Compte rendu du projet d'algorithmique 2

Arthur Blamart

2023-2024

Table des matières

1 Introduction

1.1 Introduction

L'objectif de ce projet d'algorithmie est de coder, en language C, une intelligence artificielle capable de jouer au célèbre jeu Bomberman. Si un fichier objet servant de moteur de jeu nous a été fourni, il nous incombe de coder une fonction qui étant donné le contexte d'un tour de jeu qui nous est fourni, renvoit quelle action le joueur devrait faire, que ça soit un déplacement vers l'une des 4 directions principales, haut, bas, gauche et droite ou encore la pose d'une bombe.

1.2 Bomberman, le jeu

Pour pouvoir savoir ce qu'il faut faire il faut avant toute chose savoir en quoi consiste le jeu Bomberman. Dans ce jeu, le joueur contrôle un personnage sur une carte en 2 dimensions qui peut se déplacer dans les quatre directions principales, il trouvera autour de lui des murs indestructibles et d'autres qui sont destructibles, pour pouvoir les détruire il lui faudra poser des bombes dont la portée d'explosion et le nombre varieront en fonction des bonus trouvables dans les murs destructibles, il faudra aussi faire attention à des ennemis, que l'ont peut éliminer à l'aide des bombes. Le but final étant de trouver la sortie cachée du niveau et de maximiser son score.

1.3 Contraintes

Les contraintes inhérentes au problème sont aussi à prendre en compte. Notre programme sera lancé au début d'un tour de jeu, quand le moteur réclamera une action, il nous donnera divers paramètres tels que la carte de jeu qui est sous la forme d'un arbre quaternaire prenant racine à la position du joueur et ayant comme fils les cases directement accessibles depuis celle-ci, nous disposerons aussi des valeurs de portée d'explosion des bombes et du nombre de bombes que l'on peut déposer. Deux contraintes majeures sont imposées, la première étant qu'il ne peut y avoir de mémorisation entre deux tours de jeux, hormis l'action effectuée au tour précédent qui sera directement donnée par le programme. La seconde contrainte étant que nous ne disposerons pas de l'entièreté du contenu de la carte de jeu, en effet notre joueur n'aura connaissance que de ce qu'il pourra voir jusqu'à sept cases, hors mur invisible, au travers desquels il ne voit pas.

1.4 Enjeux

Tout l'enjeux du projet est de coder une intelligence artificielle qui puisse terminer le niveau, en esquivant les dangers et en explorant, et qui permette de maximiser le score final, qui peut être augmenté en battant des ennemis ou en obtenant des bonus.

1.5 Contributions

J'ai donc implémenté un programme qui va pouvoir, dans un premier temps analyser ce qu'il a à portée de vue, et parmi ces objets, ce qui est pertinent ou non, et ensuite s'en servir pour décider d'aller chercher un objet, de fuir des dangers ou encore d'aller casser des murs pour ouvrir de nouvelles possibilités.

2 Préliminaire

Avant de développer le fonctionnement du programme de décision, il faut établir quelques points qui vont au-delà du cours d'Algorithmique 2.

2.1 Listes simplement chainées

Etant donné qu'il n'existe aucun type de liste primitif en language C, il faut se charger de l'implémenter soi-même, pour cela j'ai choisi la structure de liste simplement chainée. Une liste chainée est représentée, en language C, par un pointeur sur un objet maillon, un maillon est une structure à deux champs, d'abord sa valeur et ensuite un pointeur sur le prochain maillon.

Code 1: Implémentation d'une liste chainée d'entiers

```
struct maillon{
    int valeur;
    struct maillon* suivant;
};
typedef struct maillon* chaine;
```

2.2 Heuristique

Je vais souvent utiliser le terme heuristique dans la suite de ce compte rendu, ce que je désigne c'est une méthode de calcul qui va me permettre de trouver une solution pas forcément parfaite, mais assez bonne et assez rapide.

3 Stratégie générale

3.1 Description

La toute première chose que notre programme va regarder, c'est si le joueur est actuellement dans une situation délicate, ce qui signifie être dans le rayon d'explosion d'une bombe ou trop proche d'un ennemi. Le cas échéant il va vouloir s'éloigner du danger tout en s'ouvrant le plus de possibilités avec une heuristique qui sera détaillée plus tard.

