SPÉCIFICATION DU PROJET “Dames chinoises”

De Q1 à Q9 Edition

Table des matières

1. Types

1.1 dimension

SPÉCIFICATION – dimension	
PROFIL	$dimension \stackrel{def}{=} \mathbb{N}^*$
SÉMANTIQUE	*dimension* est une dimension d'un plateau, noté *dim* par la suite, est un paramètre qui encode la taille du plateau. Le plateau à $4\,dim+1$ lignes horizontales numérotons de bas en haut de $-2\,dim$ à $2\,dim$ et similairement pour les lignes obliques.

1.2 case

SPÉCIFICATION – case	
PROFIL	$case \stackrel{def}{=} \{(i,j,k) \in \mathbb{Z}^3 \; tel \; que \; i+j+k=0\}$
SÉMANTIQUE	*case* est définie par trois coordonnées (i,j,k), la case au centre du plateau de jeu a pour coordonnées $(0,0,0)$. Les coordonnées représentent: • ***i*** le numéro de la ligne horizontale ; • ***j*** le numéro de la ligne horizontale lorsqu'on a tourné le plateau d'un tiers de tour dans le sens anti-horaire ; • ***k*** le numéro de la ligne horizontale lorsqu'on a tourné le plateau d'un tiers de tour dans le sens horaire.

1.3 couleur

SPÉCIFICATION – couleur	
PROFIL	$couleur \stackrel{def}{=} \{Vert, Jaune, Rouge, Noir, Bleu, Maron\} \cup \{Libre\} \cup \{Code(nom)\ tel\ que\ nom \in string\ et\ (String.length\ nom) = 3\}$
SÉMANTIQUE	*couleur* c'est les couleurs des joueurs. Le constructeur *Code* permet d'entrer les noms de joueur restreint à trois caractères. La couleur *Libre* est une couleur en plus pour coder l'absence de joueur (dans une case ou pour le gagnant d'une partie).

1.4 case_coloree

SPÉCIFICATION – case colorée	
PROFIL	$case\_coloree \stackrel{def}{=} case \times couleur$
SÉMANTIQUE	*case_coloree* est un pion d'une couleur *col* se situe sur une case *c* est codé par un couple *(c, col)*.

1.5 configuration

SPÉCIFICATION – configuration du jeu	
PROFIL	$configuration \stackrel{def}{=} case\_coloree\ list \times couleur\ list \times dimension$
SÉMANTIQUE	*configuration* du jeu est donnée par un triplet formé d'une liste de cases colorées, une liste de joueurs et une dimension. La liste de cases colorées 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 de couple *(c, col)* dans la liste et non pas avec *(c, 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 *dim* 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.

1.6 coup

SPÉCIFICATION – coups unitaires et multiples	
PROFIL	$coup \stackrel{def}{=} \{Du(c1, c2) \, tel \, que \, c1, c2 \in case\} \cup \{Sm(cl) \, tel \, que \, cl \in case \, list\}$
SÉMANTIQUE	*coup* existe en de deux sortes: - *Du* pour les déplacements unitaires - *Sm* pour les sauts multiples

1.7 vecteur

SPÉCIFICATION – vecteur	
PROFIL	$vecteur \overset{def}{=} case$
SÉMANTIQUE	*vecteur* est le synonyme de *case* comme un vecteur permettant des translation avec les même propriètes.

2. Fonctions

2.1 indice_valide

SPÉCIFICATION – indice valide	
PROFIL	indice_valide : $\mathbb{N} \longrightarrow$ *dimension* $\longrightarrow \mathbb{B}$
SÉMANTIQUE	*(indice_valide x dim)* vérifie si la coordonnée *x* et valide dans la dimension *dim*.
EX. ET PROP.	*1) indice_valide(x, dim)* = vrai $\forall x \in \mathbb{Z}, \forall dim \in dimension,\ -2\,dim \leq x \leq 2\,dim$ *2) indice_valide(x, dim)* = faux $\forall x \in \mathbb{Z}, \forall dim \in dimension,\ -2\,dim < x \vee x > 2\,dim$

RÉALISATION – indice valide	
ALGORITHME	par la composition booléenne suivante $-2\,dim \leq x \leq 2\,dim$

IMPLIMENT.	`let indice_valide (x:int) (dim:dimension): bool =`
	`  -2 * dim <= x && x <= 2 * dim`
	`;;`

2.2 est_case

SPÉCIFICATION – est une case ?	
PROFIL	est_case : *case* $\longrightarrow \mathbb{B}$
SÉMANTIQUE	*(est_case c)* vérifie si *c* est une case.
EX. ET PROP.	*1) est_case((i, j, k))* = vrai, $\forall (i,j,k) \in \mathbb{Z}^3,\ i + j + k = 0$ *2) est_case((i, j, k))* = faux, $\forall (i,j,k) \in \mathbb{Z}^3,\ i + j + k \neq 0$

