ENSSAT

LANNION

Compte rendu de projet : Algorithmique 2

Lode Runner

Julien BOURDET

Florian ABADIE

ENSSAT

1ère année - Informatique

Lannion, January 2025

Table des matières

1

Introduction

Dans le cadre du module Algorithmique 2, nous avons pu appliquer les concepts étudiés en cours en réalisant un projet. Ce projet consiste à réaliser un jeu de démineur doté d'une intelligence artificielle (IA) capable de donner des conseils au joueur. Le démineur est un jeu qui demande logique et de stratégie qui se déroule sur une carte en deux dimensions où le joueur doit réussir à éviter les bombes.

L'objectif est de révéler toutes les cases d'une grille sans dévoiler une mine. Si le joueur essaie de révéler une case contenant une mine, la partie est perdue.

Au cours du développement, j'ai rencontré plusieurs défis. detailer les problèmes rencontrés

Ce rapport présente, dans un premier temps, les choix de conception du jeu, des structures de données aux algorithmes employés pour les fonctionnalités de base telles que l'affichage, la gestion des mines, et l'interaction utilisateur.

Dans un second temps, il détaille la stratégie adoptée pour concevoir une intelligence artificielle capable d'aider le joueur en identifiant les cases sûres à révéler, en s'appuyant sur des matrices d'adjacence et des modèles récursifs.

2

Préliminaires

Avant de commencer à détailler la stratégie utilisée, il est nécessaire de présenter certaines notions sur lesquelles elle repose.

2.1 Structure de données

Pour représenter la grille du jeu, nous avons décidé d'utiliser une sorte de liste chaînée. Chaque cellule de la grille est représentée par une structure `Cell` en C, qui contient les informations suivantes :

1. Les coordonnées de la cellule dans la grille, représentées par les entiers `x` et `y`.
2. Le nombre de mines adjacentes à la cellule, stocké dans `adjacentMines`.
3. Un booléen `isMine` qui indique si la cellule contient une mine.
4. Un booléen `isRevealed` qui indique si la cellule a été révélée.
5. Un booléen `isFlagged` qui indique si la cellule a été marquée d'un drapeau.
6. Un tableau de pointeurs vers les cellules adjacentes, stocké dans `adjacentCells`.
7. Un flottant `probability` qui stocke la probabilité que la cellule contienne une mine.

L'intérêt de stocker les cellules adjacentes dans un tableau de pointeurs est de pouvoir accéder facilement aux cellules voisines d'une cellule donnée. Cela facilite des algorithmes tels que la révélation des cellules adjacentes à une cellule vide, ou le calcul du nombre de mines adjacentes à une cellule donnée.

La structure `Cell` est définie en C comme suit :

```
1        typedef struct Cell {
2            int x;
3            int y;
4            int adjacentMines;
5            bool isMine;
6            bool isRevealed;
7            bool isFlagged;
```

```
8           struct Cell** adjacentCells;
9           float probability;
10      } Cell;
```

Listing 2.1: *Structure `Cell` en C.*

2.2 Cellules adjacentes

Dans un jeu de type démineur, une cellule peut avoir jusqu'à huit cellules adjacentes. Ces cellules sont celles qui partagent un côté ou un coin avec la cellule en question. Voici une illustration des cellules adjacentes pour une cellule située au centre d'une grille 3x3 :

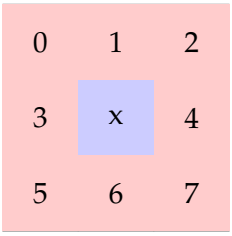

Figure 2.1: *Illustration des cellules adjacentes dans une grille 3x3. La cellule centrale est en bleu et les cellules adjacentes sont en rouge.*

Pour déterminer les cellules adjacentes d'une cellule située à la position (i, j) dans une grille, on peut utiliser les décalages suivants :

- $(i-1, j-1)$: en haut à gauche
- $(i-1, j)$: en haut
- $(i-1, j+1)$: en haut à droite
- $(i, j-1)$: à gauche
- $(i, j+1)$: à droite
- $(i+1, j-1)$: en bas à gauche
- $(i+1, j)$: en bas
- $(i+1, j+1)$: en bas à droite

Ces décalages permettent de parcourir toutes les cellules adjacentes à une cellule donnée dans la grille.

3

Modules

La stratégie se décompose en plusieurs modules, chacun ayant un rôle spécifique. Ces modules sont indépendants les uns des autres et peuvent être utilisés individuellement.

