Victor Vogel, Anthony Gussy, Benjamin Jacquot

Groupe n°3

Résumé

Ce rapport de projet a pour objectif de présenter la méthode de résolution appliquée à la problématique posée par le projet, à savoir faire bouger le mineur de sorte à ce qu'il réussisse le niveau.

IA41 Rapport de projet

Delirium 2

Table des matières

[Introduction 2](#_Toc453278326)

[Identifications des sous problèmes et résolution 2](#_Toc453278327)

[1) Identifier les éléments nécessaires à la réussite du niveau 2](#_Toc453278328)

[2) Se diriger vers un élément nécessaire 2](#_Toc453278329)

[3) Se diriger vers une zone inconnue du labyrinthe 2](#_Toc453278330)

[4) Eviter les pièges 2](#_Toc453278331)

[5) Enregistrer les états connus du labyrinthe 3](#_Toc453278332)

[Représentation des connaissances 3](#_Toc453278333)

[1) Variables d’états 3](#_Toc453278334)

[2) Machines à états finis 3](#_Toc453278335)

[3) Descriptions des machines à états finis 3](#_Toc453278336)

[a) Niveau 5 3](#_Toc453278337)

[b) Niveau 4 4](#_Toc453278338)

[c) Niveau 3 4](#_Toc453278339)

[d) Niveau 2 4](#_Toc453278340)

[e) Niveau 1 4](#_Toc453278341)

[f) Niveau 0 4](#_Toc453278342)

[Exemples 5](#_Toc453278343)

[Démo 5](#_Toc453278344)

[Difficultés rencontrées 5](#_Toc453278345)

[Améliorations possibles 5](#_Toc453278346)

# Introduction

Dans le cadre de l'UV IA41, nous sommes amenés à programmer une intelligence artificielle parmi un certain nombre de projets proposés. Nous avons choisi le projet Delirium2 car le principe nous semblait intéressant. Le principe du jeu est le suivant : nous contrôlons un mineur qui doit récolter un certain nombre de diamants pour pouvoir passer aux niveaux suivants en traversant une porte. Il a donc la possibilité de creuser des galeries pour pouvoir récupérer un diamant. Le joueur devra aussi faire attention aux monstres se baladant dans les galeries, aux rocher qui peuvent chuter si le mineur créé un chemin permettant la chute d’un rocher et la façon dont il minera : le joueur peut bloquer un chemin permettant de finir le niveau si une pierre tombe et bouche une entrée.

# Identifications des sous problèmes et résolution

## Identifier les éléments nécessaires à la réussite du niveau

Pour réussir un tableau, le mineur doit arriver à la sortie. Avant d’arriver à la sortie, le mineur doit rassembler un certain nombre de diamants. Lorsque le nombre de diamant est atteint la variable CanGotoExit passe à 1, indiquant au mineur qu’il peut se diriger vers la sortie. Pour assurer la réussite du niveau, nous devons donc vérifier tout d’abord si la variable CGE est à 1, auquel cas le mineur se dirige vers la sortie, sinon il se dirige vers un diamant. Si aucun des deux n’est en vue, il va errer dans le labyrinthe.

## Se diriger vers un élément nécessaire

Afin de se diriger vers un élément donné (que ce soit la sortie, un diamant, ou une case donnée par la fonction errer), le programme va faire appel à un algorithme A\*, qui va trouver le meilleur chemin vers la position de l’élément, si tel chemin existe, et renvoyer la direction à prendre pour s’y rendre.

## Se diriger vers une zone inconnue du labyrinthe

Lorsque le programme ne peut ni se diriger vers la sortie, ni vers un diamant, il va tenter de se diriger vers un endroit encore inexploré du labyrinthe.

## Eviter les pièges

Pour ne pas mourir, le mineur doit éviter de toucher les monstres et ne doit pas se trouver sous une pierre ou un diamant en chute libre.

## Enregistrer les états connus du labyrinthe

Afin de ne pas avoir à chercher à chaque fois où se trouve la sortie ou les diamants, on enregistre au fur et à mesure de la progression dans le labyrinthe les différents états de celui-ci.

# Représentation des connaissances

## Variables d’états

Pour notre algorithme A\* nous utilisons donc une liste de liste (une liste correspond à une ligne du labyrinthe) représentant tout le labyrinthe que l’on complète au fur et à mesure de l’avancement du mineur. Avec cela, nous fournissons à la fonction des coordonnées de fin pouvant par exemple représenté la position d’un diamant, d’une porte ou une position quelconque dans le cas où le mineur doit errer et les coordonnées du mineur.

