

Démineur

Last updated by | norrova | 13 janv. 2021 at 16:35 UTC+1

CRAMEZ Théo
NORRO Valentin

Démineur

Niveau 0

Méthodologie de travail :

Organisation des tâches : Trello

Code : <https://github.com/norrova/ia> ↗

Langage de programmation : Python

Introduction :

Le démineur est un jeu vidéo de type plateau qui a pour objectif de découvrir toutes les cellules contenant des bombes représentées dans un champ de mines avec pour seule indication des cases avec le nombre de mines adjacentes à chacune d'entre-elles.

Les règles du jeu sont :

Pour gagner, le joueur doit découvrir toutes les cases sans tomber sur une bombe.

- Si le joueur creuse sur une bombe, il perd la partie.
- Si le joueur creuse sur une case vide alors toutes les cases vides aux alentours seront creusées et les cases vides ayant un drapeau seront creusées également.
- Si lors de la première action le joueur creuse sur une bombe alors celle-ci sera déplacée dans le tableau vers une autre case qui n'a pas de bombe.
- On peut signaler autant de drapeaux qu'il y a de cases sur la grille.

Objectif général :

Écrire un programme qui permet de jouer au démineur avec trois options d'actions creuser, poser un drapeau ou retirer un drapeau. Nous implémenterons également un algorithme capable de résoudre automatiquement le jeu du démineur.

% IN :

Tableau deux dimensions contenant des cases avec des bombes et des cases avec des numéros indiquant le nombre de mines adjacentes.

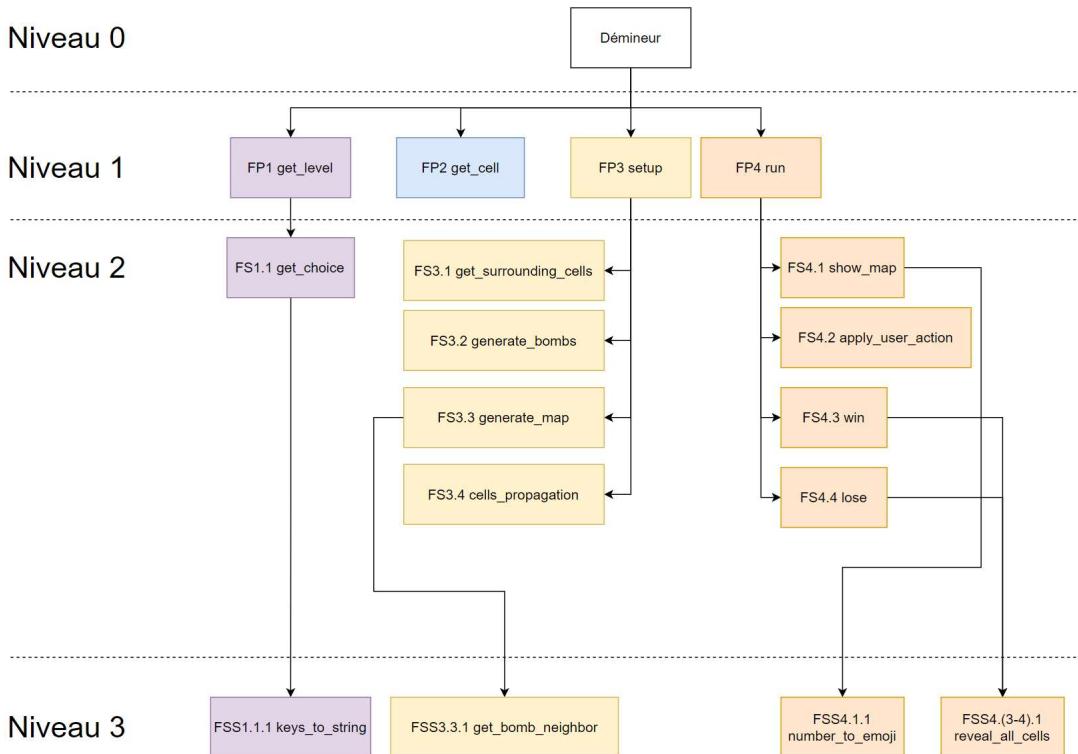
% OUT :

Affichage de la grille avec les bombes et les indicateurs de bombes.