3.1 Liste des modules

3.1.1 Démineur

Prototype	Description
`void initialize_board()`	Initialise le plateau de jeu avec des cellules vides.
`void place_mines()`	Place les mines aléatoirement sur le plateau de jeu.
`void calculate_adjacent_mines()`	Calcule le nombre de mines adjacentes pour chaque cellule.
`bool reveal_cell()`	Révèle une cellule et retourne vrai si la cellule ne contient pas de mine.
`void flag_cell()`	Marque une cellule comme contenant potentiellement une mine.
`bool isWon()`	Vérifie si toutes les mines ont été correctement marquées et toutes les autres cellules révélées.
`bool isLost()`	Vérifie si une mine a été révélée, ce qui entraîne la perte de la partie.
`void print_board()`	Affiche le plateau de jeu à l'écran.

Table 3.1: *Fonctions du jeu de démineur.*

4

Présentation des modules

4.1 Module `initialize_board`

```
1        Cell* initialize_board(int n, int m);
```

Listing 4.1: *Prototype de `initialize_board` en C.*

4.1.1 Description

Cette fonction initialise le plateau de jeu avec des cellules vides. Elle alloue de la mémoire pour un tableau 3D de cellules et établit les liens entre les cellules adjacentes.

4.1.2 Paramètres

Paramètre	Type	Description
`n`	`int`	Nombre de lignes du plateau.
`m`	`int`	Nombre de colonnes du plateau.

Table 4.1: *Paramètres de la fonction `initialize_board`.*

4.1.3 Choix d'Implémentation

Nous avons choisi d'implémenter cette fonction pour créer dynamiquement un plateau de jeu de taille variable. Cela permet de gérer facilement les cellules et leurs relations adjacentes, ce qui est essentiel pour le fonctionnement du jeu de démineur.

4.1.4 Pseudo-code

```
1 fonction initialize_board(n, m):
2     allouer de la mémoire pour un tableau 3D de cellules appelé board
```

```
    pour i de 0 à n-1:
        allouer de la mémoire pour board[i]
        pour j de 0 à m-1:
            allouer de la mémoire pour board[i][j]
            initialiser board[i][j] avec :
                x = i
                y = j
                adjacentMines = 0
                isMine = faux
                isRevealed = faux
                isFlagged = faux
                allouer de la mémoire pour board[i][j].adjacentCells avec 8 éléments
                pour k de 0 à 7:
                    board[i][j].adjacentCells[k] = NULL

    pour i de 0 à n-1:
        pour j de 0 à m-1:
            si i-1 >= 0 et j-1 >= 0:
                board[i][j].adjacentCells[0] = board[i-1][j-1]
            si i-1 >= 0:
                board[i][j].adjacentCells[1] = board[i-1][j]
            si i-1 >= 0 et j+1 < m:
                board[i][j].adjacentCells[2] = board[i-1][j+1]
            si j+1 < m:
                board[i][j].adjacentCells[4] = board[i][j+1]
            si i+1 < n et j+1 < m:
                board[i][j].adjacentCells[7] = board[i+1][j+1]
            si i+1 < n:
                board[i][j].adjacentCells[6] = board[i+1][j]
            si i+1 < n et j-1 >= 0:
                board[i][j].adjacentCells[5] = board[i+1][j-1]
            si j-1 >= 0:
                board[i][j].adjacentCells[3] = board[i][j-1]

    origin = board[0][0]
    libérer la mémoire pour le tableau 3D board
    retourner origin
```

Listing 4.2: *Pseudo-code de la fonction* `initialize_board`.

4.2 Module `place_mines`

```c
void place_mines(Cell* origin, int numMines, int n, int m, int init_x, int
↪ init_y);
```

Listing 4.3: *Prototype de `place_mines` en C.*

4.2.1 Description

Cette fonction place les mines aléatoirement sur le plateau de jeu, en évitant la zone initiale autour de la première cellule révélée.

4.2.2 Choix d'Implémentation

Nous avons choisi d'implémenter cette fonction pour garantir que les mines soient placées de manière aléatoire tout en évitant la zone initiale autour de la première cellule révélée. Cela permet de donner au joueur une chance équitable de commencer le jeu sans perdre immédiatement.

