Rapport 2i013

Serge Durand Kevin Meetooa

March 2019

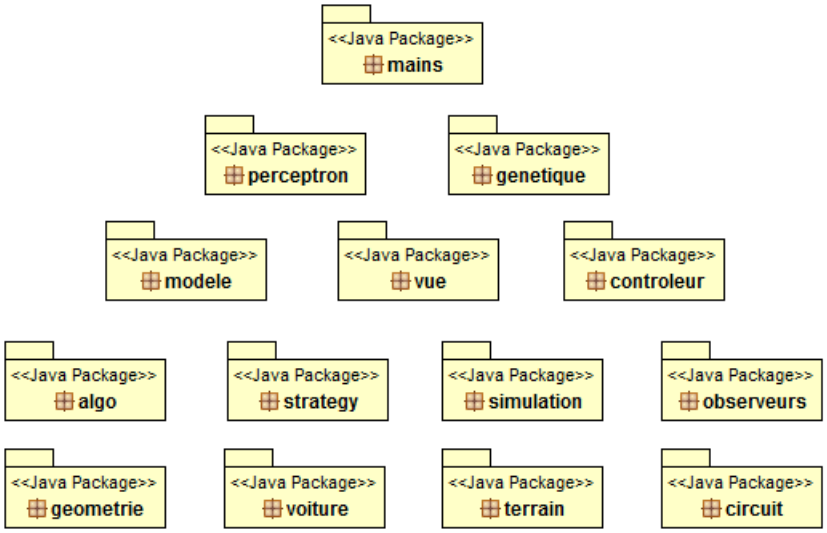
Table des matières

1 Introduction

Dans le cadre de notre cursus nous n'avions plus fait de java depuis un an. Ce projet, très riche, nous a permis de nous replonger dans la programmation objet avec Java, et d'approfondir considérablement nos connaissances en la matière. Bien balisé par le cours disponible sur le site du créateur du projet, Vincent Guigue, nous avons rapidement mis en place les bases du projet pour nous concentrer sur l'établissement de stratégies avancées, afin trouver les meilleures trajectoires selon les circuits, et pour développer une interface graphique interactive. Nous présenterons ainsi notre version fonctionnelle d'une stratégie basée sur Dijkstra, mais aussi une stratégie d'intelligence artificielle utilisant un algorithme génétique qui optimise les poids d'un réseau de neurones.

2 Mise en place du projet

Dans un souci de compréhension du code et de l'architecture, nous avons divisé le projet en plusieurs packages contenant les différentes classes du projet.



- **Géométrie :** Ce package contient la classe Vecteur. L'objet Vecteur y est défini ainsi que tous les accesseurs et méthodes intéressantes telles que les méthodes de normalisation d'un vecteur ou de récupération de la distance entre deux vecteurs. Dans la suite du projet, cet objet sert notamment à modéliser la position de la voiture.

- **Circuit :** Ce package contient la classe circuit définissant toutes les méthodes devant être implémentées par le circuit. Il contient aussi une Factory Circuit permettant de construire un circuit ainsi qu'une classe CircuitImpl implémentant l'interface citée précédemment.

- **Terrain :** Ce package contient l'interface Terrain ainsi qu'une classe implémentant cette interface. Un terrain correspond à un pixel d'un circuit. Toutes les méthodes liées aux terrains sont définies dans ce package, notamment les méthodes permettant de récupérer le type d'un terrain (route ou herbe par exemple).

- **Voiture :** Ce package contient l'interface Voiture et toutes les classes qui en découlent. Il y a notamment une Factory afin de créer un objet Voiture ainsi qu'une classe implémentant l'interface Voiture. Toutes les méthodes liées aux voitures y sont définies, notamment les fonctions permettant de conduire la voiture à partir d'une Commande donnée en argument. C'est d'ailleurs la raison pour laquelle la classe Commande se trouve aussi dans ce package.

- **Algo :** : Ce package contient les différents types de radar utilisés dans le projet. Nous avons modélisé un radar par une interface : Le but de cette interface étant de définir le comportement des radars qui seront crées dans le suite du projet. Ainsi, tous les radars que nous avons crées implémentent cette interface et suivent le comportement attendu. Parmi les radars implémentés, nous pouvons notamment retrouver le radar simple mais aussi le radar Dijkstra. Le fonctionnement de ces radars sera détaillé dans la suite du rapport.

- **Strategy :** Ici, une interface stratégie est définie afin de définir le comportement attendu de toutes les stratégies implémentées. Parmi les stratégies implémentées, on retouve des stratégies simples telles que la Stratégie Ligne Droite ou la Stratégie Radar Simple mais on retrouve aussi des stratégies un peu plus complexes telle que la Stratégie Point à Point.

- **Simulation :** Implémente notamment la fonction permettant de lancer la course en suivant une stratégie donnée.

- **Observeurs :** Implémente des Observeurs permettant d'observer l'évolution des différents objets au cours de la course.

