MATH EN INFO

TRAVAUX PRATIQUES N° 7

Destruction automatique de la nature

1 Objectifs

Notre IA commence maintenant à être capable de prendre des décisions, simples certes, mais ce n'est qu'un début. Durant le dernier TP, nous lui avons appris à aller couper des arbres mais il serait souhaitable que notre IA évite de traverser l'intégralité de la carte juste pour couper un arbre et qu'elle choisisse plutôt l'arbre le plus proche d'elle. Par la suite nous allons apprendre à notre IA de nouvelle fonctionnalité : acheter des graines, les planter, arroser... Sans changement de structure, cela risque de rendre notre méthode de prise de décision extrêmement lourde. Pour éviter cela, nous allons mettre en place une structure d'automate fini comme vu en CM.

2 Détruisons la nature, de façon méthodique !

Commençons par apprendre à notre IA à chercher l'arbre le plus proche d'elle et à aller le couper.

1. **A FAIRE : Dans le module de décision, créer une méthode "private void couperLArbreLePlusProche()" et implémenter la en suivant le pseudo-code suivant :**

```
Créer un nouvelle algorithme de Dijkstra et lancer le calcul des
    distances depuis la case du joueur.
"caseAvecArbreLaPlusProche" <- null
"distanceMinimale" <- -1
Pour toutes les cases "c" de la carte
        Si "c" contient un objet et que cet objet est un arbre
                si "caseAvecArbreLaPlusProche" est nulle ou si la
                    distance jusqu à "c" < distanceMinimale alors
                        "caseAvecArbreLaPlusProche" <- c
                        "distanceMinimale" <- distance jusqu à "c"
Si "caseAvecArbreLaPlusProche" est non nulle
        se déplacer en "caseAvecArbreLaPlusProche"
        Si la liste des actions à réaliser n'est pas vide
                Supprimer le dernier élément de cette liste
```

2. **A FAIRE : Remplacer le contenu de la méthode "determinerNouvelles-Actions()" par un simple appel à la méthode "couperLArbreLePlus-Proche()".**
3. **A FAIRE : Vérifier qu'Abigail se débarrasse maintenant relativement efficacement de toute forme de verdure sur la carte !**

Si vous êtes très attentif au comportement d'Abigail, il reste un petit souci dans son comportement. Actuellement, elle se rend systématiquement sur la case de l'arbre après l'avoir coupé. Pour éviter cela, nous allons supprimer le dernier mouvement de la liste des actions et le remplacer par deux actions de coupe de bois.

4. **A FAIRE : Dans la méthode "couperLArbreLePlusProche" ajouter le code suivant juste après l'appel à la méthode "seDeplacerEn" :**

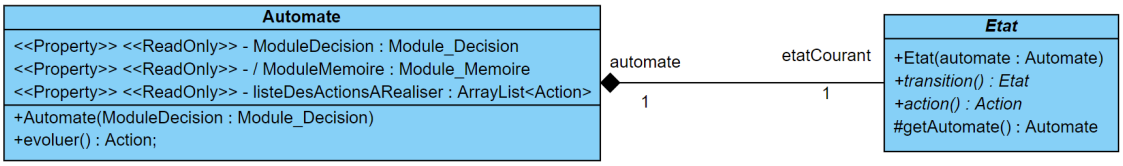
```
if(!this.listeDesActionsARealiser.isEmpty()) {
        Action action = this.listeDesActionsARealiser.get(this.
            listeDesActionsARealiser.size()-1);
        this.listeDesActionsARealiser.remove(this.
            listeDesActionsARealiser.size()-1);
        this.listeDesActionsARealiser.add(FabriqueAction.
            creerActionRecolte(TypeActionRecolte.COUPERARBRE, action.
            getDirection()));
        this.listeDesActionsARealiser.add(FabriqueAction.
            creerActionRecolte(TypeActionRecolte.COUPERARBRE, action.
            getDirection()));
}
```

5. **A FAIRE : Contempler le carnage réalisé par Abigail.**

3 Automate

Il est maintenant tant de mettre en place une structure d'automate pour structurer et alléger le module de décision.

6. **A FAIRE : Créer un package "automate" dans le package modules.**
7. **A FAIRE : Dans le package "automate", créer une classe Automate et une classe abstraite "Etat" en suivant le diagramme UML suivant :**



8. **A FAIRE : Implémenter toutes les méthodes (non abstraites) de ces deux classes. Pour la méthode "évoluer" de l'automate, on suivra le pseudo code suivant :**

```
"prochaineAction" <- null
tant que "prochaineAction" est nul
        etatCourant <- etatCourant.transition();
        "prochaineAction" <- etatCourant.action();
renvoyer "prochaineAction"
```

Comme vous pouvez le constater l'automate que nous implémentons ici diffère légèrement de celui présenté en CM, principalement la méthode "evoluer" de la classe automate. Cette différence vient du fait que dans notre cas, l'automate possèdera deux types d'états, des états de réflexion et des états d'action. A chaque "tour", l'automate évoluera jusqu'à arriver dans un état d'action, les autres états lui permettant juste de déterminer vers quel état d'action se rendre. Nous allons maintenant créer les états 1 par 1 pour obtenir l'automate présenté à la fin du document.