4.2.3 Pseudo-code

```
fonction place_mines(origin, numMines, n, m, init_x, init_y):
    minesPlaced = 0
    tant que minesPlaced < numMines:
        x = nombre aléatoire entre 0 et m-1
        y = nombre aléatoire entre 0 et n-1

        si init_x - 1 <= x <= init_x + 1 et init_y - 1 <= y <= init_y + 1:
            continuer // Éviter la zone initiale

        cell = origin
        pour i de 0 à x-1:
            cell = cell.adjacentCells[6]
        pour j de 0 à y-1:
            cell = cell.adjacentCells[4]

        si cell.isMine == faux:
            cell.isMine = vrai
            pour chaque cellule adjacente dans cell.adjacentCells:
                si cellule adjacente != NULL:
                    cellule adjacente.adjacentMines += 1
            minesPlaced += 1
    fin tant que
```

Listing 4.4: *Pseudo-code de la fonction `place_mines`.*

4.3 Module `print_board`

4.4 Affichage récursif d'une grille de démineur

Dans un jeu de type démineur, l'affichage des cellules peut se faire de manière récursive. Lorsqu'une cellule est révélée, si elle ne contient pas de mines adjacentes, toutes ses cellules adjacentes sont également révélées. Ce processus se poursuit jusqu'à ce que toutes les cellules adjacentes sans mines soient révélées.

```c
void print_board(Cell* origin, int n, int m);
```

Listing 4.5: *Prototype de `print_board` en C.*

4.4.1 Description

Cette fonction affiche le plateau de jeu à l'écran, en montrant les cellules révélées, les mines, et les drapeaux.

4.4.2 Paramètres

Paramètre	Type	Description
`origin`	`Cell*`	Pointeur vers la cellule d'origine du plateau.
`n`	`int`	Nombre de lignes du plateau.
`m`	`int`	Nombre de colonnes du plateau.

Table 4.2: *Paramètres de la fonction `print_board`.*

4.4.3 Choix d'Implémentation

Nous avons choisi d'implémenter cette fonction pour permettre au joueur de visualiser l'état actuel du plateau de jeu. Cela inclut les cellules révélées, les mines et les drapeaux, ce qui est essentiel pour la prise de décision stratégique pendant le jeu.

4.4.4 Pseudo-code

```
fonction print_board(origin, n, m):
    imprimer "  " suivi des numéros de colonnes de 0 à m-1
    imprimer une nouvelle ligne

    rowStart = origin
    pour i de 0 à n-1:
        imprimer i suivi d'un espace
        cell = rowStart
        pour j de 0 à m-1:
```

```
            si cell.isRevealed:
                si cell.isMine:
                    imprimer "M "
                sinon:
                    imprimer cell.adjacentMines suivi d'un espace
            sinon si cell.isFlagged:
                imprimer "F "
            sinon:
                imprimer ". "
            cell = cell.adjacentCells[4]  // Aller à la cellule suivante à droite
        imprimer une nouvelle ligne
        rowStart = rowStart.adjacentCells[6]  // Aller à la ligne suivante en bas
```

Listing 4.6: *Pseudo-code de la fonction* `print_board`.

4.5 Module `calculate_adjacent_mines`

```
void calculate_adjacent_mines(Cell* board, int n, int m);
```

Listing 4.7: *Prototype de* `calculate_adjacent_mines` *en C.*

4.5.1 Description

Cette fonction calcule le nombre de mines adjacentes pour chaque cellule du plateau de jeu.

4.5.2 Paramètres

Paramètre	Type	Description
`board`	`Cell*`	Pointeur vers la cellule d'origine du plateau.
`n`	`int`	Nombre de lignes du plateau.
`m`	`int`	Nombre de colonnes du plateau.

Table 4.3: *Paramètres de la fonction* `calculate_adjacent_mines`.

4.5.3 Choix d'Implémentation

Nous avons choisi d'implémenter cette fonction pour déterminer le nombre de mines adjacentes à chaque cellule. Cette information est cruciale pour le joueur afin de prendre des décisions éclairées sur les cellules à révéler ou à marquer.

4.5.4 Pseudo-code

```
fonction calculate_adjacent_mines(board, n, m):
    pour i de 0 à n-1:
```

```
        pour j de 0 à m-1:
            cell = board[i][j]
            cell.adjacentMines = 0
            pour chaque cellule adjacente dans cell.adjacentCells:
                si cellule adjacente != NULL et cellule adjacente.isMine:
                    cell.adjacentMines += 1
```

Listing 4.8: *Pseudo-code de la fonction* `calculate_adjacent_mines`.

4.6 Module `reveal_cell`

```
bool reveal_cell(Cell* board, int x, int y, int n, int m);
```

Listing 4.9: *Prototype de* `reveal_cell` *en C.*

4.6.1 Description

Cette fonction révèle une cellule et retourne vrai si la cellule ne contient pas de mine.