- **Modele/Vue/Controleur** : Ces trois packages implémentent le modèle MVC vu en cours afin d'obtenir un affichage interactif de l'évolution de la course en temps réel.

- **Perceptron/Génétique** : Implémente l'algorithme génétique utilisant un réseau de neurones, le fonctionnement de cet algorithme est détaillé plus tard dans le rapport.

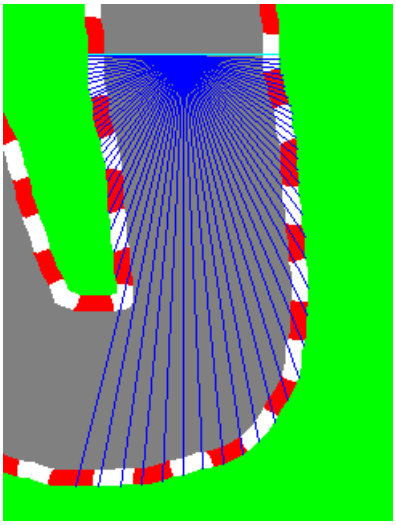
- **Mains :** : Contient les mains permettant de lancer le programme.

3 Premier algorithme

Notre premier algorithme implémente un radar simple. Sa particularité est qu'il est paramétrisable : Il faut l'instancier avec le nombre de faisceaux.

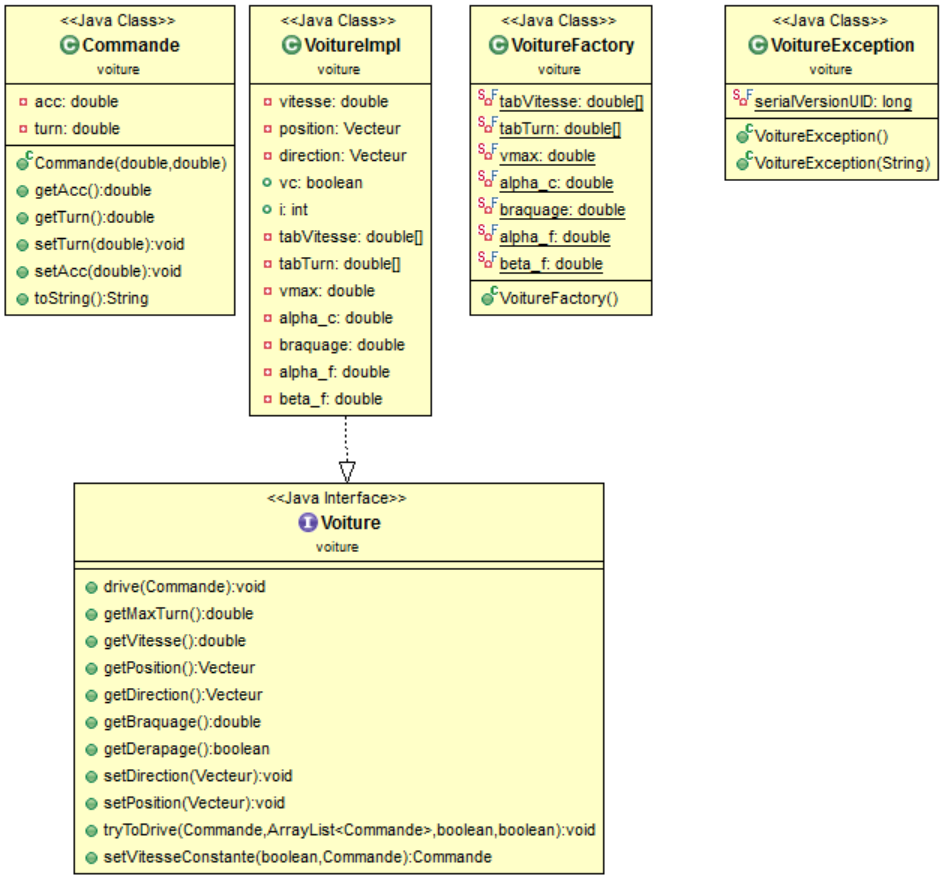
Le radar simple fonctionne comme un sonar, il envoie des faisceaux dans certaines directions devant lui jusqu'à ce que ces faisceaux rencontrent un obstacle. Dans la stratégie radar simple, on choisit de faire avancer la voiture dans la direction du faisceau de plus grande longueur à chaque itération. Ce radar fonctionne bien sur les circuits simples mais il est vite limité. Pour les circuits compliqués, il est préférable de combiner ce radar avec une autre stratégie ou de choisir un autre type de radar.

Voici une illustration de ce radar :



4 Modèle physique

Voici le diagramme UML du package Voiture illustrant les explicaitons qui vont suivre :



4.1 Fonctionnement d'une stratégie

L'interface Strategy impose à toutes les stratégies d'avoir une méthode getCommande. En effet, les différentes stratégies devraient à priori renvoyer des commandes différentes selon l'approche choisie (par exemple, comme vu précédemment, la stratégie RadarSimple renverra toujours la commande nous faisant avancer dans la direction du plus long faisceau).