Si il n'y a aucun danger immédiat nous allons, à l'aide d'un parcours de l'arbre qui réprésente la carte de jeu, récupérer tous les objets dits intéréssants,

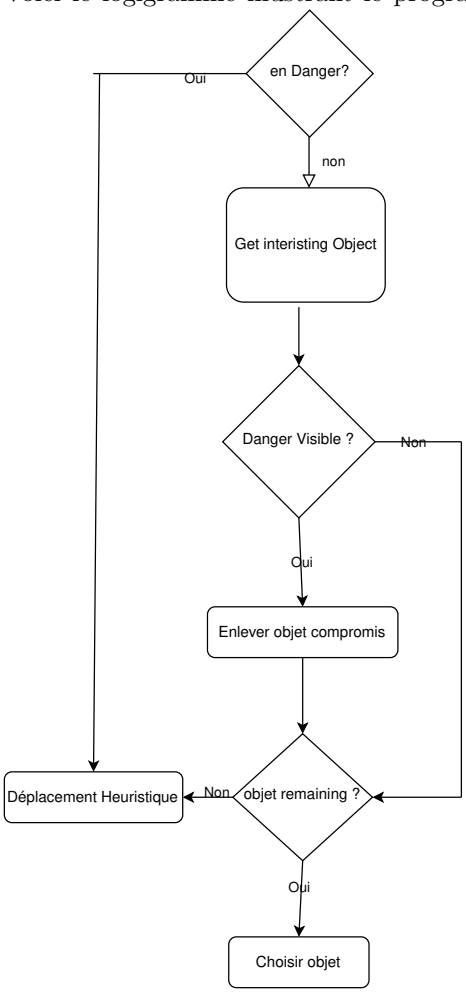
c'est-à-dire les murs cassables, bonus, sortie, et dangers dans une structure qui nous permet de stocker : la nature de l'objet, sa distance au joueur et le chemin à partir du joueur pour atteindre l'objet. Je dégage ensuite de cette liste, deux autres listes qui contiennent toutes les bombes et tous les ennemis contenu dans la liste objet, stockés sous la même structure de triplets : nature, distance et chemin.

Le problème c'est que certains de ses objets peuvent être compromis ou dangereux, c'est à dire que s'avancer vers eux impliquerait de devoir s'approcher trop d'un ennemis ou même de rentrer dans le rayon d'explosion d'une bombe. Pour éviter tout cela je lance d'abord une fonction qui, si il y a des bombes visibles, enlève de ma liste d'objets toutes les bombes et les objets compromis par celles-ci, c'est à dire, les objets pour lesquels je devrais rentrer dans le rayon de la bombe pour y accéder. Je rappelle que par la toute première étape de mon programme je suis assuré de ne pas déjà être dans le rayon d'explosion d'une bombe. J'exécute ensuite une fonction similaire mais qui agit sur les ennemis visibles cette fois.

Maintenant qu'il ne me reste plus qu'une liste d'objets que je peux aller chercher de manière sécurisée, il y a deux possibilités, soit la liste est vide, soit elle contient au moins un objet. Si la liste est vide je vais chercher à me replacer judicieusement à l'aide de la même heuristique que citée précédemment. Dans le cas contraire je vais regarder les différents objets de la liste et je vais séléctionner le plus intéressant, savoir si je dois me rapprocher d'un bonus, prendre une sortie, poser une bombe pour un mur proche ou pour attraper un ennemi.

3.2 Logigramme

Voici le logigramme illustrant le programme de décision pour un tour de jeu.



4 Points clés

Le fonctionnement du programme de décision repose sur plusieurs principes majeurs que j'ai, précédemment, évoqué succintement mais que je vais maintenant développer, dans l'ordre :

- Structures
- Détection de dangers
- Récupération d'objets dits intéressants

- Réduction d'une liste d'objets
- Choix d'objet
- Heuristique de déplacement

4.1 Structures

Je vais utiliser très régulièrement 3 structures dans mon programme :

Triplet objet : Il va me falloir une façon de représenter et de stocker des informations sur les différents objets, ennemis et bombes pour cela je vais utiliser une structure triplet qui me permettra de stocker :

- Nature
- Distance au joueur
- Chemin pour l'atteindre depuis la position du joueur

Liste chainées : Je vais aussi utiliser 2 différentes listes chainées, une qui contiendra des triplets représentant des objets, et une autre qui contiendra des actions.

4.2 Détection de dangers

L'objectif est d'avoir une fonction qui étant donné une position, qu'elle soit celle du joueur ou non, puisse nous indiquer si cette position est compromise. Pour savoir ce qu'est une position compromise il faut penser aus deux grands dangers du jeu, les bombes et les ennemis, c'est pourquoi notre fonction va d'abord calculer si la position est mise en danger par des ennemie puis si elle est mise en danger par une bombe et faire un OU logique de ces deux résultats.