## Machines à états finis

Niveau 5 : Analyser ce qui est vu par le mineur pour déceler les positions dangereuses

Niveau 4 : Mettre à jour le labyrinthe d’après ce qui est vu par le mineur et les dangers perçus

Niveau 3 : S’arrêter sur la sortie si elle a été atteinte

Niveau 2 : Se diriger vers la sortie si le compte de diamant est atteint

Niveau 1 : Se diriger vers un diamant si le compte n’est pas atteint

Niveau 0 : Errer si aucun objectif n’est visible

## Descriptions des machines à états finis

### Niveau 5

Les positions dangereuses sont matérialisées par les endroits où une pierre qui tombe, un diamant qui tombe, ou un monstre se trouvera au prochain mouvement du mineur. Ces positions sont insérées dans le champ de vue du mineur afin qu’elles soient évitées lors de l’appel de l’algorithme A\*.

L’évitement des objets qui tombent se fait en testant la valeur supérieure de chaque case du champ de vue en vérifiant qu’il s’agisse ou non d’un objet pouvant tomber et si la case actuelle est une case vide donc permettant la chute. De même des tests sont effectués sur les côtés pour les objets pouvant glisser sur des éléments.

Pour éviter les monstres, le champ de vision, la taille ainsi que la position du mineur sont envoyés à la fonction. On effectue ensuite la recherche de la position des monstres dans la liste puis, on vérifie dans quelle direction le monstre est allé et nous appliquons l’algorithme de mouvement de monstre. Une fois cela fait, nous renvoyons une liste avec la position des monstres au temps t+1.

### Niveau 4

La liste modifiée retournée par la détection des positions dangereuses est ensuite utilisée pour mettre à jour le labyrinthe. Cette mise à jour va reconstruire une nouvelle carte du labyrinthe en parcourant à la fois l’ancienne carte et la liste des objets perçus par le mineur. Les positions ne se trouvant dans aucune de ces deux listes seront mises à -1.

### Niveau 3

Une variable globale est utilisée afin de signaler que la fin a été atteinte. Cette variable donne la position de la fin une fois celle-ci atteinte. On compare la position actuelle du mineur avec cette variable et on vérifie que CGE est bien à 1. Si ces deux conditions sont réunies, le mineur attend que le prochain niveau se charge.

### Niveau 2

Si la position finale n’est pas atteinte mais que CGE est à 1, le mineur va tenter de se diriger vers la sortie. Il va donc tout d’abord vérifier qu’il connait la position de la sortie en la recherchant dans le labyrinthe. S’il connait la position de la sortie, il fera appel à l’algorithme A\* afin que celui-ci lui donne l’action à effectuer pour s’en rapprocher.

### Niveau 1

Si CGE n’est pas à 1 alors le mineur doit chercher des diamants. Il en cherche donc dans la partie du labyrinthe déjà connue et va tenter de se diriger vers celui qui est le plus proche de lui en utilisant l’algorithme A\*.

### Niveau 0

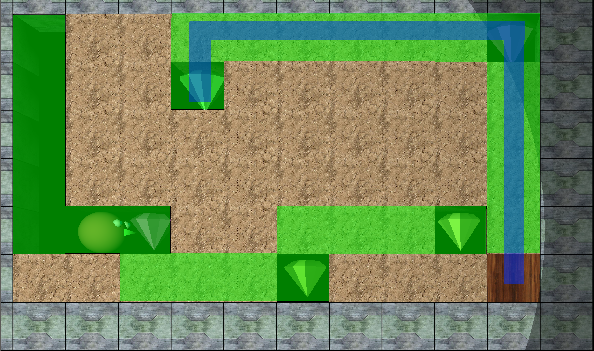
Enfin, si aucun des états précédent n’est possible, le mineur va errer dans le labyrinthe. Il va chercher la position inconnue (matérialisée par -1) la plus proche de lui. Si il n’y a pas de position inconnue dans la carte actuelle du labyrinthe, le mineur va se diriger en diagonale vers le coin inférieur droit encore inconnu de la carte.

# Exemples

## Récupération des diamants avant d'aller à la sortie (à faire) :

Dans cet exemple, le mineur doit faire en sorte de ramasser les diamants et d’aller à la sortie.

