *mettre\_a\_jour\_configuration*: *configuration*  $\rightarrow$  *coup*  $\rightarrow$  *configuration* pour quelle mette à jour le plateau pour les coups unitaires. C'est à dire que l'appel à *mettre\_a\_jour\_configuration conf cp* fait :

1. la vérification que *cp* est un coup correct;
2. effectue le coup, c'est à dire déplace le pion de la case de départ vers la case d'arrivée.

La fonction retourne la nouvelle configuration mais si une de ces étapes échoue, il est demandé d'utiliser un *failwith "explication"* où "explication" est une chaîne de caractère expliquant le problème (ex. "Ce coup n'est pas valide, le joueur doit rejouer").

## 2.4 Jouer un coup

Les sauts peuvent être enchaînés comme dans la figure 6 où le pion sélectionné enchaîne trois sauts en un seul coup. La case d'arrivée du dernier saut doit être dans le losange (i.e. il ne peut pas finir dans un des camps des joueurs sur les cotés).

Les coups qui représentent un saut multiple sont de la forme  $S_m ([c_1; c_2; \dots; c_n])$  ( $n > 1$ ) avec  $c_1$  la case de départ,  $c_n$  la case d'arrivée, chaque élément consécutif  $c_i; c_{i+1}$  de la liste correspond à un saut de  $c_i$  à  $c_{i+1}$ .

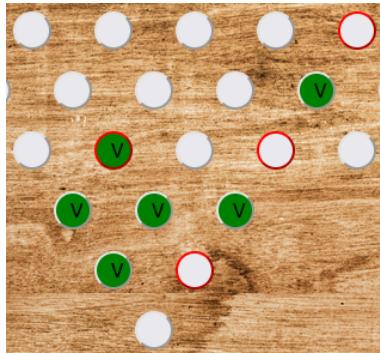


FIG. 6 : Exemple de saut multiple

- Q22.** Implémenter la fonction `est_libre_seg : case ->case ->configuration -> bool` telle que `est_libre_seg c1 c2 conf` retourne vrai si toutes les cases entre  $c_1$  et  $c_2$  sont libre,  $c_1$  et  $c_2$  sont supposées alignées. Vous pourrez utiliser une fonction auxiliaire récursive et la fonction `vec_et_dist`.
- Q23.** Implémenter la fonction `est_saut : case -> case -> configuration -> bool` qui retourne vrai si le déplacement de la première case à la seconde est un saut valide.
- Q24.** Implémenter la fonction `est_saut_multiple : case list ->configuration -> bool` qui retourne vrai si le déplacement de la première case à la dernière est un saut multiple valide passant par toutes les cases intermédiaires. On ne demande pas ici de tester si la première case est dans la configuration et si la dernière est dans le losange Nord-Sud.
- Q25.** Reprendre les trois questions de la section précédentes en y incorporant les saut multiples. C'est à dire `est_coup_valide`, `appliquer_coup` et `mettre_a_jour_configuration` doivent fonctionner pour toutes sortes de coups : déplacements unitaires et sauts multiples.

Testez bien votre code, il existe beaucoup de cas particuliers à gérer. Pensez par exemple à vérifier que :

- vous ne mélangez pas des déplacement unitaires et des sauts;
- les sauts multiples peuvent faire une étape sur une case des camps sur les cotés mais la case finale doit être dans le losange;
- le pion qui se déplace lors d'un saut multiple est provisoirement retiré du plateau : il ne peut pas se servir de lui même comme pivot et il n'est pas non plus bloqué par lui même;
- ...

### 3 Vérifier une partie

Dans cette partie il est demandé d'utiliser les opérateurs sur les listes vu en cours : `List.exists`, `List.forall`, `List.map`, ... (Il n'est pas obligatoire de tous les utiliser). L'utilisation de `let rec` est interdite (à part si les opérateurs ci-dessus n'ont pas encore été vus en cours).

On dit d'un pion du protagoniste sur une case  $(i, j, k)$  qu'il a un score  $i$ . Ceci permet de voir combien un pion est proche du but à atteindre. Le score du protagoniste est la somme des scores de ses pions.

**Q26.** Implémenter une fonction `score`: `configuration -> int` qui retourne le score du joueur protagoniste.

Implémenter une fonction `score_gagnant` : `dimension -> int` qui donne le score de la configuration gagnante pour le protagoniste, c'est à dire quand tous ses pions sont dans le camp Nord.

**Q27.** Implémenter une fonction `gagne`: `configuration -> bool` qui retourne `true` si cette configuration est gagnante pour le protagoniste.

**Q28.** Implémenter la fonction `est_partie`: `configuration -> coup list -> couleur` qui prend comme argument une configuration et une liste de coup et vérifie que cette liste de coup correspond à une partie valide et retourne la couleur du gagnant. Elle retourne `Libre` si aucun joueur n'a gagné. Attention la configuration doit être tournée à chaque fois qu'un joueur joue. Les coups sont donné toujours pour le joueur dont c'est le tour de jouer et qui se situe alors le plus au Sud. Par exemple si les deux premiers joueurs jouent le même coup la liste de coup commencera par deux fois le même coup.

## 4 Bonus

### 4.1 Calcul des coups

Dans cette partie plus délicate, nous voyons comment calculer les coups possibles d'un joueur et sélectionner celui qui maximise le score. Ceci permet d'aider les humains à jouer et peut servir pour implémenter un joueur automatique pour pouvoir «jouer contre l'ordinateur».

Les questions suivantes sont moins guidées, plusieurs solutions sont possibles, n'hésitez pas à découper les problèmes en fonctions plus petites.

**Q29.** Implémenter la fonction `coup_possibles`: `configuration -> case -> (case*coup) list` qui étant données une case  $c$  et une configuration, retourne des couples  $(c', cp)$  tels que  $c'$  est accessible depuis  $c$  en effectuant le coup  $cp$ . Tous les coups ne seront pas listés (il peut y en avoir une infinité ! pourquoi ?), par contre il faut que toutes les cases  $c'$  accessible en un coup soit présent exactement une fois dans la liste.

**Q30.** Implémenter la fonction `strategie_gloutonne`: `configuration -> coup` qui étant donné une configuration, retourne un coup valide pour le joueur dont c'est le tour qui augmente le plus son score.

### 4.2 Tous les coups sont permis

On change la règle des sauts pour qu'ils ressemblent plus à ceux des dames classiques, c'est-à-dire que le nombre de cases laissées libres avant et après le pivot peuvent être différents. Reprendre votre projet (en créant un nouveau fichier) avec ces nouveaux types de saut.