Détection d'ennemi : Pour les ennemis on va définir qu'une position est compromise si un ennemi se trouve dans un certain rayon, pour cela il suffit d'explorer récursivevment les fils de l'arbre jusqu'à une certaine profondeur. On se retouve donc avec une fonction $estPositionNearEnemy$, dont voici la signature :

- $estPositionNearEnemy$
  - Entrée : Un arbre représentant la carte de jeu
  - Sortie : Un booléen indiquant si un enemie est trop proche ou non

Détection de Bombes : Pour les bombes nous allons simplement regarder si nous sommes dans le rayon d'explosion d'une bombe.

Pour cela on va d'abord vérifier que la bombe n'est pas sur la position, surtout si c'est la case du joueur et que donc le caractère de la bombe n'est pas affiché car le joueur est dessus. Je vais ensuite avancer dans les 4 directions principales jusqu'à la distance de rayon d'explosion pour vérifier qu'aucune bombe ne s'y trouve.

Algorithme 1 : *estPositionDansRayonExplosion*

Entrées : Un arbre représentant une position de la carte de jeu, un booleen qui indique si on regarde la carte du joueur ou non, le rayon d'explosion, la dernière action effectué par le joueur
Sorties : La position est elle compromise par une bombe
Declaration
Parametre *map* en **arbre**, *estJoueur* en **booleen**, *rayonExplosion* en **entier**, *derniereAction* en **action**
Variable *resultat* en **booleen**, i en **entier**, $filsNord, filsSud, filsOuest, filsEst$ en **arbre**
début
  $resultat \leftarrow (estJoueur\ ET\ (derniereAction = BOMBARDER))$
  $i \leftarrow 0$
  $filsNord \leftarrow map.filsNord$
  $filsSud \leftarrow map.filsSud$
  $filsOuest \leftarrow map.filsOuest$
  $filsEst \leftarrow map.filsEst$
  tant que $i \leq rayonExplosion + 1$" **faire**
    si $NonestVide(filsNord)$ **alors**
      $resultat \leftarrow resultat\ OU\ (filsNord.nature = BOMB)$
      $filsNord \leftarrow filsNord.filsNord$
    fin
    si $NonestVide(filsSud)$ **alors**
      $resultat \leftarrow resultat\ OU\ (filsSud.nature = BOMB)$
      $filsSud \leftarrow filsSud.filsSud$
    fin
    si $NonestVide(filsOuest)$ **alors**
      $resultat \leftarrow resultat\ OU\ (filsOuest.nature = BOMB)$
      $filsOuest \leftarrow filsOuest.filsOuest$
    fin
    si $NonestVide(filsEst)$ **alors**
      $resultat \leftarrow resultat\ OU\ (filsEst.nature = BOMB)$
      $filsEst \leftarrow filsEst.filsEst$
    fin
    $i \leftarrow i + 1$
  fin
  retourner *resultat*
fin

Et donc grace à ces deux fonctions on obtient la fonction *esEnDanger* :

Algorithme 2 : *esEnDanger*

Entrées : Un arbre représentant une position de la carte de jeu, un booleen qui indique si on regarde la carte du joueur ou non, le rayon d'explosion, la dernière action effectué par le joueur

Sorties : La position est elle compromise

Declaration

Parametre *map* en **arbre**, *estJoueur* en **booleen**, *rayonExplosion* en **entier**, *derniereAction* en **action**

Variable *resultat* en **booleen**

début

```
resultat ←
 estPositionDansRayonExplosion(map, estJoueur, rayonExplosion, dernierAction)
resultat ← resultat OU estPositionNearEnemy(map)
retourner resultat
```

fin

4.3 Récupération d'objets

Pour pouvoir obtenir une liste d'objets je vais devoir faire un parcours de l'arbre qui représente la carte de jeu, et ce, de manière itérative.

Tout d'abord je regarde si l'arbre en entrée n'est pas NULL, si c'est le cas je renvoie une liste vide car il n'y aura en effet aucun objet. Si l'arbre n'est pas NULL je vais regarder si la nature de l'objet (un bonus, une bombe ou autre) est pertiente, c'est-à-dire si ce n'est pas juste un simple chemin.

