IA

Rapport final AI BootCamp

Guillaume Buchle, Iannick Langevin, Alexys Dussier, Pierrick Humbert, Redha Maouly

Table des matières

1 Introduction **3**

2 Description du modèle **3**
2.1 Sense 4
2.1.1 Construction de la carte 4
2.1.2 MaJ de la carte 5
2.2 Think 6
2.2.1 ObjectifsForAllNpcs 7
2.2.2 CheminsForAllNpcs 7
2.2.3 Exploitation 7
2.2.4 Score Stratégie 8
2.2.5 Exploration 8
2.2.6 Expedition 9
2.2.7 Réaffectation d'objectifs 9
2.2.8 Schéma UML des classes 10
2.3 Act 10

3 Techniques de résolutions **11**
3.1 Implémentation de l'Algorithme A* 11
3.2 Implémentation du FloodFill 11
3.3 Heuristique exploration vs exploitation 12
3.4 Gestion des portes 13
3.4.1 Notre porte préférée 13
3.4.2 Les portes à poignées 13
3.4.3 Les portes à switch, première partie 14
3.4.4 Les portes à switch, deuxième partie 14
3.5 Répartition des contraintes 16

4 Optimisations **17**
4.1 Profiler 17
4.2 Optimisations mineurs 17
4.3 Multi-Threading 19
4.4 Temporisation du tour 19

5 Analyses **20**
5.1 Justification des choix 20
5.2 Résultats obtenus 20
5.3 Avantages et Limites de notre implémentation 22

6 Conclusion **22**
6.1 Forces 23
6.2 Faiblesses 23

1 Introduction

AIBootCamp est un outil pédagogique ayant pour but d'initier à l'intelligence artificielle appliquée en jeu vidéo, sans avoir à se soucier des aspects connexes du développement de jeu vidéo comme le rendu ou les animations. Certaines limites nous sont imposées comme accomplir la carte en un nombre de tours fixe ainsi qu'une limite de temps processeur alloué pour chaque tour. De plus, les cartes comportent leur lots de défis à relever : un monde partiellement observable, gérer plusieurs agents simultanément et plusieurs objets avec des propriétés différentes.

Alors, nos agents devront développer certains comportements comme se déplacer, trouver des cases intéressantes à explorer, ouvrir des portes ou même coopérer entre eux pour arriver à leur but. En mesure de réaliser ses objectifs, nous nous sommes basés sur le modèle Sense/Think/Act pour la prise de décision de notre agent.

Les données captées par nos agents sont fournies par le moteur de jeu et elles sont utilisées pour mettre à jour le modèle de nos agents. Ce dernier permettra la prise de décision de nos entités, basée sur l'information perçue dans son environnement. Ensuite, l'action choisie est renvoyée au moteur pour l'exécuter qui renvoie les informations pour le prochain tour.

2 Description du modèle

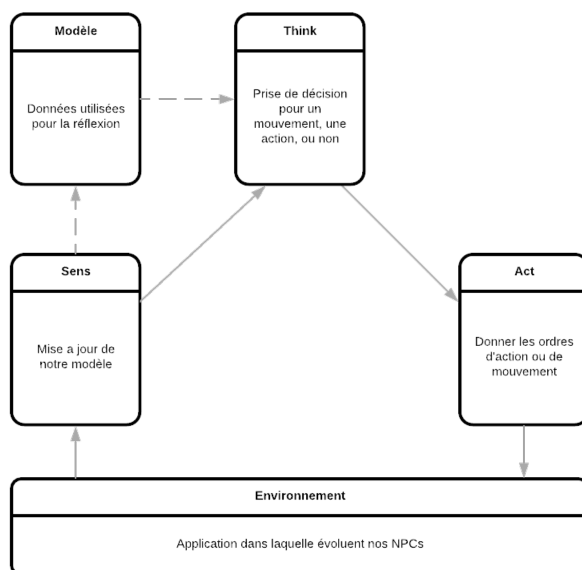


Figure 1 – Modèle Sens-Think-Act

2.1 Sense

2.1.1 Construction de la carte

Dans la méthode Init() de MyBotLogic on passe le 'levelInfo' au constructeur de notre GameManager. Puis, dans le constructeur du GameManager on crée l'objet Map en lui passant le 'levelInfo'.

Le quadrillage de la carte de notre modèle est représenté sous la forme d'un vecteur d'objets MapTile. Ainsi, on peut accéder à n'importe quelle case du niveau avec son identifiant instantanément.

L'ensemble des objets MapTile est instancié avec :
- un ID (le même que celui dans le niveau)
- les coordonnées 'x' et 'y', format de coordonnées en offset
- un type, soit TileAttributeDefault
- un statut, soit INCONNU
- un ensemble de voisins

Les voisins sont représentés par une structure regroupant l'identifiant de la tuile concernée, la direction pour se rendre a ce voisin ainsi qu'un bitset représentant les divers états de notre voisin.
Les états sont :

1. Mystérieux : une tuile adjacente à propos de laquelle on ne possède pas d'information, mais dont on sait qu'elle existe. Par exemple, une tuile juste après la frontière de vision d'un agent.
2. Accessible : une tuile adjacente sur laquelle le NPC peut se déplacer, c'est-à-dire qu'il n'y a pas d'objet bloquant, tel qu'un mur ou une fenêtre, et que la tuile n'est pas FORBIDDEN. Cela implique que les voisins mystérieux sont aussi considérés accessibles par défaut d'information supplémentaire sur leur type.
3. Visible : comprend tous les voisins avec l'état Accessible, mais aussi toute tuile adjacente que l'on peut voir par une fenêtre.

Après cela, on met à jour les objets MapTile avec les informations disponibles dans 'levelInfo'. On passe ces tiles à CONNU, on met à jour si besoin le type, si la tuile est GOAL on ajoute son ID au vecteur 'objectifs' de Map, si la tuile est FORBIDDEN on retire son ID du vecteur accessible de chacun de ses voisins et on retire son ID du vecteur mystérieux de chacun de ses voisins.

Un des gros intérêts de ce choix d'implémentation réside dans le fait que l'on peut très facilement encapsuler l'information complexe et se contenter de n'extraire que la sémantique qui nous intéresse. Par exemple nous ne sommes pas intéressé par savoir la position en x et la position en y d'une MapTile, ce qui pourrait dépendre du modèle de représentation choisi, mais nous sommes intéressé par savoir à un instant T quels sont les voisins accessibles depuis une MapTile.

C'est ce que notre modèle nous permet très facilement de faire en mettant à jour les voisins à chaque tour. Il ne nous reste ensuite plus qu'à regarder l'état des voisins pour avoir l'information voulue.

Ensuite, on enregistre chacun des objets connus de 'levelInfo' dans les vecteurs 'murs', 'portes' ou fenêtress' en fonction du type. Puis, on met à jour les états de nos voisins selon les objets a proximité. Enfin, on change le statut des tuiles sur lesquels les NPCs se trouvent de CONNU à VISITE.
On tiendra à préciser que chaque objet, comme une Porte un Mur ou un Activateur, est ensuite reconstruit selon nos propres classes éponymes pour pouvoir continuer à maintenir ce principe d'encapsulation et nous abstraire de la complexité du modèle.