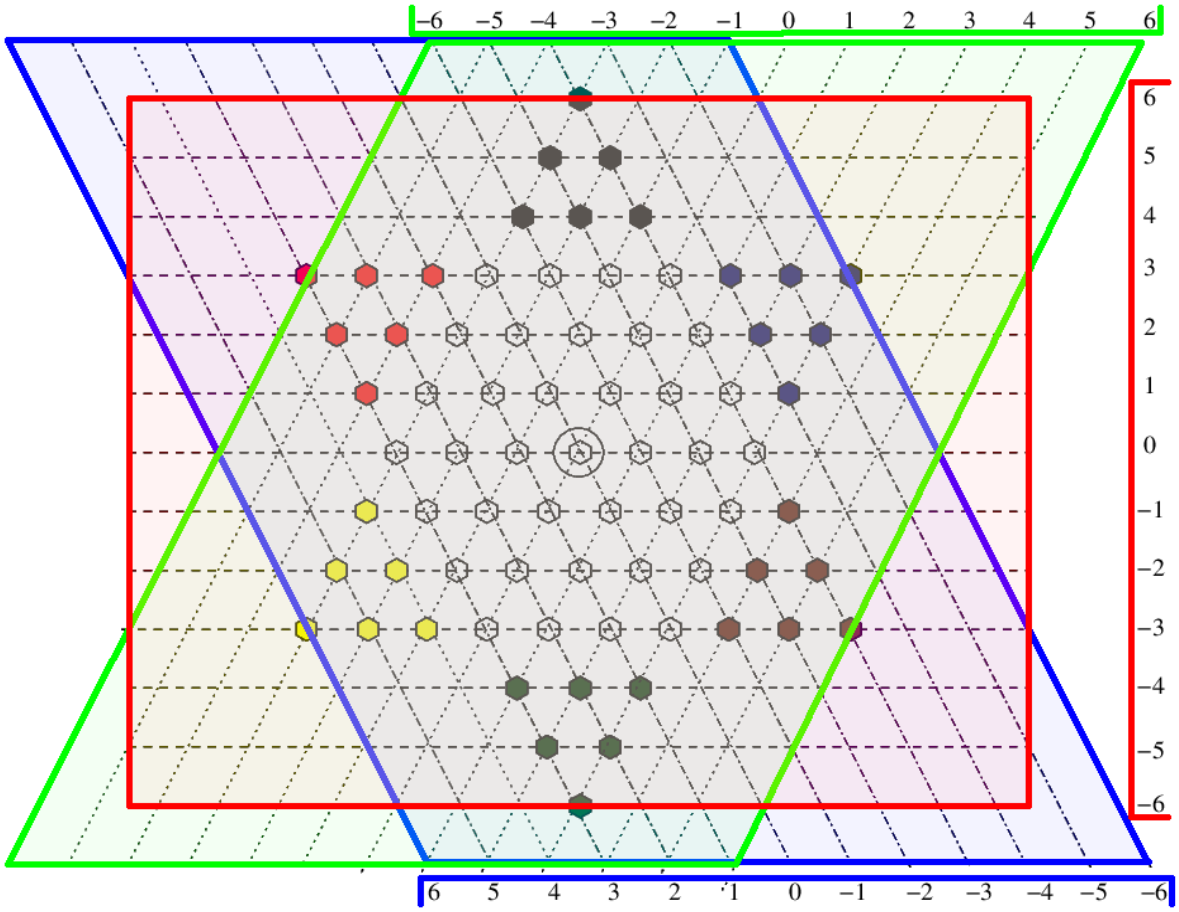

Figure 1. Représentation de la fonction *est_case* par l'intersection de trois figure ou chacune c'est un des coordonnées

RÉALISATION – est une case ?	
ALGORITHME	composition booléenne si la somme des coordonnées *i*, *j*, *k* doit sont égal à 0.

IMPLIMENT.	`let est_case (c:case): bool =`
	`  let i, j, k = c in i + j + k = 0`
	`;;`

2.3 est_dans_losange

SPÉCIFICATION – est dans losange North-South	
PROFIL	est_dans_losange : *case* ⟶ *dimension* ⟶ 𝔹
SÉMANTIQUE	*(est_dans_losange c* dim*)* vérifie si la case *c* est dans le losange North-South du plateau de dimension *dim*.
EX. ET PROP.	*1) est_dans_losange((6, -3, -3), 3)* = vrai *2) est_dans_losange((0, -3, 3), 3)* = vrai *3) est_dans_losange((0, 3, -3), 3)* = vrai *4) est_dans_losange((-6, 3, 3), 3)* = vrai *est_dans_losange((0,0,0), dim)* = vrai, ∀ *dim*∈*dimension*