Si je trouve l'objet intéressant, je crée une nouvelle instance d'un triplet qui représentera l'objet trouvé, d'abord la distance sera nulle et son chemin aussi, car nous sommes dessus, je vais ensuite appeler la fonction sur les 4 fils, Nord, Sud, Est et Ouest de mon arbre qui me renverront une liste d'objets, potentiellement vide, chacun. La première chose à faire est de mettre à jour ces listes d'objets, comme elles sont le fils du noeud ou je me trouve, je dois incrémenter la distance de tous les objets dans les listes fils de 1 et je dois mettre à jour leurs chemins, en ajoutant, par exemple l'action NORTH au début de tous les chemins des objets de la liste du fils nord.

Il me suffit ensuite de concaténer les 4 listes fils et potentiellement l'objet trouvé au noeud courant pour obtenir la liste d'objet partant de l'arbre courant, je renvoie donc cette liste finale.

Algorithme 3 : RecupererObjets

Entrées : un arbre représentant une partie de la carte de jeu
Sorties : une liste des objets contenus dans l'arbre
Declaration
Parametre *map* en **Arbre**
Variable *resultat*, *listeNord*, *listeSud*, *listOuest*, *listeEst* en **Liste Objet**, *nouvelObjet* en **Objet**
début
  $resultat \leftarrow listeObjetVide()$
  si $estInteressant(map.nature)$ **alors**
    $nouvelObjet \leftarrow$ (**Objet**)[$.nature \leftarrow map.nature$,
     $.chemin \leftarrow listeActionVide()$, $.distance \leftarrow 0$]
    $resultat \leftarrow ajoutTeteListe(resultat, nouvelObjet)$
  fin
  $listeNord \leftarrow RecupererObjets(map.filsNord)$
  $listeSud \leftarrow RecupererObjets(map.filsSud)$
  $listeOuest \leftarrow RecupererObjets(map.filsOuest)$
  $listeEst \leftarrow RecupererObjets(map.filsEst)$
  $miseAJour(listeNord, NORD)$
  $miseAJour(listeSud, SUD)$
  $miseAJour(listeOuest, OUEST)$
  $miseAJour(listeEst, EST)$
  $resultat \leftarrow$
   $concatenationListes(resultat, listeNord, listeSud, listeOuest, listeEst)$
  retourner *resultat*
fin

La fonction *RecupererObjet* utilise plusieurs fonctions et procédures pour fonctionner, voici leurs signatures :

- *estInteressant* :
  - Entrée : Un caractère qui représente un objet du jeu
  - Sortie : Un booléen qui indique si le caractère représente un objet pertinent (Bonus, Enemies, Bombes, Sorties)
- *ajoutTeteListe* :
  - Entrée : Une liste et un objet
  - Sortie : La même liste mais avec l'objet en tête
- *concatenationListes* :
  - Entrée : plusieurs listes d'objets
  - Sortie : une liste résultante de la concaténation de toutes les listes en entrée

La procédure *miseAjour* est peut-être la plus importante, elle permet de mettre à jour les listes des fils de la position actuelle en incrémentant la distance de chaque objet et en ajoutant l'action correspondante dans leur chemin. Et ce en explorant la liste d'objets récursivement avec, comme condition d'arrêt si la liste est vide.

Algorithme 4 : miseAJour

Entrées : Une liste d'objet et une action
Declaration
Parametre *liste* en **Liste d'objet**, *direction* en **Action Variable**
objetActuel en **Objet**
début
  si *Non estVide(liste)* **alors**
    $objetActuel \leftarrow liste.tete.valeur$
    $objetActuel.distance \leftarrow objetActuel.distance + 1$
    $objetActuel.chemin \leftarrow$
     $AjoutTeteListeAction(objetActuel.chemin, direction)$
    $miseAJour(liste.suivant, direction)$
  fin
fin

4.4 Réduction de la liste d'objets

La réduction d'une liste d'objet occure lorsque qu'on trouve des ennemis ou des bombes, dans ces cas on regardera chaque élément de notre liste l'un après l'autre en regardant pour chacun si une bombe ou un ennemi le compromet avec les fonctions respectives *doitOnCouperBombe*() et *doitOnCouperEnnemi*() dont voici les signatures :

- *doitOnCouperBombe*()
  - Entrée : Un objet, une liste de bombes et la distance d'explosion
  - Sortie : Un booléen qui indique si au moins l'une des bombes de la liste compromet l'objet
- *doitOnCouperEnnemi*()
  - Entrée : Un objet et une liste d'ennemis
  - Sortie : Un booléen qui indique si au moins l'un des enemies de la liste compromet l'objet

Ces deux fonctions utilisent une approche similaire, elle vont boucler sur chaque élément soit de la liste de bombes, soit de celle des ennemis, et calculer un chemin, à partir du chemin du danger, qui si il est préfixe du chemin de l'objet regardé, indiquera que ce dernier est mis en danger. La seule différence est que dans les deux cas le préfixe n'est pas calculé de la même façon.