2.1.2 MaJ de la carte

Cependant, nous devons mettre a jour les nouvelles informations a chaque début de tour, étant donné que les agents ont bougé au tour précédent et découvre de nouveaux éléments comme des tuiles ainsi que des objets. Ceci est le rôle de la fonction updateModel() du GameManager.
Une fois que l'on connaît le modèle pour le tour, il est intéressant de pré-calculer ce dont on pourrait avoir besoin pour la suite, on a donc choisi d'intégrer ici la mise à jour du flow de chaque agent via la fonction floodfill.

Lors de l'ajout de nouvelles tuiles, on regarde l'ensemble des tuiles visibles dans le turnInfo et si l'une d'entre elles est marquée comme inconnue alors on l'ajoute en utilisant la fonction addTile de la Map. Cette fonction addTile s'occupe à la fois de rajouter la tuile au modèle mais également de le maintenir dans un état cohérent. Ainsi si une tuile est découverte comme étant FORBIDDEN, non seulement son état est mis à jour, mais elle est aussi supprimée des voisins accessibles de ses propres voisins.

Lors de l'ajout d'un nouvel objet, on va vouloir l'ajouter à chaque vecteurs correspondant à son type. Un objet qui serait à la fois un mur et une fenêtre, pour prendre un exemple au hasard, sera donc ajouté à la fois à la liste des murs et à la liste des fenêtres de notre Map.

Une fois un objet ajouté, il nous faut également vérifier qu'elles sont les tuiles affectées par ce dernier. L'état des voisins concernés sera modifié pour représenter les caractéristiques de l'objet ajouté, comme un voisin n'est plus accessible si un mur se retrouve entre les deux tuiles.

Les flows de chaque agent nous permettent de pré-calculer l'ensemble des cases accessibles de cet agent, et en même temps de calculer la distance réelle (c'est-à-dire prenant en compte les obstacles) de notre agent à chacune de ses cases accessibles. Ces informations sont stockées dans l'agent lui-même. Elles ne sont valables que pour le tour en cours.
L'ensemble des cases accessibles d'un agent est stocké dans sa variable ensembleAccessible qui lui est propre. Cet attribut est représenté par une vecteur d'une structure composée

d'un identifiant de tuile ainsi que de la distance de cette case à partir de notre tuile actuelle.
Nous avions 2 choix d'implémentation pour l'algorithme FloodFill :

1) Choisir de ne calculer que l'ensemble des cases accessibles de chaque Npc via un système de partitions unifiées. Pour un temps de calcul faible.

2) Choisir de calculer l'ensemble des cases accessibles de chaque Npc mais également leurs distance dudit Npc. Pour un temps de calcul bien plus important.

Nous avons choisi la deuxième solution dans le but d'être plus flexible dans notre parti Think, en plus de pouvoir implémenter le comportement qui nous paraissait optimal sans avoir à faire de compromis.
Alors bien sûr ce choix est accompagné de nombreuses conséquences, aussi positives que négatives, et nous avons donc pris l'ensemble des décisions ultérieures en prenant cela en compte. Il s'agit d'un choix mûrement réfléchi.

2.2 Think

En ce qui concerne la prise de décision de nos agents, nous avons opté pour un BehaviorTree qui définit le comportement de tous nos Npcs simultanément. Ce dernier est décomposé en 3 étapes :

— Exploitation : si notre modèle est suffisamment complet pour trouver un chemin de chaque joueur à chaque objectif, alors on calcule ces chemins et les agents vont se diriger vers ces objectifs ! C'est le cas où tout se passe bien.

— Expédition : si on voit suffisamment d'objectifs pour tous nos agents, mais que l'on n'arrive pas à les atteindre, alors tous les agents vont associer un score à toutes leurs cases accessibles. Ce score est une heuristique de la probabilité que cette case nous rapproche d'un objectif. Les agents se dirigeront ensuite vers leur case de meilleur score. /

— Exploration : si on ne voit rien, ou pas suffisamment d'objectifs par rapport au nombre d'agents, alors tous les agents vont partir à la recherche de nouveaux objectifs ! Même ceux qui sont déjà proches d'un objectif, si quelqu'un n'a pas d'objectif, alors tout le monde l'aide à en trouver un. Car la carte n'est finie qu'une fois que tous le monde arrive à son but : on ne laisse personne derrière !

— CheckingHiddenDoors : Il existe également une quatrième étape consacré à la fouille des portes cachées. Comme cette branche est spécifique à la résolution d'un problème précis, nous n'en discuterons pas ici mais dans la section appropriée.

Pour implémenter ce BehaviorTree, nous avons utilisé le patron de conception Composite. Ainsi, on a une classe générale abstraite BT_Noeud qui correspond à tous les noeuds possible de notre arbre. BT_Composite est un noeuds qui possède au moins un enfant, il s'agit par exemple d'un séquenceur ou d'un sélectionneur tout comme il peut également s'agir d'un négateur ou autre ! Enfin, les feuilles de notre arbre sont des BT_Feuilles et modélisent les actions de calculs de notre algorithme tel que Exploitation, Expédition et Exploration. Chacun de nos nœuds retourne un état qui peut soit être Réussite, Echec ou En_Cours.

Voici le Behavior Tree de notre IA générale que nous avons implémenté :

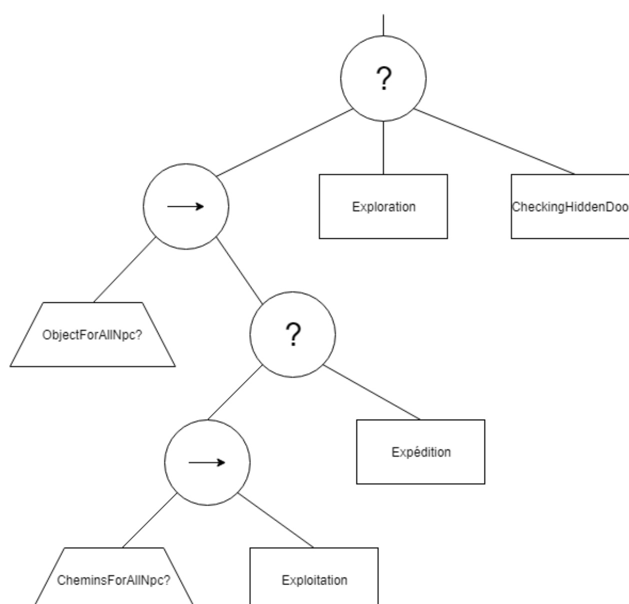


Figure 2 – Représentation du Behavior Tree

2.2.1 ObjectifsForAllNpcs

Pour vérifier si chacun des NPCs à un but attribué, on regarde simplement si le nombre d'objectif attribué est supérieur ou égal au nombre de NPC sur la carte.

2.2.2 CheminsForAllNpcs

Dans le cadre de ce test, on parcours l'ensemble des NPCs. Pour chaque Npc, on regarde si il est possible de lui assigner un objectif, si c'est le cas on ne lui assigne pas tout de suite mais on retire l'objectif de la liste des objectifs. Si ce n'est pas le cas, le test retourne ECHEC. Si chacun des NPCs trouve un objectif assignable, alors le test retourne REUSSI.

2.2.3 Exploitation

Pour chacun des NPCs on regarde sa distance à tous les objectifs, et on retiens cette distance dans un vecteur de d'objectifs possibles.
Dès qu'on l'on connait toutes les distances, on commence par assigner un objectif au Npc qui a sa distance minimale aux objectifs la plus longue, puis on passe au Npc dont la distance minimal aux objectifs est la deuxième plus longue. Et ainsi de suite pour tous les Npcs.

