Rapport démineur

Quentin BRULÉ et Lucie BIHET

Janvier 2025

Table des matières

1 Introduction

Le démineur est un jeu qui se joue seul face à une grille de N par N cases contenant P mines. Chaque case est d'abord caché et le joueur peut les découvrir une par une. Le but du jeu est de découvrir toutes les cases qui n'ont pas de mines.
S'il découvre une mine c'est perdu pour lui. Il peut poser des drapeaux sur les cases qu'il considère être une mine. De plus, chaque case découverte indique le nombre de mines qui lui sont adjacentes dans les 8 cases qui l'entoure.
Il nous ai demandé de créer le jeu du démineur jouable dans le terminal. Le principe sera d'afficher la grille de la partie en cours au joueur qui aura le choix entre 4 possibilités : découvrir une case, poser un drapeau, retirer un drapeau et demander conseil à une IA sur le coup à jouer. L'IA doit utiliser un arbre de décision pour fonctionner.

2 Fonctionnalités et choix algorithmiques

2.1 Structures créées

```
typedef struct {
  bool flag;
  bool decouvert;
  int interieur;
} carre;
```

Cette structure permet de définir toutes les propriétés nécessaires pour une case du jeu, c'est à dire si elle possède un drapeau (true ou false), si elle a été découverte (true ou false) et ce qu'elle contient (une valeur ou une bombe).
Le jeu commence donc avec toutes les cases sans drapeau, non découverte et avec leur valeur assignée.

2.2 Fonctionnalités

C'est un jeu basique de démineur. La partie commence avec une grille donc toutes les cases sont non découvertes, les lignes et colonnes sont numérotés et on demande au joueur l'action qu'il souhaite réaliser.

On reviendra au cas de l'IA dans la partie sur la stratégie.
Pour toutes les autres actions, on va demander au joueur de choisir la ligne et la colonne de l'action qu'il veut réaliser. S'il choisi une action impossible ou fait une mauvaise entrée, il lui sera demandé de recommencer.

La partie continue tant que le joueur n'a pas découvert de mines ou n'a pas découvert toutes les cases n'en contenant pas.

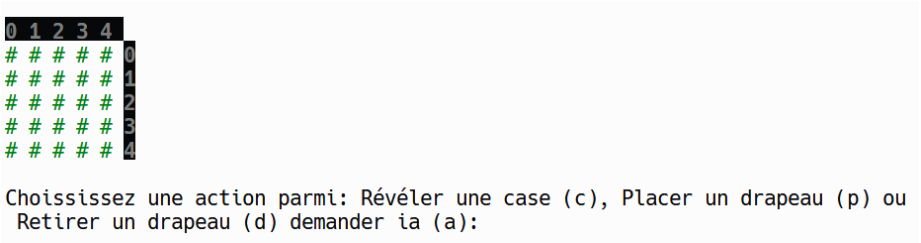

FIGURE 1 – Vue du début du jeu

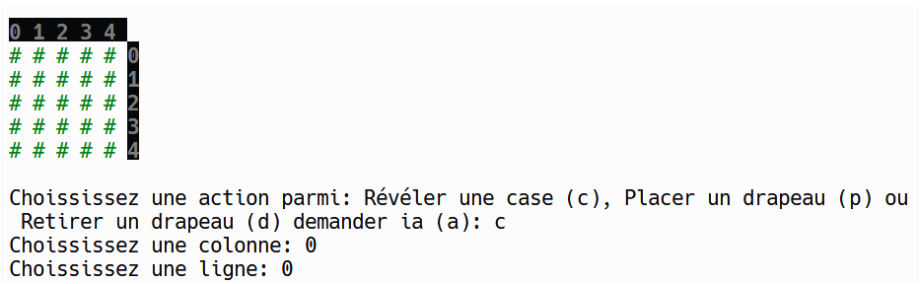

FIGURE 2 – Choix de la case

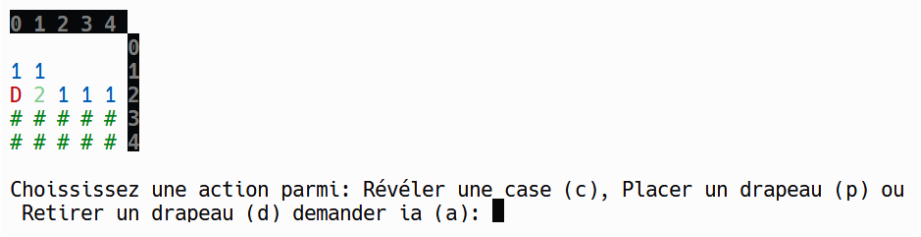

FIGURE 3 – Exemple d'interface en cours de jeu

Lorsque celle ci se termine, on a le choix de recommencer le jeu avec une nouvelle grille ou de quitter la partie.