Arbre des dépendances

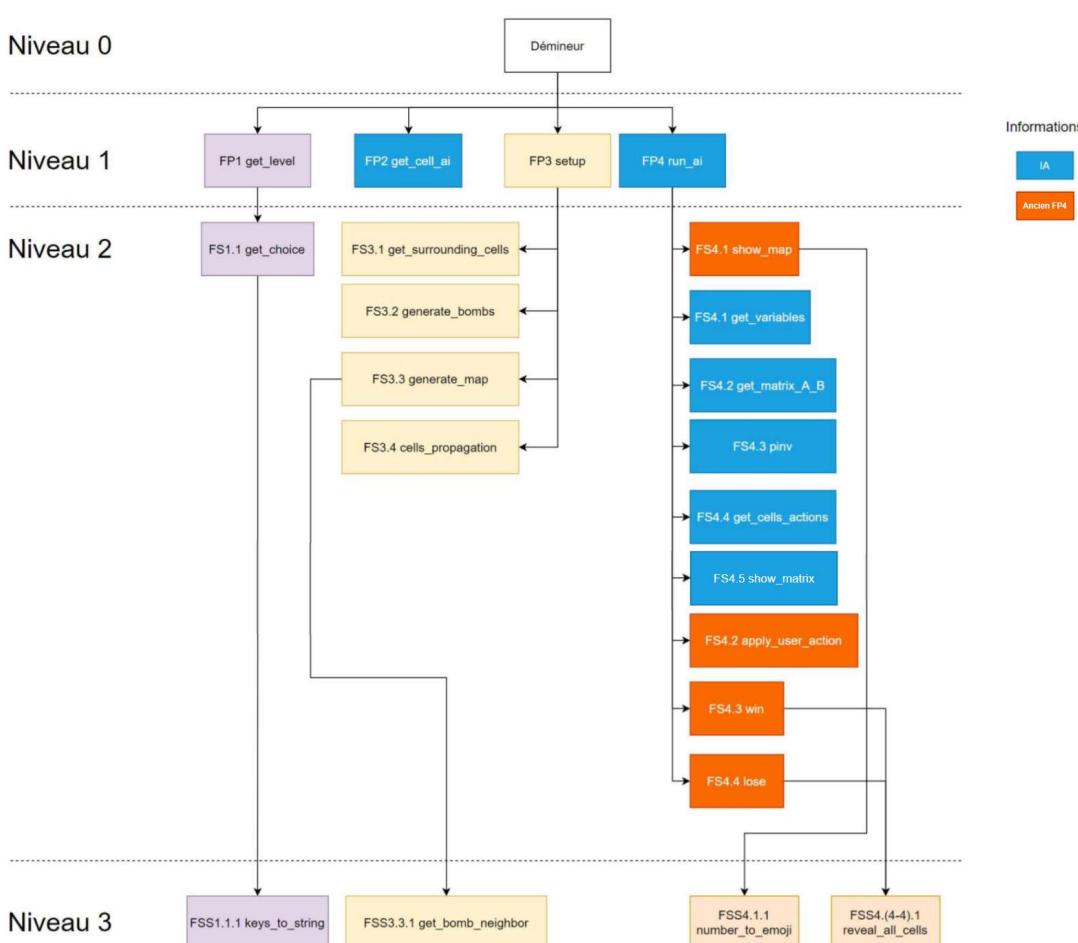
Partie sans bot :

Niveau 0



Partie avec bot :

Niveau 0



Informations générales au projet

Paramétrage du jeu :

Utilisation d'un fichier json contenant 3 niveaux avec une largeur, une hauteur et le nombre de mines :

- Débutant
- Intermédiaire
- Expert

```
{  
    "Beginner" : {  
        "width" : 9,  
        "height" : 9,  
        "mines" : 10  
    },  
    "Intermediate" : {  
        "width" : 16,  
        "height" : 16,  
        "mines" : 40  
    },  
    "Expert" : {  
        "width" : 30,  
        "height" : 16,  
        "mines" : 99  
    }  
}
```

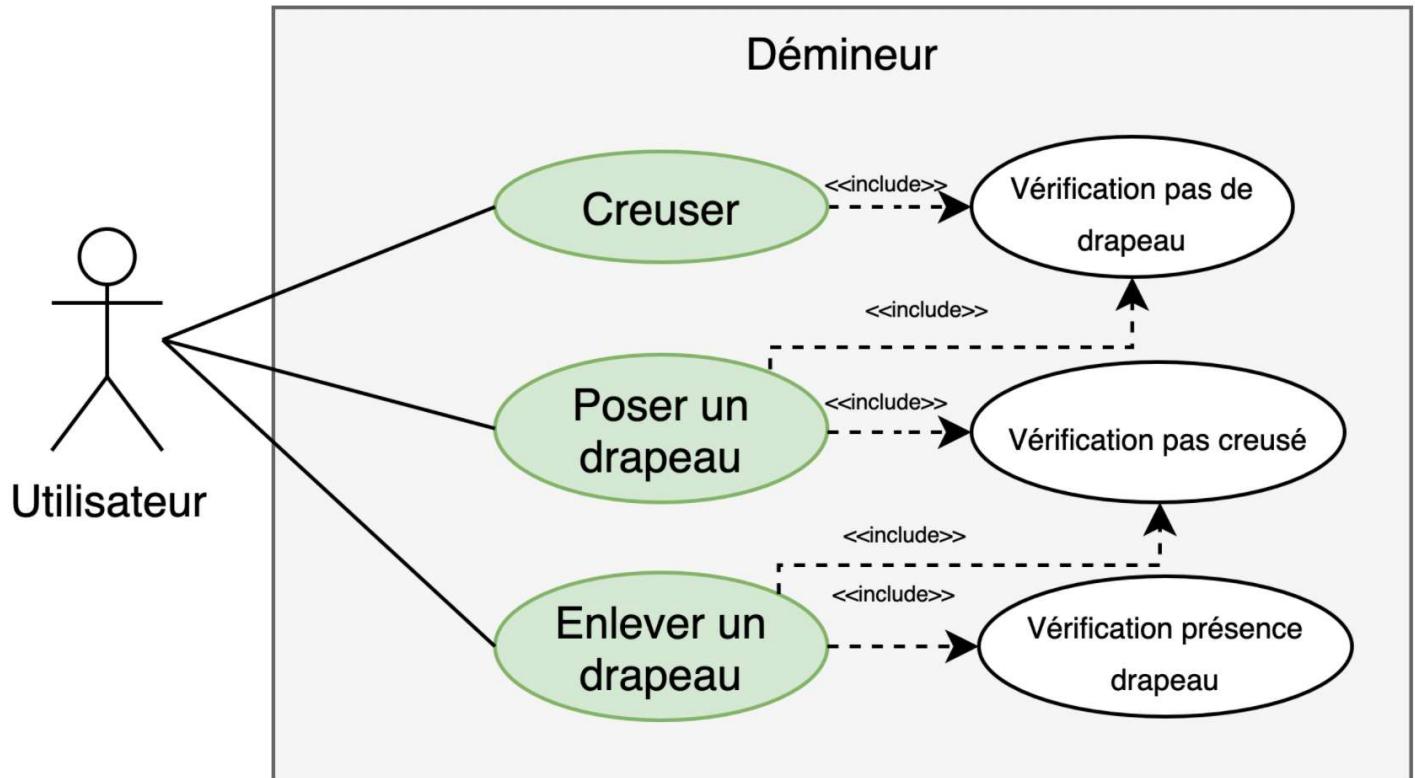
Structuration du tableau deux dimensions des voisins :

```
[  
    y0 [{x: 0, y:0}, {x: 1, y:0}]  
    y1 [{x: 0, y:1}, {x: 1, y:1}]  
]
```

Structuration du tableau bombe, drapeau posé, cellule creusée :

```
[{x: 0, y:0},...]
```

Cas d'utilisation



Niveau 1

FP1 get_level

Permet de récupérer le niveau choisi par l'utilisateur.