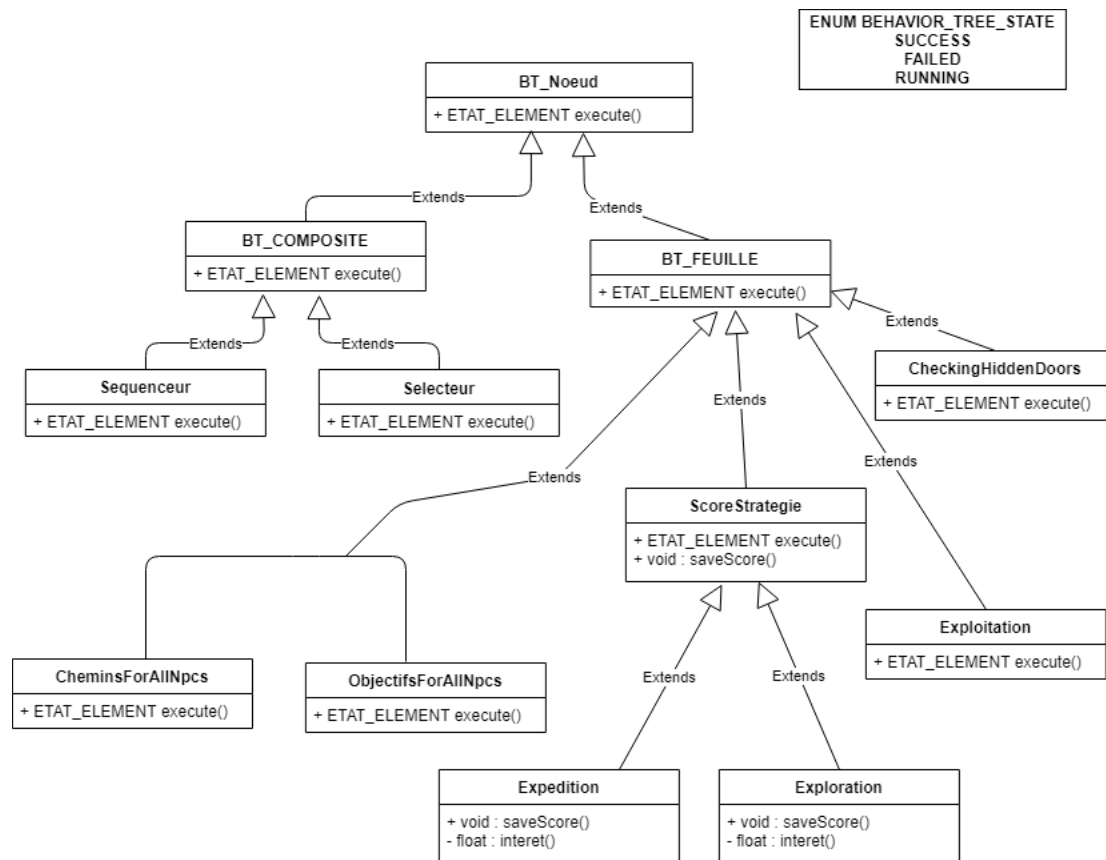


FIGURE 3 – Architecture choisie pour le Behavior Tree

2.2.4 Score Stratégie

Si on ne connaît pas assez d'objectifs pour tous nos Npcs, ou si l'on ne connaît pas de chemins pour aller vers ceux-ci, alors deux stratégies peuvent être mises en place, agissant de manière très similaire.

On décide d'attribuer un score à toutes les cases atteignable d'un Npc selon différents facteurs, influencé chacune par un coefficient propre. Et la tuile ayant le score le plus grand, sera la case vers laquelle notre NPC va se diriger.

Si une tuile a déjà été visité, ou si elle ne possède pas de voisins mystérieux, alors nous n'attribuons pas de score à cette tuile, car elle n'est pas intéressante à visiter.

2.2.5 Exploration

Tant que nous ne connaissons pas assez d'objectifs pour tous nos Npcs, nous appliquerons la stratégie **Exploration**. Nous considérons, que tant que nous avons de la main d'œuvre, autant l'utiliser pour explorer un maximum notre grille et obtenir de l'information.

L'Exploration prendra en compte dans son score :

— La *distance de la tuile visée à la position du Npc* : Notre Npc favorisera les tuiles proches de lui, afin de limiter un maximum les aller-retours, mais surtout car nous

voulions que si le Npc prend une décision (choisi une direction), qu'il s'y tienne au maximum.
— La *distance de la tuile visée aux tuiles visées par les autre Npcs* : On souhaite s'éparpiller un maximum pour éviter d'explorer la même zone avec plusieurs Npcs, quand un seul suffit.
— L'*intérêt de la tuile* : Le nombre de voisins mystérieux de la tuile.
— L'*intérêt accessible de la tuile* : Le nombre de voisins mystérieux accessibles de la tuile.
— L'*intérêt visible de la tuile* : Le nombre de voisins mystérieux visibles de la tuile.

On notera un des intêrets voulus de cette méthode d'Exploration : tant qu'au moins 1 Npc n'a pas d'objectif vers lequel se diriger, alors tous les Npcs vont l'aider à en trouver un. En effet le but est que tout le monde arrive le plus vite possible, et non pas qu'une partition d'entre-eux soit rapide tandis que le reste soit à la traîne.
D'ailleurs, l'entraîde est au coeur des valeurs du DDJV :)

2.2.6 Expedition

Si nous connaissons au moins autant de tuiles objectifs que de npcs, mais pas de chemins pour les atteindre, nous chercherons donc à nous diriger vers ces objectifs pour déterminer des chemins nous y menant, et appliquerons la stratégie **Expédition**. Nous calculerons le score de la même manière qu'en Exploration en ajoutant un facteur :
— la *distance hexagonale de la tuile visée aux objectifs* : Comme on ne connaît pas de chemin vers ces tuiles objectifs, on va chercher à s'approcher relativement d'elles, en considérant la distance hexagonale, c'est-à-dire la distance entre les deux cases sur la grille sans objets.

2.2.7 Réaffectation d'objectifs

A ce point du code, tous nos Npcs savent exactement où ils veulent aller. Seulement ils ont presque choisi leur objectifs indépendemment les uns des autres. Et il pourraît souvent être intéressant pour eux de collaborer pour échanger d'objectifs.
C'est ce que l'on fait dans la méthode reaffecterObjectifsSelonDistance où pour tous les couples de Npcs, on regarde si le temps de trajet global serait strictement diminué si l'on faisait s'échanger les objectifs de nos 2 Npcs, si c'est le cas alors on fait l'échange, sinon on ne fait rien. Comme un échange pourrait entraîner un autre échange alors on se doit de boucler : tant qu'il y a eus au moins 1 échange, alors vérifier à nouveau toutes les permutations pour s'assurer qu'il n'y pas de nouvel échange possible.