Les stratégies sont donc des classes implémentant une méthode getCommande. La méthode getCommande est appelée à chaque fois que la voiture avance jusqu'à ce qu'on atteigne éventuellement la ligne d'arrivée.

4.2 Modèle physique imposé par le projet

Pour que les courses soient réalistes, il y a un modèle physique à respecter. Dans la réalité, une voiture ne peut pas effectuer un virage serré si elle possède une vitesse trop importante : Il en est de même ici.

Les différentes contraintes physiques fournies par les sujets de TME sont implémentées dans la classe VoitureFactory. Il y a notamment un tableau de vitesses tabVitesse et un tableau d'angles tabTurn : À chaque vitesse dans tabVitesse on associe un angle de rotation maximal en valeur absolue situé dans tabTurn. Par exemple, la première case de tabVitesse contient la valeur 0.1 et la première case de tabTurn contient la valeur 1. Cela signifie que si la vitesse actuelle est inférieure à 0.1*vmax où vmax est une constante correspondant à la vitesse maximale de la voiture, alors on peut renvoyer une commande de rotation entre -1 et 1 : On peut tourner de l'angle qu'on veut.

Cependant, si la vitesse est comprise entre 0.9*vmax et vmax, on peut seulement renvoyer une commande de rotation entre -0.05 et 0.05 : Il s'agit de l'implémentation du modèle physique imposée par l'énoncé du projet.

4.3 Fonction tryToDrive

Les différentes stratégies renvoient des commandes indépendamment des contraintes physiques. Une fonction drive nous est fournie par l'énoncé du projet et elle permet de faire avancer la voiture selon une commande donnée en argument. Cependant, certaines contraintes sont physiquement impossible (effectuer un demi-tour à grande

vitesse par exemple) et il ne faut pas renvoyer ce type de commandes, il faut donc filtrer les commandes que l'on veut donner à la fonction drive : C'est le but de la fonction tryToDrive que nous avons crée.

Nous avons implémenté la fonction tryToDrive dans la classe VoitureImpl et à chaque fois que l'on récupère une commande, on appelle la fonction tryToDrive avec cette commande en paramètre afin de vérifier si la commande est physiquement cohérente.

Le fonctionnement de la fonction est assez simple : On vérifie si la commande de rotation donnée est supérieure en valeur absolue à la commande de rotation maximale admissible donnée dans le tableau tabTurn.

- Si ce n'est pas le cas, alors la commande respecte le modèle physique : On la renvoie telle quelle et on peut appeler la fonction drive sur cette commande.

- Sinon, la rotation est trop grande. Tant que la rotation reste trop grande, on divise sa valeur par 2 jusqu'à ce que la rotation respecte les contraintes. On renvoie ensuite la commande obtenue avec la valeur de rotation modifiée et une accélération négative afin de décélérer dans le virage. En effet, si l'on accélérait pendant le virage, le virage effectué serait beaucoup trop large comparé à un virage effectué à faible vitesse.

Voici un récapitulatif du fonctionnement de la fonction tryToDrive :

5 Dijktsra

L'algorithme Dijkstra est un des algorithmes les plus classiques pour la recherche de plus court chemin. Nous avons utilisé son principe, vu en cours, adapté à notre projet, non sans difficulté.

5.1 Implémentation dans le cadre du projet

Il est possible d'implémenter l'algorithme de Dijkstra en prenant comme graphe un circuit dont tous les pixels seraient les noeuds. Le noeud de départ demandé par l'algorithme est simplement le point de départ du circuit.

Ainsi, l'algorithme de Dijkstra s'implémente naturellement de façon analogue à l'algorithme général à quelques différences près.

- Au lieu d'utiliser une liste puis de chercher son minimum comme dans l'algorithme, nous avons choisi d'implémenter une PriorityBlockingQueue qui correspond à une liste triée à tout instant. Ainsi, le minimum de la liste se situe toujours au début de celle-ci.

- Pour avoir les voisins d'un pixel P nous avons crée une sous-fonction qui explore autour de P dans un rayon de 3x3 pixels

- Dans le cas particulier où l'on se situe sur la ligne de départ, il faut faire attention à regarder uniquement les voisins situés dans la direction de départ et ignorer ceux dans la direction opposée.

- Nous avons choisi d'affecter des poids en fonction du type de terrain sur lequel nous roulons, de sorte à ce qu'une bande blanche/rouge ait plus de poids que la route classique.

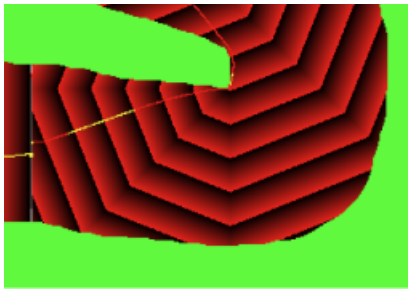
Sinon, l'algorithme de Dijkstra pour le circuit fonctionne de manière identique à celui sur les graphes généraux.