% IN : [levels]

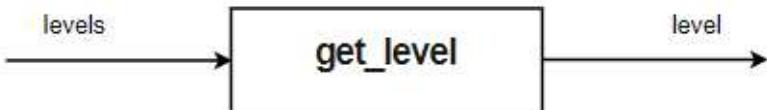
- levels : Tableau associatif contenant des clés et des valeurs : Contient les différents niveaux

% OUT : [level]

- level : Tableau associatif : Contient le niveau sélectionné

Pseudo-code :

```
Fonction get_level :  
    Paramètre par valeur :  
        - levels : Tableau associatif  
    Variables :  
        - choice : Entier  
        - level : Tableau associatif  
Début  
    choice <- get_choice(levels)  
    level <- levels[Tableau(levels)[choice]]  
    Retourne level  
Fin
```



FP2 get_cell

Permet de récupérer les coordonnées x et y d'une cellule

% IN : [height, width]

- height : Entier : Hauteur de la grille
- width : Entier : Largeur de la grille

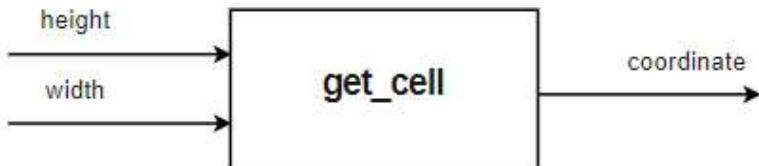
% OUT : [coordinate]

- coordinate : Tableau associatif : Représente une cellule avec les coordonnées x et y
- Pseudo-code :

```

Fonction get_cell :
    Paramètre par valeur :
        - height : Entier
        - width : Entier
    Variable :
        - axes : Tableau associatif
        - valid : Booléen
        - entry : Chaîne de caractères
        - coordinate : Tableau associatif
    Début
        axes <- {"x": height, "y": width}
        coordinate <- {}
        Pour axe dans axes (Récupération des clés + transformation en tableau)
            response <- ("Axe " + axe + " between 1 and " + axes[axe] + " : ")
            valid <- Faux
            Tant que valid == Faux Faire
                Lire entry
                Si entry >= 1 et entry <= axes[axe]
                    valid <- Vrai
                    coordinate.update({axe: entry - 1})
                Fin si
            Fin tant que
        Fin pour
        Retourne coordinate
    Fin

```



FP3 setup

Permet d'initialiser la partie.

% IN : [level, cell]

- level : Tableau associatif : Contient le niveau sélectionné
- cell : Tableau associatif : Représente une cellule avec les coordonnées x et y

% OUT : [game_param]

- game_param : Tableau associatif : Contient toutes les informations nécessaires au déroulement du jeu.

Pseudo-code :

```

Fonction setup :
    Paramètre par référence :
        - game_param : Tableau associatif
        - cell : Tableau associatif

    Variable :
        - game_param : Tableau associatif
        - excluded_cells : Tableau associatif

Début
    game_param = level
    game_param ajoute {nb_bombs = level["mines"]}
    game_param ajoute {"nb_total_cells" : game_param["height"] * game_param["width"]} - game_param["mines"]
    excluded_cells <- get_surrounding_cells(game_param["height"], game_param["width"], cell)
    game_param ajoute {"bombs" : generate_bombs(game_param["height"], game_param["width"], game_param["mines"])}
    game_param ajoute {"map" <- generate_map(game_param["height"], game_param["width"], game_param["bombs"])}
    game_param ajoute {"revealed" : [], "flagged" : []}
    game_param ajoute {"actions" : {
        "Dig": 0,
        "Flag": 1,
        "Unflag": 2
    }}
    cells_propagation(game_param, [cell])
    Retourne game_param
Fin
  
```



FP4 run

Permet de lancer une partie.

% IN : [game_param]

- game_param : Tableau associatif : Contient toutes les informations nécessaires au déroulement du jeu.

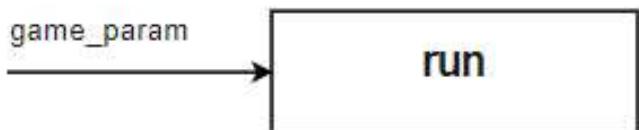
% OUT : []

Pseudo-code :

```

Procédure run :
    Paramètre par référence :
        - game_param : Tableau associatif
    Variable :
        - stop : Booléen
Début
    stop <- Faux
    show_map(game_param)
    Si win(game_param) == Faux:
        Tant que stop == Faux Faire
            show_map(game_param)
            choice <- get_choice(game_param["actions"])
            cell <- get_cell(game_param["height"], game_param["width"])
            Si cell n'existe pas dans game_param["revealed"]
                apply_user_action(game_param, choice, cell)
            Fin si
            Si win(game_param) ou lose(game_param, cell)
                stop <- Vrai True
            Fin si
        Fin tant que
    Fin si
Fin

```



Niveau 2

FS1.1 get_choice

Permet de récupérer un choix utilisateur selon le dictionnaire passé en paramètre.