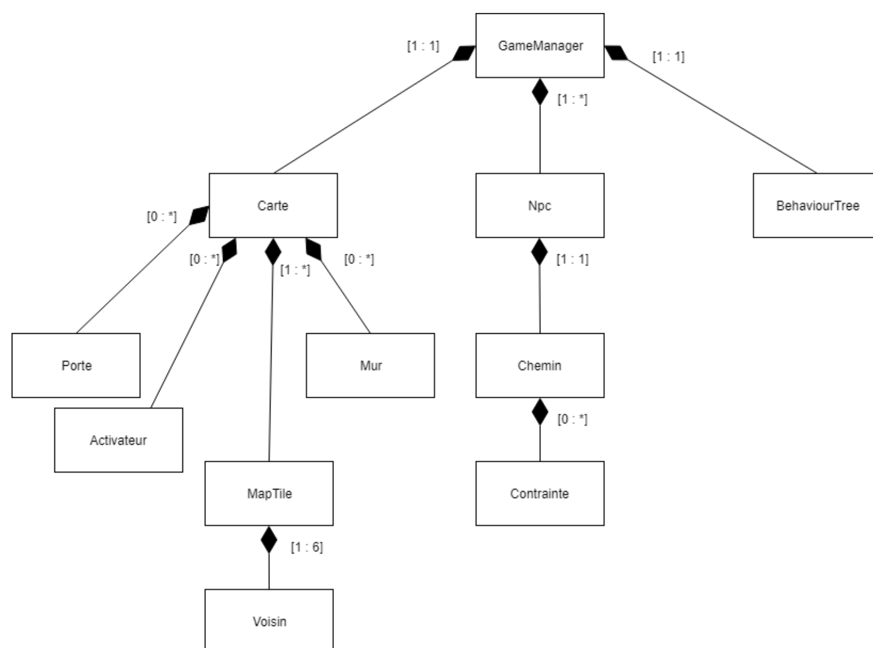
Il existe aussi des permutations cycliques à partir de 3 Npcs. Il se trouve que ces permutations sont équivalentes en temps globales, mais peuvent faire tourner en rond notre algorithme. Pour nous prévenir de ce cas, nous avons rajouté un nombre d'itérations maximum.

On notera qu'il s'agit d'un algorithme d'affectation optimal, cependant cet algorithme est très coûteux car on doit connaître la distance de chaque Npc à chaque objectifs poten-

tiel. Ce qui si on a autant d'objectifs que de Npcs, alors on se retrouve à devoir calculer $O(n^2)$ distances où n est le nombre de npcs.
C'est une des raisons pour lesquels nous avons choisi d'implémenter notre FloodFill comme nous l'avons fait, pour pouvoir être optimal dans la réaffectation des objectifs. Car bien sur, il n'était pas envisageable d'effectuer $O(n^2)$ aStars !

Pour le moment, tous nos Npcs savent exactement où ils veulent aller et combien de tours cela va leur prendre, mais ils n'ont toujours pas effectuer le coûteux calcul de chemin. C'est ce que l'on fera une fois pour chaque Npc à la fin des réaffectations.

2.2.8 Schéma UML des classes



2.3 Act

Maintenant que nos Npcs savent ce qu'ils doivent faire, encore faut-il qu'ils le fassent ! Un Npc peut vouloir faire plusieurs choses différentes, comme se déplacer, activer une porte ou encore vérifier si un mur est une porte cachée. C'est notre fonction getAllMouvements() qui s'occupera de récupérer les actions qu'à calculé notre Npc pendant le tour.

On va maintenant ordonner nos mouvements, pour que les plus importants soient les premiers à être exécutées. Les mouvements les plus importants sont ceux qui vont sur une case de switch activant une porte. Et les mouvements les moins importants sont ceux qui en sortent.

Une fois en possession de tous ces mouvements, il est encore possible que certains entre en conflits, par exemple si deux Npcs veulent aller sur la même case. Dans ce cas-là on laisse la priorité à celui dont le chemin est le plus long, l'autre Npc restant sur place. Bien

sur cela est récursif, pour que si un Npc est stoppé alors il puisse stopper d'autres Npcs éventuels attendant derrière lui.

Il ne nous reste plus qu'à appliquer ces mouvements, à savoir les rentrer dans notre actionList pour les envoyer au serveur et mettre notre modèle à jour en fonction des mouvement effectivement effectués par nos Npcs.

3 Techniques de résolutions

3.1 Implémentation de l'Algorithme A*

L'algorithme A* est utilisé dans le domaine du jeu vidéo principalement pour trouver un chemin optimal dans un graphe composé d'une multitude de noeud. Chaque noeud parcouru se voie attribuer un score, déterminé par une fonction d'évaluation et une fonction de coût, qui indiquent si ce noeud rapproche notre agent de son but.
Le coût d'un noeud représenta la distance de ce noeud à notre agent.
L'évaluation d'un noeud représente notre estimation de la distance de ce noeud au point d'arrivée. Pour cette estimation nous avons choisi la distance hexagonale, c'est-à-dire le nombre de déplacements à faire du noeuds au point d'arrivée sans tenir compte des obstacles.

L'heuristique d'un noeud est la somme du coût et de l'évaluation de celui-ci, c'est cette valeur que l'on cherchera à minimiser pour trouver le chemin le plus court. On conserve dans une structure de données les tuiles parcourues depuis la case de départ de manière à être capable de reconstruire le chemin parcouru lorsqu'on atteint la tuile recherchée. A chaque itération, on choisit le noeud possédant le plus petit score et on évalue ses cases voisines. Si on choisit une tuile qui a déjà été évaluée, on doit conserver celle qui détient le meilleur score pour empêcher les aller retour à l'infini. Notre algorithme de recherche de chemin gère également les portes.

3.2 Implémentation du FloodFill

L'algorithme de FloodFill est utilisé pour permettre de déterminer, a la première vue, toutes les tuiles théoriquement accessibles pour un agent dans le tour présent.
Généralement, on associe une valeur identique à chacune des tuiles atteignables par notre agent, ce qui nous évite de faire un A* sur une case qui n'est simplement pas atteignable pour le moment. Dans notre implémentation, chacun des agents possède sa propre carte et elle est recalculée a chaque mise a jour du modèle. Cependant, la valeur attribuée est différente du cas général, elle représente le temps pour que l'agent se rende de sa tuile actuelle à la tuile inondée. Ces données sont extrèmement utile lorsque nous voulons déterminer rapidement le meilleur but pour un agent.

Le temps de déplacement calculé dans le floodfill représente la plupart du temps la

distance de l'agent à la tuile sur laquelle il veut se rendre, mais lorsque l'on commence à prendre en compte les portes, ce temps de déplacement est utile car il représente également le temps que prendra l'agent à ouvrir une porte. Ou il représente encore le temps qu'un agent devra attendre devant une porte à switch le temps qu'un autre agent active le switch de la dite porte.

3.3 Heuristique exploration vs exploitation

Dans le but de définir les objectifs de chaque NPC, nous avons décidé de définir un score pour chaque tuile qui va représenter l'utilité de visiter cette tuile.
Ce score va être légèrement différent selon si on est en phase d'exploration ou d'expédition. En phase d'exploitation, on ne calcule bien évidemment pas ce score car dans ce cas la nous avons déjà connaissance des buts et de leurs chemins.

Pour l'exploration, le but est de découvrir de nouveaux buts. Le score calculé pour chaque tuile va être calculé selon différents paramètres. Le premier étant que la tuile doit pouvoir être accessible par le NPC en question. Pour ça on ne calcul un score que dans l'ensemble des cases accessibles par ce Npc pour s'en assurer. Ensuite on calcule le score en ajoutant les différents paramètres que l'on multiplie par un coefficient correspondant. C'est en jouant sur ces paramètres que l'on peut obtenir facilement des résultats variées et intéressants. Ces paramètres sont :

- La distance du NPC à la tuile. Car on veut chercher en priorité les cases proche de nous.
- La distance moyenne de la tuile aux autres tuiles qui seront visités. Pour éviter que tous les Npcs cherchent dans la même direction.
- Le degré d'intérêt de la tuile. Car il est plus intéressant de chercher une tuile avec beaucoup de potentiel d'ouvertures qu'une tuile dans un cul de sac.