Résultat obtenu :

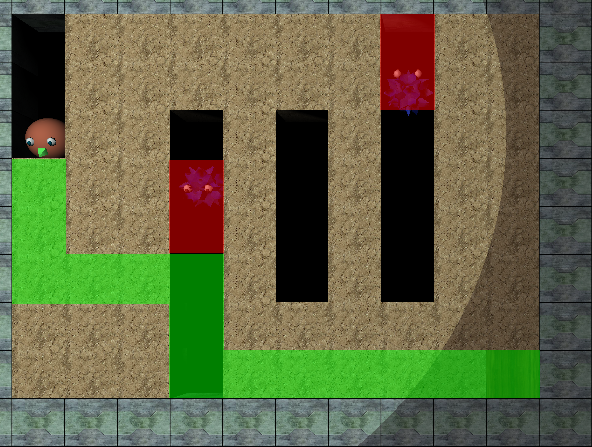


Le mineur va prendre successivement les diamants les plus proches de sa position actuelle. Une fois tous les diamants (nécessaires pour accéder à la sortie) récupérer, le mineur va revenir sur ses pas pour éviter de passer dans la clay et ainsi accéder à la sortie.

## Éviter les monstres

Dans cet exemple, le mineur doit faire en sorte d’éviter un monstre en mouvement. Pour ce faire, nous fournissons donc à notre méthode la liste contenant la vision du mineur. Grâce à cette liste et une variable globale contenant la position des monstres au temps t-1, il va donc comparer la position au temps t-1 et t pour savoir dans quelle direction telle monstre est allé. Avec ceci, il va utiliser l’algorithme de déplacement d’un monstre pour nous donner la prochaine action du monstre. En utilisant cette action, nous modifions la liste contenant la vision du mineur pour pouvoir ressortir une liste contenant la position du monstre au moment t+1. Finalement, la liste est retournée au prédicat principal ce qui va permettre d’éviter de marcher sur la case contenant le monstre au temps t+1.

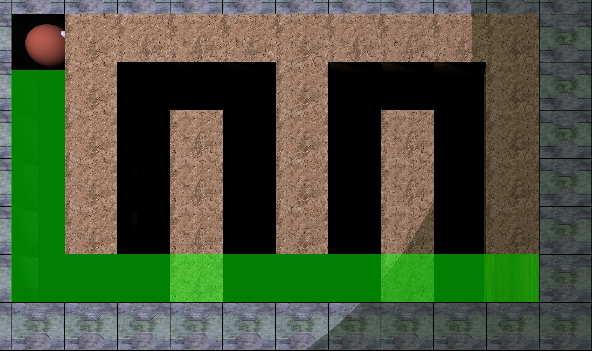
Résultat obtenu :

Les zones rouges représentent les zones de dangers calculées avant le trajet choisi par le mineur. Ainsi le mineur va contourner les monstres en évitant leurs zones de danger. Toutefois, comme les monstres vont se déplacer, le trajet sera recalculé tout au long de scénario pour pouvoir arriver à la sortie sans encombre.

## Recherche d’une destination finale :

Dans cet exemple, nous devons nous déplacer jusqu’à une certaine zone. Pour ce faire, l’algorithme A\* va nous aider. Nous prenons donc la position de notre mineur, le cout totale pour arriver à cette case (ici 0 car nous commençons de cette case), le cout du chemin qui est égale à la distance entre la case courante et la case finale auquel nous ajoutons +3 si la case correspond à de la terre et +10 si elle correspond à une pierre, et son père (ici définit comme étant [-1,-1] car il n’a pas de père). Nous ajoutons cette information dans une variable globale noté closeList et nous cherchons les mouvements possibles. Nous vérifions que ces mouvements n’existent pas dans les variables globales open et close, si tel est le cas, nous calculons son coût, son heuristique et nous l’ajoutons dans open. Si cela n’est pas le cas, nous vérifions que son coût total actuel est inférieur au cout total contenu dans une des deux variables globales. Si tel est le cas, on sait que nous pouvons atteindre cette position d’une façon plus efficace, nous extrayons donc ce mouvement de la liste globale qui le contient, nous modifions ses valeurs et le rajoutons dans openList car nous devons revoir les chemins qui le succèdent car leurs coût pourrait aussi changer. Si aucune de ces deux conditions sont réussi, nous abandonnons la position courante et passons à la suivante. Une fois cela fait, nous prenons la position ayant la meilleure valeur f (heuristique + coût) de openList et nous vérifions si la position finale est atteinte. Si cela n’est pas le cas nous recommençons les mêmes étapes que précédemment.