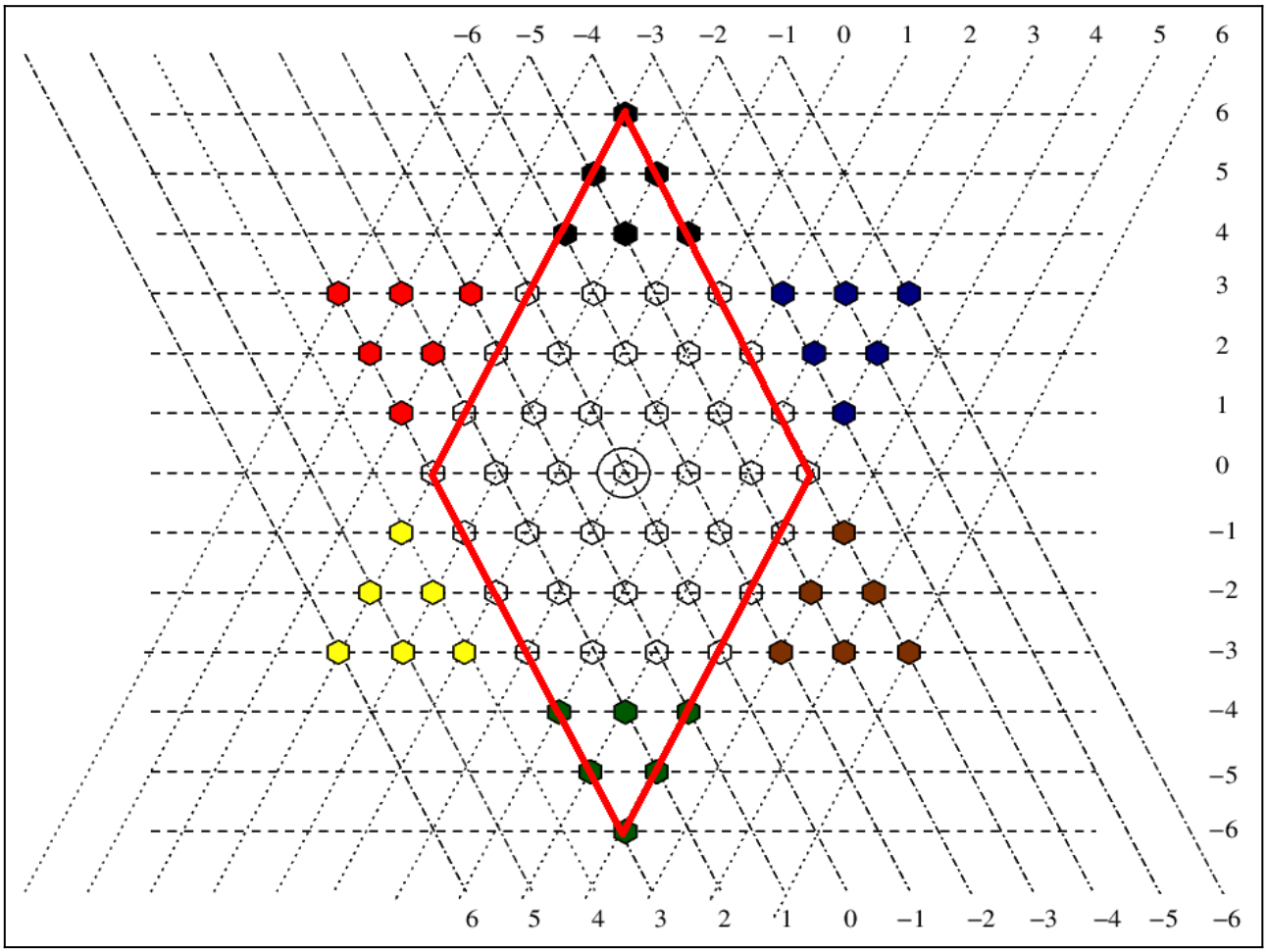

Figure 2. North-South losange

RÉALISATION – est dans losange North-South	
ALGORITHME	composition booléenne si la case est dans le losange North-South de la figure 2

IMPLIMENT.	`let est_dans_losange (c:case) (dim:dimension): bool =`
	`  let _, j, k = c in`
	`  -dim <= j && j <= dim &&`
	`  -dim <= k && k <= dim`
	`;;`

2.4 est_dans_losange_2

SPÉCIFICATION – est dans losange Northwest-Southeast	
PROFIL	est_dans_losange_2 : *case* ⟶ *dimension* ⟶ 𝔹
SÉMANTIQUE	*(est_dans_losange_2 c dim) v*érifie si la case *c* est dans le losange Northwest-Southeast du plateau de dimension *dim*.
EX. ET PROP.	*1) est_dans_losange_2((3, 3, -6), 3)* = vrai *2) est_dans_losange_2((3, -3, 0), 3)* = vrai *3) est_dans_losange_2((-3, -3, 6), 3)* = vrai *4) est_dans_losange_2((-3, 3, 0), 3)* = vrai *(est_dans_losange_2 (0,0,0), dim)* = vrai, ∀ *dim*∈*dimension*

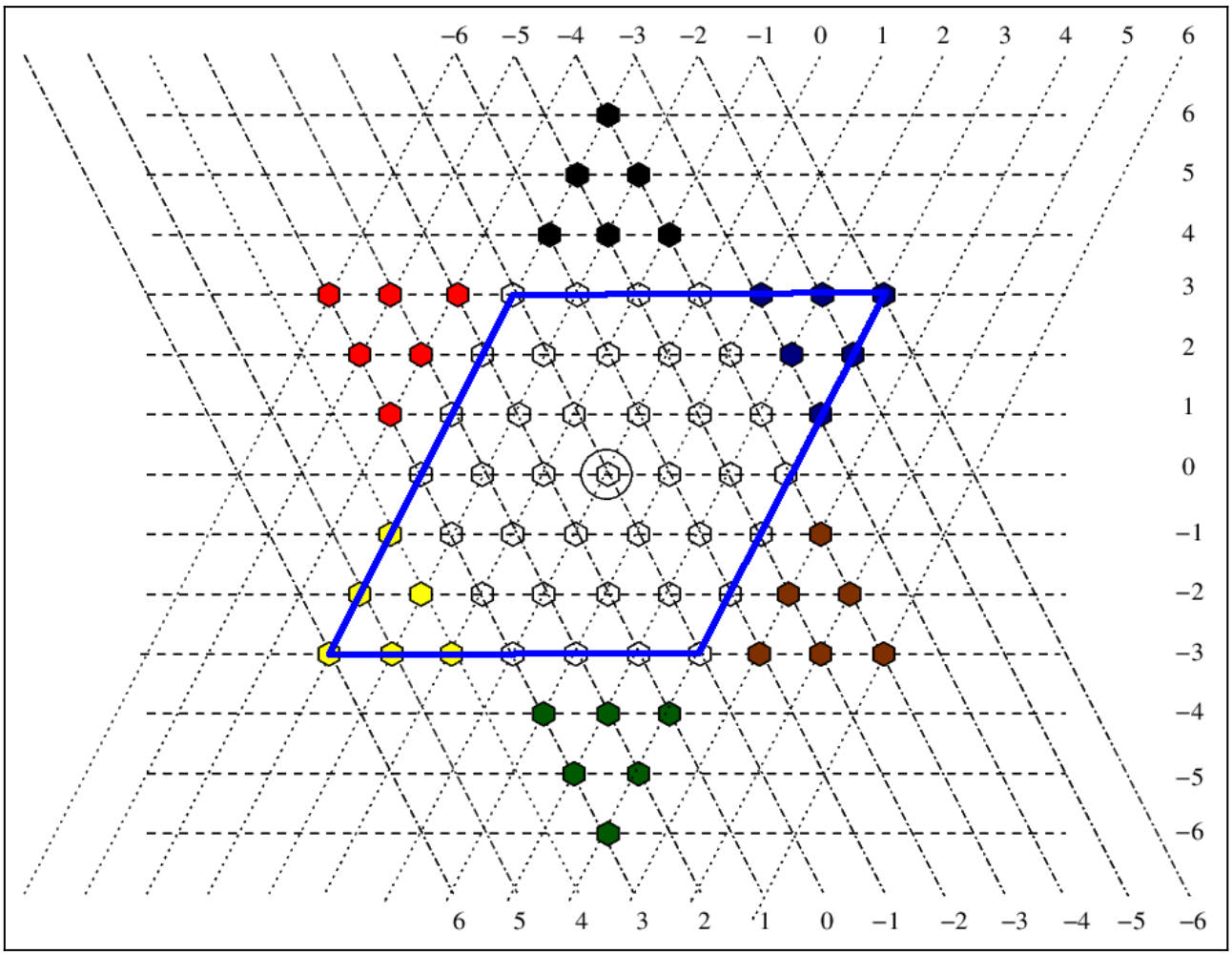

Figure 3. Northwest-Southeast losange

RÉALISATION – est dans losange Northwest-Southeast	
ALGORITHME	composition booléenne si la case est dans le losange Northwest-Southeast de la figure 3
IMPLIMENT.	

```
let est_dans_losange_2 (c:case) (dim:dimension): bool =
  let i, _, k = c in
    -dim <= i && i <= dim &&
    -dim <= k && k <= dim
;;
```