% IN : [dictionnaire]

- dictionnaire : Tableau associatif contenant des clés et des valeurs

% OUT : [entry]

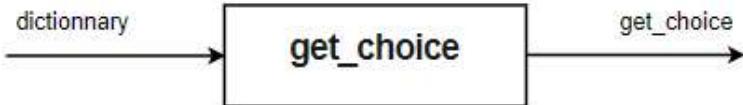
- entry : Entier : Choix utilisateur

Pseudo-code :

```

Fonction get_choice :
    Paramètre par valeur :
        - dictionnaire : Tableau associatif
    Variables :
        - message : Chaîne de caractères
        - valid : Booléen
        - entry : Entier
    Début
        message <- keys_to_string(dictionnaire)
        valid <- Faux
        Tant que Faux == valid Faire
            Lire entry
            Si entry >= 1 et entry <= Taille(dictionnaire)
                valid <- Vrai
            Fin si
        Fin tant que
        Retourne entry - 1
    Fin

```



FS3.1 get_surrounding_cells

Permet récupérer les cellules avoisinantes une cellule.

% IN : [height, width, bombs]

- height : Entier : Hauteur de la grille
- width : Entier : Largeur de la grille
- cell : Tableau associatif : Représente une cellule avec les coordonnées x et y

% OUT : [surrounding_cells]

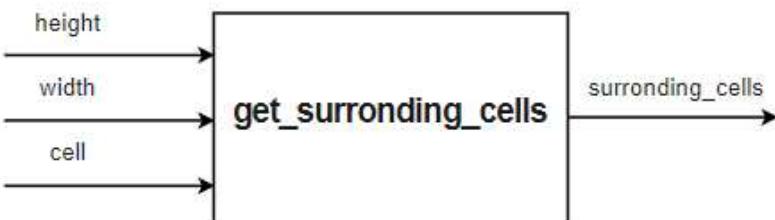
- surrounding_cells : Tableau : Contient des tableaux associatifs donc les cellules voisines

Pseudo-code :

```

Fonction get_surrounding_cells:
    Paramètre par valeur :
        - height : Entier
        - width : Entier
        - cell : Tableau associatif
    Variables :
        - surrounding_cells : Tableau
        - res_y : Entier
        - res_x : Entier
    Début
        surrounding_cells <- []
        Pour chaque y allant de -1 à 2
            res_y <- cell["y"] + y
            Si res_y >= 0 et res_y < height
                Pour chaque x allant de -1 à 2
                    res_x <- cell["x"] + x
                    Si res_x >= 0 et res_x < width
                        surrounding_cells ajoute {"x": res_x, "y": res_y}
                    Fin si
                Fin pour
            Fin si
        Fin pour
        Retourne surrounding_cells
    Fin

```



FS3.2 generate_bombs

Permet générer les bombes

% IN : [height, width, nb_bomb, excluded_cells]

- height : Entier : Hauteur de la grille
- width : Entier : Largeur de la grille
- nb_bomb : Entier : Nombre de bombes à générer
- excluded_cells : Tableau : Cellule exclue

% OUT : [bombs]

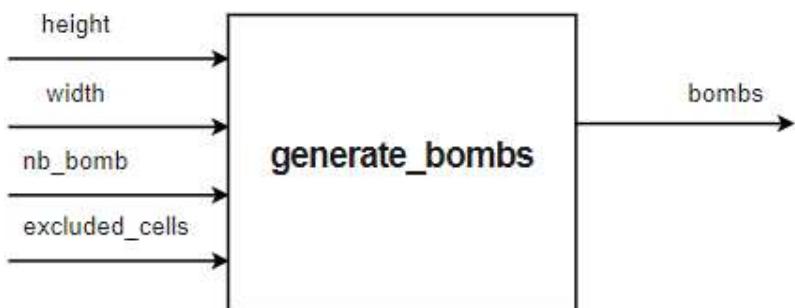
- bombs : Tableau : Contient les bombes générées aléatoirement dans la grille.

Pseudo-code :

```

Fonction generate_bombs :
    Paramètre par valeur :
        - height : Entier
        - width : Entier
        - nb_bomb : Entier
        - excluded_cells : Tableau
    Variables :
        - bombs : Tableau
        - valid : Booléen
        - coordinate : Tableau associatif
Début
    bombs <- []
    Pour index allant de 0 à nb_bomb Faire
        valid = Faux
        Tant que valid == Faux Faire
            coordinate = {"x": randrange(0, width), "y": randrange(0, height)}
            Si (coordinate n'existe pas dans bombs et coordinate n'existe pas dans excluded_cells)
                valid <- Vrai
                bombs ajoute coordinate
        Fin tant que
    Fin pour
    Retourne bombs
Fin

```



FS3.3 generate_map

Permet de générer la grille incluant le calcul des voisins lors de la création des cellules.

% IN : [height, width, bombs]

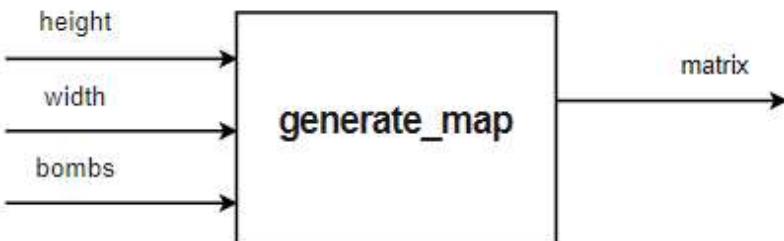
- height : Entier : Hauteur de la grille
- width : Entier : Largeur de la grille
- bombs : Tableau : Contient les bombes disposées dans la grille

% OUT : [matrix]