5.2 Radar Dijkstra

Implémentation du radar

Le radar Dijkstra que nous avons implémenté est un radar combinant l'algorithme de Dijkstra avec un radar simple dont nous avons vu le fonctionnement précédemment.

Voici une illustration de l'algorithme de Dijkstra appliqué à un circuit :



Le score d'une case est le score croisé entre la longueur du plus long faisceau et le poids de la case selon l'algorithme de Dijkstra. Le but est qu'une case très proche de l'arrivée avec un faisceau court soit plus privilégiée qu'une case éloignée de l'arrivée mais avec un long faisceau, ce qui n'était pas possible avec le Radar Simple.

Poids de l'algorithme de Dijkstra

Nous avons vu que le score renvoyé par le radar Dijkstra était le score croisé entre le score du radar simple et le score renvoyé par l'algorithme de Dijkstra (qui correspond à la distance de la case à la ligne d'arrivée).

Il est possible d'obtenir différents résultats en attribuant des poids différents au radar simple et au radar Dijkstra.

En effet, un radar Dijkstra dont le poids des deux composantes est uniformément réparti entre 50% de radar simple et 50% d'algorithme de Dijkstra fonctionne bien sur les circuits simples mais n'est pas optimal sur les cicruits compliqués.

Pour les circuits compliqués, une répartition associant un poids logarithmique au radar simple et un poids linéaire au radar Dijkstra est meilleure : Le score obtenu avec l'algorithme de Dijkstra est grandement favorisé par rapport à la longueur du faisceau renvoyée par le radar simple. Cette approche est particulièrement efficace sur les circuits compliqués tels que les labyrinthes.

Voici une illustration de la différence que peut faire le poids des radar :

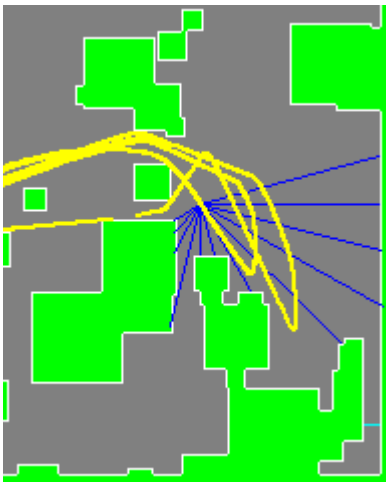


FIGURE 1 – Malgré de nombreuses tentatives, l'algorithme avec le poids 50% radar simple et 50% radar Dijkstra n'arrive pas à trouver la sortie

FIGURE 2 – Lorsque le poids du radar simple est logarithmique et que le poids du radar Dijkstra est linéaire, la voiture trouve rapidement la sortie

6 Stratégie Prudente et Point à point

6.1 Échecs rencontrés avec les premières stratégies

Les stratégies basées sur le radar simple et sur le radar Dijkstra sont efficaces sur les circuits simples mais elles deviennent limitées lorsque les circuits deviennent complexes et possèdent un nombre important d'obstacles. Sur ce type de circuits, la voiture termine très souvent sa course dans un mur car elle roule trop rapidement pour éviter les nombreux obstacles. Il faut donc envisager d'autres stratégies, quitte à finir le circuit en un très grand nombre de commande.

6.2 Stratégie Prudente

Une approche possible pour ce type de circuits est une approche "prudente" : Lorsque l'on se situe près d'un mur, on ralentit fortement quitte à effectuer plus de commandes. Cela nous garantit que nous pourrons faire des rotations avec des angles très prononcés, ce qui n'est pas possible à grande vitesse.

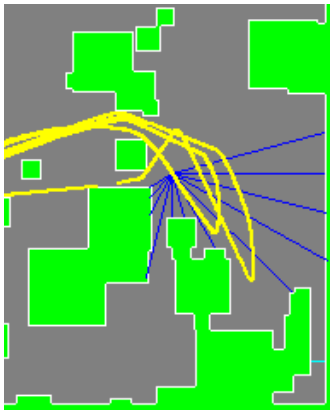
Voici une illustration de la Stratégie Prudente :

FIGURE 3 – Stratégie Radar Simple : La voiture dont la trajectoire est représentée en jaune roule trop vite et termine sa course dans un obstacle.