Résultat obtenu :

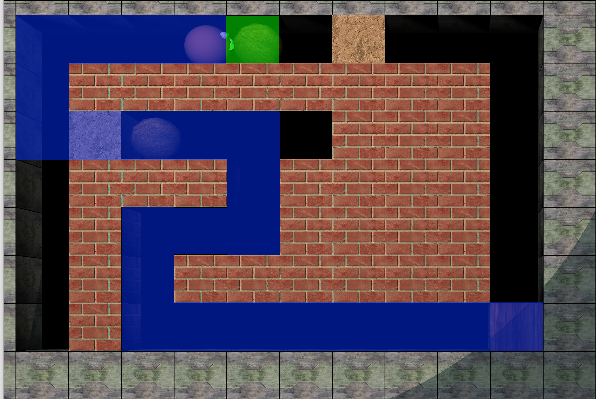


Le scénario est basique, le mineur calcule son chemin pour arriver jusqu'à la sortie, il n'évite pas la clay car le détour serai trop long.

## Ne pas passer par les rocher bloqués

Comme vu dans l’exemple précédent, l’algorithme A\* permet de trouver le chemin jusqu’à la sortie. Celui-ci accepte le passage par une pierre mais en considérant que le passage est plus « difficile », que son coût est plus important. Donc lorsque la pierre est suivie directement par un bloc de terre, l’algorithme considère que le chemin est faisable, mais pour un coût important. Or en réalité, ce chemin est impossible car une pierre ne peut être déplacée si elle est bloquée par « derrière ». Pour corriger ce problème, il faut identifier les pierres déplaçable et celles qui ne le sont pas. Pour ce faire, lorsqu’une pierre est un état successeur, on vérifie la direction vers laquelle arrive le mineur et ainsi on peut observer la case symétrique par rapport à la pierre, celle qui est possiblement bloquante pour la pierre, et tester si cette case est vide. Dans le cas où celle-ci est vide, le chemin est possible.

Résultat obtenu :

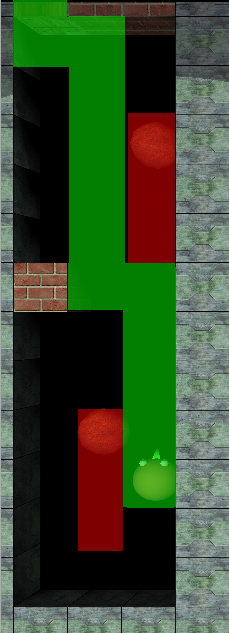


Le mineur va d'abord pousser le rocher (vert) d'une case car il se trouve entre lui et la sortie (sur le chemin le plus court). Mais une fois déplacé le rocher sera bloqué par la clay, c'est pourquoi il recalculera un autre chemin lui permettant d'accéder à la sortie. Ce second chemin est montré en bleu.

## Éviter les rochers en chute libre

VICTOR JE SAIS PAS :’(

Résultat obtenu :



Comme pour l'exemple avec les monstres, les zones en rouge sont les zones de danger des rochers en chute libre (deux cases sous les rocher pour éviter la collision). Le mineur va donc calculer son chemin, en tenant compte des rochers, pour pouvoir accéder à la sortie. Encore une fois, la position des rocher étant mise à jour tout au long du scénario, le chemin vert ne sera pas respecté car un des rocher va faire dévier le mineur.

# Difficultés rencontrées

Une des plus grandes difficultés que l’on a rencontrées était de savoir comment allons-nous faire pour errer dans le labyrinthe. Nous avons discuté de plusieurs solutions pour ce faire (méthode tabou, etc.). Finalement, nous sommes partis sur … VICTOR JE SAIS PAS :’(

# Améliorations possibles

Une amélioration possible pour notre IA serait l’optimisation des méthodes seDirigerVers et errer. La trop grande quantité de donnée à analyser en une seule fois rend l’algorithme assez lent. De ce fait, il faudrait trouver une solution pour élaguer des chemins ou faire en sorte de mettre de côté certains chemin (en utilisant une méthode tabou par exemple), ce qui permettrait de réduire le nombre de calcul et ainsi permettre une décision de direction plus rapide.