

Projet Design Pattern - Pacman

Olivier Goudet

September 12, 2023

1 Description du sujet

Le jeu de plateau séquentiel que vous allez développer dans ce projet est composé d'un monde de taille limitée dans lequel évolue un certain nombre d'agents. Le déroulé général du jeu est le suivant :

- Au tour $t = 0$ le plateau du jeu est initialisé suivant une configuration définie par l'utilisateur. Les agents sont créés et placés sur le plateau.
- A chaque tour t , chaque agent peut réaliser une action prédéfinie par un ensemble de règles et qui a un impact sur l'environnement. L'ensemble de ces actions vont conduire à un nouvel état du monde au temps $t + 1$.
- Une fois que le nombre maximum de tours est atteint ou bien qu'une condition spécifique de fin de jeu est atteinte, le jeu s'arrête.

Nous allons créer un jeu de plateau de type Pacman qui comprend un ou plusieurs agents *pacman* ainsi que plusieurs agents *fantôme*. Voir la fiche Wikipedia pour plus d'information sur le jeu <https://fr.wikipedia.org/wiki/Pac-Man>.

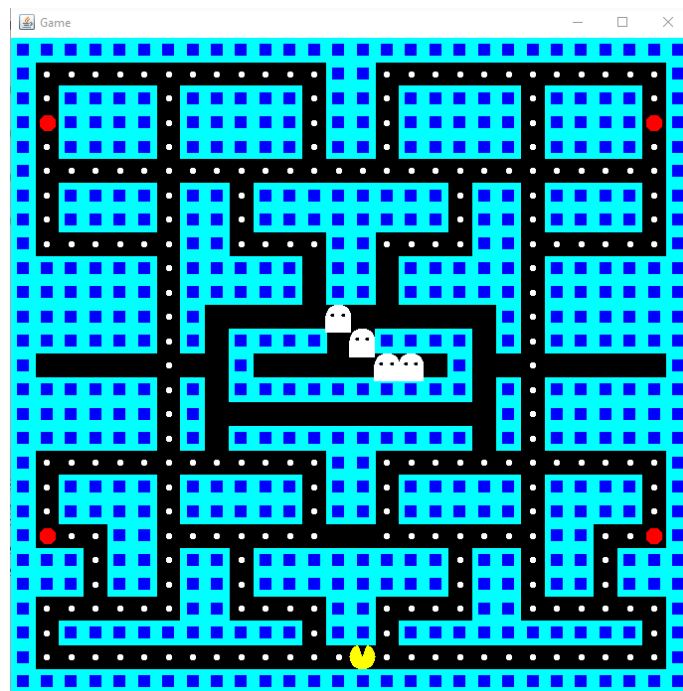


Figure 1: Plateau du jeu de pacman

1.1 Règles du jeu

Les règles du jeu de Pacman sont les suivantes :

- L'environnement est constitué d'un labyrinthe avec des cases libres et des cases murs. Sur chaque case libre peut se trouver des pac-gommes que les agents pacmans doivent manger pour gagner la partie. Une vue du plateau est présentée sur la figure 1.
- Chaque agent pacman ou fantôme se déplace d'une case à chaque tour dans la direction de son choix mais ne peut pas aller sur une case mur.
- Si un agent pacman se trouve sur la même case qu'un agent fantôme il est éliminé et disparaît du plateau.
- Des capsules spéciales sont disposées sur le plateau. Une fois mangée par un pacman, elles rendent les fantômes vulnérables pendant 20 périodes au cours desquelles les Pacmans peuvent les manger.
- Le jeu s'arrête si les agents fantômes mangent l'ensemble des pacmans (victoire des fantômes), ou bien si les pacmans mangent l'ensemble des pac-gommes (victoire des pacmans).

1.2 Objectifs du projet

L'objectif de ce projet est d'implémenter un jeu de Pacman interactif avec une interface visuelle de la façon la plus modulable et extensible possible en utilisant les Design Patterns vus en cours. Une fois la base du jeu réalisée, il s'agira d'implémenter les comportements et les stratégies des agents pour remporter la partie. On pourra implémenter plusieurs modes de jeu en coopératif ou en affrontement avec un ou deux joueurs humain ainsi que des intelligences artificielles. Ce projet est à faire individuellement.