FIGURE 4 – Stratégie Radar Prudente : Plus la voiture est proche d'un mur, plus la voiture roule lentement. Ainsi, elle n'a pas de problème pour passer dans les passages très étroits

6.3 Stratégie Point à Point

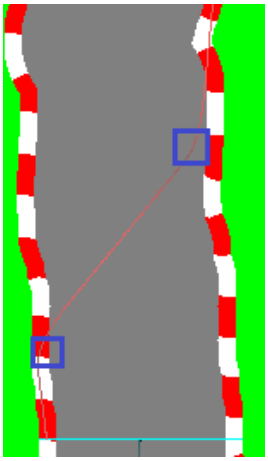
Sur certains circuits, la voiture se rapproche de la sortie mais n'arrive jamais à la franchir. C'est le cas sur la figure donnée précédemment :



Pour éviter ce type de problèmes, il peut être intéressant de créer une stratégie point à point dont le fonctionnement est le suivant :

On place plusieurs points $p_1, p_2, ..., p_n$ sur le circuit. L'idée est qu'à partir du moment où l'on arrive sur un point p_i, on oublie la stratégie courante et on essaye uniquement d'atteindre le point p_{i+1}

Ainsi, pour résoudre le problème de la voiture qui n'atteint jamais la ligne d'arrivée, il suffirait de placer des points bien choisis près de la ligne d'arrivée qui "guideraient" la voiture vers la ligne d'arrivée.



L'image ci-dessus est une illustration de la Stratégie Point à Point que nous avons implémenté. Nous avons placé deux points aux endroits entourés en bleu.

Le comportement est bien celui attendu : Lorsque la voiture franchit le premier point bleu, plutôt que de continuer tout droit et d'aller directement vers la ligne d'arrivée, elle se dirige d'abord vers le second point bleu.

7 Intelligence Artificielle

Pour aller plus loin dans les stratégies visant à trouver les meilleurs chemins nous avons choisi d'utiliser une tactique basée sur les algorithmes génétiques comme suggéré par le cours, mais en utilisant de plus un réseau de neurones. L'idée est d'utiliser le réseau de neurones pour diriger la voiture, et d'utiliser l'algorithme génétique pour trouver quels sont les meilleurs paramètres du réseau.

7.1 Perceptron Multicouche

Nous avons donc d'abord implémenté la structure d'un réseau de neurones classique : le perceptron multicouche, aussi appelé *feedforward neural network*, illustré dans la figure 5.

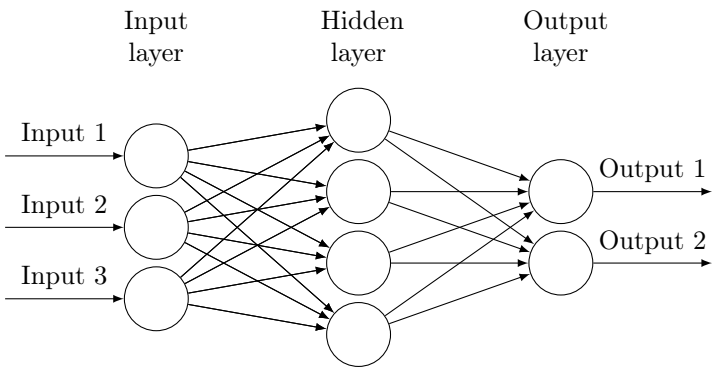


FIGURE 5 – Exemple de perceptron multicouche

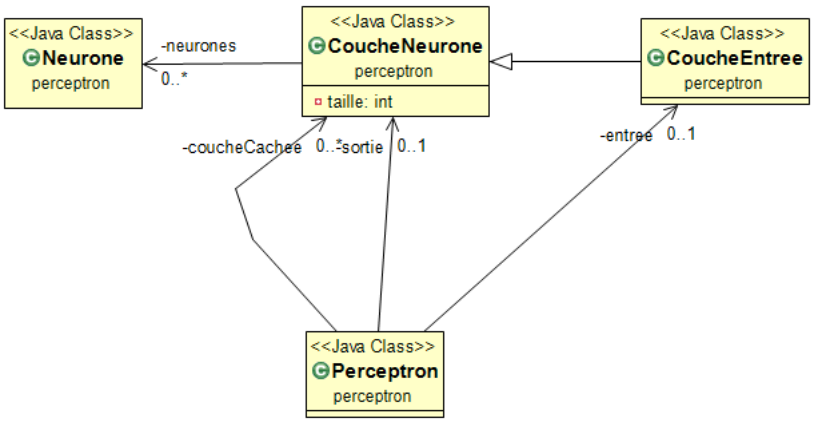


FIGURE 6 – UML du package Perceptron

Nous avons utilisé le paradigme objet classique pour la création des classes nécessaires à l'implémentation de perceptrons, schématisé par la figure 6. Nous avons essayé d'avoir une architecture souple : il est possible de créer n'importe quel type de réseau en terme de nombre de neurones par couche, y compris pour la couche d'entrée et la couche de sortie, et le nombre de couches cachées. Dans notre cas nous n'avons besoins que de deux neurones dans la couche de sortie : un pour l'accélération et un pour la rotation. En effet le but du réseau est d'envoyer une commande. Pour la couche d'entrée, elle contient à minima autant de neurones que de faisceaux du radar utilisé. En effet les inputs correspondent aux informations transmises par le radar : la distance à l'obstacle le plus proche dans chaque direction. Il est possible de rajouter des neurones pour la couche d'entrée, afin de donner plus d'information en entrée au réseau, comme la vitesse courante ou la rotation maximale possible.