2.5 est_dans_losange_3

SPÉCIFICATION – est dans losange Northeast-Southwest	
PROFIL	est_dans_losange_3 : *case* ⟶ *dimension* ⟶ 𝔹
SÉMANTIQUE	*(est_dans_losange_3)* vérifie si la case *c* est dans le losange Northeast-Southwest du plateau de dimension *dim*.
EX. ET PROP.	*1) est_dans_losange_3((3, -6, 3), 3)* = vrai *2) est_dans_losange_3((3, 0, -3), 3)* = vrai *3) est_dans_losange_3((-3, 6, -3), 3)* = vrai *4) est_dans_losange_3((-3, 0, 3), 3)* = vrai *(est_dans_losange_3 (0,0,0), dim)* = vrai, ∀ *dim*∈*dimension*

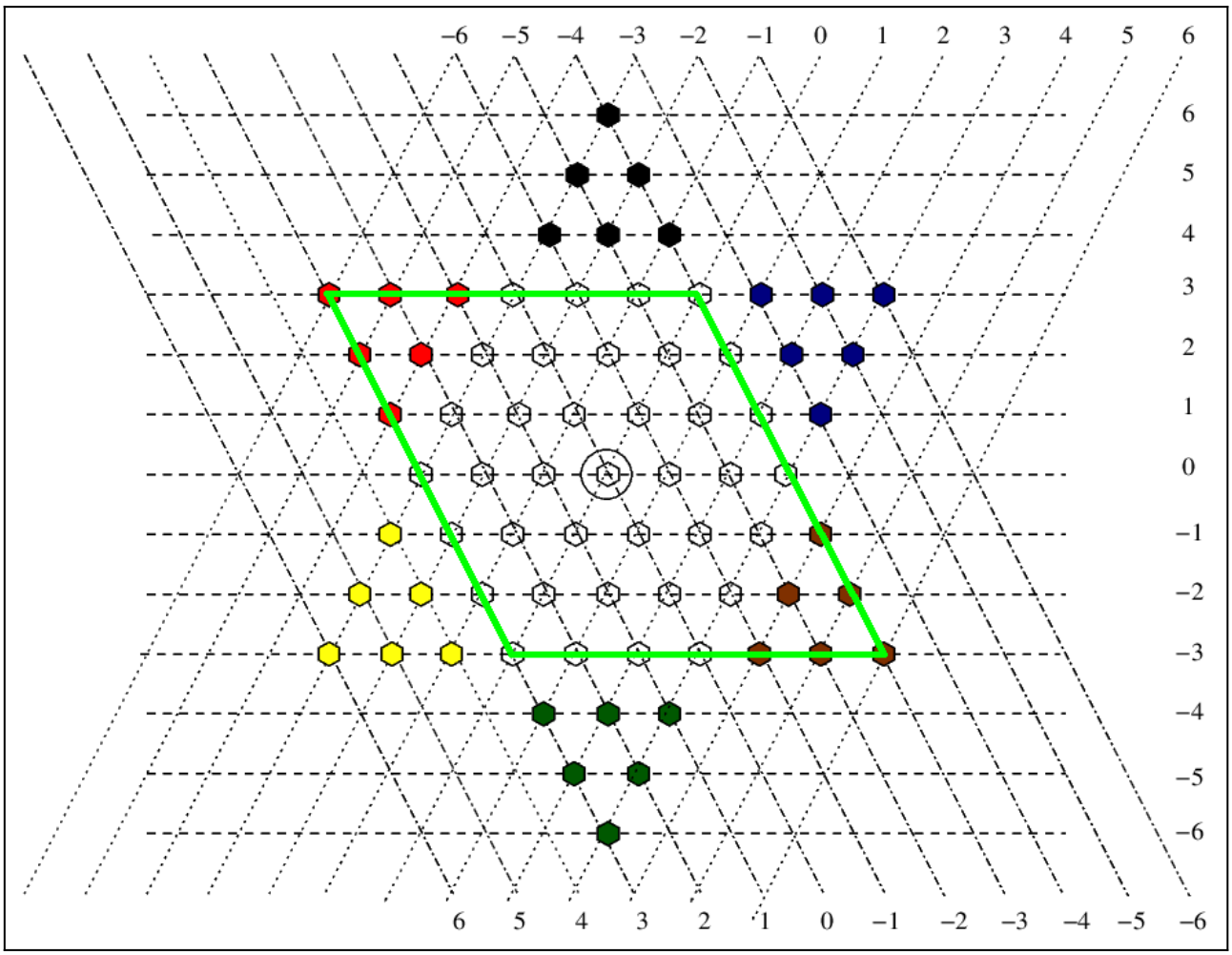

Figure 4. Northeast-Southwest losange

RÉALISATION – est dans losange Northeast-Southwest	
ALGORITHME	composition booléenne si la case est dans le losange Northeast-Southwest de la figure 4
IMPLIMENT.	

```
let est_dans_losange_3 (c:case) (dim:dimension): bool =
  let i, j, _ = c in
    -dim <= i && i <= dim &&
    -dim <= j && j <= dim
;;
```

2.6 est_dans_etoile

SPÉCIFICATION – est dans étoile	
PROFIL	est_dans_etoile : *case* ⟶ *dimension* ⟶ 𝔹
SÉMANTIQUE	*(est_dans_etoile)* vérifie si la case *c* est dans l'étoile du plateau de dimension *dim*
EX. ET PROP.	*1) est_dans_etoile((6, -3, -3), 3)* = vrai *2) est_dans_etoile((3, -6, 3), 3)* = vrai *3) est_dans_etoile((-3, -3, 6), 3)* = vrai *4) est_dans_etoile((-6, 3, 3), 3)* = vrai *5) est_dans_etoile((-3, 6, -3), 3)* = vrai *6) est_dans_etoile((3, 3 -6), 3)* = vrai *(est_dans_etoile (0,0,0), dim)* = vrai, ∀ *dim*∈*dimension*