L'intérêt de la tuile est calculé en multipliant le nombre de voisins de la tuile qui sont accessibles ou non accessibles mais visibles , par un coefficient correspondant.

La gestion des portes (qui sera présenté dans la partie suivante du rapport) est elle pris en compte, quand on calcule la distance entre le NPC et la tuile. C'est à dire que l'on aggrandie la distance si il est nécessaire d'atteindre un objectif intermédiaire (plaque de pression) avant d'arriver au but.

Nous avons choisi un poids de -12 pour la distance du Npc à la tuile car on veut pénaliser la distance, et que l'on veut lui donner un poids qui surpasse toujours le poids associé à l'intêret, ainsi l'intêret avec son poids de 1 (pour les fenêtes) ou 2 (pour les cases accessibles) sera toujours inférieur à une variation de 1 de la distance du Npc à la tuile, ce qui fait qu'une tuile peut intéressante mais proche sera toujous priorisé sur une tuile très intéressante mais plus éloingné, c'est le résultat voulu.

De même, le poids de la distance de la tuile aux autres tuiles visités est de 12 donc le même poids que la distance du Npc à sa tuile car on veut maximiser la distance entre les tuiles et que l'on veut faire valoir cela autant que la distance au Npc.

Pour l'expédition, le but est de découvrir les chemins qui mènent à un but. Le score calculé va légèrement changer de la phase d'exploration. Lors du calcul du score on va rajouter un paramètre (ayant lui aussi son coefficient correspondant) : la distance moyenne de la tuile à tous les objectifs. Ce paramètre nous permet donc de concentrer les recherches de chemins vers les objectifs. Nous avons choisis un poids de -12 également car il nous semble aussi important d'explorer des cases proches que de se diriger vers l'objectif.

En implémentant ces différents calculs de score nous arrivons donc à avoir un comportement de NPC qui permet de résoudre logiquement une grande partie des cartes. Il reste néanmoins la gestion des portes et de la collaboration en NPC, qui seront expliqués dans les prochaines parties.

3.4 Gestion des portes

Tant que l'on ne rajoute pas les portes, notre algorithme possède une logique optimal, nous avons donc voulu conserver ses propriétés en rajoutant les portes.

Nous avons donc cherché à intégrer les portes de tel sorte qu'elles soient aussi transparentes que possibles dans notre implémentation et que nous n'ayons pas à nous en préoccuper lorsque nous écrivons des algorithmes de plus haut niveau.

Nous avons donc suivi un principe tout au long de cette implémentation des portes : Ignorer les portes.

3.4.1 Notre porte préférée

Pour cette raison, la porte qui rentre le mieux dans notre modèle est la porte ouverte. En effet il suffit de ne jamais se préoccuper de cette porte et tout continuera de se passer de la même façon que auparavant, puisque cette porte se comporte exactement comme si elle n'existait pas !

3.4.2 Les portes à poignées

Les portes à poignées se comportent finalement, même si elles sont fermées, très similairement à une porte ouverte. En effet, même si l'on ne peut pas voir à travers, on peut toujours passer au travers, il nous suffit de l'ouvrir une fois devant.
Ainsi nos algorithmes tel que le floodfill ou le aStar ou pour ordre d'ignorer une porte si c'est une porte à poignée, car on sait que le Npc pourra toujours la traverser. Ainsi floodfill et aStar traversent ces portes, mais en augmentant leur poids d'une valeur de 1 supplémentaire car il faut bien prendre en compte le tour pour activer la porte.

Si l'on peut abstraire les portes à poignées dans le Think, on ne peut cependant pas le faire dans l'Act, en effet si un Npc cherche à passer à travers une porte à poignée fermée il va avoir des problèmes. Pour cela, dans notre Act, on vérifie si ce cas-là se présente, et

dans ce cas le Npc reste sur place et active la porte à la place.

Seulement il nous reste un problème. Dans notre implémentation actuelle, on ne cherche à accéder à une case que si un Npc la voit, c'est-à-dire que si une tuile positionné juste derrière une porte n'est pas visible, alors nos Npcs n'auront jamais l'idée d'y aller car ils ne savent juste pas qu'elle existe ! Pour pallier à ce problème, nous avons utilisé une technique d'illusionnisme, qui finalement caractérise le fait que l'on sait qu'il y a quelque chose derrière une porte : les deux cases adjacentes à une porte sont supposées comme existante, même si elle ne sont pas visibles.

3.4.3 Les portes à switch, première partie

Les portes à switchs sont un problème bien plus complexe car elle nécessitent l'interventions de plusieurs Npcs en simultané pour permettre à un seul Npc de se mouvoir à l'endroit souhaité.

Nous nous sommes fixé comme objectif de pouvoir passer n'importe quel configuration de portes et de switchs qui soit solvable.
Et c'est ce que fait notre algorithme en théorie.

La première chose que l'on a essayé de faire a été de tenter d'ignorer les portes à switch. Et il existe un cas où cela est possible : le cas où il existe un switch juste devant la porte que l'on veut franchir. Dans ce cas-là nous n'avons pas besoin de l'aide d'un autre Npc pour la franchir, il nous suffit juste d'ignorer la porte, comme avant !
On notera toutefois 2 choses :

1) on ne peut ignorer ce genre de porte la plupart du temps que dans 1 seul sens.

2) dans le cas où il n'y a qu'un seul Npc du coté traversable de la porte, il est possible que cette porte coupe la carte en 2, dans ce cas notre Npc y serait piégé.

Nous n'avons pas géré ce deuxième cas pour le moment. Un humain ne se laisserai pas avoir par un tel piège mais nos Npcs actuels si. Si l'on voulait y remédier, il faudrait simplement vérifier si l'espace après la porte est potentiellement disjoint de l'espace avant la porte et qu'il n'y a pas d'objectif dans l'espace après la porte, si tel est le cas on mettrai un poids de -1000000 à cette case, de telle sorte à l'explorer mais après s'être assuré qu'il n'y a vraiment plus rien à faire dans la première zone.
Cependant, cette heuristique dépendrait également du nombre de tour restant et pourrait créer d'autres problèmes sur d'autres cartes où l'on aurait besoin de rapidement traverser une telle carte. Pour cette raison, nous ne l'avons pas implémenté.

3.4.4 Les portes à switch, deuxième partie

Maintenant, comment gérer le cas qui finalement est le plus intéressant ?

Par exemple comment gérer le cas où pour que le Npc A passe par la porte 1 il faille

que le Npc B passer par la porte 2 pour activer le switch α et donc que le Npc C atteigne le switch β ?

Il y a en fait 2 problèmes distincts que l'on veut résoudre, mais nous allons essayer de les résoudres simultanément :

1) On veut pouvoir savoir si une porte à switch est traversable dans le floodfill, sans forcément savoir comment organiser nos Npcs de manière à ouvrir cette porte.
2) On veut pouvoir savoir, sachant qu'une porte à switch est traversable par un Npc, comment organiser nos Npcs de manière à ouvrir cette porte ?