9. **A FAIRE : Créer un package "etats" dans le package "automate". Ce package contiendra toutes les classes des états.**

10. **A FAIRE : Créer une classe pour chacun des états de l'automate en suivant la nomenclature donnée à la fin du document, en les faisant hériter de la classe "Etat". (Votre code ne doit pas contenir d'erreur.)**

3.1 EtatInitial

L'état initial sert de point de départ à l'automate. Il s'agit d'un état de réflexion (la méthode action renvoie null).

11. **A FAIRE : Implémenter les méthodes "action" et "transition" de la classe EtatInitial. (Avant de demander à votre professeur ou votre chargé de TD pourquoi la question ne vous indique pas quoi faire dans la méthode "transition", regardez bien l'automate donné à la fin du sujet !)**
12. **A FAIRE : Modifier le constructeur de l'automate pour qu'il initialise "etatCourant" à un nouvel état initial.**

3.2 EtatCheckCarte

L'état EtatCheckCarte sert à tester si l'IA possède la carte ou non. Il s'agit d'un état de réflexion.

13. **A FAIRE : Implémenter la méthode "action" de la classe "EtatCheckCarte" pour qu'elle renvoie null.**
14. **A FAIRE : Implémenter la méthode "transition" de la classe "EtatCheckCarte" pour qu'elle renvoie un nouvel état EtatCheckAction si l'IA possède déjà la carte et un nouvel état EtatDemanderCarte sinon.**

3.3 EtatDemanderCarte

L'état EtatDemanderCarte fait demander la carte au serveur. C'est notre premier état d'action.

15. **A FAIRE : Implémenter la méthode "action" de la classe "EtatDemanderCarte" pour qu'elle renvoie une demande de carte.**
16. **A FAIRE : Implémenter la méthode "transition" de la classe "EtatDemanderCarte".**

3.4 EtatCheckAction

L'état EtatCheckAction est un état de réflexion qui teste si une action à déjà été planifiée ou non (on notera que c'est maintenant l'automate qui est en charge de gérer cette liste).

17. **A FAIRE : Implémenter la méthode "action" de la classe "EtatCheckAction" pour qu'elle renvoie null.**
18. **A FAIRE : Implémenter la méthode "transition" de la classe "EtatCheckAction" pour qu'elle renvoie un nouvel état "EtatRealiserAction" si l'automate dispose d'une action planifiée et un nouvel état "EtatAllerVersArbre".**

3.5 EtatRealiserAction

L'état EtatRealiserAction est un état d'action qui réalise la première action de la liste des actions.

19. **A FAIRE : Implémenter la méthode "transition" de la classe "EtatRealiserAction" pour qu'elle renvoie un nouvel état EtatCheckAction.**

20. **A FAIRE : Implémenter la méthode ”action” de la classe ”EtatRealiserAction” en suivant le pseudo-code suivant :**

```
”action” <— premier élément de la liste des actions à réaliser
si action est de type mouvement
        ”coordonneeDestination” <— coordonnée obtenu en partant de
            celle du joueur et en se déplaçant suivant la direction de l
             ”action”
        ”caseDestination” <— la case située aux ”
            coordonneeDestination”
        si la case de destination possède un objet et que celui—ci est
            un arbre
                ajouter en position 0 deux actions ”COUPERARBRE”
                    de même direction que l ”action”.
”action” <— premier élément de la liste des actions à réaliser
Supprimer le premier élément de la liste des actions à réaliser
Renvoyer ”action”
```

3.6 EtatAllerVersArbre

L'état EtatAllerVersArbre est un état de réflexion qui détermine l'arbre le plus proche du joueur et détermine le chemin pour s'y rendre (en ajoutant les actions à la liste des actions à réaliser).

21. **A FAIRE : Ajouter un attribut ”arbreTrouve” de type booléen à la classe et l'initialiser à faux dans le constructeur.**
22. **A FAIRE : Implémenter la méthode ”transition” de la classe ”EtatAllerVersArbre” pour qu'elle renvoie un nouvel état ”EtatAllerDormir” si arbreTrouve est à faux et un nouvel état ”EtatCheckAction” sinon.**

La méthode action va déterminer l'arbre le plus proche puis déterminer le chemin pour s'y rendre et le couper. Cet état ne sera pas le seul à demander à l'IA de se rendre sur une case, nous allons donc créer une méthode directement dans la classe abstraite ”Etat” déterminant le chemin pour se rendre à un lieu et ajoutant les actions correspondantes dans la liste des actions à réaliser.

