MATH EN INFO

TRAVAUX PRATIQUES N°9 ET PLUS

PME

1 Objectif du TP

Il est temps de finir ce que nous avons commencé durant le TP 8 et de permettre à Abigail de planter et récolter ses premiers légumes.

2 Plantons des plantes

Si l'on souhaite qu'Abigail puisse planter des plantes, puis les récolter, il faut que notre IA les gère, or actuellement, nous n'avons rien créé pour pouvoir manipuler la notion de plante. Il est temps d'y remédier.

1. **A FAIRE : Ajouter deux nouveaux éléments à l'énumération TypeObjet : PANAIS et CHOUFLEUR.**
2. **A FAIRE : Ajouter une classe abstraite Plante héritant de la classe Objet. Implémenter sa méthode estBloquant pour indiquer que les plantes ne seront jamais bloquantes pour notre personnage.**
3. **A FAIRE : Ajouter deux classes PlantePanais et PlanteChouFleur héritant de la classe Plante. Ne pas oublier d'implémenter les méthodes getType et getLoot de chacune d'elles (on rappelle qu'une plante donne 2 unités de légumes matures à la récolte).**
4. **A FAIRE : Dans la FabriqueObjet, ajouter une méthode**

   public static Plante creerPlante(Case position, TypeObjet type)

   et l'implémenter (la méthode renverra null si type n'est ni PANAIS ni CHOUFLEUR).

La classe abstraite plante aura plusieurs intérêts que nous verrons par la suite, elle permettra de manipuler indistinctement le panais et les choux-fleurs tout en différenciant les plantes des autres objets.

Notre IA a besoin de se souvenir des plantes qu'elle a plantées, il nous faut donc modifier le module mémoire.

5. **A FAIRE : Ajouter au module mémoire un attribut "listePlantes" de type ArrayList<Plante> (bien penser à l'initialiser...) et son getter.**
6. **A FAIRE : Ajouter au module mémoire une méthode :**

   private void addPlante(TypeObjet type)

   Cette méthode demande à la fabrique objet de créer une plante du type donné en paramètre et sur la case du joueur, puis ajouter cette plante à la case du joueur et enfin ajoute cette plante à la liste des plantes.
7. **A FAIRE : Ajouter au module mémoire une méthode :**

private void recolterPlante()

Cette méthode récupère l'objet situé la case du joueur. Récolte l'objet puis le retire de la liste des plantes et enfin supprime l'objet de la case.

Notez au passage que l'on utilise ici une spécificité du JAVA sur la gestion des listes. Cela ne pose pas de problème de retirer un "Objet" d'une liste de "Plante".

Il nous faut maintenant créer une nouvelle action : "Planter".

8. **A FAIRE : Ajouter un nouvel élément à l'énumération TypeAction : PLANTER.**
9. **A FAIRE : Créer une nouvelle classe "ActionPlanter" héritant de la classe "Action". Cette classe possèdera un attribut de type TypeRessource (qui représentera le type de graine plantée) qui sera initialisé à une valeur donnée en paramètre du constructeur.**
10. **A FAIRE : Ajouter à la fabrique des actions une méthode :**

    public static Action creerActionPlanter(TypeRessource typeRessource)

Nous pouvons maintenant nous attaquer à l'automate.

11. **A FAIRE : Ajouter un attribut "aPlante" de type booléen dans l'état EtatAllerPlanter et l'initialiser à faux.**
12. **A FAIRE : Implémenter la méthode transition de cet état pour qu'elle renvoie un nouvel EtatCheckAction si aPlante est vrai et un nouvel EtatAllerDormir sinon.**
13. **A FAIRE : Implémenter la méthode action de cet état pour qu'elle suive le pseudo-code suivant :**

```
Si le joueur possède au moins 1 graine de panais
        Créer un nouvelle algorithme de Dijkstra et lancer le calcul
            des distances depuis la case du joueur.
        "caseTerreVide" <- null
        "distanceMinimale" <- -1
        Pour toutes les cases "c" de la carte
                Si "c" est une case de terre sans objet
                        si "caseTerreVide" est nulle ou si la distance
                            jusqu à "c" < distanceMinimale alors
                                "caseTerreVide" <- c
                                "distanceMinimale" <- distance jusqu à
                                    "c"
        Si "caseTerreVide" est non nulle
                se déplacer en "caseTerreVide"
                ajouter une nouvelle action "planter une graine de
                    panais" à la liste des actions.
                aPlante <- true
renvoyer null
```

14. **A FAIRE : Modifier EtatAcheter pour que la méthode transition renvoie une nouvel état EtatAllerPlanter quand elle renvoyait un nouvel EtatAllerDormir.**

Si vous testez votre IA maintenant vous verrez qu'Abigail, après avoir acheté des graines, se rend sur une case de terre et plante sa première graine ! Mais décide alors de planter toutes les autres graines au même endroit... Il faut que le module mémoire réagisse à l'action planter pour se rappeler de la présence de la nouvelle plante.