Les premières séances de TP seront très guidées de façon à ce que chacun puisse réaliser une base du jeu. Une fois cette base réalisée, plus de liberté sera accordée pour le développement de l'interface, des commandes et des stratégies des agents.

L'avancement lors des séances de TP sera pris en compte pour la notation finale. Les séances de TP sont obligatoires.

Il y a 5 séances de TP en tout. Les séances de TP vont s'articuler dans les grandes lignes de la façon suivante :

Séance 1 : Réalisation d'un patron de conception général de jeu de plateau séquentiel. Implémentation d'un prototype de jeu très simple. Test de l'architecture. Début d'implémentation des principaux éléments de l'affichage graphique.

Séance 2 : Implémentation d'une architecture Modèle-Vue-Contrôleur (MVC) pour gérer l'affichage graphique et les commandes de l'utilisateur. Initialisation de l'environnement de jeu Pacman. Création des agents sur la plateforme. Affichage du jeu.

Séance 3 : Implémentation des règles du jeu de Pacman, des états des agents ainsi que des comportements.

Séance 4 : Ajout de stratégies pour les agents. Ajout d'un mode interactif.

Séance 5 : Implémentation de stratégies avancées : stratégie de recherche dans des arbres, algorithme A, coopération multi-agent...

2 Création de l'architecture d'un jeu séquentiel

Il s'agit tout d'abord de créer une architecture générale d'un jeu séquentiel à l'aide du pattern *Patron de méthode* (voir cours séance 1). Ce patron de méthode sera implémenté de façon concrète par un prototype de jeu très simple dans un premier temps avec une classe *SimpleGame*. On pourra étendre cette même architecture par la suite avec un jeu plus complexe du type *PacmanGame*.

1. Créer une classe abstraite *Game* avec les éléments suivants :

- un entier *turn* qui permettra de compter le nombre de tours du jeu.
 - un nombre de tours maximum *maxturn* dont la valeur est prédéfinie au moment de la création du jeu via le constructeur de *Game*.
 - un boolean *isRunning* qui permet de savoir si le jeu a été mis en pause ou pas.
 - une méthode concrète *void init()* qui initialise le jeu en remettant le compteur du nombre de tours *turn* à zéro, met *isRunning* à la valeur *true* et appelle la méthode abstraite *void initializeGame()*, non implémentée pour l'instant. Elle sera implémentée par les classes concrètes qui héritent de *Game*.
 - une méthode concrète *void step()* qui incrémente le compteur de tours du jeu et effectue un seul tour du jeu en appelant la méthode abstraite *void takeTurn()* si le jeu n'est pas terminé. Pour savoir si le jeu est terminé, on appellera la méthode abstraite *boolean gameContinue()* et on vérifiera si le nombre maximum de tours est atteint. Si le jeu est terminé, elle doit mettre le booléen *isRunning* à la valeur *false* puis faire appel à la méthode abstraite *void gameOver()* et qui permettra d'afficher un message de fin du jeu.
 - une méthode concrète *pause()* met en pause le jeu en mettant le flag booléen *isRunning* à *false*.
 - une méthode concrète *void run()* lance le jeu en pas à pas avec la méthode *step()* tant que le flag booléen *isRunning* reste à *true*.
2. Créer une classe concrète *SimpleGame* qui hérite de cette classe abstraite *Game*. A chaque fois que quelque chose se passe dans le jeu, faites une sortie console. Cela servira à vérifier que la structure générale du jeu marche bien. Par exemple quand la méthode *takeTurn()* est appelée, on affichera "Tour x du jeu en cours", où x est la valeur courante du compteur de tour *turn*. La méthode *boolean gameContinue()* renvoie pour l'instant simplement la valeur *true* systématiquement.
3. Implémentez une classe *Test* avec une méthode *main* qui permet de créer un objet *SimpleGame*. On appellera ensuite la méthode *init()* de *SimpleGame*, puis on testera les méthodes *step()* et *run()* de *SimpleGame*. Vérifiez que tout fonctionne correctement avec les affichages en sortie console.
4. Comme vous pouvez le constater, le déroulé du jeu est très rapide jusqu'à la fin.

