Compte rendu Projet Algorithmique 1

Arthur BLAMART

Table des matières

1 Introduction

L'objectif de ce projet d'algorithmie est de coder en language C l'intelligence artificielle d'un joueur de bomberman, c'est à dire un programme de décision. Un moteur de jeu nous est fournit par l'intermédiaire d'un fichier objet il ne reste plus qu'à implémenter un programme qui en fonction des différents paramètres connus prendra une décision d'action parmis les 5 suivantes : aller au nord, au sud, à l'ouest, à l'est ou poser une bombe. Pour pouvoir penser cette intelligence il faut d'abord prendre en compte les différentes contraintes et règles qui s'appliquent, tout d'abord il-y-a les règles du jeu Bomberman. Bomberman est un jeu ou un joueur peut se déplacer dans les 4 principales directions et peut poser des bombes, perdu dans un labyrinthe, son but est de fuir ou vaincre les monstres présents pour trouver la sortie cachée dans un mur, cassable grâce à ses bombes explosant en croix, il sera, de plus, aidé des différents bonus récupérables dans les murs cassables.

Les contraintes du projet sont toutes aussi importantes, à chaque tours de jeu une multitude de paramètres sont envoyées à notre programme qui en fonction de ceux-ci devra décider de l'action à faire à ce tour de jeu. Le premier point à souligner est que nous ne pouvons sauvegarder aucune donnée à travers des différentes executions de notre programme, nous ne pouvons pas, par exemple, demander à notre programme de sauvegarder le chemin parcouru depuis le début, la seule chose dont nous disposons est à chaque tours de la dernière action effectuée. Pour ce qui est des autres paramètres nous disposons d'une matrice de caractère représentant la carte du jeu, la taille de celle ci, le nombre de bombe restantes à notre disposition (nous ne pouvons qu'avoir un certains nombre de bombes en simultané sur la carte), le rayon d'explosions d'une bombe, les significations des différents caractères sur la matrice de la carte du jeu.

Tout l'enjeux du projet est de coder une intelligence artificielle qui puisse avant tout terminer les différents niveaux, et ensuite maximiser le score qui est obtenue en détruisant des murs, des enemies ou en ramassant des bonus. J'ai donc implémenter un programme qui vérifie avant tout si elle est en sécurité, puis ensuite qui va chercher des bonus ou la sortie qui va s'y diriger si elle sont accessible, dans le cas contraire mon programme va chercher les murs cassable les plus proches pour pouvoir aller les détruires, ouvrant ainsi de nouvelles possibilitées.

2 Préliminaires

Pour comprendre les algorithmes mit en place, les notion de graphe non-orienté, de file, et de parcours en largeur ou Breadth First Search (BFS), sont primordiales.

Graphe : Le graphe est une structure de données qui permet de représenter des liens entre différents éléments et permet des opérations intéréssantes comme par exemple le parcours en largeur, à l'aide de sommets, ou node, représentant les objets et des arêtes, ou vertex, représentant les liens les unissant.

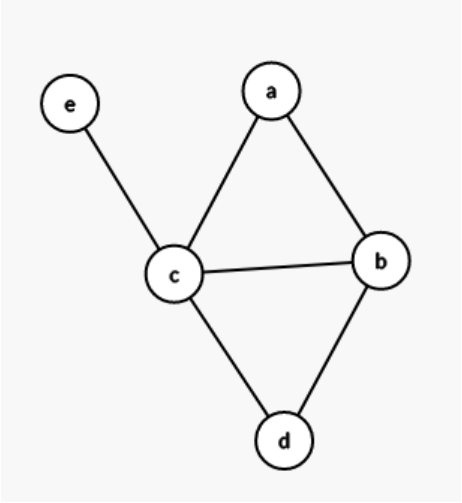

Figure 1: Exemple d'un graphe à 5 sommets et 6 arêtes

File : La file est un type abstrait basé sur le principe First in First out (FIFO) qui sert à stocker des données. On peut la comparer à une file dans un magasin par exemple, des gens rentrent continuellement dans la file à l'arrière et le premier qui sort c'est celui qui est entrée en premier, puis le suivant. Les primitives associées sont :

- Enfile : Ajoute un objet à la fin de la file
- Défile : Retire le premier élément de la file
- Premier : Obtenir le premier élement de la file
- EstVide : Es-ce qu'une file est vide ?
- FileVide : Créer une file vide

Parcours en profondeur : Le parcours en profondeur permet d'obtenir une liste de sommet d'un graphe donné, tel que plus le sommet est "loin" dans la liste plus il est loin du point de départ du parcours.
Dans l'exemple précédent [e, c, a, d, b] est un exemple de parcours en profondeur.