Pour les ennemis : On va vouloir éviter tout objet se trouvant derrière l'ennemi ou juste devant donc le préfixe interdit sera, pour un ennemi donné, le chemin de cet ennemi moins la derniere action.

Pour les bombes : Pour une bombe donnée on va vouloir éviter tout objet nous demandant de passer au dessus de la bombe ou tout objet dans son rayon d'explosion, c'est pourquoi nous allons prendre comme préfixe interdit les $bombe.distance - distanceExplosion$ premières actions de $bombe.chemin$

Pour mieux illustrer voici le code de $doitOnCouperBombe()$:

Algorithme 5 : $doitOnCouperBombe$

Entrées : Un objet, une liste de bombes et la distance d'explosion
Sorties : Un booléen qui indique si au moins l'une des bombes de la liste compromet l'objet

Declaration
Parametre $objetToCheck$ en **Objet**, $listeBombe$ en **liste Objet**, $rayonExplosion$ en **Entier**
Variable $resultat$ en **Booleen**, $currentInBomb$ en **liste Bomb**, $currentBomb$ en **Objet**, $prefixe$ en **liste Action**
début
  $resultat \leftarrow faux$
  $currentInBomb \leftarrow listeBombe$
  tant que $(NonestVide(currentInBomb) ET Non resultat)$ **faire**
    $currentBomb \leftarrow currentInBomb.valeur$
    $prefixe \leftarrow raccourcir(currentBomb.chemin, currentBomb.distance - rayonExplosion)$
    $resultat \leftarrow estPrefixe(prefixe, objetToCheck.chemin)$
    $currentInBomb \leftarrow currentInBomb.suivant$
  fin
  retourner $resultat$
fin

Voici les signatures des fonctions utilisées :

- $raccourcir$
  - Entrée : une liste d'actions et un entier n
  - Sortie : une liste d'actions qui contient les n premiers élements de la liste initiale
- $estPrefixe$
  - Entrée : deux liste d'actions
  - Sortie : un booleen qui indique si la première liste est bien préfixe de la seconde

Une fois que l'on peut savoir quels objets sont à exclure ou à garder il suffit de faire une fonction qui coupe les éléments à couper de la liste objets et qui garde ceux qu'ils faut garder.

Pour cela on va donc prendre une liste vide qu'on appellera *resultat*, on itère sur chaque élément de la liste d'objets. Pour chaque objet on détermine d'abord si on doit le "couper" ou si on veut le garder. Si on veut le couper : il faut mettre de coté le maillon de la liste d'objets suivant le maillon courant et on va libérer la mémoire du maillon courant avant de prendre comme maillon courant, celui que nous avons mis de côté, pour la prochaine itération. Si on veut garder le maillon : on va tout d'abord mettre de côté notre *resultat*, donc ce qu'on a trouvé pour l'instant, car on veut que l'objet courant devienne la tête de notre résultat, et nous gardons de côté la suite de la liste objet qu'il reste à analyser, qui se trouve dans le champ suivant du maillon courant, notre maillon courant devient la tête de notre liste *resultat* et nous mettons, pour la prochaine itération, comme maillon courant, le maillon que nous avions précédemment gardé de côté qui nous permet de retrouver la suite de la liste à analyser.A la fin, il ne restera plus qu'à retourner la liste *resultat* qui contiendra tous les objets vers lesquels on peut se diriger sans risque.

Algorithme 6 : *couperListeObjet*

Entrées : Un objet, une liste de bombes, une d'ennemi et la distance d'explosion

Sorties : Un booléen qui indique si au moins l'un des dangers des deux listes compromet l'objet

Declaration

Parametre $listeBombe, listeEnemie, listeObjet$ en **liste objet**, $rayonExplosion$ en **entier**