Vous allez maintenant changer un peu la classe *Game* de façon à ce que la vitesse de déroulement du jeu soit contrôlable :

- Faites en sorte que la classe *Game* implémente l'interface *Runnable* (bibliothèque java.lang.Runnable).
- Ajoutez un attribut *thread* de type *Thread* dans la classe *Game*.
- Ajouter une méthode concrète *launch()* dans la classe *Game* qui met l'attribut *isRunning* à la valeur *true* puis instancie l'attribut *thread* avec un nouvel objet *Thread* qui contient le jeu courant (*thread = new Thread(this);*) et enfin fait appel à la méthode *start()* de cet objet *thread* pour lancer le jeu. Remarque : l'appel à cette méthode lancera automatiquement la méthode *run()* de l'objet *Game* de type *Runnable*.
- Ajouter un temps d'arrêt paramétrable dans la méthode *run()* après chaque tour de jeu : *Thread.sleep(time)*, où *time* est un attribut de type long de la classe *Game*. Il correspond au temps de pause entre chaque tour en millisecondes.
- Tester à nouveau avec *SimpleGame* que tout fonctionne correctement. Vérifiez que le changement de la valeur de l'attribut *time* a un impact sur la vitesse de déroulement de la simulation.

Remarque : on verra par la suite que le fait d'exécuter le jeu dans un thread présentera d'autres avantages que le simple contrôle du temps de la simulation. Cela permettra notamment à l'utilisateur de garder la main sur l'interface graphique pendant que le jeu s'exécute en tâche de fond, ainsi que de lancer un grand nombre de simulations de jeu en parallèle pour évaluer différentes stratégies de comportement des agents.

3 Création des premiers éléments de l'interface graphique

On souhaite maintenant créer les premiers éléments de l'interface graphique pour ce jeu type de jeu séquentiel.

Pour l'interface graphique, il s'agira de réaliser deux classes :

- une classe *ViewSimpleGame* qui permet l'affichage du jeu *SimpleGame*. Au début, l'affichage du jeu consistera simplement à écrire le tour courant du jeu sur un panneau comme sur la Figure 2.

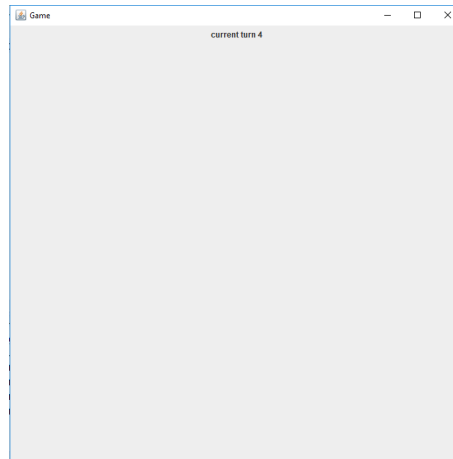


Figure 2: Affichage du jeu

- une classe *ViewCommand* qui affiche les principales commandes pour l'utilisateur (initialisation du jeu, lancement, mise en pause, etc...), ainsi qu'un compteur de tours du jeu, comme présenté sur la figure 3.

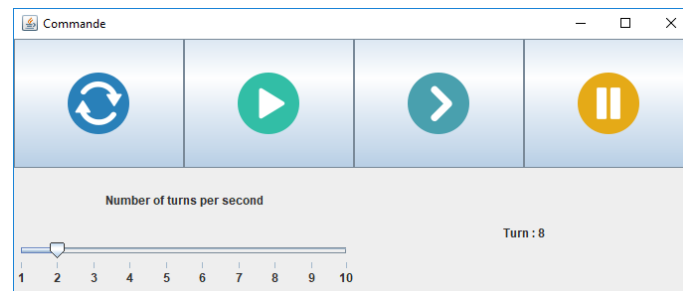


Figure 3: Interface de commande du jeu

Chacune des deux classes *ViewSimpleGame* et *ViewCommand* contiendra un élément de type *JFrame* qui permettra de placer les différents composants graphiques. Ces deux *JFrame* peuvent être disposées à deux endroits différents de l'écran. Pour dimensionner et positionner un élément *JFrame* par rapport au centre de l'écran, on peut utiliser par exemple le code suivant :

```
JFrame jFrame = new JFrame();
jFrame.setTitle("Game");
jFrame.setSize(new Dimension(700, 700));
Dimension windowSize = jFrame.getSize();
GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
Point centerPoint = ge.getCenterPoint();
int dx = centerPoint.x - windowSize.width / 2 ;
```

```
int dy = centerPoint.y - windowSize.height / 2 - 350;
jFrame.setLocation(dx, dy);
```

Il faudra préalablement importer les librairies *import javax.swing.;* et *import java.awt.;*.