4.6.2 Paramètres

Paramètre	Type	Description
`board`	`Cell*`	Pointeur vers la cellule d'origine du plateau.
`x`	`int`	Coordonnée x de la cellule à révéler.
`y`	`int`	Coordonnée y de la cellule à révéler.
`n`	`int`	Nombre de lignes du plateau.
`m`	`int`	Nombre de colonnes du plateau.

Table 4.4: *Paramètres de la fonction* `reveal_cell`.

4.6.3 Choix d'Implémentation

Nous avons choisi d'implémenter cette fonction pour permettre au joueur de révéler une cellule. Si la cellule ne contient pas de mine, elle est révélée et, si elle n'a pas de mines adjacentes, les cellules adjacentes sont également révélées récursivement.

4.6.4 Pseudo-code

```
fonction reveal_cell(board, x, y, n, m):
    cell = board
    pour i de 0 à x-1:
        cell = cell.adjacentCells[6]
    pour j de 0 à y-1:
        cell = cell.adjacentCells[4]
```

```
7
8       si cell.isMine:
9           retourner faux
10      sinon:
11          cell.isRevealed = vrai
12          si cell.adjacentMines == 0:
13              pour chaque cellule adjacente dans cell.adjacentCells:
14                  si cellule adjacente != NULL et non cellule adjacente.isRevealed:
15                      reveal_cell(board, cellule adjacente.x, cellule adjacente.y, n, m)
16          retourner vrai
```

Listing 4.10: *Pseudo-code de la fonction `reveal_cell`.*

4.7 Module `flag_cell`

```c
void flag_cell(Cell* board, int x, int y);
```

Listing 4.11: *Prototype de `flag_cell` en C.*

4.7.1 Description

Cette fonction marque une cellule comme contenant potentiellement une mine.

4.7.2 Paramètres

Paramètre	Type	Description
`board`	`Cell*`	Pointeur vers la cellule d'origine du plateau.
`x`	`int`	Coordonnée x de la cellule à marquer.
`y`	`int`	Coordonnée y de la cellule à marquer.

Table 4.5: *Paramètres de la fonction `flag_cell`.*

4.7.3 Pseudo-code

```
fonction flag_cell(board, x, y):
    cell = board
    pour i de 0 à x-1:
        cell = cell.adjacentCells[6]
    pour j de 0 à y-1:
        cell = cell.adjacentCells[4]

    cell.isFlagged = vrai
```

Listing 4.12: *Pseudo-code de la fonction `flag_cell`.*

4.8 Module isLost

```
1        bool isLost(Cell* board, int n, int m);
```

Listing 4.13: *Prototype de isLost en C.*

4.8.1 Description

Cette fonction vérifie si une mine a été révélée, ce qui entraîne la perte de la partie.

4.8.2 Paramètres

Paramètre	Type	Description
board	Cell*	Pointeur vers la cellule d'origine du plateau.
n	int	Nombre de lignes du plateau.
m	int	Nombre de colonnes du plateau.

Table 4.6: *Paramètres de la fonction isLost.*

4.8.3 Pseudo-code

```
1  fonction isLost(board, n, m):
2      pour i de 0 à n-1:
3          pour j de 0 à m-1:
4              cell = board[i][j]
5              si cell.isRevealed et cell.isMine:
6                  retourner vrai
7      retourner faux
```

Listing 4.14: *Pseudo-code de la fonction isLost.*

5

Stratégie

Même si le démineur est un jeu simple, une IA parfaite n'aurait pas 100% de victoire. En effet, c'est un jeu de probabilités, et il existe donc des situations dans lesquelles la possibilité de trouver un bombe dans chaque case non-révélée est équiprobable. Dans ce cas, l'IA doit faire un choix aléatoire.
Notre stratégie sera donc de calculer la probabilité de chaque case non-révélée de contenir une bombe, et de choisir une des cases avec la plus faible probabilité.

5.1 Preliminaires

Dans toute la suite de ce chapitre, on considère une grille de démineur de taille $n \times m$ avec b bombes, et on note G la grille, $G_{i,j}$ la case à la ligne i et la colonne j.
$G_{i,j} = -1$ si la case n'est pas révélée, $G_{i,j} = k$ si la case est révélée et contient k bombes autour d'elle.