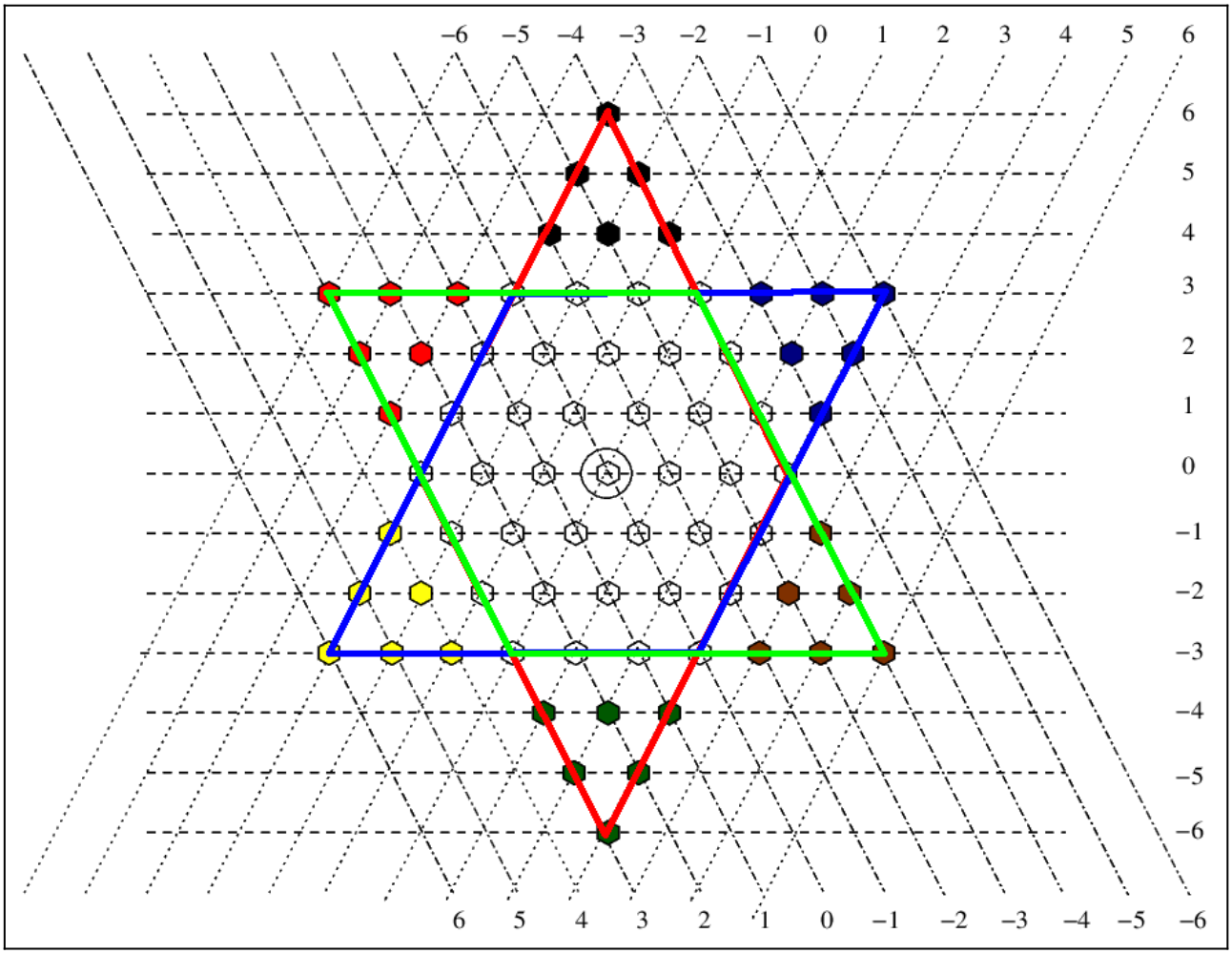

Figure 5. L'union de trois losanges forme une étoile

RÉALISATION – est dans étoile	
ALGORITHME	composition booléenne et fonctions si la case est dans l'union de trois losanges
IMPLIMENT.	

```
let est_dans_etoile (c:case) (dim:dimension): bool =
  (* l'union de trois losange est un étoile *)
  est_dans_losange c dim ||
  est_dans_losange_2 c dim ||
  est_dans_losange_3 c dim
;;
```

2.7 tourner_case

SPÉCIFICATION – tourner case	
PROFIL	tourner_case : $\mathbb{N} \longrightarrow case \longrightarrow case$
SÉMANTIQUE	*(tourner_case m c)* c'est la case *c* après avoir fait tourner le plateau de *m* sixième de tour dans le sens anti-horaire.
EX. ET PROP.	*1) tourner_case(0, (4, -2, -2))* = (4, -2, -2) *2) tourner_case(1, (4, -2, -2))* = (2, -4, 2) *3) tourner_case(2, (4, -2, -2))* = (-2, -2, 4) *4) tourner_case(3, (4, -2, -2))* = (-4, 2, 2) *5) tourner_case(4, (4, -2, -2))* = (-2, 4, -2) *6) tourner_case(5, (4, -2, -2))* = (2, 2, -4) *7) tourner_case(6, (4, -2, -2))* = (4, -2, -2) $\forall n \in \mathbb{N}, \forall (i,j,k) \in case, m = n \bmod 6,$ $sim = 0,\ tourner\_case(m,(i,j,k)) = (i,j,k)$ $sim \neq 0,\ tourner\_case(m,(i,j,k)) = tourner\_case((m-1),(-k,-i,-j))$