Pour rendre la fenêtre visible, il faudra ajouter aussi la ligne *jFrame.setVisible(true);*.

3.0.1 Création de l’affichage du jeu

Pour la partie affichage du jeu, créez simplement pour l’instant un panneau qui contient un JLabel avec un petit texte. On affichera plus tard à cet endroit ce qui se passe dans le jeu (affichage du message qu’un nouveau tour se déroule, affichage d’un message quand le jeu est fini, ...).

Remarque :

- On peut centrer ce texte avec l’option JLabel.CENTER précisée dans le constructeur du JLabel.
- On peut changer la police avec la méthode *setFont()* du JLabel.

3.0.2 Affichage des commandes du jeu

Dans le constructeur de la classe *ViewCommand*, créez les éléments graphiques pour les commandes utilisateur. Un exemple d’affichage est proposée sur la figure 3.

Pour réaliser cette affichage de commande il faut utiliser les classe *JSlider*, *JLabel* et *JButton*. On peut les placer avec des Panels composés de *GridLayout* (positions en quadrillage) : un premier avec un *GridLayout* (2,1) qui permet de couper le panneau global en deux suivant le sens de la hauteur. Un deuxième qui sera situé en haut avec un *GridLayout* (1,4) pour positionner quatre boutons et un deuxième en bas avec un *GridLayout* (1,2) pour positionner le slider et la zone de texte.

Pour ajouter des icônes (placés dans un dossier nommé *icons* et situé à la racine du projet), on peut utiliser la commande suivante :

```
Icon restartIcon = new ImageIcon("icons/icon_restart.png");
JButton restartButton = new JButton(restartIcon);
```

Les fichiers d’icône sont disponibles dans le dossier *icons* de la section TP du projet sur Moodle.

Pour le slider, on pourra le régler pour accepter des vitesses entre 1 et 10, et utiliser les méthodes *setMajorTickSpacing*, *setPaintTicks* et *setPaintLabels* pour afficher les choix de vitesse possibles.

4 Création du Modèle-Vue-Contrôleur

Voir la cours de la séance 2 sur le MVC. Dans notre cas, qui est la vue et qui est le modèle ? Comment faire en sorte que les deux fenêtres graphiques développées lors de la première séance réagissent à des modifications de l'état du jeu ?

4.1 Mise en place du design pattern Observateur

Mettez en place le design pattern Observateur de façon à ce qu'à chaque fois que le jeu change d'état, les différents éléments graphiques soient mis à jour. On affichera pour l'instant simplement le nombre de tours dans le jeu sur chacun de ces composants graphiques.

Pour cela, inspirez vous de l'exemple vu en cours. Il s'agit d'effectuer des notifications aux observateurs quand il se passe quelque chose dans le jeu : après l'initialisation, après qu'un tour soit effectué, etc.

Il y a deux possibilités d'implémentation :

1. créer vous même une interface d'observable que vous implémentez.
2. utiliser directement la librairie *java.util.Observable* de Java.
3. utiliser l'API *propertyChangeListener*.

Remarque : si vous utilisez la librairie *java.util.Observable* de Java, attention à bien appeler la méthode *setChanged()* sur l'observable pour dire qu'il a changé d'état juste avant de notifier les observateurs (voir l'exemple du cours).

Faites un test pour vérifier si les différentes interfaces se mettent bien à jour au bon moment quand il y a un changement dans le jeu.

4.2 Création du contrôleur pour l'interface graphique

On va ajouter un objet "contrôleur" qui va faire le lien entre la vue et le jeu. Dans le modèle MVC, le "contrôleur" est la stratégie pour la vue.

Ce contrôleur doit permettre de contrôler le jeu quand des actions ont été effectuées par l'utilisateur dans la Vue (par exemple lorsque l'utilisateur clique sur un bouton). Voir l'exemple du cours.

De façon à rester le plus générique possible et à factoriser du code commun aux contrôleurs des différents jeux (*SimpleGame*, puis plus tard *SnakeGame*), on peut réaliser une classe abstraite de contrôleur avec les fonctionnalités de base que doit assurer un contrôleur pour ce type de jeux.

