

Rapport de projet

Stardew Valley Automaton

Module M2201 – Graphes et Langages

Groupe A1

Maxime CHARLET, Geoffrey DELVAL, Simon FAUVRE

I. Avant-propos

La dernière amélioration (le dopplegänger) étant susceptible d'occasionner des bugs qui ne sont pas présents sans celle-ci, nous avons décidé de joindre 2 projets netBeans : un avec cette amélioration et un autre sans.

II. Présentation du projet

L'objet du projet est de développer, en langage JAVA, de l'IA d'Abigail dans un mini-jeu inspiré du jeu vidéo indépendant Stardew Valley.

Abigail est une fermière qui devra notamment réaliser ces actions particulières :

- Ramasser des œufs
- Traire la vache (si elle a produit du lait)
- Faire du fromage avec le lait récupéré

Elle peut également attendre, se déplacer, manger le fromage et boire le lait.

Une fois les 3 fonctionnalités de base implémentées, le programme devra intégrer 4 améliorations :

1. Abigail devra réaliser ces actions en respectant des règles liées à des besoins (faim, soif, fatigue).
2. Il est demandé d'implémenter un cycle jour/nuit, une journée étant découpée en 3 parties : matin, après-midi et soir. En fin de journée (c'est-à-dire le soir), Abigail devra se placer sur une case adjacente à sa maison et attendre le matin.
3. Les poules ne devront pas être capables de sortir de leur enclos
4. Implémenter la sœur jumelle d'Abigail qui possède une IA similaire. Il faudra que seule la plus proche des deux sœurs ne se dirige vers un œuf pour le ramasser (si ses besoins le lui permettent).

Le travail attendu consiste en une archive au format 7z contenant le projet netBeans ainsi que le présent rapport.

III. Modifications majeures apportées au programme

1. Nouvelles classes :

- Etat (abstraite), Attendre, Boit, CollecteOeuf, FabriqueFromage, Mange, ReposMaison, Trait : classes relatives à l'automate. Voir la partie qui traite de l'automate pour plus de détails.
- Moment_Journée : énumération des différents moments de la journée
- Couple, Graphes : classes qui permettent la création du graphe. Importées des TPs du module Graphe et modifiées.

2. Nouvelles méthodes :

Classe Graphe :

- void ajouterArete(int *debut*, int *fin*) : ajoute une arête de pondération 1
- void construireGrille(int *taille*) : construit une grille de taille passée en paramètre
- void obstacle(int *l*, int *c*, int *taille*) : modifie le graphe pour considérer un obstacle en position *l,c*

- void afficheGraphe(int *taille*) : utile en phase de test, permet d'afficher dans la console une représentation du graphe s'il a été conçu comme une grille dont la taille (largeur et hauteur) est passée en paramètre
- static Couple sommet2Coordonnees(int s) : retourne les coordonnées en fonction du sommet du graphe
- static int coordonnees2Sommet(int l, int c) : retourne le numéro de sommet du graphe
- int distanceDijkstra(int *depart*, int *arrivee*) : calcul la distance entre 2 sommets (valeur de retour) et stocke le parcours dans l'attribut *parcours* du graphe

Classe Carte :

- void InitGraphe() : création du graphe correspondant à la carte

Classe Abigail :

- Nouveaux assesseurs (correspondant aux nouveaux attributs)

Classe IA_Abigail :

- boolean aDuLait(), boolean aFaim(), boolean aSoif(), ... : méthodes retournant un booléen, utiles pour le fonctionnement des automates
- Abigail getAbigail() : permet de récupérer l'objet Abigail
- Enum_Action gestionBesoinsAction(Enum_Action *action*) : méthode qui gère la gestion des besoins d'Abigail en fonction des prochaines actions à effectuer
- Void deplacement(ArrayList <Integer> *parcours*) : méthode qui ajoute à l'attribut *listeActions* les prochains déplacements en fonction du parcours passé en paramètre

3. Génération du graphe

Un attribut *graphe* a été ajouté à la classe Carte. Le graphe est généré par la méthode *initGraphe()* qui fonctionne comme suit :

- Elle construit d'abord un graphe, sous forme de grille, de taille (taille de la carte)² de sorte qu'il y ai autant de sommets que de cases.
- Elle parcourt l'ensemble des cases de la carte et si elle rencontre un objet qui ne soit pas un œuf (et par conséquent dont on ne peut passer au travers), elle isole ce sommet du reste du graphe à l'aide de la méthode *obstacle* du graphe.