* matrix : Tableau deux dimensions : Contient toutes les cellules du tableau

Pseudo-code :

```
Fonction generate_map :
    Paramètre par valeur :
        - height : Entier
        - width : Entier
        - bombs : Tableau
    Variables :
        - matrix : Tableau deux dimensions
        - matrix_x : Tableau contenant des tableaux associatifs {"x": ?, "y": ?}
Début
    matrix <- []
    Pour y allant de 0 à height Faire
        matrix_x <- []
        Pour x allant de 0 à width Faire
            cell <- {"x": x, "y": y}
            Si cell existe dans bombs Faire
                matrix_x ajoute (-1)
            Sinon
                matrix_x ajoute (get_bomb_neighbor(height, width, bombs, cell))
            Fin si
        Fin pour
        matrix ajoute matrix_x
    Fin pour
    Retourne matrix
Fin
```



FS3.4 cells_propagation

Permet de révéler des cellules n'ayant pas de voisin par propagation

% IN : [game_param, to_discover]

- game_param : Tableau associatif : Contient toutes les informations nécessaires au déroulement du jeu.
- to_discover : Tableau : Contient les cellules à découvrir

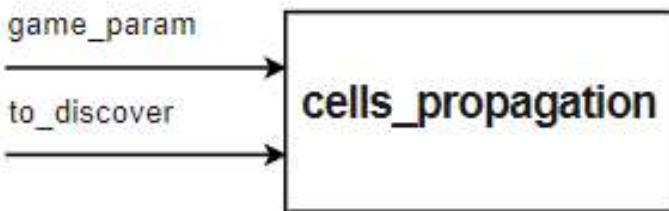
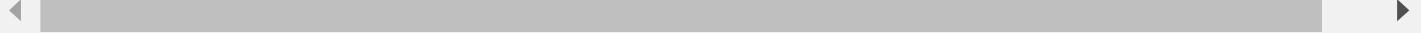
% OUT : []

Pseudo-code :

```

Procédure cells_propagation :
    Paramètre par référence :
        - game_param : Tableau associatif
    Paramètre par valeur :
        - to_discover : Tableau
    Variable :
        - tmp_cells : Tableau
Début
    tmp_cells <- []
    Pour chaque cell_discover dans to_discover
        Si cell_discover n'existe pas dans game_param["revealed"]
            game_param["revealed"] ajoute cell_discover
        Fin si
        Pour chaque cell dans get_surrounding_cells(game_param["height"], game_param["width"], cell_discover)
            Si cell n'existe pas dans to_discover et cell n'existe pas dans game_param["revealed"]
                Si 0 == game_param["map"][cell["y"]][cell["x"]]
                    tmp_cells ajoute cell
                Sinon
                    game_param["revealed"] ajoute cell
                Fin si
            Fin si
        Si Taille(tmp_cells) > 0
            cells_propagation(game_param, tmp_cells)
        Fin si
    Fin

```



FS4.1 show_map

Permet d'afficher la carte

% IN : [game_param]

- game_param : Tableau associatif : Contient toutes les informations nécessaires au déroulement du jeu.

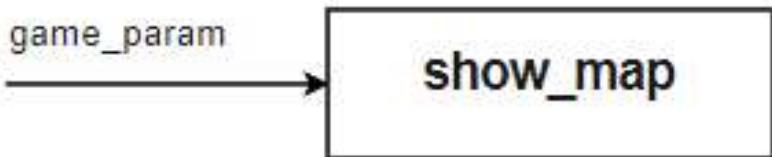
% OUT : []

Pseudo-code :

```

Procédure show_map :
    Paramètre par valeur :
        - game_param : Tableau associatif
    Variable :
        - emoji : Chaîne de caractères
Début
    Afficher "mines remaining : " + game_param["nb_bombs"]
    Pour y allant de game_param["height"] - 1 à 0
        Pour x allant de game_param["width"] - 1 à 0
            cell <- {"x": x, "y": y}
            emoji <- null
            Si cell existe dans game_param["revealed"]
                Si cell existe dans game_param["bombs"]
                    emoji <- "💣"
                Sinon si 0 == game_param["map"][y][x]
                    emoji <- "🚩"
                Sinon
                    emoji = number_to_emoji(game_param["map"][y][x])
                Fin si
            Sinon Si cell existe dans game_param["flagged"]
                emoji <- "🚩"
            Sinon
                emoji <- "◻"
            Fin si
            Afficher emoji
        Afficher ""
    Fin

```



FS4.2 apply_user_action

Permet d'appliquer l'action de l'utilisateur (creuser, poser un drapeau, enlever un drapeau).

% IN : [game_param, choice, cell]

- game_param : Tableau associatif : Contient toutes les informations nécessaires au déroulement du jeu
- choice : Entier : Choix de l'utilisateur sur l'action à effectuer (creuser, poser un drapeau, enlever un drapeau)
- cell : Tableau associatif : Représente une cellule avec les coordonnées x et y

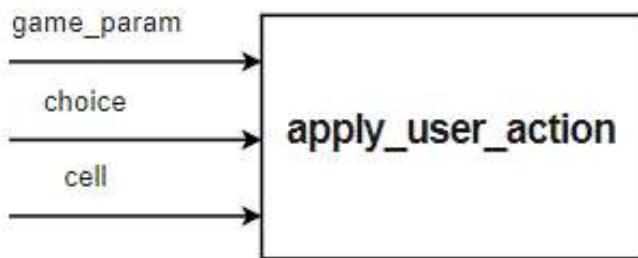
% OUT : []

Pseudo-code :

```

Procédure apply_user_action :
    Paramètre par référence :
        - game_param : Tableau associatif
    Paramètre par valeur :
        - choice : Entier
        - cell : Tableau associatif
Début
    Si choice == game_param["actions"]["Dig"]
        Si 0 == game_param["map"][cell["y"]][cell["x"]]
            cells_propagation(game_param, [cell])
        Sinon si cell n'existe pas dans game_param["flagged"]
            game_param["revealed"] ajoute cell
        Fin si
    Sinon si (choice == game_param["actions"]["Flag"]
        et cell n'existe pas dans game_param["flagged"])
        game_param["flagged"] ajoute cell
    Sinon si (choice == game_param["actions"]["Unflag"]
        et cell existe game_param["flagged"])
        game_param["flagged"] retire cell
    Fin si
Fin

```



FS4.3 win

Permet de vérifier si le joueur est gagnant.