Nous avons pour cela modifié à la fois le floodfill et le aStar. Les deux ont subit la même modification, lorsque l'on essaye d'ajouter un voisin, on regarde si pour ajouter ce voisin on passe par une porte. Si c'est le cas, alors on n'ajoute notre voisin que si il existe une façon de traverser cette porte.
Floodfill ne retiens pas comment on passe par une porte, mais il retient le temps que ça prend de traverser cette porte.
AStar retiens comment l'on doit traverser cette porte, il stockera cette information dans le chemin qu'il retournera.

Pour savoir comment traverser une porte, nous avons introduit le concept de Contrainte. Une contrainte caractérise le fait que pour un Npc spécifique parcourt un chemin, nous avons besoin de passer par une porte à switch dont le switch n'est pas trivial.
Une contrainte est considérée comme résolue une fois que l'on a trouvé un Npc disponible, donc qui n'est pas le Npc du chemin actuel, duquel existe un chemin à un des switch de notre porte.

Ainsi une contrainte résolue est constituté de la porte qui nous intéresse, mais également du Npc associée et du chemin que doit prendre cet Npc pour activer cette porte. Mais donc, le chemin de cette contrainte peut également contenir d'autres contraintes qui elles-mêmes peuvent contenir d'autres chemins etc. Il s'agit d'une structure de donnée récursive.

Il nous reste maintenant à remplir cette structure récursive.
Lorsque l'on regarde si l'on peut traverser une porte, on regarde pour tous les Npcs disponibles si au moins l'un d'entre eux peut atteindre un des switchs de cette porte. Pour cela on appel notre algorithme aStar de la position du Npc choisi à la position du switch choisi, ce qui fait que l'on pourrait éventuellement être amené à vérifier si on peut traverser une autre porte.

Notre algorithme récursif est donc en place, il nous reste à traiter les cas de base :

- Si aucun Npc ne parvient à trouver un chemin de sa propre position à l'une des positions des switchs, alors on ne peut pas passer cette porte.
- Si un Npc parvient à trouver un chemin pour ouvrir cette porte mais que ce chemin nécessite qu'un autre Npc passe lui-même par la même porte que l'on essaye de franchir, alors ce chemin n'est pas valide.
- Si au moins un Npc parvient à trouver un chemin de sa propre position à l'une des

positions des switchs, alors on peut passer cette porte.
— Si plusieurs Npc parviennent à trouver un chemin, alors on garde le chemin le plus rapide en temps.

Lorsque l'on garde un chemin qui passe par une porte, celui-ci contient alors des contraintes qui sont résolues, c'est-à-dire avec un autre Npc associé et un chemin pour cet Npc.

3.5 Répartition des contraintes

Maintenant, tous nos Npcs savent où ils veulent aller et comment y aller. Ils savent également de qui ils ont besoin pour les aiders à atteindre leur but (qui n'est pas forcément un objectif soit dit en passant).

Cependant, si un Npc A a besoin d'un Npc B pour passer une porte, mais que le Npc a besoin du Npc A lui aussi pour ouvrir une porte, ils vont chacun aller sur leur switchs respectif et attendre à l'infinie sans espoir que la situation se débloque.
Pour cette raison, on a besoin de définir un système de priorité pour savoir quel Npc va imposer ses contraintes aux autres, et quels Npcs devront se soumettre aux contraintes des autres.
Nous avons choisi un critère simple, le Npc qui a le plus de chemin à faire est le Npc le plus prioritaire, tous les autres doivent l'aider car c'est lui qui prendra le plus de temps.
Pour implémenter cela, on affecte d'abord le Npc avec le trajet le plus court, si son chemin contient au moins 1 contrainte, alors on affecte uniquement sa première contrainte au Npc correspondant. On n'affecte que la première contrainte car il ne sert à rien de réquisitionner plusieurs Npcs trop tôt, chaque choses en son temps. Puis pour ce nouvel Npc affecté, on affecte à nouveau sa première contrainte et ainsi de suite.
Une fois le Npc avec le chemin le plus court affecté, on passe au Npc avec le deuxième chemin le plus court, et cette fois-ci on écrase les Npcs qui ont déjà étés affectés en changeant leur chemin si besoin. Ainsi, lors de l'affectation du dernier Npc, celui avec le chemin le plus et le chemin et le plus prioritaire, on est sur que toutes ses contraintes seront remplis ! Et on est également sur que les autres Npcs auront tous quelque chose à faire de cohérent.

On notera que pour affecter les contraintes de manière optimal, il est important que l'algorithme de réaffectation d'objectifs ait déjà été effectué. Cependant, il est important d'effectuer l'algorithme de réaffectation d'objectifs à la fin du calcul des chemins des Npcs. Pour régler ce problème, nous devons appeler la fonction de réaffectation d'objectifs une fois avant d'affecter les contraintes, et une autre fois après leurs affectations. Mais comme cet algorithme est très rapide, ce n'est pas une grande perte !

4 Optimisations

4.1 Profiler

Quand on parle d'optimisation, on parle de performance, et donc de mesure. En effet, il est centrale, quand on cherche a optimiser, d'étudier son programme, et pour cela de mesurer le temps d'exécution de tel ou tel partie de celui-ci. Nous avons donc investi des d'efforts dans la mise en place d'un Profiler [1], un outils simple qui permet facilement de servir à la fois de chronomètre pour l'exécution d'une fonction que de logger [2], et ainsi gagner en simplicité d'utilisation pour le développeur et la personne qui débugge. En effet une instance de notre profiler écrit dans le logger qu'on lui a indiqué sa durée de vie. Ainsi pour mesurer le "cout" en temps d'une méthode et lui fournir la possibilité de laisser une trace textuelle, il suffit d'instancier un profiler au début de cette méthode.

En revanche, le but est d'avoir un objet qui prenne le moins de temps à l'éxecution, notamment dans la version livrée (version "Release"), et donc deux techniques on été pensées pour permettre de conserver cet outil de debuggage utile mais avec un impact limité sur la performance du programme :

- n'écrire dans le logger qu'à la mort du profiler
- avoir une version avec des méthodes qui ont une implémentation vide (facilement activable à la compilation)

Pour rendre le profiling plus agréable à lire, nous lui faisons générer un fichier exploitable par l'outil de Profiling de Google Chrome afin d'obtenir des graphique tel que ceux partie 5.2

4.2 Optimisations mineurs

- Créer une classe Voisin
  Nous avions quatre structures de données pour emmagasiner les différents états de nos voisins (Existe, Accessible, Visible et Mystérieux) ainsi que plusieurs méthodes avec quatre variantes pour représenter chacun des états. Nous avons donc créé une structure avec un bitset représentant chacun des états, modifié les méthodes en rajoutant en paramètre le type de l'état recherché et supprimer ceux qui n'étaient plus nécessaire. De plus, nous avons ajouté une méthode qui retourne tous les voisins partageant un état spécifique pour simplifier certaines boucles.

- Alléger les retours de fonctions
  Certaines de nos fonctions retournaient toutes les propriétés de l'objet au lieu de la propriété qui nous intéressait. Par exemple, une méthode retournait une instance de la classe Tuile, alors que seulement son indice était nécessaire pour la suite des calculs.