5.2 Algorithme naïf

Calculer la probabilité de chaque case non-révélée de contenir une bombe est une tâche complexe : une information à un bout de la grille peut influencer une case à l'autre bout.
Intuitivement, on commence par essayer de faire du dénombrement : on calcule toutes les combinaisons possibles de bombes, et ne garde que les combinaisons valides, et on compte le nombre de fois où chaque case est une bombe.

5.2.1 Dénombrement

On construit un algorithme qui, pour une grille G donnée calcule toutes les combinaisons possibles de bombes, et ne garde que les combinaisons valides.
Il y a deux difficultés majeures à résoudre :

- Comment valider une combinaison ?
- Comment éviter de tester des combinaisons qui sont évidemment invalides ?

Pour répondre à ces problèmes, on introduit une idée simple, mais qui nous sera utile dans la suite : on va grouper des cases ensemble.
On définit un groupe comme un ensemble de cases non-révélées.
Dans cet algorithme, pour chaque case révélée, on crée un groupe avec les cases adjacentes.
On peut alors, pour chaque combinaison de 2 groupes, trouver l'intersection de ces derniers, et poser une condition sur les cases de cette intersection.

5.2.2 Pseudo-code

```
1   Fonction Probabilite
2   Parametres :
3       n en entier, m en entier
4       G en matrice d'entiers de taille n x m
5   Declarations :
6       groupes en liste de listes de couples d'entiers
7       intersections en liste de listes de couples d'entiers
8       P en matrice d'entiers
9       combinaisons en liste de listes de couples d'entiers
10      combinaisons_valides en entier
11  Sortie :
12      P en matrice de reels de taille n x m
13  Debut
14      P = matrice de 0 de taille n x m
15
16      groupes = liste des groupes des cases révélées
17
18      intersections = liste des intersections de chaque combinaison de 2 groupes
19
20      combinaisons = []
21      Pour k allant de 1 a |groupes| faire
22          Ajouter a combinaisons une liste de toutes les combinaisons possibles
            ↪ de G[i][j] bombes dans les cases de groupes[k]
23      FinPour
24
25      combinaisons_valides = 0
26      Pour chaque element du produit cartesien de combinaisons faire
27          Pour chaque intersection dans intersections faire
28              Si l'intersection contient plus de bombes que le nombre de bombes
                ↪ autour de chaque case dans l'intersection alors
29                  Continuer
30              FinSi
31          FinPour
32
33          Pour chaque case dans l'element faire
34              P[case] += 1
35          FinPour
36          combinaisons_valides += 1
37      FinPour
```

```
38
39              Pour i allant de 1 a n faire
40                  Pour j allant de 1 a m faire
41                      Si G[i,j] = -1 alors
42                          P[i,j] /= combinaisons_valides
43                      FinSi
44                  FinPour
45              FinPour
46
47              Retourner P
48          Fin
```

Listing 5.1: *Pseudo-code de l'algorithme naif de calcul de probabilité*

5.2.3 Complexité

La complexité de cet algorithme est très élevée, car il faut calculer le produit cartésien de toutes les combinaisons possibles de bombes : la complexité est exponentielle en le nombre de cases non-révélées. Cet algorithme est donc inutilisable pour des grilles de taille moyenne.
Si on veut calculer les probabilités de chaque case, il va falloir radicalement optimiser cette stratégie.

5.3 Algorithme optimisé

Pour optimiser cet algorithme, on va représenter notre problème sous une forme différente.
On commence par définir un graphe G', où chaque sommet est une case, et où il y a une arête entre deux cases si elles sont adjacentes. On construit alors la matrice A de taille $n * m \times n * m$ telle que $A_{(i,j),(i',j')} = 1$ si (i, j) et (i', j') sont adjacents, et 0 sinon. C'est la matrice d'adjacence de G'.
On remarque alors que l'ont peut très facilement retrouver la matrice G complète (c'est à dire avec toutes les cases révélées) à partir de A et de la position des bombes.

Notons B la colonne de taille $n * m$ telle que $B_i = 1$ si la case i contient une bombe, et 0 sinon. De même, soit v la colonne de taille $n * m$ telle que $v_i = G_{i,j}$ si la case i est révélée, et -1 sinon.

On a alors la relation suivante :

$$A \cdot B = v$$

Cette relation est un système linéaire, que l'on peut résoudre pour trouver toutes les colonnes B qui satisfassent v (c'est à dire toutes les positions possible de bombes qui respectent les cases révélées). Nous pouvons alors calculer la probabilité de chaque

case non-révélée de contenir une bombe.