3 Stratégie générale

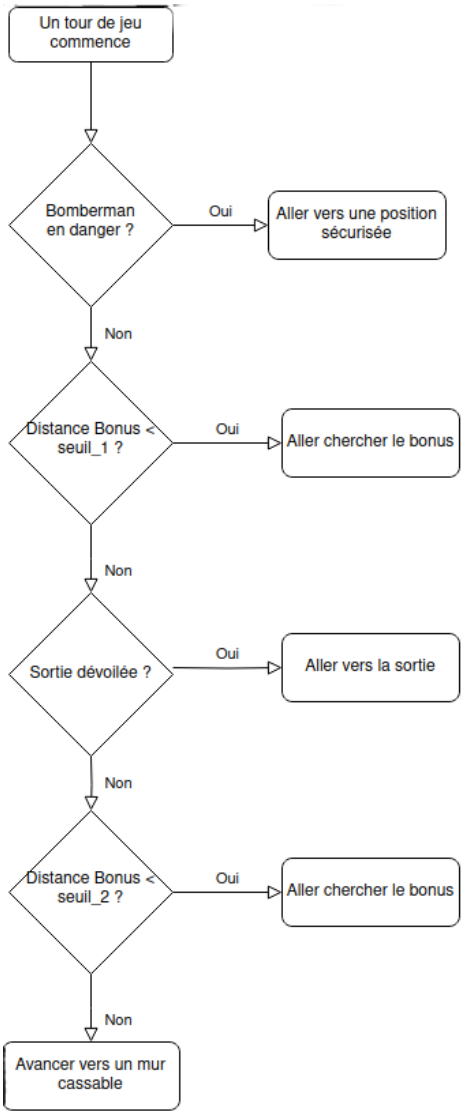

Figure 2: Logigramme de décision

Toute la stratégie tourne autour d'un graphe qu'on construit de tel sorte : les sommets sont les différents cases chemin de la carte de jeu, c'est à dire toutes les cases sur lesquelles le joueur peut marché donc ça inclus les monstres et les bonus, et les arêtes sont définies ainsi : soit a,b deux arêtes du graphe, il y a

une arête entre a et b si et seulement si on peut aller de a à b en un mouvement, donc si a et b sont contigüe. Ce graphe est implémenté par matrice d'adjacence et est manipulable à l'aide de diverses fonctions qui permettent d'ajouter des cases dans le graphe et qui se redimensionne automatiquement si nécessaire.
Les cases sont représenté par des **Objets** qui ont un caractère (celui indiqué sur la map) et une position, ces objets couplés à un **id ou nb**, représentent des sommets dans le graphe, le graphe qui contient une liste de sommets et une matrice d'adjacence qui sont utilisable grâce aux identifiants des sommets, le graphe mémorise aussi **n** : le nombre de sommet courant et **nmax** le nombre de sommet maximum afin de se redimensionner automatiquement si n=nmax.

```
typedef struct objet{
  char quoi;
  int x;
  int y;
}objet;

typedef struct sommet{
  int nb;
  objet quoi;
}sommet;

typedef struct graphe{
  int nmax;
  int n;
  int** mat;
  sommet* sommetList;
}graphe;
```

A chaque tours de jeu on lance 4 parcours en largeur de graphe qui partent des 4 cases entourant le joueur, celle du nord, du sud, de l'ouest et de l'est. Nous allons pouvoir utiliser les quatres parcours pour essayer de trouver les élements qui nous intéressent, c'est à dire : des bonus, la sortie ou encore des murs cassables, nous allons donc parcourirs les quatres parcours en parallèle et dès que l'on trouve l'élèment cherché dans un parcours nous connaissont donc le mouvement le plus pertinent pour s'en rapprocher. Nous lançons tous les testes et les résultats nous sont rendus sous forme d'action qui nous permet de directement connaître l'action à effectuer, il y a aussi une valeur par défaut dans le cas où le teste à échouer, par exemple si on ne trouve pas de sortie, au lieu de renvoyer une direction au hasard on renverra BOMBING

Recherche de danger et fuite Avant toute chose, on souhaite vérifier que la position dans laquelle se trouve le joueur n'est pas dangereuse, es-ce qu'une bombe nous met en danger, est-elle sous nos pieds ou encore es-ce qu'un monstre est trop proche du joueur, pour cela on va vérifier dans un certains rayon qu'il n'y ait pas de monstre, en parrallèle nous allons regardé aux 4 points cardinaux si une bombe ne nous menace pas, prenant en compte son rayon d'explosion qui

peut varier avec les bonus. Si pour une raison ou pour une autre un danger est detecté nous utilisons les 4 parcours pour nous diriger vers une case à l'abris de tout danger, pour savoir si une case est sans danger, il suffit de refaire le même teste que précédemment sur celle-ci. Il faudra de plus faire attention à avoir au minimum 2 cases sécurisée contigüe, car si on est forcé de retourner sur la case dangereuse au prochain tours, car on est obligé de faire un mouvement, et bien on perdras

Recherche de bonus Pour la recherche de bonus on utilise le graphe pour trouver une liste de tous les bonus présent sur la map et on utilise les 4 parcours pour savoir si ils sont assez proche.