- Bien choisir les structures de données
  Dans notre code, nous utilisions beaucoup de std : :map qui facilitait la recherche

1. Analiseur de code
2. Enregistreur d'historique des événements

en associant la valeur a une clé, ce dernier étant généralement l'identifiant de la valeur. Cependant, cette structure offre des performances moindres qu'un vecteur lorsque nous devons itérer sur les membres, car elle n'est pas contiguë en mémoire comme le vecteur. Alors, nous avons défini une structure composée de la clé ainsi que de la valeur et si nous avons besoin de rechercher une certaine donnée, nous pouvons utiliser les algorithmes std : :find_if avec une lambda pour spécifier ce qui nous intéresse.

— Journal de données
  Pour nous permettre de comprendre rapidement un problème, nous avons un journal de données enregistrant différentes étapes cruciales de nos agents. Certaines méthodes définissaient une chaîne de caractère en début de la fonction et concaténait l'information jugée nécessaire pour ensuite la passer a notre système de journalisation à la fin de la fonction. Chaque concaténation entraîne une allocation d'une nouvelle string, ce qui peut être facilement optimiser grâce a un flux comme std : :stringstream. Une seule allocation est faite lorsqu'on choisit de construire ce dernier.

— Passage par référence
  La majorité des paramètres ou du retour de nos fonctions étaient des copies. Alors, nous les avons modifié pour passer en référence (ou référence constante) lorsque c'était possible. Ceci évite de créer inutilement des instances lorsque nous savons que l'objet ne sera pas modifié dans la fonction.

— A*
  Au moment de choisir le noeud avec l'heuristique la plus intéressante, nous trions préalablement notre vecteur de noeud selon leur score attribué. Cette opération coûtait très cher étant donné que seulement le plus petit score nous intéressait. Nous avons donc utilisé la méthode find_if de la std pour trouver le meilleur candidat de manière a l'échanger avec le dernier élément du vecteur. Ainsi, notre prochain noeud a évaluer se retrouve toujours a la fin du vecteur, ce qui nous évite de le trier !
— Optimisation des boucles A de nombreux moments dans notre programmes, nous avons des boucles de plusieurs instructions. Il se trouve que le compilateur est incroyablement plus rapide si une boucle n'est constitué que d'une seule instruction, cette amélioration de performance est de l'ordre de 4 ou 5 fois plus rapide.
  Pour en profiter au maximum, nous avons remplacé nos boucles à plusieurs instructions les plus cruciales par des appels de fonctions qui font exactement la même chose que la boucle, mais en une seule instruction. Dans le but de permettre au compilateur d'optimiser.

4.3 Multi-Threading

Malgré les améliorations vu jusqu'ici, notre programme n'est pas encore assez rapide pour assurer de prendre une décision dans le temps impartie pour un tour. Afin d'augmenter notre rapidité de calcul sur les étapes les plus coûteuses de notre programme nous avons choisi d'utiliser du multi-threading. L'intérêt est ici double, d'un coté on augmente notre capcacité de calcul, et de l'autre nous pouvons laisser nos threads de calcul continuer même aprés la fin du tour durant le temps de calcul du serveur. Ce dernier point est très important car le temps de calcul du serveur est long (plus de 10 ms). Ainsi, si on détecte que nos threads de calcul ne sont pas tous terminés, il suffit de ne pas donner d'ordre de mouvement au NPCs et ainsi attendre le prochain tour pour avoir le résultat des calculs.

Le fait que l'on ne donne pas d'ordre de mouvements à nos Npcs est ici crucial, il permet de s'assurer que l'état du jeu n'aura pas changé entre le tour du début de nos calcul et le tour où l'on prendra une décision. Si l'on a pas d'information complète on ne fait rien. Si l'on a pas pû effectuer l'ensemble de nos calculs on ne fait rien non plus. Tels sont nos critères.

Pour que le multithreading soit efficace il faut l'utiliser à des endroits du programme qui ne nécessite pas d'écritures concurrentielles et que la tache à effectuer soit suffisamment conséquente pour que payer le cout de création des threads soit rentable. Les deux endroits qui s'y prêtent bien sont le calcul du FloodFill, ou un thread est créé pour chaque NPC et l'étape du "Think" qui a un thread pour elle seule, car c'est une étape longue. Mettre du multi-threading pour l'étape "Act" n'est en revanche pas judicieux car elle est déjà très rapide.

(voir les résultats section 5.2)

4.4 Temporisation du tour

Comme abordé dans la section précédente, nous avons fait un travail sur la mesure du temps restant à chaque tour pour effectuer nos calculs. Afin d'assurer que notre temps de calcul ne dépasse pas celui du tour, nous avons eu comme stratégie de considérer que nous avions toujours une milliseconde de moins de disponible. Grâce à cela, nous pallions les possible imprécisions dans le calcul des temps, liée à l'échelle de temps où nous sommes (la microseconde), et nous laissons ainsi une marge d'erreur suffisamment grande pour éviter tout dépassement du temps autorisé qui est, nous le rappelons, éliminatoire. Dans le peu probable cas où nos threads n'ont pas fini leur calculs même après le temps serveur, nous forçons notre thread principal (main) d'attendre jusqu'à la fin du temps du tour afin d'utiliser ce temps, et de ne pas "gacher" plusieurs tour pour rien. Sans cela, chaque fois que les threads de calculs n'auraient pas fini, alors on passerait au tour suivant directement sans utiliser le temps qui nous est alloué. En revanche, afin d'avoir le maximum de précision, nous sommes obligé de faire de l'attente active, ce qui est coûteux, mais ici nécessaire car la fonction sleep des threads à une précision de l'ordre de la milliseconde, ce qui n'est pas ici acceptable. (voir les résultats section 5.2)

5 Analyses

5.1 Justification des choix

— Algo A*
Les deux algorithmes les plus connus pour la recherche de chemin dans le monde du jeu vidéo sont le A* et celui de Dijkstra. Nous avons choisi le premier, car ce qui nous intéresse est de trouver le meilleur chemin possible sans nécessairement tester toutes les possibilités possibles. L'algorithme A* nous offre cette possibilité grâce a sa fonction heuristique qui détermine les candidats potentiels qui nous rapproche de notre but. L'algorithme de Dijkstra explore toutes les possibilités, ce qui est plus coûteux comparé au A* si seulement le chemin optimal nous intéresse.

— FloodFill
Nous nous sommes vites rendu compte que calculer un A* n'est pas gratuit et encore moins lorsque nous utilisons cette outil pour valider si la case est accessible. C'est pourquoi nous avons décidé d'utiliser le floodfill, car il nous permet de valider que notre agent peut se rendre a une case spécifique sans payer le coût du A*, qui est très coûteux s'il ne trouve pas.

— Behavior Tree
Au cours des trois itérations du projet, les comportements de notre agent ont énormément évolué. Nous avons utilisé un arbre de comportement au lieu d'une machine a états pour plusieurs raisons. Premièrement, une machine a états nécessitera forcément a évaluer l'état courant et ceci peut avoir un coût conséquent si notre agent peut se retrouver dans plusieurs états possible. Deuxièmement, les différents états peuvent se retrouver étroitement relié a d'autres états, ce qui signifie qu'une modification peut en entraîner une ou même plusieurs dans les autres états. Ceci le rend beaucoup moins flexible si notre machine a état se voit évoluer dans le temps comme AIBootCamp. Et ce n'est pas scalable. Un arbre de comportement ne contient pas ces désavantages et les noeuds peuvent être réutilisés sans avoir besoin de dupliquer le code comme ce serait le cas pour une machine a états.

