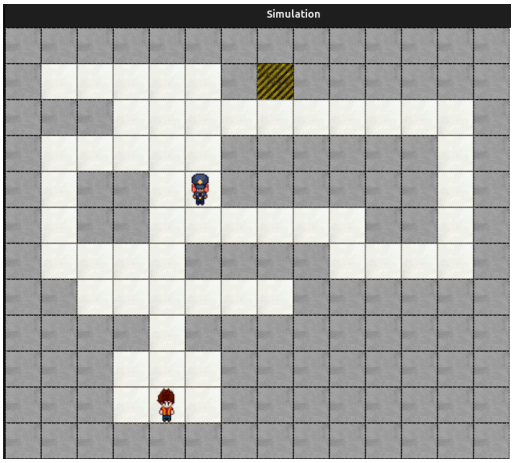
Bilan itération n°1:
Simulation de poursuite-évasion compétitive entre agents informatiques intelligents



Tuteur: Guénaël Cabanes

Matias Amaglio
Maëlle Bitsindou
Luc Dechezleprêtre
Célie Ponroy
Année universitaire 2024- 2025

Table des matières

I. Objectif de la première itération

L'objectif de cette première itération était d'avoir une simulation avec des bases solides et un premier comportement intelligent non évolutif. Pour optimiser au mieux les 22 heures la liste suivante a donc était faite:

- Base du projet
- Base du moteur de jeu
- Création de la vue du menu
- Création de la vue de la simulation interactive
- Mise en place de l'inférence bayésienne
- Création de l'arbre de décision du gardien
- Ajout du contrôle du prisonnier

Par rapport à notre étude préalable, nous avons principalement réussi à implémenter chacune des fonctionnalités ci-dessus sauf la mise en place de l'inférence bayésienne et la création de l'arbre de décision du gardien qui ne l'ont pas totalement été par défaut de temps.

I. Fonctionnalités développées

Lors de cette itération, nous nous sommes réparties le travail de façon équitable.

➢ Base du projet (Célie)

Pour bien commencer ce projet, il fallait commencer par la base du jeu : la grande majorité du jeu est géré par la classe **Simulation** cette classe contient une carte a deux personnages : prisonnier et gardien et gère les déplacements de ses personnages. La classe abstraite **Personnage** représente les personnages(gardien ou prisonnier). Deux classe en hérite:

- La classe **Joueur** qui représente le joueur de la simulation interactive
- La classe **Agent** qui représente une intelligence artificielle de la simulation.

➢ Base du moteur de jeu (Célie & Luc)

Pour le début de notre projet, nous avons décidé de créer un moteur de jeu permettant au joueur de jouer en tour par tour. Dans le sens où l'adversaire du joueur ne se déplacera pas lorsque le joueur ne se déplacera pas. Et il se déplacera lorsque le joueur se déplacera. L'ajout du temps réel dans la simulation est considéré comme une fonctionnalité avancée de notre application. Pour l'architecture globale de notre application nous avons fait le choix d'une architecture **MVC** (Modèle-Vue-Contrôleur) car notre application n'échangera pas avec des services extérieurs tel qu'une base de données par exemple. Et le MVC nous garantit une bonne séparation des différents composants de notre application. Ce qui nous permettra de travailler ensemble sans crainte de porter atteinte au travail d'un membre du groupe.

En l'état présent, notre modèle est une classe **Simulation** qui permet le déroulement d'une partie de poursuite/évasion. Cette classe possède des "observateurs" (ou "vues") qui se construisent en se basant sur ses données. Un objet de cette classe est créé au lancement de l'application et est connu du contrôleur pour qu'il puisse appeler les méthodes correspondantes aux actions de l'utilisateur. Dès que le modèle exécute une méthode qui va engendrer un changement pour ses attributs, il appellera une méthode qui se chargera de prévenir tous les observateurs pour qu'ils se mettent à jour. L'affichage graphique des vues est fait grâce à javaFX, le plus souvent nos vues héritent d'un objet javaFX pour éviter tout problème de comportement que javaFX ne comprendrait pas.