4. Fonctionnement de l'IA

L'IA d'Abigail fonctionne de manière très simple en 2 temps :




1. Si Abigail n'a aucune action à effectuer (la collection *listeActions* est vide), elle réfléchit aux prochaines actions à effectuer et les stockent dans l'attribut *listeActions* (collection de Enum_Action)
2. Si Abigail a des actions à effectuer (*listeActions* n'est pas vide), elle effectue (c'est-à-dire affecte à *resultat*) la prochaine action de la collection (indice 0) et la retire.

Ainsi, la prochaine action à effectuer sera toujours la première action de la liste des actions. Si Abigail n'a aucune action à effectuer, elle attend.

IV. Automate

L'IA d'Abigail est gérée par un automate. Ainsi, Abigail répond constamment à un état (impliquant certaines actions) et un état suivant (conditionnel selon ce qu'elle vient de faire, ses besoins ...).

Pour que l'automate fonctionne, il a fallu établir la gestion des priorités lorsqu'il est possible de passer à plusieurs états dans la même boucle de jeu afin de savoir quel choix l'IA doit faire. Ces priorités ont été modélisées dans le graphe ayant servi à la construction de l'automate, en la légende suivante :

	PRIORITE 1
	PRIORITE 2
	PRIORITE 3
	PRIORITE 4
	PRIORITE 5 (arc fin)
	PRIORITE 6 (arc fin)
	PRIORITE 7 (arc fin)
	PRIORITE 8 (arc fin)

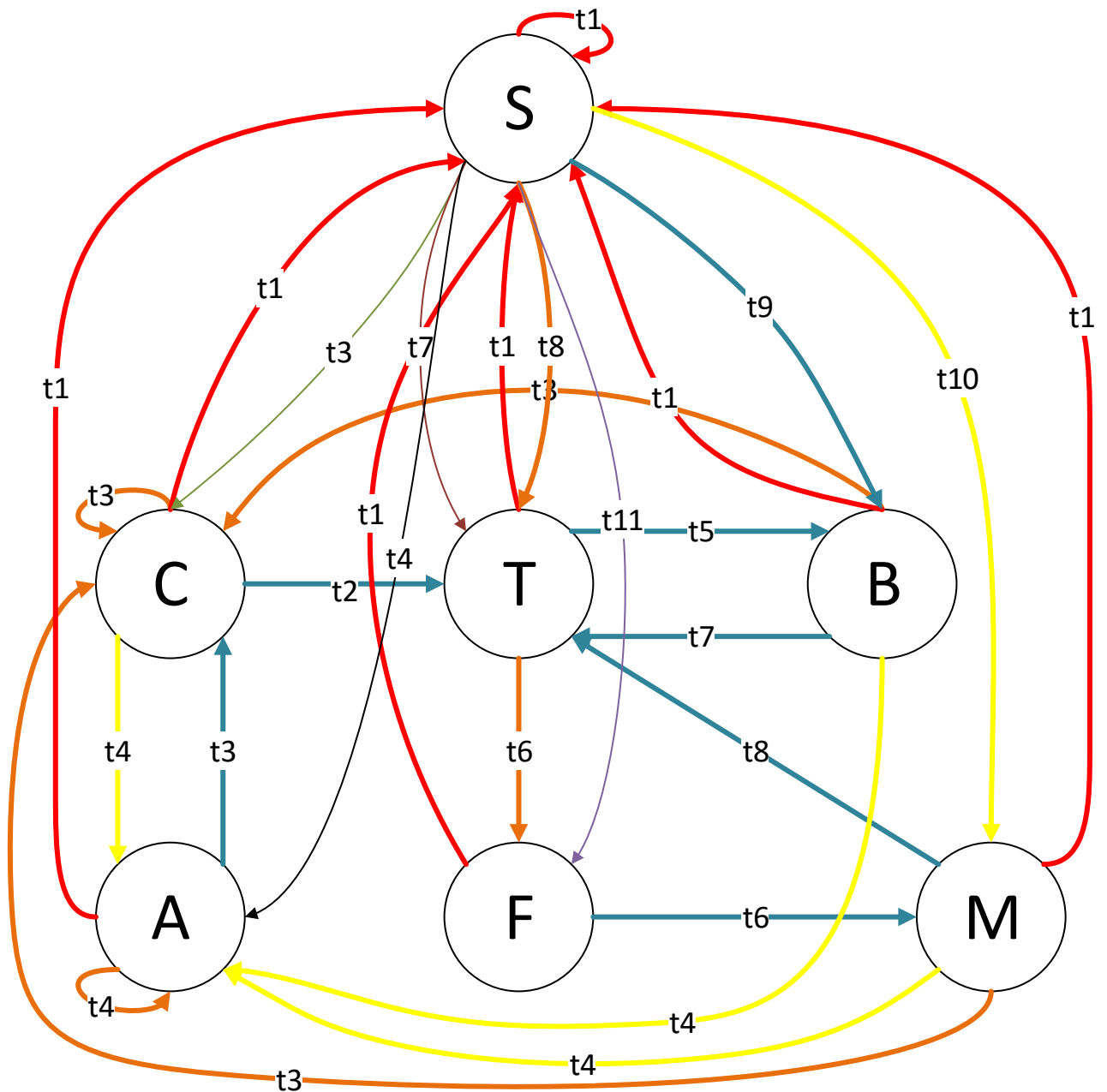
Les sommets représentent les états suivant :