% IN : [game_param]

- game_param : Tableau associatif : Contient toutes les informations nécessaires au déroulement du jeu.

% OUT : [win]

- win : Booléen : Joueur gagnant ou non

Pseudo-code :

```

Fonction win :
    Paramètre par référence :
        - game_param : Tableau associatif
    Variable :
        - win : Booléen
Début
    win <- Faux
    Si Taille(game_param["revealed"]) == game_param["nb_total_cells"]
        reveal_all_cells(game_param)
        show_map(game_param)
        Afficher "Congrats you win !"
        win <- Vrai
    Fin si
    Retourne win
Fin

```



FS4.4 lose

Permet de vérifier si le joueur est perdant.

% IN : [game_param, cell]

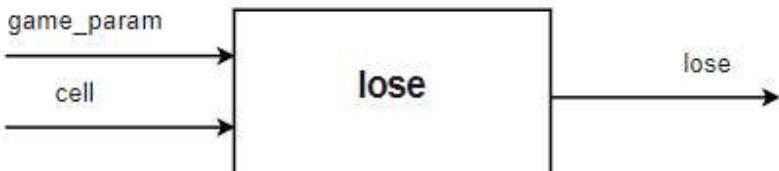
- game_param : Tableau associatif : Contient toutes les informations nécessaires au déroulement du jeu.
- cell : Tableau associatif : Représente une cellule avec les coordonnées x et y

% OUT : [lose]

- lose : Booléen : Joueur est perdant ou non

Pseudo-code :

```
Fonction lose :
    Paramètre par référence :
        - game_param : Tableau associatif
        - cell : Tableau associatif
    Variable :
        - lose : Booléen
Début
    lose <- Faux
    Si cell existe dans game_param["bombs"] et cell n'existe pas dans game_param["flagged"] et cell existe
        reveal_all_cells(game_param)
        show_map(game_param)
        Afficher "You lose"
        lose <- Vrai
    Retourne lose
Fin
```



Niveau 3

FSS1.1.1 keys_to_string

Permet de transformer des clés à partir d'un dictionnaire en chaîne de caractères.

% IN : [dictionnaire]

- dictionnaire : Tableau associatif contenant des clés et des valeurs

% OUT : [response]

- response : Chaîne de caractères : Contient toutes les clés avec un formatage

Pseudo-code :

```

Fonction keys_to_string :
    Paramètre par valeur :
        - dictionnaire : Tableau associatif
    Variables :
        - keys : Liste de chaîne de caractères
        - response : Chaine de caractères
Début
    keys <- # Récupération des clés + transformation en tableau
    response <- ""
    Pour index allant de 0 à taille(keys) Faire
        response <- ((index +1) + "." + keys[index] + " ")
    Fin Pour
    Retourne response
Fin
  
```



FSS3.3.1 get_bomb_neighbor

Permet de récupérer le nombre de bombes entourant une cellule.

% IN : [height, width, bombs]

- height : Entier : Hauteur de la grille
- width : Entier : Largeur de la grille
- bombs : Tableau : Contient les bombes disposées dans la grille
- cell : Tableau associatif : Représente une cellule avec les coordonnées x et y

% OUT : [bomb_neighbor]

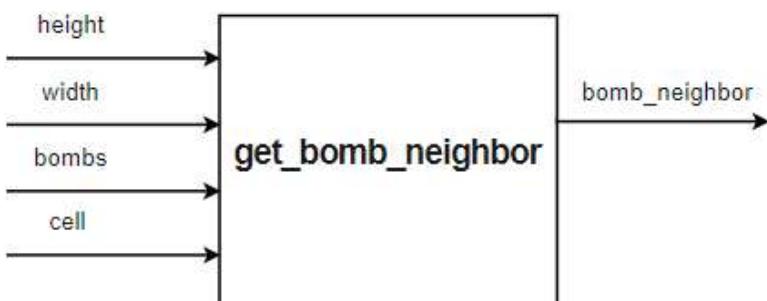
- bomb_neighbor : Entier : nombre de bombes entourant une cellule

Pseudo-code :

```

Fonction get_bomb_neighbor :
    Paramètre par valeur :
        - height : Entier
        - width : Entier
        - bombs : Tableau
    Variables :
        - cells : Tableau deux dimensions
        - bomb_neighbor : Entier : bombe voisine
    Début
        cells <- get_surrounding_cells(height, width, cell)
        cell <- null
        bomb_neighbor <- 0
        Pour chaque cell dans cells
            Si cell existe dans bombs
                bomb_neighbor <- bomb_neighbor + 1
            Fin si
        Fin Pour
        Retourne bomb_neighbor
    Fin

```



FSS4.1.1 number_to_emoji

Permet de transformer un chiffre en emoji

% IN : [number]

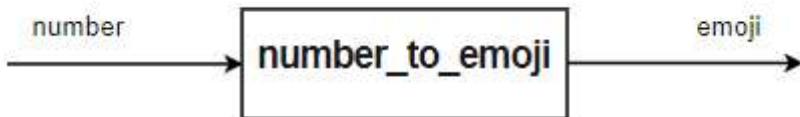
- number : Entier : Chiffre allant de 0 à 8

% OUT : [emoji]

- emoji : Chaîne de caractères : représente le chiffre en emoji

Pseudo-code :

```
Fonction number_to_emoji :  
    Paramètre par valeur :  
        - number : Entier  
    Variable :  
        - emoji : Chaîne de caractères  
Début  
    Si number > 8  
        Propage une exception  
    Fin si  
    emojis <- ['0', '1', '2', '3', '4', '5', '6', '7', '8']  
    emoji <- emojis[number]  
    Retourne emoji  
Fin
```



FSS4.(3-4).1 reveal_all_cells

Permet de révéler toutes les cellules.