2.3 Choix algorithmiques

Les fonctions `jeu` et `recommencer` sont des fonctions récursives croisées : `jeu` appelle `recommencer` à la fin de la partie, et `recommencer` appel `jeu` si l'utilisateur veut refaire une partie. Ce choix d'implémentation permet de séparer les différentes fonctionnalités du programme pour une meilleure lisibilité. Elles vont donc toutes les deux être des procédures car n'ont pas besoin de renvoyer quelque chose.

Procédure ou Fonction :

Les procédures sont utilisées pour les actions qui modifient l'état du jeu car elles vont faire des actions sur des éléments passé en paramètre par adresse comme pour modifier la grille qui est passée par adresse ou pour des fonctions d'affichage sur le terminal qui ne vont donc rien renvoyer. On retrouve donc : `afficherCarre`, `afficherGrille`, `detruireGrille`, `poseMines`, `remplissage`, `poseDrapeau` , `retireDrapeau`, `propagationDuZero`, `poseMinesCoup1`, `gestionPremierCoup`.

Les fonctions, en revanche, retournent une valeur sans modifier l'état global (e.g., indice_indexation, nbMinesVoisines), car elles sont destinées à des calculs simples et des traitements spécifiques comme compter des mines, créer la grille ou donner la bonne couleur. Elles peuvent aussi servir à assurer le bon fonctionnement de la boucle de jeu principale comme pour dans `dialogueUtilisateur` qui renverra un booléen. Ces fonctions sont :
`couleurNumero`, `indice_indexation_auxiliaire`, `indice_indexation`, `creerGrille`, `nbMinesVoisines`, `finDuJeuPerdu`, `finDuJeuGagne`, `toutesCasesDecouvertes`, `decouvreCase` , `dialogueUtilisateur`.

Boucles for et while :

Ces structures permettent de parcourir la grille et d'effectuer des opérations sur toutes les cases. Elles sont essentielles pour les tâches répétitives comme la génération de mines, la découverte des cases voisines ou le remplissage des valeurs des mines.

Récursivité (`propagationDuZero` et `indice_indexation_auxiliaire`) :

La récursivité est choisie ici pour traiter efficacement les cases voisines d'une case contenant le chiffre zéro. Cela permet de découvrir toutes les cases adjacentes sans avoir à itérer à travers chaque case.
Pour la deuxième fonction, la récursivité permet de gérer efficacement la recherche pour éviter une complexité excessive

3 Stratégie

3.1 Structures créés

Cette structure permet de définir l'arbre qui servira pour le fonctionnement de l'IA.
La structure se compose d'une case avec les coordonnées entières x et y, puis le nœud possède 2 enfants, une dans le cas où la case est considérée sûre et une si elle est considérée comme possédant une mine. Cela va permettre de créer des scénario testant toutes les possibilités de position de mine. On verra cela en

```c
struct arbre_s {
  int x;
  int y;
  struct arbre_s *mine;
  struct arbre_s *sure;
};
typedef struct arbre_s *arbre;
```

détail par la suite.

Remarque : La structure d'arbre a été abandonné peu de temps avant le rendu du projet car l'IA fonctionne aussi bien sans que avec et utiliser l'arbre ne faisait que ralentir le jeu. La méthode choisie n'utilise donc pas d'arbre mais se repose bien sur les même fonctionnalité d'un arbre. Effectivement, le backtracking va bien visiter toutes les possibilités de manière binaire, tout en visitant toutes les solutions de manières systématiques avec une structure arborescente.