Créer tout d'abord une classe abstraite de contrôleur *AbstractController* :

- Cette classe *AbstractController* doit avoir un attribut de type *Game*.
- Ajouter des méthodes dans *AbstractController*: *void restart()* (arrêt et réinitialisation), *void step()* (passage manuel d'une étape), *void play()* (passage automatique des étapes), *void pause()* (interruption du passage automatique des étapes), *void setSpeed(double speed)* (réglage de la vitesse du jeu). A chaque fois qu'une méthode est appelée on effectuera le ou les actions correspondantes sur le jeu *Game*.
- Attention, dans la méthode *setSpeed*, il y a une conversion à faire pour régler le temps de pause entre chaque tour dans le jeu.

Créez une ensuite classe concrète *ControllerSimpleGame* qui étend *AbstractController*. Cette classe doit avoir un constructeur qui permet de créer puis d'initialiser son attribut *Game* avec un jeu concret de type *SimpleGame*, mais aussi de créer les interfaces graphiques de type *ViewSimpleGame* et *ViewCommand*. Pour cette dernière, le contrôleur peut se passer en attribut car c'est une "stratégie" pour *ViewCommand*.

4.3 Ajout des écouteurs sur les différents composants de l'interface de commande

Ajouter ensuite des écouteurs d'actions sur les différents composants de l'interface graphique de *ViewCommand*.

Voici le code par exemple pour réaliser une action quand l'utilisateur clique sur le JButton *restartButton*.

```
restartButton.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent evenement) {  
        ...  
    }  
});
```

Lorsqu'une action est effectuée par l'utilisateur, on appellera ici son attribut de type *AbstractController* qui sait comment gérer les actions des utilisateurs et les communiquer au modèle. C'est la stratégie pour la vue.

4.4 Activation et désactivations des boutons en fonctions des actions effectuées par l'utilisateur

Une fonctionnalité intéressante peut être d'ajuster les choix possibles des boutons pour l'utilisateur en fonction du choix qu'il vient de faire.

Par exemple, au début seront activés tous les boutons sauf le bouton "restart" et le bouton "pause". Dès que l'utilisateur clique sur le bouton "play", on peut désactiver les boutons "play" et "step" (car le jeu est déjà lancé). Par contre à ce moment, on active les boutons "restart" et "pause", car ce sont de nouveaux choix proposés à l'utilisateur. On fera la même chose pour la gestion des autres boutons.

Quel design pattern peut être utile pour gérer ça ?

Faites un diagramme de conception, parlez en avec l'encadrant de la séance et mettez en oeuvre la solution retenue.

4.5 Test de l'architecture MVC

Créer maintenant une classe *Test* avec une méthode *main()* qui instancie un objet de type *ControlleurSimpleGame*.

Testez que tout fonctionne correctement au niveau de l'affichage, des différents boutons et du réglage du temps de chaque tour de jeu.

Organisez votre projet en différents packages.

5 Création du jeu Pacman

On va pouvoir maintenant facilement réutiliser l'architecture du jeu développée jusqu'ici pour créer le jeu Pacman. Certains éléments vous seront fournis pour ne pas perdre trop de temps, notamment pour la réalisation de l'interface graphique du jeu.

5.1 Matériel fourni

Dans la section TP pacman de Moodle se trouvent différents éléments qui vous seront utiles pour créer le jeu :

- Il y a tout d'abord un répertoire nommé *layouts* qui contient une base de labyrinthes au format texte
- Dans le répertoire *Sourcesjava* se trouvent quatre fichiers :
 1. *Maze.java* vous fournit une classe qui charge l'ensemble du labyrinthe dans un objet *Maze*. Le constructeur de cette classe prend comme argument le nom du fichier texte du labyrinthe à charger.

2. *PositionAgent.java* permet simplement de définir la position d'un agent et sa direction
3. *AgentAction.java* permet de définir une action de déplacement choisie par l'agent avec son vecteur de déplacement correspondant.
4. Le fichier *PanelPacmanGame.java* est créé en prenant en entrée un labyrinthe. C'est un JPanel qui permet l'affichage le jeu en fonction du labyrinthe mais aussi des positions courantes des agents. A noter que la méthode *paint* sera appelée automatique lorsque l'appel *setVisible(true)* sera effectué sur le JPanel. Une mise à jour du plateau peut être effectuée en appelant la méthode *repaint*.