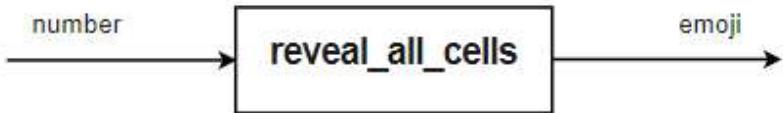
% IN : [game_param]

- game_param : Tableau associatif : Contient toutes les informations

% OUT : []

Pseudo-code :

```
Procédure reveal_all_cells :  
    Paramètre par référence :  
        - game_param : Tableau associatif  
Début  
    game_param["revealed"] <- []  
    Pour y allant de game_param["height"] - 1 à 0  
        Pour x allant de game_param["width"] - 1 à 0  
            cell = {"x": x, "y": y}  
            game_param["revealed"] ajoute cell  
        Fin pour  
    Fin pour  
Fin
```



IA

Niveau 1 :

FP2 get_cell_ai

Permet de générer les coordonnées x et y d'une cellule aléatoirement.

% IN : [height, width]

- height : Entier : Hauteur de la grille
- width : Entier : Largeur de la grille

% OUT : [coordinate]

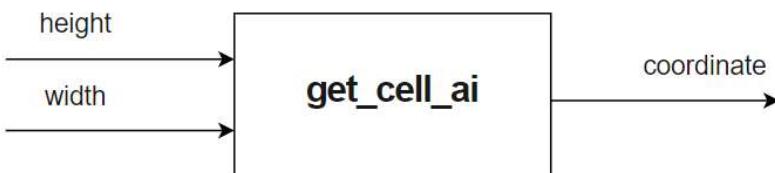
- coordinate : Tableau associatif : Représente une cellule avec les coordonnées x et y

Pseudo-code :

```

Fonction get_cell_ai:
    Paramètre par valeur :
        - height : Entier
        - width : Entier
    Variable :
        - x : Entier
        - y : Entier
        - coordinate : Tableau associatif
    Début
        x <- chiffre aléatoire entre 0 et width - 1
        y <- chiffre aléatoire entre 0 et height - 1
        coordinate <- {"x": x, "y": y}
    Retourne coordinate
Fin

```



FP4 run_ai

Permet de lancer une partie avec la résolution du problème du démineur par le bot.

% IN : [game_param]

- game_param : Tableau associatif : Contient toutes les informations nécessaires au déroulement du jeu.

% OUT : []

Pseudo-code :

```

Procédure run_ai :
    Paramètre par valeur :
        - number : Entier
    Variable :
        - stop : Booléen
        - variables : Tableau associatif
        - matrix : Tableau à deux dimensions
        - numbers : Tableau à deux dimensions (résultat de pinv(A)*B)
Début
    stop <- False
    Si Faux == win(game_param):
        Tant que Faux == stop:
            show_map(game_param)
            // Récupération des variables
            variables <- get_variables(game_param)
            // Calcul des deux matrices A et B
            matrix <- get_matrix_A_B(game_param, variables)
            numbers <- pinv(matrix['A'], matrix['B'])
            // Récupérer les cellules à effectuer une action
            cells_actions <- get_cells_actions(game_param, variables, numbers)

            Pour chaque cell dans cells_actions
                apply_user_action(game_param, cell[0], cell[1])
                Si win(game_param) ou lose(game_param, cell[1])
                    stop <- True
                    Sortir
            Fin pour
        Fin tant que
    Fin si
Fin

```



Niveau 2

FS4.1 get_variables

Récupérer les variables dans la grille

% IN : [game_param]

- game_param : Tableau associatif : Contient toutes les informations nécessaires au déroulement du jeu.

% OUT : [variables]

- variables : Tableau associatif : Contient les variables

Pseudo-code :

```

Fonction get_variables:
    Paramètre par référence :
        - game_param : Tableau associatif
    Variable :
        - variables : Tableau associatif
Début
    variables <- []
    Pour y allant de game_param["height"] - 1 à 0
        Pour x in allant de 0 à game_param["width"]
            cell <- {"x": x, "y": y}
            Si (cell n'existe pas dans game_param["revealed"]
                et cell n'existe pas dans game_param["flagged"])
                variables ajoute cell
            Fin si
        Fin pour
    Fin pour
    Retourne variables
Fin

```



FS4.2 get_matrix_A_B

Récupérer les variables dans la grille

% IN : [game_param, variables]

- game_param : Tableau associatif : Contient toutes les informations nécessaires au déroulement du jeu.
- variables : Tableau associatif : Contient les variables

% OUT : [matrices]

- matrices : Tableau associatif : Contient les deux matrices A et B

code page suivante...