➢ Ajout du contrôle du prisonnier (Luc)

Pour l'instant le prisonnier est contrôlable par le joueur soit avec les neuf touches suivantes:

- A : aller en haut à gauche
- Z : aller en haut
- E : aller en haut à droite
- S : ne pas bouger
- Q : aller à gauche
- D : aller à droite
- W: aller en bas à gauche
- X : aller en bas
- C : aller en bas à droite

Nous avons aussi ajouté la possibilité de jouer avec le pavé numérique (respectivement 7, 8, 9, 5, 4, 6, 1, 2, 3) mais toutes les machines ne possèdent pas nécessairement un pavé numérique.

La gestion des évènements clavier est assurée par une classe **Clavier** qui implémente une interface javaFX : EventHandler<KeyEvent>. Cette implémentation nous garantit que notre classe s'occupera uniquement des événements déclenchés par le clavier. L'interface javaFX nous impose de définir une méthode nommée "handle" qui recevra l'événement clavier et nous le traitons en conséquence dans la méthode, c'est-à-dire en appelant la méthode du modèle correspondant à l'action souhaitée par l'utilisateur.

➢ Création de la vue du menu (Maëlle)

Pour pouvoir afficher le menu principal, il a fallu créer une classe **VueMenu**. Dans cette classe se trouve la méthode *afficherMenu* qui va créer une VBox contenant les différents boutons du menu. Néanmoins, même si la vue du menu principal a été implémentée, elle n'a pas encore été reliée à la simulation. Cette classe sera donc concrètement visible dans l'application lors de la prochaine itération.

➢ Création de la vue de la simulation interactive (Maëlle)

La classe **VuePrincipale** est la classe gérant de l'affichage de notre jeu. Elle implémente l'interface **DessinJeu**, qui définit la méthode *update*. Cette méthode est essentielle pour mettre à jour graphiquement l'état du jeu en fonction des données du modèle. Lorsqu'elle est appelée, elle redessine les éléments du jeu sur un *Canvas*, en se basant sur les informations reçues de l'objet Jeu.

Dans notre application, **VuePrincipale** est intégrée dans un *BorderPane*, où elle occupe l'espace central de l'interface. Dès qu'un changement est détecté dans le modèle de jeu, la vue principale, ajoutée comme observateur du modèle, est "automatiquement" informée. Elle exécute alors la méthode *update* pour refléter les modifications, par exemple lorsqu'un personnage se déplace ou qu'un autre événement se produit.

L'utilisation de cette classe apporte plusieurs avantages :

- Elle permet de maintenir l'affichage toujours à jour avec l'état du jeu.
- Elle respecte la séparation des responsabilités : la logique du jeu reste dans le modèle, tandis que l'affichage est géré par la vue.

Avec cette organisation, nous avons pu poser les bases graphiques de notre simulation.

➢ Ajout de la vision des personnages (Célie)

Nous avons mis en place un moyen de calculer ce que le personnage voit, sachant que sa vision est bloquée par les murs. Le personnage aurait la capacité de voir sur un carré de taille 7 cases sur 7 cases, cela lui confère alors 3 cases de vision de chaque côté. Sous les conseils de notre tuteur nous avons fait les calculs du champ de vision au préalable et séparément du lancement de l'application. Cela est possible car la carte reste la même d'une partie à une autre. Pour ce faire nous avons créé la classe **CalculVision** qui a une méthode pour calculer à une position donnée la vision du personnage: *calculerVision*. Cette fonction va retourner une liste de cases visibles selon la topologie de la carte. La méthode *calculerCarteVision* récupère ce résultat pour toutes les positions de la carte et le stocke à son tour dans une HashMap<Position,ArrayList<Position>>. Pour stocker ses résultats dans la machine le méthode *ecrireVision* va créer un fichier **vision.txt** *recupererVision* permet de remettre les résultats sous forme de HashMap pour pouvoir les réutiliser.

➢ Mise en place de l'inférence bayésienne (Matias & Luc)

L'inférence bayésienne est une méthode d'inférence statistique permettant de calculer des probabilités à partir d'observations. Appliquée à notre projet, l'inférence permet de créer des cartes mentales pour les personnages, non-joueurs contenant les probabilités de présence des autres personnages de la simulation.