5.3.1 Optimisations

Représenter le problème sous forme de système linéaire ne change pas la complexité de la résolution du problème, résoudre le système entier revient à la même complexité que l'algorithme naïf. Mais cette représentation nous permet de faire des optimisations qui vont nous permettre de réduire la complexité de l'algorithme.

Prenons un exemple simple :

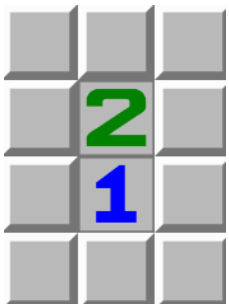

Figure 5.1: *Exemple de grille.*

On obtient les matrices suivantes : (A est symétrique, on ne montre que la moitié de la matrice)

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ & & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ & & & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ & & & & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ & & & & & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ & & & & & & 0 & 1 & 0 & 1 & 1 & 0 \\ & & & & & & & 0 & 1 & 1 & 1 & 1 \\ & & & & & & & & 0 & 0 & 1 & 1 \\ & & & & & & & & & 0 & 1 & 0 \\ & & & & & & & & & & 0 & 1 \\ & & & & & & & & & & & 0 \end{pmatrix} \qquad v = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \\ 2 \\ -1 \\ -1 \\ 1 \\ -1 \\ -1 \\ -1 \\ -1 \end{bmatrix}$$

On commence par ne garder que les lignes de A et de v qui correspondent à des cases révélées. On peut enlever ces lignes car on ne connait pas la valeur de ces cases dans v, les utiliser pour résoudre le système ne nous apporterait aucune information.

On enleve aussi les colonnes de A qui correspondent à des cases révélées, car il ne peut y avoir de bombes dans ces cases.

On obtient alors :

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \qquad v = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

On remarque alors que plusieurs colonnes de A sont identiques, on va donc les regrouper. Ce regroupement n'est pas intuitif, on est en fait en train de grouper les cases qui sont influencées par les mêmes cases révélées. Ce dernier est possible car on sait que chaque case d'un groupe à la même probabilité de contenir une bombe.
Dans notre exemple, on obtient ces groupes :

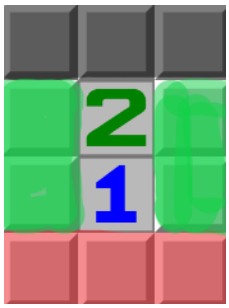

Figure 5.2: *Grille avec les groupes en couleur.*

On groupe alors ces colonnes :

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \qquad v = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Trouver un solution à ce système revient à distribuer des bombes dans les groupes. On connait le nombre de cases dans chaque groupe, on peut alors facilement calculer la probabilité de chaque case de contenir une bombe.

On utilise l'algorithme de Gauss-Jordan pour réduire la matrice A en forme échelonnée réduite :

$$A = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & 1 \end{pmatrix} \qquad v = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Dans un cas classique, ce système aurait une infinité de solutions, mais dans notre cas, on peut mettre des bornes sur les variables (le nombre de bombes par groupe). Initialement, ces bornes sont $[0, |groupe|]$, mais on peut les réduire grâce au système $A|v$.
On cherche alors les variables libres, c'est à dire les colonnes de A qui ne sont pas des pivots. Les solutions sont alors les combinaisons de ces variables qui respectent les bornes.

Dans notre exemple, on a une seule variable libre, la dernière colonne de A. Les solutions sont alors :

$$\begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} \qquad \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

Retrouver la probabilité de chaque case revient à trouver la probabilité de chaque solution. On obtient alors :

$$\mathbb{P}(\text{Solutions}_k) = \prod_{i=1}^{|\text{Groupes}|} \binom{|\text{Groupe}_i|}{\text{Solution}_i}$$

$$\mathbb{P}(\text{Groupes}_k) = \sum_{i=1}^{|\text{Solutions}|} \frac{\mathbb{P}(\text{Solutions}_i) \times \text{Solution}_k}{|\text{Groupes}_k|}$$

Finalement, toute les cases d'un groupe ont la probabilité de ce groupe de contenir une bombe.

6

Conclusion

Ce projet nous a permis de concevoir et d'implémenter une intelligence artificielle pour jouer au jeu Démineur. Malgré des contraintes, nous avons pu élaborer une stratégie robuste basée sur des algorithmes de recherche et de probabilités pour gérer au mieux les situations rencontrées. Nous avons également pu mettre en place une interface graphique pour visualiser le jeu et les actions de l'IA.