Le réseau exploite ensuite ces informations ("inputs") pour obtenir une commande adéquate. Les informations sont transmises couche par couche via une transformation affine classique avec les poids de chaque connexion, les biais de chaque neurone et la fonction d'activation. Cette transmission d'information au sein d'un neurone est schématisé dans la figure 7. Les w_i représente les poids entrants du neurone, et b est son biais. Il y a autant de poids entrant que de neurones dans la couche précédente. Les x_i sont les valeurs des neurones de la couche précédente. Pour la première couche ce sont simplement les valeurs données par le radar. Chaque neurone fait une somme de ces valeurs, pondérée par les poids, y ajoute son biais et le passe aux neurones suivants après application de la fonction d'activation. Cette fonction est une fonction tangente hyperbolique qui nous a semblé la plus adapté pour notre cas : on veut récupérer une valeur entre -1 et 1.

La question qui se pose ensuite est : quels sont les meilleurs poids et les meilleurs biais ? L'apprentissage classique d'un perceptron se déroule par entraînement supervisé sur un jeu de données dont les caractéristiques sont connues. Pour nous cela n'était pas applicable : nous ne disposons pas de milliers de courses jouées par exemple. Mais il est possible d'utiliser un algorithme génétique pour entraîner notre perceptron.

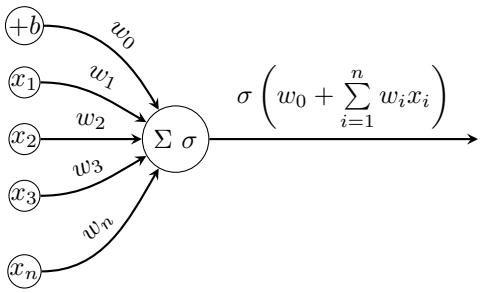


FIGURE 7 – transmission de valeur au sein d'un neurone

7.2 Algorithme génétique

Guidé par le cours nous avons implémenté un package génétique permettant de faire évoluer nos perceptrons jusqu'à trouver des poids et biais intéressants.

Dans notre approche les gènes sont simplement les poids et les biais, et le génome est un ensemble de ces poids et ces biais, organisé selon une structure donnée en paramètre sous la forme d'une liste d'entiers. Par exemple [4,5,2] représente la structure d'un réseau composé d'une couche d'entrée de 4 neurones, d'une couche cachée de 5 neurones et d'une couche de sortie de 2 neurones.

Ici aussi nous avons voulu avoir une architecture souple, afin de réaliser de nombreux tests sans trop de modifications de code. Lors de l'appel à l'algorithme génétique dans le main on peut indiquer la structure du réseau de neurones voulu, le nombre de génomes dans la population, la part de la population que l'on conserve pour le croisement, le nombre de générations maximum ...

Afin d'affiner l'évaluation de nos génomes nous avons distingué plusieurs cas lors de l'arrêt prématuré de la course, par la création d'exceptions spécifiques exploitées par la fonction play de simulation :

1. Arrivée franchie dans le mauvais sens
2. La voiture fait du surplace
3. La voiture atteint un obstacle

Nous avons fortement pénalisé ces deux premiers cas avec un score rédhibitoire. Dans le deuxième cas le score est la distance à l'arrivée, connue grâce à notre classe Dijkstra, majoré par un malus. Par ailleurs ce deuxième critère permet de gagner en efficacité : nous testons dans la simulation si la voiture fait du surplace toutes les 500 itérations, si c'est le cas la simulation est arrêtée.

Pour les voitures qui réussissent à franchir l'arrivée, le score est le nombre de commandes. Le but est alors de minimiser ce score selon l'approche classique de la sélection / croisement / mutation. Nous avons fait quelques choix permettant d'avoir des performances correctes en terme de vitesse de convergence :

1. Avantages aux meilleurs pour le croisement : si les deux parents sont dans le top 10 ils donnent 3 enfants.
2. Les 3 meilleurs génomes sont conservés dans la génération suivante.
3. Pour compenser cette approche "élitiste" on fait une mutation importante - plus de 50% des nouveaux génomes sont mutés, et quand ils le sont, tous leur gènes sont mutés.

7.3 Résultats

Cette approche a plutôt bien marché sur les circuits simples (de 1 à 5). Pour les circuits 1 et 5 c'est particulièrement rapide, une vingtaine de génération avec une trentaine de génomes par génération suffit à obtenir un score correct qui ne sera pas amélioré avec plus d'individus ou de générations. C'est un peu plus long pour les circuits 2,3 et 4 mais on y arrive tout de même. Une approche différente est de prendre 1000 individus et de les faire évoluer sur 5 générations uniquement, les résultats sont analogues. Nous avons également pu obtenir des résultats intéressants sur les circuits bond et Een2, avec cette dernière approche basée sur un très grand nombre d'individus.