15. **A FAIRE : Modifier la méthode effectuerAction pour que dans le cas d'une action planter elle ajoute une nouvelle plante de type Panais (en utilisant la méthode addPlante) et retire 1 graine de panais de l'inventaire.**
16. **A FAIRE : Tester l'IA. Abigail doit planter des graines de panais jusqu'à ne plus avoir d'argent.**

3 A boire !

Si Abigail a appris à planter des graines, elle ne les arrose pas. Changeons cela ! Commençons par apprendre à Abigail une nouvelle action : arroser.

17. **A FAIRE : Ajouter un attribut "estArrosee" de type booléen dans la classe "Plante", initialisé à faux ainsi que son setter et son getter.**
18. **A FAIRE : Créer un nouvel élément à l'énumération TypeAction : "ARROSER".**
19. **A FAIRE : Créer une nouvelle action : ActionArroser.**
20. **A FAIRE : Ajouter dans la fabrique des actions une nouvelle méthode :**

    public static Action creerActionArroser()

Ajoutons maintenant l'arrosage dans l'automate de prise de décision.

21. **A FAIRE : Si ce n'est pas déjà fait, ajouter un getter pour l'attribut position de la classe Objet.**
22. **A FAIRE : Ajouter un attribut "aArrose" de type booléen à l'état EtatAllerArroser, initialisé à faux.**
23. **A FAIRE : Modifier la méthode transition de l'état pour qu'elle renvoie un nouvel état "EtatCheckAction" si aArrose est vrai et un nouvel état "EttatAllerDormir" sinon.**
24. **A FAIRE : En s'inspirant de ce qu'on a fait pour l'état EtatAllerPlanter, et en se rappelant que le module mémoire connaît la liste des plantes, modifier la méthode action pour qu'elle :**
    - **Détermine la case contenant une plante non arrosée la plus proche.**
    - **Si elle en trouve une, elle met "aArrose" à vrai, ajoute à la liste des actions à réaliser le déplacement jusqu'à la case en question (en utilisant seDeplacerEn puis ajoute une action ARROSER à la liste des actions.**
    - **Renvoyer null.**
25. **A FAIRE : Modifier la méthode transition de l'état EtatAllerPlanter pour qu'elle renvoie un nouvel EtatAllerArroser quand elle renvoyait un EtatAllerDormir.**
26. **A FAIRE : Modifier la méthode "effectuerAction" du module mémoire pour qu'en cas d'action ARROSER elle parcourt la liste des plantes et arrose celle située sur la même case que le joueur.**

En testant l'IA, on se rend vite compte que si Abigail arrose bien les plantes le premier jour, elle ne le fait plus (sauf pour les nouvelles plantes) dès le deuxième jour. En effet, pour notre IA, une fois arrosée, une plante reste définitivement arrosée. Corrigeons cela !

27. **A FAIRE : Modifier la méthode "effectuerAction" pour que dans le cas d'une ACTIONSTATIQUE, en plus de mettre le stock du magasin à null, l'IA considère que toutes les plantes sont "non arrosées".**

Si maintenant Abigail arrose bien toutes les plantes le deuxième jour, un autre problème apparaît, son arrosoir est vide et Abigail ne se rend pas compte !

28. **A FAIRE : Ajouter un élément "EAU" à l'énumération TypeRessource.**
29. **A FAIRE : Dans le constructeur du module mémoire, s'assurer que 20 unités d'eau soient mises dans l'inventaire.**
30. **A FAIRE : Modifier la méthode effectuerAction pour diminuer le nombre d'unités d'eau de 1 dans l'inventaire lors d'une action ARROSER.**

Abigail peut maintenant se rendre compte que son arrosoir est vide, apprenons-lui à aller le remplir.

31. **A FAIRE : Ajouter un nouvel élément "REMPLIR" à l'énumération TypeAction.**
32. **A FAIRE : Ajouter une nouvelle action "ActionRemplir" ayant un attribut direction de type TypeMouvement initialisé dans le constructeur par une valeur donnée en paramètre.**
33. **A FAIRE : Ajouter à la fabrique des actions la méthode suivante :**

    public static Action creerActionRemplir(TypeMouvement direction)