A	Attendre
C	Collecte un œuf
T	Trait la vache
B	Boit du lait
F	Fabrique du fromage
M	Mange un fromage
S	Se repose près de la maison

Enfin, la liste des transitions :

t1	C'est le soir	t7	A faim ET a la vache a du lait
t2	(A soif ET la vache a du lait) OU (a faim ET la vache a du lait)	t8	A soif ET a la vache a du lait
t3	Il y a un œuf sur la carte	t9	A soif ET transporte du lait
t4	Il n'y a pas d'œuf sur la carte	t10	A faim ET transporte du fromage
t5	A soif	t11	A faim ET transporte du lait
t6	A faim		

Représentation du graphe :



Du point de vue de la programmation, les états sont modélisés par des classes qui héritent de la classe abstraite *Etat*. Chaque état contient deux méthodes en plus du constructeur :

- `Etat etatSuivant()` : effectue un test conditionnel pour connaître le prochain état de l'IA. Renvoie l'état suivant.
- `void action()` : actions à effectuer lorsque le personnage est dans cet état.

Les états possèdent un attribut *ia* de type *IA_Abigail*. C'est une variable qui référence l'objet *IA_Abigail*, dans le but d'avoir accès aux méthodes implémentées dans *IA_Abigail* pour gérer les différentes actions à effectuer (par exemple, le déplacement).

A chaque boucle de jeu, l'attribut *etat* de l'objet IA_Abigail (qui représente l'état courant) effectue l'action associée à cet état puis passe à l'état suivant ; et ceci à la condition que la liste des actions soit vide (dans le cas contraire, cela signifie qu'elle a encore des actions à effectuer : elle n'a donc pas besoin d'effectuer à nouveau l'action associée à cet état ni de passer à l'état suivant).

NB : il n'est pas précisé dans le sujet si Abigail doit traire la vache dès que possible ou uniquement quand c'est requis pour remplir un besoin. En amie des animaux, Abigail n'exploite donc pas cruellement la vache en volant systématiquement le lait qui revient à son veau : elle ne traie la vache que pour subvenir à un besoin et si cette dernière a produit du lait.

V. Améliorations

1. Gestion des besoins

Dans un premier temps, il a fallu ajouter à la classe Abigail les attributs relatifs aux besoins (*fatigue*, *faim* et *soif*) initialisés à 0.

D'une part, l'automate permet de gérer l'action suivante en fonction des valeurs des besoins d'Abigail.

D'autre part, les besoins sont incrémentés dans la méthode `gestionBesoinsAction(Enum_Action action)` de la classe IA_Abigail. Cette méthode gère les différents phénomènes liés aux actions d'Abigail : les déplacements augmentent la fatigue et la soif, ramasser un œuf augmente la faim. Un indice permet de doubler l'incrémentation de la fatigue selon le moment de la journée.