Pour les autres circuits la stratégie génétique échoue, en tout cas avec un temps de calcul raisonnable. Contrairement aux premiers circuits, il est indispensable pour les circuits compliqués comme les labyrinthes de recourir à l'utilisation d'un Radar Dijkstra : le but de la voiture n'est plus d'apprendre à bien aborder des courbes jusqu'à faire un tour de circuit, mais de trouver une trajectoire particulière d'un point A à un point B sur un circuit comportant de nombreux obstacles.

Malheureusement l'algorithme génétique ne fonctionne pas avec Dijkstra, il est très lent et converge très lentement. Nous avons pu le faire tourner sur 80 générations sur un circuit au prix d'une dizaine d'heure de calculs, sans succès.

8 Modèle MVC

8.1 Intérêt

Le modèle MVC est une architecture divisée en trois composantes :
— Le modèle : Implémente le programme et gère les données
— La vue : Implémente l'interface graphique du programme
— Le contrôleur : Contrôle les entrées de l'utilisateur et les convertit en commandes pour le modèle et/ou la vue

8.2 Parallèle avec le projet

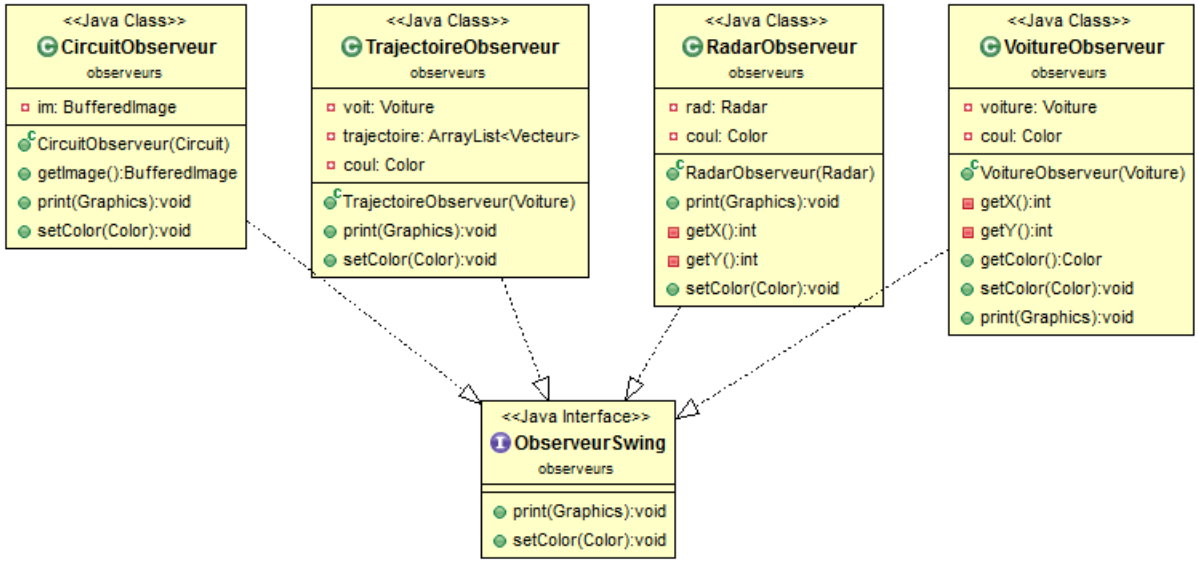
Dans le cadre de notre projet, il peut être intéressant d'implémenter le modèle MVC afin de donner une interactivité qui était inexistante jusqu'à présent. Il serait possible grâce à cette architecture d'observer l'évolution de la course en temps réel mais aussi de choisir le circuit et la stratégie directement depuis l'interface graphique sans avoir besoin de modifier le code source : Cela représenterait un réel avantage en terme d'utilisation.

8.3 Implémentation

Vue

Nous avons implémenté la Vue en utilisant un JFrame : Il s'agit d'une classe permettant de créer une fenêtre dans laquelle nous pourrons observer l'évolution de la course.

Nous avons ensuite implémenté des classes Observeur pour tous les objets que nous souhaitions observer. Voici le diagramme UML des classes Observeur que nous avons implémenté.



Contrôleur

Le contrôleur est un écouteur d'évènements qui gère une liste d'observeurs et qui écoute les actions de l'utilisateur. Le contrôleur que nous avons crée implémente les interfaces UpdateEventListener et ActionListener C'est cette interface impose notamment l'implémentation de deux fonctions :

- La fonction manageUpdate qui permet de gérer ce qu'il se passe lorsque qu'un évènement se produit (par exemple lorsque la voiture avance).

- La fonction actionPerformed qui permet de gérer ce qu'il se passe lorsque l'utilisateur effectue une action (par exemple un changement de circuit).

Il faut ensuite modifier l'architecture de la simulation afin que la simulation devienne émettrice d'évènements : Cela permettra d'observer en temps réel les évènements à l'aide du Contrôleur et de la Vue.