Pour réaliser cette fonctionnalité, nous avons créé une classe **Bayesien** contenant un tableau de tableau de double représentant les probabilités de présence pour chaque case de la carte. La classe contient aussi un tableau de case dite valide (case n'étant pas un mur) ainsi qu'une méthode **calculerProbaPresence**. Cette méthode prenant en paramètre un tableau de tableau double représentant la carte avant le nouveau calcul de probabilité et une liste de casesVues correspondants aux cases vue par un agent. La probabilité Pn+1 pour une case i,j et un tour n est calculés à partir de l'équation suivante :

$$P_{n+1}(i,j) = \sum_{(k,l)\in Voisins(i,j)} P_n(k,l)\cdot T((i,j)|(k,l)),$$

où $T((i,j)|(k,l))$ *est la probabilité de transition, égale* à $\frac{1}{NbVoisins(k,l)}$.

Les objets **Bayesien** sont créés dans la classe Simulation lorsqu'un nouveau personnage non joueur est créé. On ajoute la carte bayésienne et le personnage dans une Hashmap **carteBayesiennes** qui permet d'associer les personnages non joueurs et leur carte mentale. On récupère ensuite cette carte du tour en cours pour calculer les probabilité de la nouvelle carte pour le tour suivant. On actualise alors **carteBayesiennes** avec les nouvelles probabilités calculées.

➢ Création de la vue inférence bayésienne(Matias)

La classe **VueInférenceBayésienne** est la classe gérant l'affichage des probabilités de la carte mentale d'un personnage non-joueur. Comme pour **VuePrincipale**, implémente l'interface **DessinJeu** avec sa méthode update. À la différence de **VuePrincipale**, **VueInférenceBayésienne** affiche les probabilités de présence liée à la mentale d'un personnage non-joueur. Chaque case possède un filtre rouge avec une opacité variant en fonction de la probabilité de présence, plus elle est importante plus la case est rouge.

Comme cette vue concerne les personnages non-joueurs, elle a pour vocation d'être affichée lorsque l'utilisateur veut voir l'historique de la partie ou quand une partie est lancée en mode non-interactif.

➢ Création de l'arbre de décision du gardien (Célie)

Malheureusement par manque de temps et par contrainte de l'attente des calculs bayésiens l'arbre de décision n'as pas pu être implémenté dans la totalitée.

II. Difficultés rencontrées

Durant cette itération, nous avons pû faire la plupart des fonctionnalités prévues. Néanmoins, certaines ont été retardées à cause de certaines difficultés.

La première difficulté a été la mise en place de la vision. Pour implémenter cette fonctionnalité, il a d'abord fallu calculer et établir le champ de vision du prisonnier, ce qui a pris plus de temps que prévu.

La deuxième difficulté rencontrée a été lors des différents tests sur l'inférence bayésienne. N'ayant pas trouvé la façon "conforme" de tester, nous avons effectué des tests empiriques. Cela à donc pris un certain temps.

Les 2 difficultés rencontrées lors de cette itération ont engendré un retard sur les différentes tâches prévues lors de l'itération 1, dont la conception de l'arbre de décision du gardien qui était un tâche dépendante de l'inférence bayésienne. C'est pourquoi lors de l'itération 2, nous prévoyons moins de fonctionnalités pour parvenir à rattraper ce contretemps.

III. Prochaine étape: itération n°2

Notre premier but dans cette deuxième itération sera de rattraper notre retard sur les fonctionnalités d'inférence bayésienne et d'arbre de décision. Ces fonctionnalités sont importantes pour par la suite mettre en place le réseau de neurones.

Les fonctionnalités prévues seront donc :

- Finition de l'arbre de décision du gardien (en retard)
- Mise en place de l'inférence bayésienne (en retard)
- Ajout du réseau de neurones sur le prisonnier
- Ajout du contrôle du gardien
- Implémentation d'un système d'analyse
- Implémentation du lancement de l'apprentissage de l'application dans le terminal
- Ajout des vues des historiques