Recherche de sortie Pour la recherche de sortie, on regarde nos 4 parcours pour regarder si on trouve la sortie dans l'un d'eux, si c'est le cas le premier parcours où on l'a trouvée nous permet de savoir vers où nous diriger.

Replacement intelligent Si on n'a rien trouvé d'autre à faire on va vouloir aller casser des murs pour se créer des opportunités, découvir des bonus ou la sortie, pour cela on utilise encore une fois nos 4 parcours en regardant à chaque case si une bombe serait utile, c'est à dire, compte tenue de la portée de nos bombes, es-ce qu'une bombe posée ici pourrait toucher un mur, si c'est le cas on la pose.

4 points clés

Construction du graphe : Le premier point algorithmique clé est la construction du graphe. Nous utiliserons d'abord une fonction CreerGraphe à qui à partir de la carte de jeu et de ses dimensions nous renverras le graphe, cette fonction utilisera la procédure AjouterDansGraphe qui ajoute un objet dans un graphe. On partiras aussi du principe qu'on a une fonction qui nous permet de créer l'arêtes entre 2 sommets appelé a et b dans un graphe g appelé AjouterVoisin(g, (a.x,a.y), (b.x, b.y)).

CreerGraph : Creer graphe est une fonction car on veut obtenir un pointeur sur un nouveau graphe. Pour obtenir ce graphe on va d'abord l'initialiser puis parcourir la carte de jeu, de gauche à droite puis de bas en haut en essayant d'identifier toutes les cases définies comme chemin, cela inclut les monstres et les bonus car on peut marcher sur leurs cases, à chaque fois qu'on en trouve une on l'ajoute dans notre graphe et on va regarder si les cases adjacentes à gauche et au dessus de celle-ci sont aussi des chemins car si elle sont adjacentes c'est qu'il faut créer une arêtes entre notre nouveau sommet et ceux correspondants à ces deux cases et par construction du graphe ce sont les seuls cases adjacente qui ont déjà été mises dans le graphe. pour le parcours du chemin on choisit d'utiliser deux boucle for car on veut parcourir l'entiereté de la carte de bas en haut et de gauche à droite et aillant deux entier x,y qui nous permettent de connaitre notre actuelle position dans la carte.

```
Fonction CreerGraph en pointeur sur graphe
Declaration
  Parametre map en matrice de caractere, tailleX en entier, tailleY en entier
  Variable g en pointeur sur graphe, temp en objet
Debut
        g <- grapheVide()
      Pour y allant de 0 a tailleY-1
            Pour x allant de 0 a tailleX-1
                  Si map[y][x] = chemin Alors
                        temp.caractere = map[y][x]
                        temp.x = x
                        temp.y = y
                        AjouterDansGraphe(g, temp)
                        Si map[y-1][x] Alors
                              AjouterVoisin(g, (x,y) (x,y-1))
                        FinSi
                        Si map[y][x-1] Alors
                              AjouterVoisin(g, (x,y) (x-1,y))
                        FinSi
                  FinSi
            FinPour
```

```
        FinPour
        Retourner g
Fin
```

AjouterDansGraphe : Pour ajouter un sommet temp dans notre graphe on choisit une procédure car on veut modifier un graphe et donc agir au travers d'un pointeur sur la mémoire. Pour cela on regarde d'abord si le graphe n'a pas atteint sa taille maximum, le cas échéant on doit augmenter la taille du graphe grâce, notamment, grace à de la réallocution de mémoire. On peut ensuite incrémenter la taille du graphe et ajouter le sommet dans le graphe.

```
Procedure AjouterDansGraphe
Decalaration
        Parametre g pointeur sur graphe, temp en objet
        Variable nouveauSommet en sommet
Debut
        Si g->taille >= g->tailleMax Alors
                AugmenterTailleGraphe(g)
        FinSi
        nouveauSommet.id = g->taille
        nouveauSommet.objet = temp
        g->listeSommet[g->taille] <- Ajouter(nouveauSommet)
        g->taille <- g->taille+1
Fin
```