5.2 Réalisations à effectuer pour obtenir une première simulation de jeu

A partir de maintenant les consignes sont volontairement moins précises, c'est à vous de concevoir et implémenter le jeu. Un fil directeur général est cependant proposé :

1. Mettez à jour l'architecture MCV déjà implémentée avec cette fois-ci les classes du jeu Pacman : *PacmanGame*, *ControleurPacmanGame* et *ViewPacmanGame*.
2. Chargez et affichez le plateau du jeu avec la position de base à partir d'un layout donné.
3. Créez une hiérarchie de classe Agent qui permet de modéliser les différents agents du jeu (soit Pacman, soit Fantôme).
4. Créez une classe *PacmanGame* qui étend la classe générique *Game* et qui contient notamment un labyrinthe, ainsi qu'une liste d'agents.
5. Implémentez la méthode *initializeGame* pour initialiser le jeu et notamment créer les différents types d'agent à partir de leurs positions initiales sur le plateau. Quel design pattern peut être utile dans ce cas ?
6. Vous pouvez ajouter si vous le souhaitez un menu déroulant dans l'interface qui permettra de choisir le layout à charger, pour cela regardez la classe *JFileChooser* de Java.
7. Implémentez les règles du jeu avec simplement des déplacements aléatoires des agents et la possibilité de manger les pac-gommes (sans la gestion des capsules pour l'instant et sans la possibilité donné à un fantôme de manger un pacman). Pour cela il peut être utile de créer deux méthodes :
 - une méthode *isLegalMove* prend en entrée un agent et une action et renvoie vrai ou faux si l'action est possible dans le labyrinthe
 - une méthode *moveAgent* prend en entrée un agent et une action et met à jour la nouvelle position de l'agent
8. Implémentez la méthode *taketurn* qui effectue une action pour chaque agent puis met à jour le labyrinthe (par exemple si une pac-gomme a été mangée)
9. Faites en sorte que la méthode *update* de la Vue mette à jour le panneau du jeu en fonction des nouvelles positions des agents et de l'état du labyrinthe. Une fois les positions redéfinies dans l'objet *PanelPacmanGame*, il s'agira d'appeler la méthode *repaint* du JPanel pour effectuer la mise à jour graphique.
10. Implémentez un premier comportement d'agent sous la forme d'un déplacement aléatoire à chaque tour. Pour cela, implémentez une stratégie avec une méthode *getAction* qui prend en entrée un agent et le labyrinthe et renvoie une action valide à effectuer.
11. Implémentez un autre comportement d'agent simple. Quel Design Pattern peut être utile ?
12. Implémentez la mort d'un agent Pacman lorsqu'il rencontre un fantôme
13. Implémentez la règle avec les capsules, qui enclenche un *timer* de 20 périodes au cours desquelles les pacmans peuvent manger les fantômes.
14. Implémentez les conditions de fin du jeu

15. On veut maintenant créer des agents avec des stratégies qui dépendent du fait que le *timer* lié à la capsule soit enclenché ou non. Par exemple quand le *timer* est enclenché les fantômes vont avoir tendance à s'enfuir plutôt qu'à se rapprocher des Pacmans. Quel design pattern vu en cours peut être utile dans ce cas ?
16. Implémentez une stratégie qui est en mode interactif. C'est l'utilisateur qui choisit l'action (pour le pacman ou pour le fantôme). Ce choix d'action peut être fait avec le clavier ou bien avec la souris sur différents boutons.

5.3 Stratégies et conceptions avancées

Il est demandé de proposer des extensions au jeu. Le choix est libre mais entrera dans la note finale. Choisissez les thèmes thème qui vous intéresse puis discutez-en avec votre encadrant.

Plusieurs orientations sont possibles :

- Améliorer l'interface et la gestion du jeu (compte des points, du nombre de vies des pacmans), gestion des niveaux, etc...
- Ajouter un mode multijoueur (sur un seul ordinateur). Par exemple un joueur gère le pacman et un joueur gère le fantôme
- Ajouter de l'IA dans les agents (algorithme A pour les pacmans et/ou les fantômes, coopération entre les fantômes afin d'encercler le Pacman).