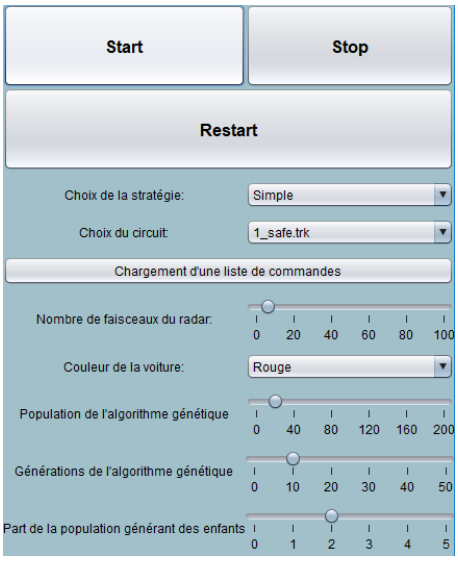
Modèle

Le modèle est une classe dont le but est de gérer le comportement de la vue et du programme lorsqu'un évènement se produit ou lorsque l'utilisateur effectue une action.

Pour ce faire, le modèle réutilise de nombreuses classes déjà créees (Radar, Voiture, Circuit, Stratégie, ...)

Ainsi, le modèle implémente par exemple les fonctions permettant de changer de circuit ou de stratégie.

8.4 Illustration et fonctionnalités implémentées



Voici une illustration de notre interface MVC. Nous avons implémenté trois boutons pour démarrer/arrêter/redémarrer la course.

De plus, nous avons ajouté trois menus déroulants permettant de choisir respectivement la stratégie, le circuit et la couleur de la voiture. Le nombre de faisceaux du radar est paramétrisable et s'affiche sur l'interface graphique tout en s'ajustant dans le modèle : On obtient une plus grande précision lorsque le radar possède un plus grand nombre de faisceaux.

Les trois derniers sliders permettent de régler des paramètres pour la stratégie génétique dont le fonctionnement est détaillé plus tard dans le rapport.

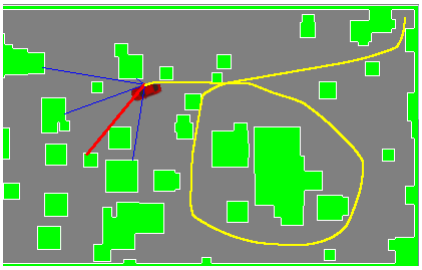


FIGURE 8 – Ici, le radar possède uniquement 5 faisceaux. Sa précision est donc faible et la voiture fait une boucle inutile

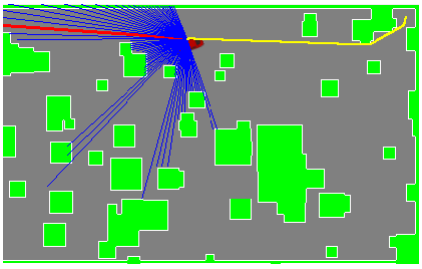


FIGURE 9 – Le radar possède maintenant 80 faisceaux : La voiture n'a aucun mal à trouver la bonne trajectoire

Les deux figures ci-dessus montrent l'utilité de la fonctionnalité de paramétrisation du radar. Le faisceau rouge représente le faisceau de plus grande longueur : C'est la direction dans laquelle la voiture se dirige lorsque l'on utilise une Stratégie Radar Simple

9 Résultats finaux et conclusion

Nous récapitulons les résultats dans un tableau. Les scores ont été validé par l'outil proposé sur le site, le jar de M. Baskiotis.

Stratégies	1 safe	2 safe	3 safe	4 safe	5 safe	6 safe	7 safe	8 safe
Radar Simple	3173	4047	∞	4173	2392	∞	∞	∞
Dijkstra	3079	3948	∞	4196	2271	1990	1335	∞
Prudente	11838	14057	18773	17933	10036	9462	5052	15350
Genetique	3425	4358	5500	5506	2525	∞	1560	∞

‘

Stratégies	aufeu	bond	een	labymod	perso	labyperso	t2009	li260
Radar Simple	∞	∞	∞	∞	∞	∞	∞	∞
Dijkstra	∞	∞	∞	∞	∞	∞	∞	∞
Prudente	12541	5582	7467	∞	∞	∞	∞	∞
Genetique	∞	2060	2837	∞	∞	∞	∞	∞

Conclusion

Il reste de nombreuses améliorations possibles, ce qui est naturel sur un tel projet. En particulier nous aurions voulu pouvoir utiliser la stratégie génétique avec Dijkstra, mais également implémenter un éditeur graphique de circuit pour pouvoir rajouter des obstacles et les rendre plus simples. Nous avons quand même beaucoup appris de ce projet. Il nous a permis d'approfondir nos connaissances techniques en java, et plus théoriques en algorithmique. Nous avons également bien progressé sur la manière de nous organiser pour le travail en binôme sur un projet au long cours. Par exemple nous avons appris à utiliser le gestionnaire de contrôle de version git ainsi que la plateforme github pour héberger notre code et gérer l'avancement du projet. Tout cela nous sera certainement utile dans nos futures carrières !