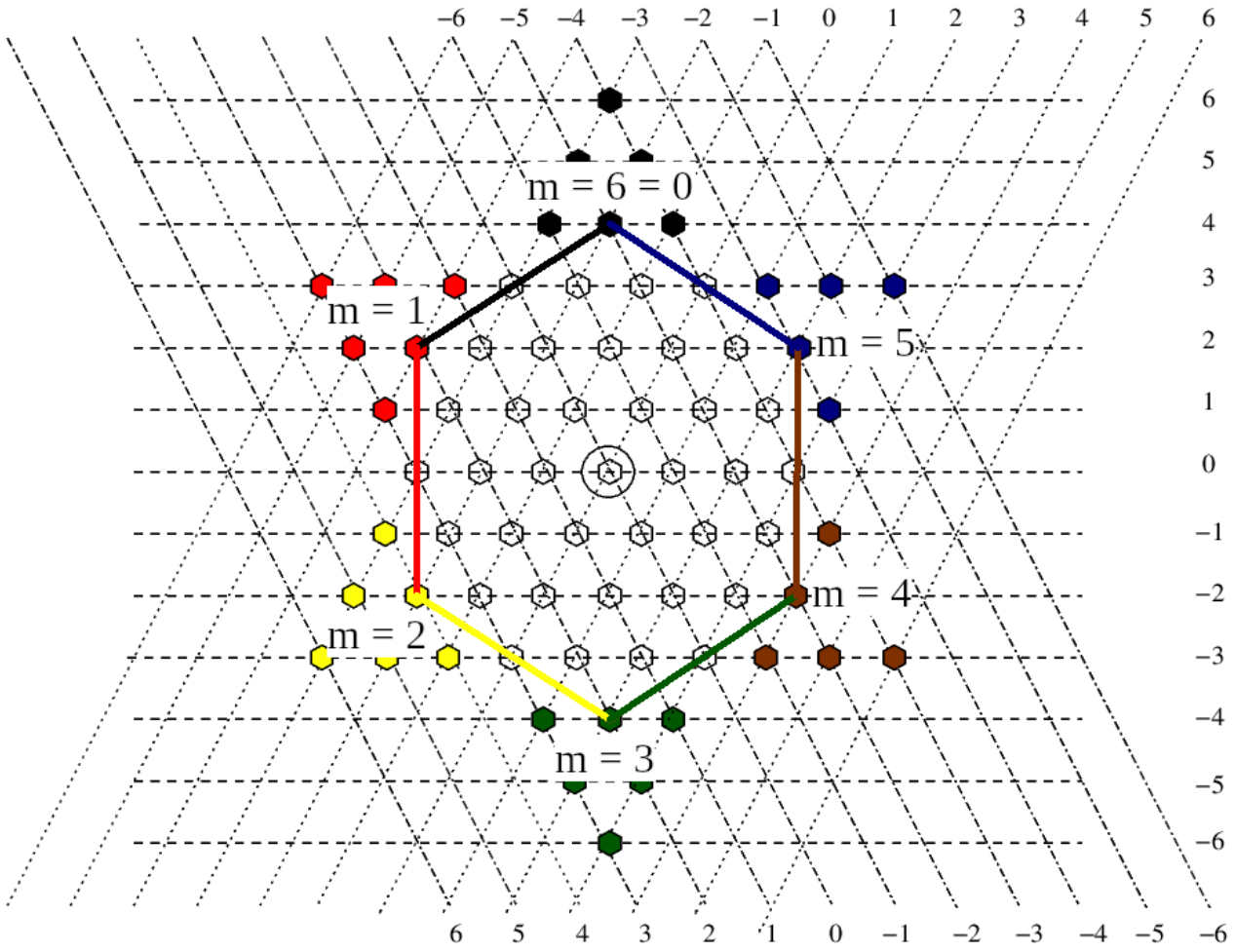

Figure 5. La case après avoir tourner 6 fois

RÉALISATION – tourner case	
On va utiliser une fonction locale *tourner* avec une équation de récurrence : $(1) tourner(0,(i,j,k)) = (i,j,k)$ $(2) tourner(m,(i,j,k)) = tourner((m-1),(-k,-i,-j))$	
ALGORITHME	composition conditionnelle, composition fonction et analyse par cas de filtrage

IMPLIMENT.	`let tourner_case (m:int) (c:case): case =`
	`  (* réduction de nombre de fait pour tourner *)`
	`  let m = m mod 6 in`

```
  (* équation récursive *)
    let rec tourner_case_rec (m:int) (c:case): case =
      let i, j, k = c in
      match m with
      | 0 -> i, j, k
      | m -> tourner_case_rec (m - 1) (-k, -i, -j)
    in tourner_case_rec m c
;;
```

2.8 translate

SPÉCIFICATION – translate	
PROFIL	translate : *case* ⟶ *vecteur* ⟶ *case*
SÉMANTIQUE	*(translate c v)* calcule la case par le translation du vecteur *v* à partir de *c*.
EX. ET PROP.	*1) translate((0, -2, 2), (0, 1, -1))* = (0, -1, 1) *2) translate((0, 0, 0), (0, -2, 2))* = (0, -2, 2) *3) translate((0, -2, 2), (0, 0, 0))* = (0, -2, 2) *4) translate((0, 0, 0), (0, 0, 0))* = (0, 0, 0) $\forall (c1, c2, c3), (v1, v2, v3) \in case, (c1+v1, c2+v2, c3+v3) \in case$

RÉALISATION – translate	
ALGORITHME	addition de coordonnées de case *c* et vecteur *v*
IMPLIMENT.	

```
let translate (c:case) (v:vecteur): case =
  (* les coordonnées de la case c *)
  let c1, c2, c3 = c
  (* les coordonnées du vectuer v *)
  and v1, v2, v3 = v in
    (* translation des coordonnées de v vers c *)
    v1 + c1, v2 + c2, v3 + c3
;;
```

2.9 diff_case

SPÉCIFICATION – diff_case	
PROFIL	diff_case : *case* ⟶ *case* ⟶ *vecteur*
SÉMANTIQUE	*(diff_case c1 c2)* c'est le vecteur de translation de *c2* vers *c1*, calculer par la différence entre les cases *c1* et *c2*.
EX. ET PROP.	*1) diff_case((0, -2, 2), (0, 1, -1))* = (0, -3, 3) *2) diff_case((0, 0, 0), (0, -2, 2))* = (0, 2, -2) *3) diff_case((0, -2, 2), (0, 0, 0))* = (0, -2, 2) *4) diff_case((0, 0, 0), (0, 0, 0))* = (0, 0, 0) $\forall (i1, j1, k1), (i2, j2, k2) \in case, (i1-i2, j1-j2, k1-k2) \in vecteur$