Cette méthode est donc appelée à chaque fois que IA_Abigail s'apprête à effectuer une action. Si l'action fait augmenter la fatigue à 100 points ou plus, l'`Enum_Action attendre` est ajoutée à la liste des actions pour que la prochaine action d'Abigail soit de se reposer. La méthode renvoie l'action passée en paramètre (le premier élément de la collection *listeActions*) et l'affecte à *resultat* afin d'effectivement réaliser l'action à la fin de la boucle de jeu.

2. Fin de journée

L'énumération `Moment_Journée` énumère les trois périodes : *Matin*, *Après_Midi* et *Soir*. La mise en place d'un cycle jour/nuit s'effectue dans la classe `TimerIAHandler`. Une journée vaut 24 heures et les périodes sont découpées comme suit :

- Matin : de 5h à 12h (inclus)
- Après-midi : de 13h à 20h (inclus)
- Soir : de 21h à 4h (inclus)

Un *compteur* s'incrément à chaque boucle de jeu et équivaut à une unité de temps. Quand ce compteur vaut une certaine valeur (nous avons arbitrairement choisi 100 et préféré effectuer un test à l'aide de l'opérateur modulo, mais nous aurions également pu décider de rendre à *compteur* la valeur 0 à chaque fois qu'il vaut 100), alors on incrémente l'heure de 1. Puis on effectue une série de tests pour vérifier si le moment de la journée a changé.

Concernant la gestion des besoins lié au moment de la journée, ce point a été explicité dans la partie précédente (gestion des besoins).

La modélisation de l'automate permet également à Abigail d'aller se reposer près de sa maison quand le moment de la journée est *Soir*.

3. Des poules bien dressées

La résolution de ce problème s'effectue dans la classe `IA_Poule`. Lorsque le programme établit la liste des actions possibles pour la poule (à chaque boucle de jeu), il suffit de modifier le code pour vérifier que la poule se trouve face à une ouverture dans la barrière.

Pour cela, il faut ajouter des tests dans les conditions `if` pour vérifier que 2 barrières ne se trouvent pas sur les cases adjacentes en diagonale à la poule. Pour cela, on peut utiliser la méthode `estLibre()` des cases que l'on souhaite tester (puisque les seuls objets rencontrés dans l'enceinte de l'enclos sont des barrières). Si des barrières se trouvent à ces endroits, alors c'est que la poule est face à une ouverture de l'enclos et on n'ajoute pas le déplacement qui lui permettrait de sortir à la liste des actions possibles.

4. Doppelgänger

Pour gérer la sœur jumelle d'Abigail, que nous avons nommé Simone, la création d'une classe `GestionIAHumain` a été requise. Cette classe a pour but de gérer la cohabitation des deux IAs. Elle contient 2 méthodes qui renvoient des booléens et qui seront appelées dans la méthode `action()` de l'automate (état `CollecteOeuf`) afin de définir si l'IA est plus proche de l'œuf ou non et si l'autre IA est en train d'effectuer une série d'actions.

Il a également fallu dupliquer l'automate d'Abigail pour créer un automate pour Simone. Ce nouvel automate répond aux mêmes règles mais possède comme attribut une variable de type `IA_Simone` référençant l'IA de Simone.

Pour décider du personnage qui ira collecter l'œuf, il faut prendre en compte deux éléments :

- est-ce que ce personnage est le plus proche de l'œuf ?
- est-ce que l'autre personnage est occupé ? (sous-entendu : il ne peut pas aller récupérer l'œuf en cet instant)

Si l'une de ces deux conditions est validée, alors le personnage peut aller collecter l'œuf.

Lorsqu'un personnage se dirige vers un œuf pour le collecter, il faut également que l'autre personnage arrête de considérer cet œuf dans le calcul de l'œuf le plus proche. Pour cela, un attribut *marque* a été ajouté à la classe `Œuf`. Cet attribut, initialisé à faux, passe à vrai lorsqu'un personnage valide le parcours vers un œuf et s'apprête donc à effectuer la série d'actions visant à aller le collecter.