3.2 Fonctionnement de l'IA

L'IA fonctionne selon le principe du backtracking : la solution est construite petit à petit, en explorant toutes les possibilités. Pour une case donnée, si elle n'est pas encore découverte, nous n'avons pas d'information à son propos : on suppose qu'elle contient une mine, on essaye de construire récursivement le reste de la solution, puis on suppose qu'elle ne contient pas de mine, et on essaye de construire récursivement le reste de la solution. Étant donnée qu'on ne peut pas toujours être sûr quand on joue, il risque d'y avoir plusieurs solutions possibles : pour remédier à ce problème, nous supposons que les cases ont plus de chance d'être dans l'état dans lequel elles sont si elle le sont dans la majorité des cas simulés : l'IA propose donc le coup qui possède la plus grande majorité.

3.3 Choix algorithmiques

Nous avons choisis d'implémenter l'IA en utilisant le backtracking car cela reprend les notions du cours d'algorithmique 2 : les fonctions récursives et les arbres. La structure d'arbre est ici implicite : les nœuds sont en faite les appels récursif et les étiquette sont les 3 premiers paramètres de la fonction `backtracking` : la grille dans cette éventualité, les entiers x et y qui représentent les coordonnées de la case qui n'est pas encore déterminée. Pour prendre une décision, nous n'avons besoin que des feuilles de cette arbre, car à ce moment là, toutes les cases de la grille ont été déterminées, et nous savons que nous sommes dans une feuille lorsque les coordonnées x et y sont toutes les deux en dehors de la grille, c'est donc à ce moment là que nous mettons à jour `grille_out`, en incrémentant les cases sûres et en décrémentant les cases minés.

3.4 Taux de victoire

La victoire est quasiment assurée grâce à l'IA. Cependant, à certains moments on tombera forcément sur des cas où le positionnement des mines tombent sous la probabilités du 50/50. C'est à dire qu'il n'y a aucun moyen de savoir si la mine est dans une case ou l'autre.

3.5 Limites de l'IA

Une fois le programme terminé, nous nous sommes rendus compte que nous n'utilisions pas vraiment l'arbre que nous avons construit : nous stockons les appels récursifs mais ils ne servent à rien car nous n'avons pas besoin de cet objet pour déterminer si nous avons atteint la fin du parcours de la grille. Cependant, le fonctionnement de notre IA suit le même principe qu'un arbre, c'est à dire que l'on suit les "scénarios" possibles (cases avec ou sans mine) à travers des branches. C'est comme un arbre où chaque nœud possède 2 enfants. Nous avons décidé de retirer la structure d'arbre pour alléger le programme.

De plus, la recherche dans la grille sont très lourdes algorithmiquement parlant car on visite toutes les possibilités possible. On se retrouve donc, si on lance l'IA avant d'avoir creuser de cases, en complexité $2^{GRILLE \times GRILLE}$ où GRILLE est la taille de la grille. Il faut donc lancer le jeu sur de plus petite grille ou n'appeler l'IA que quand assez de cases sont creusées. Le temps d'exécution de l'IA va évidemment aussi dépendre du nombre de mines présentes sur la grille.

3.6 Pour aller plus loin

Nous aurions pu faire un arbre général qui est construit une fois au début de la partie, et dont on supprime les branches au fur et à mesures que les cases sont révélées. Le problème avec cette méthode est que le démarrage est lent : il a besoin d'allouer $2^{GRILLE \times GRILLE}$ nœuds pour construire l'arbre de départ, et cela prends beaucoup de temps avec une grande grille.

Il pourrait aussi y avoir des moyens de décomplexifier l'algorithme comme en ne prenant en compte seulement les cases "relativement" proche de cases déjà découvertes (si l'on est trop loin du moindre "indice", cela devient vite inintéressant de l'étudier).

4 Conclusion

La résolution du démineur est considéré comme un problème NP-complet. Dans notre cas, on doit donc parcourir toutes les possibilités de positionnement de mines afin de s'assurer qu'une case est sûre ou minée. Cela demande beaucoup en terme de complexité au programme qui recalcule un arbre de décision

complet à chaque tour.
Comme présenté précédemment, finalement nous n'utilisons pas vraiment l'arbre pour décider de quel coup jouer mais plutôt pour suivre en parallèle du backtracking la recherche des possibilité.