RÉALISATION – diff_case	
ALGORITHME	différence entre les coordonnées des cases *c1* et *c2*
IMPLIMENT.	

```
let diff_case (c1:case) (c2:case): vecteur =
  let i1, j1, k1 = c1
  and i2, j2, k2 = c2 in
    (* la différence entre les coordonnées c1 et c2 *)
    i1 - i2, j1 - j2, k1 - k2
;;
```

2.10 sont_cases_alignee

SPÉCIFICATION – sont_cases_alignee	
PROFIL	sont_cases_alignee : *case* ⟶ *case* ⟶ $\mathbb{B}$
SÉMANTIQUE	*(sont_cases_alignee c1 c2)* vérifie si les cases *c1* et *c2* sont alignées.
EX. ET PROP.	*1) sont_cases_alignee((0, -2, 2), (0, 2, -2))* = vrai *2) sont_cases_alignee((-2, 0, 2), (2, 0, -2))* = vrai *3) sont_cases_alignee((-2, 2, 0), (2, -2, 0))* = vrai 4) sont_cases_alignee((-2, 2, 0), (2, 0, -2)) = faux pas de coordonnées qui est égal à l'autre $\forall$ *c*∈*case*, *sont_cases_alignee(c, c)* = faux on effet c'est vrai, car chaque de coordonnées est égal, mais on idée les cases doivent être de différentes valeurs, donc c'est faux

RÉALISATION – sont_cases_alignee	
ALGORITHME	analyse par cas de filtrage
IMPLIMENT.	

```
let sont_cases_alignee (c1:case) (c2:case): bool =
  let i1, j1, k1 = c1
  and i2, j2, k2 = c2 in
    match () with (* comparasion entre chaque de coordonnées *)
    (* les doivent être de différentes valeurs *)
    | _ when c1 = c2 -> false
    (* s'ils sont alignées sur i, donc vrai *)
    | _ when i1 = i2 -> true
    (* s'ils sont alignées sur j, donc vrai *)
    | _ when j1 = j2 -> true
    (* s'ils sont alignées sur k, donc vrai *)
    | _ when k1 = k2 -> true
    | _ -> false (* si les cases ne sont pas alignées *)
;;
```

2.11 dist_entre_coordonnees

SPÉCIFICATION – dist_coords	
PROFIL	dist_coords : *case* ⟶ *case* ⟶ $\mathbb{N}^3$
SÉMANTIQUE	*(dist_coords c1 c2)* est un triplet de distances entre les coordonnées des cases *c1* et *c2*.
EX. ET PROP.,	*1) dist_coords((0, 0, 0), (0, -2, 2))* = (0, 2, 2) *2) dist_coords((0, 2, -2), (0, 0, 0))* = (0, 2, 2) *3) dist_coords((0, 2, -2), (0, -2, 2))* = (0, 4, 4) $\forall$ *c*∈*case*, *dist_coords(c, c)* = (0, 0, 0)

RÉALISATION – dist_coords	
ALGORITHME	calculation de chaque coordonnées utilisont la fonction *abs*
IMPLIMENT.	

```
let dist_coords (c1:case) (c2:case): int * int * int =
  let i1, j1, k1 = c1
  and i2, j2, k2 = c2 in
    let di = abs (i1 - i2) (* distance entre les coordonnées i *)
    and dj = abs (j1 - j2) (* distance entre les coordonnées j *)
    and dk = abs (k1 - k2) (* distance entre les coordonnées k *)
      in di, dj, dk (* triplet des distances entres i, j et k *)
;;
```

2.12 max_dist_cases

SPÉCIFICATION – max_dist_cases	
PROFIL	max_dist_cases : *case* ⟶ *case* ⟶ ℕ
SÉMANTIQUE	*(max_dist_cases c1 c2)* est la distance maximale entre les coordonnées des cases *c1* et *c2*.
EX. ET PROP.	*1) max_dist_cases((0, 0, 0), (0, -2, 2))* = 2 *2) max_dist_cases((0, 2, -2), (0, 0, 0))* = 2 *3) max_dist_cases((0, 2, -2), (0, -2, 2))* = 4 ∀ *c*∈*case*, dist_coords(c, c) = 0

RÉALISATION – max_dist_cases	
ALGORITHME	utilisation des fonctions dist_coords et max

IMPLIMENT.	`let max_dist_cases (c1:case) (c2:case): int =`
	`  let di, dj, dk = dist_coords c1 c2 in`
	`  max di (max dj dk)`
	`;;`

2.13 min_dist_cases

SPÉCIFICATION – min_dist_cases	
PROFIL	min_dist_cases : *case* ⟶ *case* ⟶ ℕ
SÉMANTIQUE	*(min_dist_cases c1 c2)* est la distance minimale entre les coordonnées des cases c1 et *c2*.
EX. ET PROP.	*1) min_dist_cases((0, 0, 0), (0, -2, 2))* = 0 *2) min_dist_cases((-2, 3, -1), (0, 0, 0))* = 1 *3) min_dist_cases((0, 3, -3), (-2, -2, 4))* = 2 *3) min_dist_cases((0, 3, -3), (-3, -3, 6))* = 3 ∀ *c*∈*case*, dist_coords(c, c) = 0

RÉALISATION – min_dist_cases	
ALGORITHME	utilisation des fonctions dist_coords et min

IMPLIMENT.	`let min_dist_cases (c1:case) (c2:case): int =`
	`  let di, dj, dk = dist_coords c1 c2 in`
	`  min di (min dj dk)`
	`;;`

2.14 compte_cases

SPÉCIFICATION – compte_cases	
PROFIL	compte_cases : *case* ⟶ *case* ⟶ ℕ
SÉMANTIQUE	*(compte_cases c1 c2)* est le nombre de cases entres les cases *c1* et *c2*. Pour determiner ce nombre on prendent la distance maximale si ils sont alignées, sinon la distance minimale.
EX. ET PROP.	*1) compte_cases((0,-1, 1), (0, 1,-1))* = 1 *2) compte_cases((2,-3, 1), (2, 0,-2))* = 2 *3) compte_cases((0,-2, 2), (0, 2,-2))* = 3 *4) compte_cases((3,-3, 0), (-2, 2, 0))* = 4 *5) compte_cases((0,-3, 3), (0, 3,-3))* = 5 ∀ *c*∈*case*, *compte_cases(c, c)* = 0