Variable $resultat, currentInObjet, temporaire, temporaire2$ en **liste objet**, $currentObjet$ en **objet**, $doitOnCouper$ en **booleen**

début

- $resultat \leftarrow listeObjetVide()$
- $currentInObjet \leftarrow liste$
- **tant que** $NonestVide(currentInObjet)$ **faire**
  - $currentObjet \leftarrow currentInObjet.valeur$
  - **si** $doitOnCouperBombe$ *OU* $doitOnCouperEnemies$ **alors**
    - $temporaire \leftarrow currentInObjet.valeur$
    - $liberer(currentInObjet)$
    - $currentInObjet \leftarrow temporaire$
  - **sinon**
    - $temporaire \leftarrow resultat$
    - $temporaire2 \leftarrow currentInObjet.suivant$
    - $resultat \leftarrow currentInObjet$
    - $resultat.suivant \leftarrow temporaire$
    - $currentInObjet \leftarrow temporaire2$
  - **fin**
- **fin**
- **retourner** $resultat$

fin

4.5 Choix d'objet

La fonction *choixAction* est utilisée après l'exécution de la fonction *couperListeObjet* il reste des objets vers lesquels on pourrait se déplacer ou interagir, sans se mettre en danger.

Elle va nous permettre de déduire à partir d'une liste d'objet fournie en entrée de choisir l'objet le plus intéressant à exploiter et nous renverra quelle action il faut donc faire.

Pour cela on va devoir établir un ordre de priorité des différents objets, le voici en partant du plus au moins intéressant :

- Bonus
- Ennemi atteignable
- Mur cassable atteignable

- Sortie
- Mur éloigné

Atteindre des murs : un mur atteignable correspond à un mur qu'une bombe posée à ce tour-ci pourrait atteindre, ouvrant ainsi de nouvelles possibilités, tandis qu'un mur éloigné correpond à un mur qui n'est pas à portée d'explosion, mais duquel on pourrait se rapprocher. Pour savoir si un mur est atteignable par une bombe je vais utiliser une fonction $isBombingUsefulOn$ dont voici la signature :

- $isBombingUsefulOn$
  - Entrée : un objet et un entier représentant la portée d'explosion des bombes
  - Sortie : un booléen qui indique si l'objet donné en entrée serait atteignable avec une bombe

Pour savoir si l'objet fourni est atteignable il suffit simplement de regarder son champ *chemin* pour vérifier que celui-ci soit bien une ligne droite, depuis le joueur, et dont la taille est inférieure ou égale à la portée d'explosion des bombes

Atteindre des ennemis : Pour savoir si un ennemi est atteignable par une bombe qui serait posée au tour de jeu actuel on va utiliser une fonction $isAnEnemyeExplosable$ qui est en fait une heuristique simple mais terriblement efficace, dont voici la signature :

- $isAnEnemyeExplosable$
  - Entrée : un arbre représentant la carte de jeu et deux entiers représentant la portée d'explosion et le temps qu'une bombe posée met avant d'exploser
  - Sortie : un booléen qui indique si il existe un ennemi qui pourrait être touché par une bombe posée à ce tour-ci

Le but est de savoir, dans l'hypothèse où on poserait une bombe à ce tour-ci, si au moment où la bombe explosera, l'ennemi sera dans son rayon d'explosion.

Dit autrement nous voulons savoir si au bout du $delaisd'explosion$, l'ennemi sera dans $lerayond'explosion$, hors ce sont deux paramètres que nous connaissons. Nous pouvons donc en déduire que si nous posons une bombe à ce tour-ci elle a une chance d'atteindre un ennemi si celui-ci se trouve à une distance de $delais + rayon$, pour ne pas que cette fonction se lance trop souvent j'ai décidé de simplement soustraire une constante que j'ai arbitrairement choisi à 2.

Pour cela il me suffit de parcourir l'abre, depuis la position du joueur jusqu'à une profondeur de $delais+rayon-2$ pour en déduire si il y a au moins un ennemi atteignable par une bombe qui serait posée à ce tour-ci.

Choix de l'objet : Pour décider quel objet est le plus intéressant à exploiter et quel action le joueur doit faire je vais parcourir linéairement la liste des objets en comparant à chaque itération le meilleur objet trouvé, pour l'instant, et l'objet actuel.

Pour les comparer je vais tout d'abord regarder si leurs natures sont identiques, si c'est le cas, je vais comparer leurs distance au joueur, le plus proche sera le mieux. Attention au cas des murs cassables, car un mur cassable n'a pas la même valeur qu'un mur cassable et atteignable, et si un mur cassable est atteignable, il ne sert à rien de comparer leurs distances car ils sont tous les deux atteignables et donc une bombe posée à ce tour-ci atteindra les deux murs.

Je vais ensuite regarder le cas où le candidat précédemment trouvé n'est pas un bonus, c'est à dire la priorité numéro 1, car si on déjà trouvé un Bbonus on ne voudra pas d'un autre objet. Dans le cas où nous n'avions pas trouvé de bonus on va regarder si l'objet courant en est un, ou si c'est un mur atteignable (et qu'il nous reste des bombes bien sur) ou encore si l'objet courant est la sortie du niveau.