parcours en profondeur : Le parcours en profondeur sera implémenter par une fonction car on veut récupérer une liste chainée, représentant le parcours, pour cela on va partir du graphe, pour cela on va avoir besoin d'une file qui nous permetteras de stocker les sommets à explorer.
D'abord on enfile le sommet duquel pars le parcours, ens uite on va, jusqu'a ce que la file soit vide, justifiant ainsi l'utilisation d'une boucle tant que, récupérer sa tête, l'ajouter dans le parcours et enfiler tous ses voisins qui n'ont pas encore été vu, grace à une boucle pour qui nous permet de parcourirs la liste des sommets du graphe.

```
Fonction Parcours en listeChainee
Declaration
Parametre g en pointeur sur graphe, depart en sommet
Variable tableauVu en tableau[g->taille] sur booleen, file en file,
current en sommet, parcours en listeChainee
Debut
        tableauVu <- allouer(taille(booleen)*g->taille)
        file <- fileVide()
        enfiler(file, depart)
```

```
        Tant Que(fileNonVide(file)) Faire
                current <- tete(file)
                AjouterListe(parcours, current)
                tableauVu[current.nb] <- true
                Pour i allant de 0 a (g->taille)-1 Faire
                        Si g->matriceAdj[current.nb][i] Faire
                                Si tableauVu[current.nb]=faux Faire
                                        enfiler(file, g->sommetListe[i])
                                FinSi
                        FinSi
                FinPour
        FinTantQue
        Retourner parcours
Fin
```

recherche dans parcours : La recherche de parcours est utilisée de différentes facon selon le besoin, par exemple on peut avoir besoin d'un certains entier qui permet de borner la profondeur de recherche ou alors on peut vouloir s'arreter si on rencontre certains objets, mais voici la forme générale d'une recherche : On va donc vouloir chercher dans les parcours une cases avec une certaines caractéristique et en déduire l'action correpondante, pour cela on regarde avant tout sur quelles cases parmis Nord, Sud, Est et Ouest on peut aller, et ensuite on va utiliser une boucle TantQue qui va nous permettre de continuer à chercher dans les 4 parcours tant qu'ils ne sont pas tous vite et tant que l'on as pas trouvé ce que l'on veut.

```
Fonction RechercheAction en action
Parametre listeNord en listeChainee, listeSud en listeChainee,
listeOuest en listeChainee, listeEst en listeChainee,
Variable NordSafe en boolen, SudSafe en boolen, OuestSafe en boolen
 EstSafe en boolen, caseNord en sommet, caseSud en sommet, caseOuest en sommet,
fin en boolen, resultat en action, posX en entier, posY en entier
Debut
        resultat <- BOMBING
        fin <- faux

        NordSafe <- EsCeSafe(posX, posY-1)
        SudSafe <- EsCeSafe(posX, posY+1)
        OuestSafe <- EsCeSafe(posX-1, posY)
        EstSafe <- EsCeSafe(posX+1, posY)


        TantQue (resultat!=BOMBING) ET Non(fin) Faire
                Si NordSafe ET ListeNonVide(listeNord) Alors
                        caseNord = listeNord->tete
```

```
                Si RechercheCeQuonVeut(caseNord) Alors
                        resultat = Nord
                FinSi
                listeNord = listeNord->queue
        FinSi

        Si SudSafe ET ListeNonVide(listeSud) Alors
                caseSud = listeSud->tete
                Si RechercheCeQuonVeut(caseSud) Alors
                        resultat = Sud
                FinSi
                listeSud = listeSud->queue
        FinSi

        Si OuestSafe ET ListeNonVide(listeOuest) Alors
                caseOuest = listeOuest->tete
                Si RechercheCeQuonVeut(caseOuest) Alors
                        resultat = Ouest
                FinSi
                listeOuest = listeOuest->queue
        FinSi

        Si EstSafe ET ListeNonVide(listeEst) Alors
                caseEst = listeEst->tete
                Si RechercheCeQuonVeut(caseEst) Alors
                        resultat = Est
                FinSi
                listeEst = listeEst->queue
        FinSi
    Fin = (EstVide(listeNord) OU NordSafe) ET (EstVide(listeSud) OU SudSafe)
    ET (EstVide(listeOuest) OU OuestSafe) ET (EstVide(listeEst) OU EstSafe)
    FinTantQue
    Retourner resultat
Fin
```

5 Conclusion

Au final on obtient, un programme qui permet au joueur de rester toujours à une certaine distance du danger et qui peut décider si il prefere prendre des bonus ou une sortie, et qui permet, dans le cas où le joueur ne trouve ni bonus ni sortie ni danger, de trouver la case la plus proche pour faire une action utile, casser un mur cassable pour ouvrir de nouvelles possibilités. La faiblesse du programme est qu'il n'a pas d'approche particulière pour éliminer les enemies. Une approche possible aurais été de prendre en compte toutes les intersection

entre un monstre et le joueur pour déterminer la probabilité que le monstre ait d'arriver à porté bombe au bon moment, permettant de determiner un taux de probabilité de réussite qui nous permetterais de décider que faire.