5.2 Résultats obtenus

Tout d'abord, l'ensemble des cartes d'entraînement ont été passées avec succès, notre Bot a donc bien rempli sa mission. Les cartes donnant suffisamment de temps de calcul (au moins 10 ms par tour) sont toutes passées avec un nombre faible de tours. En revanche, pour celles qui laissent peu de temps de calcul, notre Bot met plus de tour, car il étale ses calculs sur plusieurs tour (cf. Partie 4.3). Bien qu'il utilise plus de tour, notre Bot reste assez rapide car dans le pire des cas il prend 2 tours de réflexion pour agir, et cela n'arrive finalement que très peu, car il faut qu'il y ait beaucoup de NPCs (cartes avec plus de 5 NPCs), que notre temps disponible soit faible (5 ms), que la carte soit grande et que l'on soit à une étape couteuse (l'Exploitation dans notre Behavior Tree).
Nous mettons ici les visuels d'étude des performances de notre Bot mesurées à la fois sur nos PC de développements et sur le serveur. Le niveau visualisé ici est volontairement un où il y a beaucoup de NPCs, afin de bien visualiser à la fois la claire différence de puissance

entre les deux plateformes, ainsi que la gestion des threads et du temps d'éxecution sur plusieurs tours.

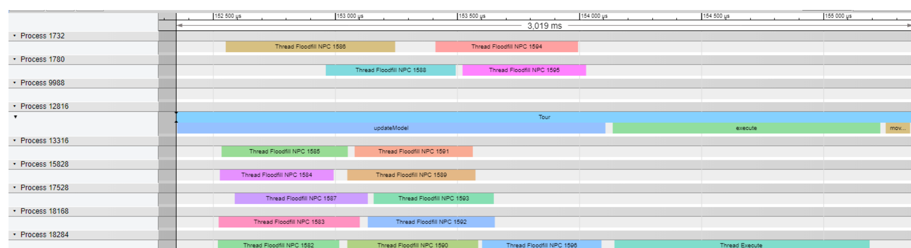


FIGURE 4 – Performance sur le PC de développement

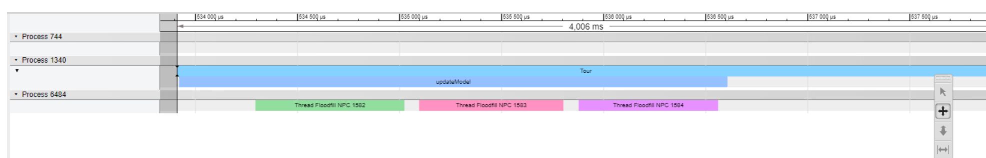


FIGURE 5 – Performance sur le serveur cible : tour 1, notre Bot n'a pas le temps de finir toutes ses tâches dans le tour

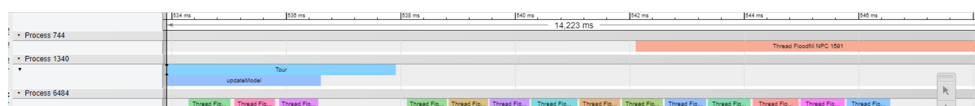


FIGURE 6 – Performance sur le serveur cible : tour 1, notre Bot utilise donc le temps serveur pour finir ses calculs

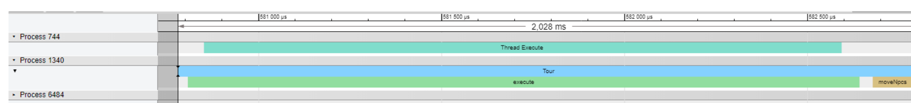


FIGURE 7 – Performance sur le serveur cible : tour 2, notre Bot a finit ses calculs, il passe à l'étape suivante

5.3 Avantages et Limites de notre implémentation

-MultiThreading
Notre utilisation du parallélisme dans notre code est un couteau a double tranchant. Idéalement, nous croyons qu'il aurait été préférable de recommencer AIBootCamp du départ en conservant certaines parties que de continuer a développer par dessus. Le principal problème était relié au fait que nos fonctions étaient trop coûteuse. Nous avons réglé ce problème grâce au parallélisme, mais il aurait été préférable d'optimiser ces fonctions. Cependant, nous avons vite réalisé qu'il y avait beaucoup de dépendances dans notre code, ce qui signifie que plusieurs modifications auraient été nécessaires pour en faire théoriquement qu'une seule.

-Implémentation en vues des cartes mystères
L'ensemble de notre implémentation à été réalisé en vue de valider un maximum de cartes mystères. Même si nous ne connaissons aucune des ces cartes nous avons imaginés quelques cas supplémentaires (en particulier dans le placement de certaines portes et interrupteur) afin de pousser notre implémentation plus loin. Nous ne pouvons pas savoir d'avances si nous allons réussir à passer un grand nombre de ces cartes mais nous validons sans problèmes nos nouvelles cartes.

6 Conclusion

Tout d'abord rappellons nous que notre Bot n'a pas vocation à être une IA scientifique, mais bien une IA pour le jeu-vidéo.

Cette subtilité implique qu'il n'est pas cruciale que les décisions prisent soient les meilleures où qu'elle soient semblables à celles que nous aurions prisent en temps qu'humain. Voila pourquoi la première conclusion à tirer de cette expérience est qu'il est plus important d'avoir une IA qui remplie sa mission dans un temps convenable et en ayant un temps de calcul court, plutôt qu'une IA très élaborée mais qui demande trop de ressources.

S'il est envisageable d'avoir une IA relativement générique dans le domaine scientifique, cela est semble très compliqué dans le cadre d'un jeu-vidéo. Ce challenge nous a fait expérimenter l'obligation, au vue du temps de calcul disponible, de faire une IA ajustée au design de notre jeu qui ne fait pas plus que nécessaire. Et cela s'est fortement ressenti lors de l'ajout de nouvelles contraintes au design du jeu : un modèle est fait pour répondre à un certain problème, et si celui-ci est changé, alors il faut repenser le modèle.
Même si nous avons fait de notre mieux pour que notre modèle soit facilement adaptable à l'ajout de nouvelles fonctionnalités.

Bien qu'il n'est pas demandé à notre IA d'être d'un très haut niveau de complexité, elle doit quand même remplir sa fonction (dans un temps raisonnable), et il n'est finalement pas choquant de la voir ne pas agir sur certain tour. A l'échelle d'un jeu-vidéo, cela représente quelque images, qui ne sont pas visibles par le joueur, dans un jeu à 30

images/seconde.

Le dernier point est que la grande difficulté fut de développer un programme pour une machine qu'on ne connaît pas et sur laquelle nous n'avons pas la main. Cette douloureuse expérience, nous rappelle bien que quand un jeu est crée, il doit pouvoir fonctionner sur plusieurs plate-formes qui n'ont pas toutes la même puissance de calcul (comme vu Section 5.2). D'où l'importance de garder en tête la nécessité de faire un programme adapté à notre cible, et de bien se renseigner sur celle-ci.

6.1 Forces

— Ne dépend pas du type de carte (Vision, Porte, etc.)
— Meilleur contrôle sur le temps de calcul
— Outils de debug adaptés et efficaces

6.2 Faiblesses

.
— Bot coûteux en temps de calcul
— Dépendance de nos modules/outils rendant difficile les modifications