Une fois la liste d'objets finie, si aucun bonus n'a été trouvé il faut regarder si un ennemi est éventuellement atteignable avec la fonction $isAnEnemyeExplosable$, car je rappelle que c'est notre seconde priorité, le cas échéant, le joueur posera une bombe.

Algorithme 7 : choixAction

```
Entrées : Un arbre représentant la carte de jeu, une liste d'objet, deux entiers
          représentant la portée d'explosion et le nombre de bombes disponibles et
          l'action faite au dernier tour
Sorties : L'action la plus intéressantes à faire à ce tour-ci
Declaration
Parametre map en arbre, listeObjet en liste d'objet,
 rayonExplosion, bombesRestantes en entier, derniereAction en action
Variable resultat en action, currentInObjet en liste d'objet,
 candidat, objetCourant en objet
début
    currentInObjet ← listeObjet
    candidat ← currentInObjet.valeure
    si candidat.nature = MurCassable ET bombeRestantes > 0 ET
     isBombingUsefulOn(candidat, rayonExplosion) alors
        resultat ← BOMBING
    fin
    resultat ← candidat.chemin.valeure
    tant que NonestVide(currentInObjet) faire
        currentObjet ← currentInObjet.valeure
        si candidat.nature = currentObjet.nature alors
            si candidat.nature! = MurCassable ET
             candidat.distance > currentObjet.distance alors
                candidat ← currentObjet
                resultat ← candidat.chemin.valeure
            fin
            sinon si estMurCassable(currentObjet.nature) alors
                si resultat = Bombing ET bombesRestantes > 0 ET
                 isBombingUsefulOn(currentObjet, rayonExplosion) alors
                    candidat ← currentObjet
                    resultat ← Bombing
                fin
                sinon si NonisBombingUsefulOn(currentObjet, rayonExplosion)
                 ET currentObjet.distance > 1 ET
                 candidat.distance > currentObjet.distance alors
                    candidat ← currentObjet
                    resultat ← candidat.chemin.valeure
                fin
            fin
        fin
        sinon si NonestBonus(candidat) alors
            si NonestBonus(candidat) alors
                candidat ← currentObjet
                resultat ← candidat.chemin.valeure
            fin
            sinon si estMurCassable(currentObjet) ET bombeRestantes > 0 ET
             isBombingUsefulOn(currentObjet, rayonExplosion) alors
                candidat ← currentObjet
                resultat ← Bombing
            fin
            sinon si estSortie(currentObjet.nature) ET resultat! = Bombing
             alors
                candidat ← currentObjet
                resultat ← candidat.chemin.valeure
            fin
        fin
        currentInObjet ← currentInObjet.suivant
    fin
    si (NonestBonus(candidat) ET
     isAnEnemyeExplosable(map, rayonExplosion, bombesRestantes) alors
        resultar ← Bombing
    fin
    retourner resultat
fin
```

- *estBonus*
  - Entrée : un objet
  - Sortie : L'objet donné est-il un bonus
- *estMurCassable*
  - Entrée : un objet
  - Sortie : L'ojet donné est-il un mur cassable
- *estSortie*
  - Entrée : un objet
  - Sortie : L'ojet donné est-il une sortie

4.6 Heuristique de déplacement

Dans les cas où il n'y a aucun objet pour se décider vers où aller ou quand le joueur est en danger, la fonction *whereMaxCases* nous permettra de prendre une décision. Son nom est assez explicite : elle va calculer, pour les quatre directions principales, le nombre de cases sûres atteignables à partir de ces directions on va ensuite regarder laquelle amène au plus de cases et s'y diriger en pensant à vérifier que ce mouvement ne nous met pas en danger.

Un problème majeur qui arrivait était que si le joueur arrivait à éliminer tous les objets présent dans la partie centrale de la carte de jeu, en laissant la sortie dans un des coins de la carte, le programme avait toujours tendance à se diriger vers le centre de la carte, restant ainsi bloqué car les objets dans les coins serait hors de son champ de vision.

Pour palier à ce problème j'ai fixé un seuil qui une fois que toutes les valeurs le dépassent plutôt que de choisir la direction qui offre le plus de possibilités, le déplacement serait aléatoire, garantissant, à un moment ou un autre de se rapprocher assez près de ces coins pour voir les objets qui s'y trouvent.