23. **A FAIRE : Dans la classe ”Etat”, ajouter la méthode :**

    protected void seDeplacerEn(Coordonnee coordonnee)

24. **A FAIRE : En s'inspirant de la méthode du même nom du module de décision, implémenter cette méthode.**
25. **A FAIRE : En s'inspirant de la méthode ”couperLArbreLePlusProche” du module de décision, implémenter la méthode ”action” de l'état ”EtatAllerVersArbre” (attention, on utilisera bien la méthode seDeplacerEn de la classe état et non celle du module de décision). Bien penser à passer le booléen ”arbreTrouve” à vrai si un arbre est trouvé. L'état étant un état de réflexion, la méthode renverra null.**

3.7 EtatAllerDormir

L'état EtatAllerDormir est un état de réflexion renvoie le joueur au point de départ pour dormir.

26. **A FAIRE : Implémenter la méthode ”transition” de la classe ”EtatAllerDormir” pour qu'elle renvoie un nouvel état ”CheckAction”.**

27. **A FAIRE : En s'inspirant de la méthode "allerDormir" du module de décision, implémenter la méthode "action" de la classe "EtatAllerDormir". La méthode renverra null.**

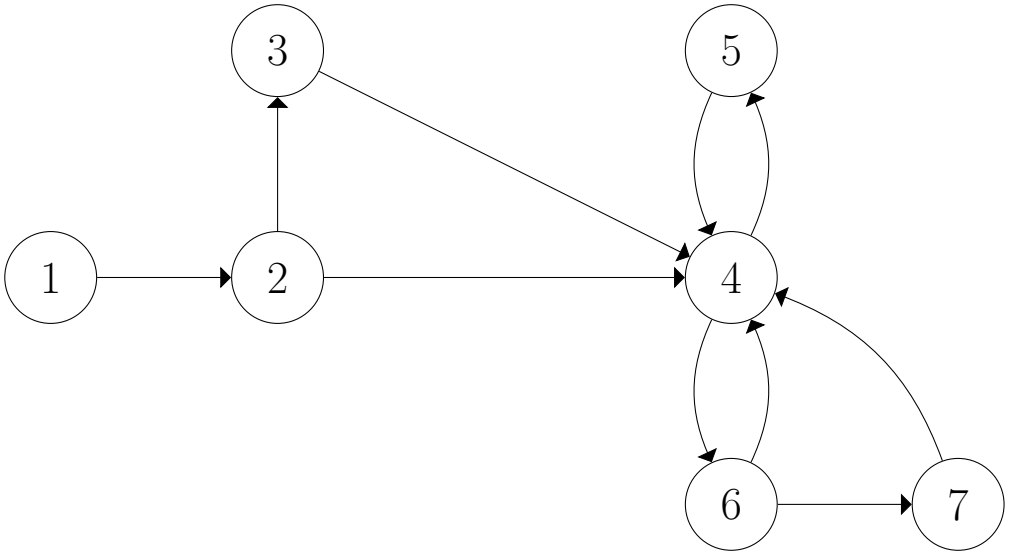
3.8 Module de décision

Notre automate est maintenant prêt à être utilisé. Il ne reste plus qu'à expliquer à notre module de décision comment l'utiliser.

28. **A FAIRE : Dans le module de décision, supprimer les méthodes "déterminerNouvellesActions", "seDeplacerEn", "allerDormir", et "couperLArbreLePlusProche".**
29. **A FAIRE : Supprimer le contenu (uniquement le contenu) de la méthode "determinerNouvelleAction". (Pensez à supprimer les imports maintenant inutiles)**
30. **A FAIRE : Ajouter un attribut "automate" de type Automate et créer le dans le constructeur.**
31. **A FAIRE : Modifier la méthode "determinerNouvelleAction" pour qu'elle récupère l'action renvoyée par la méthode évoluer de l'automate, demande au module mémoire d'effectuer cette action et renvoie le message de l'action.**
32. **A FAIRE : Tester l'IA.**

Notre IA est de nouveau capable de déforester l'intégralité de la carte rapidement. Bien que nous n'ayons pas encore rajouté de nouvelle fonctionnalité, nous disposons maintenant d'une structure propre, simple et qui sera facilement modifiable par la suite. Dans le prochain TP nous pourrons donc (enfin) commencer de planter des choux (enfin des panais...) !

4 Description de l'automate



N°	Nom de la classe	Descriptif
1	EtatInitial	Point de départ de l'automate
2	EtatCheckCarte	Test si la carte existe
3	EtatDemanderCarte	Demande la carte au serveur
4	EtatCheckAction	Regarde si une action est déjà planifiée
5	EtatRealiserAction	Réalise l'action planifiée
6	EtatAllerVersArbre	Détermine l'arbre le plus proche
7	EtatAllerDormir	Détermine comment rentrer pour dormir