RÉALISATION – compte_cases	
ALGORITHME	analyse par cas de filtrage et utilisation des fonctions *sont_cases_alignee*, *max_dist_cases* et *min_dist_cases*.
IMPLIMENT.	

```
let compte_cases (c1:case) (c2:case): int =
  match () with
  (* si les cases sont égal *)
  | _ when c1 = c2 -> 0
  (* si les cases sont alignées *)
  | _ when sont_cases_alignee c1 c2 -> max_dist_cases c1 c2 - 1
  (* sinon ... *)
  | _ -> min_dist_cases c1 c2 - 1
;;
```

2.15 sont_cases_voisines

SPÉCIFICATION – sont_cases_voisines	
PROFIL	sont_cases_voisines : *case* ⟶ *case* ⟶ $\mathbb{B}$
SÉMANTIQUE	*(sont_cases_voisines c1 c2)* vérifie si les cases *c1* et *c2* sont voisines.
EX. ET PROP.	*1) sont_cases_voisines((0, 0, 0), (0,-1, 1))* = vrai *2) sont_cases_voisines((0, 0, 0), (0, 1,-1))* = vrai *3) sont_cases_voisines((0, 0, 0), (-1, 0, 1))* = vrai $\forall$ *c*$\in$*case*, *sont_cases_voisines(c, c)* = faux

RÉALISATION – sont_cases_voisines	
ALGORITHME	composition booléenne et utilisation des fonctions *sont_cases_alignee* et *max_dist_cases*.
IMPLIMENT.	

```
let sont_cases_voisines (c1:case) (c2:case): bool =
 (* si les cases sont alignées et la distances entre eux est 1 *)
  sont_cases_alignee c1 c2 && max_dist_cases c1 c2 = 1
;;
```

2.16 calcul_pivot

SPÉCIFICATION – calcul_pivot	
PROFIL	calcul_pivot : *case* ⟶ *case* ⟶ *case option*
SÉMANTIQUE	*(calcul_pivot c1 c2)* calcul le pivot entre les cases *c1* et *c2* si ils sont alignées et le nombre de cases entre les deux est impair, sinon *None*.
EX. ET PROP.	*1) calcul_pivot((0,-1, 1), (0, 1,-1))* = Some (0, 0, 0) *2) calcul_pivot((1, 0,-1), (-1, 0, 1))* = Some (0, 0, 0) *3) calcul_pivot((-1, 1, 0), (1,-1, 0))* = Some (0, 0, 0) *4) calcul_pivot (0,-2, 2) (2, 0,-2)* = None *5) calcul_pivot (2, 0,-2) (-2, 2, 0)* = None *6) calcul_pivot (-2, 2, 0) (0,-2, 2)* = None $\forall$ *c*$\in$*case*, *case_pivot(c, c)* = None

RÉALISATION – calcul_pivot	
ALGORITHME	composition conditionnelle et utilisation de fonctions *mod*, *compte_cases*, *translate* et *sont_cases_alignee*.
IMPLIMENT.	

```
let calcul_pivot (c1:case) (c2:case): case option =
  (* si le nombre de cases entre c1 et c2 est impair *)
  let est_impair = (compte_cases c1 c2) mod 2 = 1
  (* les coordonnées du vecteur de translation de c2 vers c1 *)
  and i, j, k = diff_case c1 c2 in
  (* le vecteur de translation de c2 vers le mi-chemin de c1 *)
```

```
    let v = i/2, j/2, k/2 in
    (* les coordonnées de pivot *)
    let p = translate c2 v in
    if est_impair && sont_cases_alignee c1 c2
      then Some(p) (* si impair et alignées, pivot existe *)
      else None    (* sinon, pivot n'existe pas *)
;;
```

2.17 vec_et_dist

SPÉCIFICATION – vec_et_dist	
PROFIL	vec_et_dist : *case* ⟶ *case* ⟶ *vecteur* × ℕ
SÉMANTIQUE	*(vec_et_dist c1 c2)* est le couple *(v, d)* avec *v* le vecteur de translation d'un déplacement unitaire des cases alignées *c1* vers *c2* et avec *d* la distance entre c'est cases. Si le vecteur unitaire n'existe pas, alors en renvoie *((0, 0, 0), 0)*.
EX. ET PROP.	1) vec_et_dist((0, -2, 2), (0, 0, 0)) = ((0,1,-1), 2) 2) vec_et_dist((0, 0, 0), (0, -2, 2)) = ((0,-1,1), 2) 3) vec_et_dist((0, -2, 2), (0, -2, 2)) = (0, 0, 0), 0) 4) vec_et_dist((0, -2, 2), (-2, 2, 0)) = (0, 0, 0), 0) vec_et_dist(c, c) = ((0, 0, 0), 0), ∀ *c*∈*case*

,

RÉALISATION – vec_et_dist	
ALGORITHME	composition conditionnelle et utilisation des fonctions *sont_cases_alignee*, *max_dist_cases* et *diff_case*.
IMPLIMENT.	

```
let vec_et_dist (c1:case) (c2:case): vecteur * int =
  (* si c1 = c2 ou non alignées renvoie nuls *)
  if c1 = c2 || not (sont_cases_alignee c1 c2) then (0, 0, 0), 0
  else (* sinon ... *)
    (* la distance entres les cases *)
    let d = max_dist_cases c1 c2
    (* les coordonnées du vecteur de translation de c2 vers c1 *)
    and i, j, k = diff_case c1 c2 in
    (* les coordonnées du vecteur de translation unitaire de *)
    (* c2 vers c1 *)
    let i, j, k = i/d, j/d, k/d in
    (* le vecteur de translation unitaire de c1 vers c2 *)
    let v = i * (-1), j * (-1), k * (-1) in
      if est_case v
        then v, d (* si c'est un vecteur *)
        else (0, 0, 0), 0 (*'sinon  *)
;;
```