Algorithme 8 : wherePlusDeCases

Entrées : un arbre représentant la carte de jeu, un entier pour la portée d'explosion des bombes et l'action effectuée au dernier tour
Sorties : Dans quelle direction le joueur devrait se diriger
Declaration
Parametre *map* en **arbre**, *rayonExplosion* en **entier**, *dernierAction* en **action**
Variable *resulat* en **action**,
$nbNord, nbSud, nbOuest, nbEst, nbMaxCases, seuilAleatoire$ en **entier**,
$estNordSafe, estSudSafe, estOuestSafe, estEstSafe, isThereAnyPosSafe$ en **booleen**
début
  $resultat \leftarrow Bombing$
  $isNordSafe \leftarrow$
    $NonesEnDanger(map.filsNord, faux, rayonExplosion, dernierAction)$
  $isSudSafe \leftarrow$
    $NonesEnDanger(map.filsSud, faux, rayonExplosion, dernierAction)$
  $isOuestSafe \leftarrow$
    $NonesEnDanger(map.filsOuest, faux, rayonExplosion, dernierAction)$
  $isEstSafe \leftarrow$
    $NonesEnDanger(map.filsEst, faux, rayonExplosion, dernierAction)$
  $isThereAnyPosSafe \leftarrow$
    $(estNordSafe\ OU\ estSudSafe\ OU\ estOuestSafe\ OU\ estEstSafe)$
  $nbNord \leftarrow howManyCase(map.filsNord)$
  $nbSud \leftarrow howManyCase(map.filsSud)$
  $nbOuest \leftarrow howManyCase(map.filsOuest)$
  $nbEst \leftarrow howManyCase(map.filsEst)$
  si $isThereAnyPosSafe$ **alors**
    $nbMaxCases \leftarrow max(nbNord, max(nbSud, max(nbOuest, nbEst)))$
    si $nbNord = nbMaxCases$ **alors**
      $resultat \leftarrow NORD$
    fin
    sinon si $nbSud = nbMaxCases$ **alors**
      $resultat \leftarrow SUD$
    fin
    sinon si $nbOuest = nbMaxCases$ **alors**
      $resultat \leftarrow OUEST$
    fin
    sinon
      $resultat \leftarrow EST$
    fin
  sinon
    $seuilAleatoire \leftarrow 45$
    si $min(nbNord, min(nbSud, min(nbOuest, nbEst))) = seuilAleatoire$ **alors**
      $resultat \leftarrow directionAleatoire()$
    sinon
      $nbMaxCases \leftarrow nbNord$
      $resultat \leftarrow NORD$
      si $isSudSafe\ ET\ nbMaxDeCasesnb < nbSud$ **alors**
        $nbMaxCases \leftarrow nbSud$
        $resultat \leftarrow SUD$
      fin
      si $isOuestSafe\ ET\ nbMaxDeCasesnb < nbOuest$ **alors**
        $nbMaxCases \leftarrow nbOuest$
        $resultat \leftarrow OUEST$
      fin
      si $isEstSafe\ ET\ nbMaxDeCasesnb < nbEst$ **alors**
        $nbMaxCases \leftarrow nbEst$
        $resultat \leftarrow EST$
      fin
    fin
  fin
  retourner $resultat$
fin

Voici les signatures des fonctions auxiliaires utilisées :

- *max*
  - Entrée : deux entiers
  - Sortie : le plus grand des deux entiers donnés
- *min*
  - Entrée : deux entiers
  - Sortie : le plus petit des deux entiers donnés
- *directionAleatoire*
  - Entrée : rien
  - Sortie : aléatoirement une des actions : NORD, SUD, OUEST ou EST

5 Conclusion

Dans ce projet les défis étaient multiples : implémenter une intelligence artificielle pour le jeu bomberman, et le faire avec des informations restreintes, pas de mémorisation entre les tours de jeu et un champ de vision réduit. J'ai réussi à implémenter diverses procédures et fonctions qui me permettent à la fois d'analyser les dangers, savoir si le joueur est dans une situation délicate et où il devrait aller, d'aller chercher les différents objets à portée de vue en m'assurant bien sûr qu'il n'y a pas de risque à aller les chercher, pouvoir essayer d'aller attaquer les monstres pour maximiser le score. Et j'ai pu faire face au problème de vue restreinte en laissant une part d'aléatoire se déplacer lorsque rien n'est as porté de vue.

Le résultat est une intelligence artificielle, fonctionelle qui réussit à trouver la sortie dans la plupart des cas et qui arrive à ramasser le maximum d'objets et attaquer les ennemis afin de maximiser son score.