Pseudo-code :

```

Fonction get_matrix_A_B :
    Paramètre par référence :
        - game_param : Tableau associatif
        - variables : Tableau associatif
    Variable :
        - cell : Tableau associatif
        - flag : entier
        - line : Tableau d'entier
        - neighbors : Tableau contenant des tableau associatif
        - matrices : Tableau associatif
        - A : Tableau associatif
        - B : Tableau associatif
Début
    A <- []
    B <- []
    Pour y allant de game_param["height"] - 1 à 0
        Pour x in allant de 0 à game_param["width"]
            cell <- {"x": x, "y": y}

            Si cell existe dans game_param["revealed"]
                neighbors <- main.get_surrounding_cells(game_param["height"], game_param["width"], cell)
                surronding_variables <- []
                flag <- 0

                Pour chaque neighbor dans neighbors
                    Si neighbor existe dans game_param["flagged"]
                        flag += 1
                    Sinon si neighbor existe dans variables
                        surronding_variables ajoute neighbor
                    Fin si
                Fin Pour

                nb_surronding_variables = len(surronding_variables)
                Si nb_surronding_variables > 0

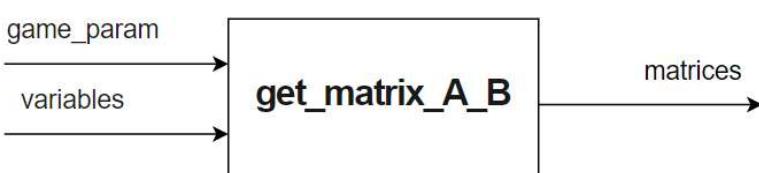
                    // Créer un tableau de 0 selon le nombre de variables
                    line <- [0] * Taille(variables)

                    Pour chaque variable dans surronding_variables
                        // Récupérer l'index d'
                        index <- variable index // Récupérer l'index dans variables
                        line[index] <- 1
                    Fin pour

                    // Eviter les nombres négatifs dans le tableau, choix du chiffre le plus grand
                    res = maximum(0, game_param["map"][cell['y']][cell['x']] - flag)
                    A ajoute line
                    B ajoute res
                Fin si
            Fin Si
        Fin pour
    Fin pour

    Si (Taille(A) > 0)
        A.append([1] * Taille(variables))
        B.append([ maximum(0, game_param["nb_bombs"] - Taille(game_param["flagged"]))])
    Fin si
    matrices <- {'A': A, 'B': B}
    Retourne matrices
Fin

```



FS4.3 pinv

Calcul de la pseudo inverse

% IN : [A, B]

- A : Tableau à deux dimensions : Indices des cases ouvertes
- B : Tableau à deux dimensions : Indice nombre de bombes alentour - Indices des cases avec drapeaux

% OUT : [variables]

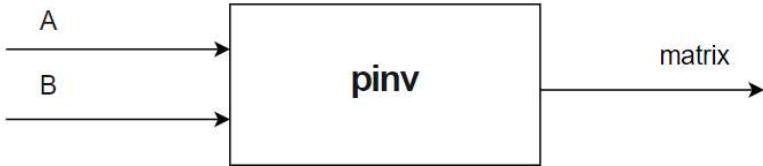
- matrix : Tableau à deux dimensions : résultat de la méthode de la pseudo inverse

Pseudo-code :

```

Fonction pinv :
    Paramètre par référence :
        - A : Tableau à deux dimensions
        - B : Tableau à deux dimensions
    Variable :
        - matrix : Tableau à deux dimensions
Début
    matrix <- pinv(A) * B
    Retourne matrix
Fin

```



FS4.4 get_cells_actions

Calcul de la pseudo inverse

% IN : [game_param, variables, matrix]

- game_param : Tableau associatif : Contient toutes les informations nécessaires au déroulement du jeu.
- variables : Tableau associatif : Contient les variables
- numbers: Tableau à deux dimensions : résultat de la méthode de la pseudo inverse

% OUT : [cells_actions]

- cells_actions : Tableau associatif : Contient chaque action a effectué par cellule (creuser, poser un drapeau)

Pseudo-code :

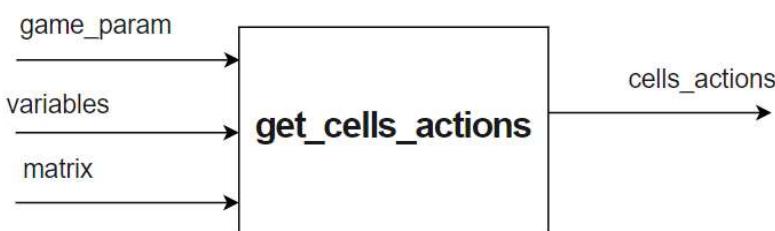
```

Fonction get_cells_actions:
    Paramètre par référence :
        - game_param : Tableau associatif
        - variables : Tableau associatif
        - numbers : Tableau à deux dimensions
    Variable :
        - cells_actions : Tableau associatif
        - index : entier
        - neighbors : Tableau contenant des tableau associatif
        - cell_revealed : Booléen
Début
    cells_actions <- []
    index <- 0
    Tant que index < Taille(variables)

        neighbors <- get_surrounding_cells(game_param["height"], game_param["width"], variables[index])
        cell_revealed <- Faux
        Pour chaque neighbor dans neighbors:
            Si neighbor existe dans game_param["revealed"]:
                cell_revealed <- Vrai
                break
        Fin pour

        Si Vrai == cell_revealed
            number <- numbers[index]
            Si number >= 0.99
                // Poser un drapeau
                cells_actions ajoute [1, variables[index]]
            Sinon si number <= 0
                // Creuser
                cells_actions ajoute [0, variables[index]]
            Fin si
        FinSi
        index += 1
    Si (1 == Taille(variables) et 0 == Taille(cells_actions)):
        cells_actions ajoute [0, variables[0]]
    Retourne cells_actions
Fin

```



FS4.5 show_matrix

Afficher les différentes matrices lors de la résolution algébriques

% IN : [A, B]

- variables : Tableau associatif : Contient les variables
- matrix_A_B : Tableau à deux dimensions : Contient la matrice A et B
- numbers: Tableau à deux dimensions : résultat de la méthode de la pseudo inverse

% OUT : []

Pseudo-code :

```
Fonction pinv :
    Paramètre par valeur:
        - variables : Tableau associatif
        - matrix_A_B : Tableau à deux dimensions
        - numbers : Tableau à deux dimensions
Début
    Afficher variables
    Afficher matrix_A_B
    Afficher numbers
    Entrée // Sert de pause pour voir l'avancement
Fin
```