34. **A FAIRE : Modifier la méthode transition de l'état EtatAllerRemplir pour qu'elle renvoie un nouvel EtatCheckAction.**
35. **A FAIRE : Modifier la méthode action de cet état pour qu'elle suive le pseudo-code suivant :**

```
Créer un nouvelle algorithme de Dijkstra et lancer le calcul des
    distances depuis la case du joueur.
"caseBordDeLEauLaPlusProche" <- null
"distanceMinimale" <- -1
Pour toutes les cases "c" de la carte
        Si "c" est une case accessible
                "bordDeLEau" <- faux
                Pour chacune des cases voisines "v" de "c"
                        Si "v" est une case d eau
                                "bordDeLEau" <- vrai
                Si "bordDeLEau" est vrai
                        Si "caseBordDeLEauLaPlusProche" == null ou
                           si distance(c) < "distanceMinimale"
                                "caseBordDeLEauLaPlusProche" <- c
                                "distanceMinimale" <- distance(c)
se rendre à "caseBordDeLEauLaPlusProche"
"direction" <- null
Pour chacun des voisins "v" de "caseBordDeLEauLaPlusProche"
        si "v" est une case d eau
                "direction" <- direction pour aller de "
                   caseBordDeLEauLaPlusProche" à "v"
ajouter une nouvelle action remplir dans cette direction.
Renvoyer null
```

36. **A FAIRE : Modifier la méthode "effectuerAction" du module mémoire pour que dans le cas d'une action de type REMPLIR, le nombre d'unités d'eau dans l'inventaire repasse à 20.**

37. **A FAIRE : Modifier la méthode transition de l'état EtatAllerArroser pour que si aArrose est faux, si Abigail n'a plus d'eau, cela renvoie un nouvel état EtatAllerRemplir et si elle a encore de l'eau, cela renvoie un nouvel état EtatAllerDormir.**
38. **A FAIRE : Modifier la méthode action de EtatAllerArroser pour que tout ce qui y est fait ne soit réalisé que si Abigail possède de l'eau dans son inventaire.**
39. **A FAIRE : Tester l'IA.**

4 Recolte

Abigail s'occupe maintenant bien de ses plantes. Il est temps de gérer la récolte. Pour cela, il nous faut gérer la maturité des plantes.

40. **A FAIRE : Ajouter un attribut "age" de type entier à la classe plante, initialisé à 0 ainsi qu'un getter protégé pour cet attribut.**
41. **A FAIRE : Ajouter dans la classe Plante une méthode :**

    public void grandir()

    qui augmente de 1 l'âge de la plante.
42. **A FAIRE : Ajouter dans la classe Plante une méthode abstraite :**

    public abstract boolean estMature() ;

    et l'implémenter dans les classes filles (on rappelle qu'un panais est mature si son âge est supérieur ou égal à 5 et un chou-fleur est mature si son âge est supérieur ou égal à 13).
43. **A FAIRE : Modifier la méthode "effectuerAction" du module mémoire pour que toutes les plantes arrosées grandissent lors d'une action statique (attention à bien le faire avant de considérer qu'aucune plante n'est arrosée !).**

Il faut maintenant apprendre à Abigail à aller récolter les plantes matures :

44. **A FAIRE : Ajouter un nouvel élément "CUEILLIR" à l'énumération TypeAction.**
45. **A FAIRE : Ajouter une nouvelle action "ActionCueillir".**
46. **A FAIRE : Ajouter à la fabrique des actions la méthode suivante :**

    public static Action creerActionCueillir()

47. **A FAIRE : Ajouter un attribut de type booléen "aRecolte" l'état EtatAllerRecolter, initialisé à faux.**
48. **A FAIRE : Modifier la méthode transition pour que si "ARecote" est vrai, elle renvoie un nouvel état EtatCheckAction et sinon elle renvoie un nouvel état EtatAcheter.**
49. **A FAIRE : En s'inspirant de la méthode action de l'état EtatAllerArroser, modifier la méthode action de l'état EtatAllerRecolter pour qu'elle cherche la plante mature la plus proche, si elle en trouve une, elle se déplace sur cette case, récolte la plante et met aRecolte à vrai.**
50. **A FAIRE : Modifier la méthode transition de l'état EtatCheckAction pour qu'elle renvoie un nouvel EtatAllerRecolter quand elle renvoyait un état EtatAcheter.**

51. **A FAIRE : Modifier la méthode "effectuerAction" du module de mémoire pour que dans le cas d'une action de type CUEILLIR, elle rajoute 2 unités de la bonne ressource dans l'inventaire, retire la plante de la liste des plantes et supprime l'objet situé sur la case du joueur.**
52. **A FAIRE : Tester l'IA.**

5 Petite ferme deviendra grande !

Abigail sait maintenant s'occuper de ses plantes et les récolter. Elle va pouvoir enfin se faire un peu d'argent.

53. **A FAIRE : En s'inspirant des différents TPs, apprendre à Abigail à vendre ses légumes.**

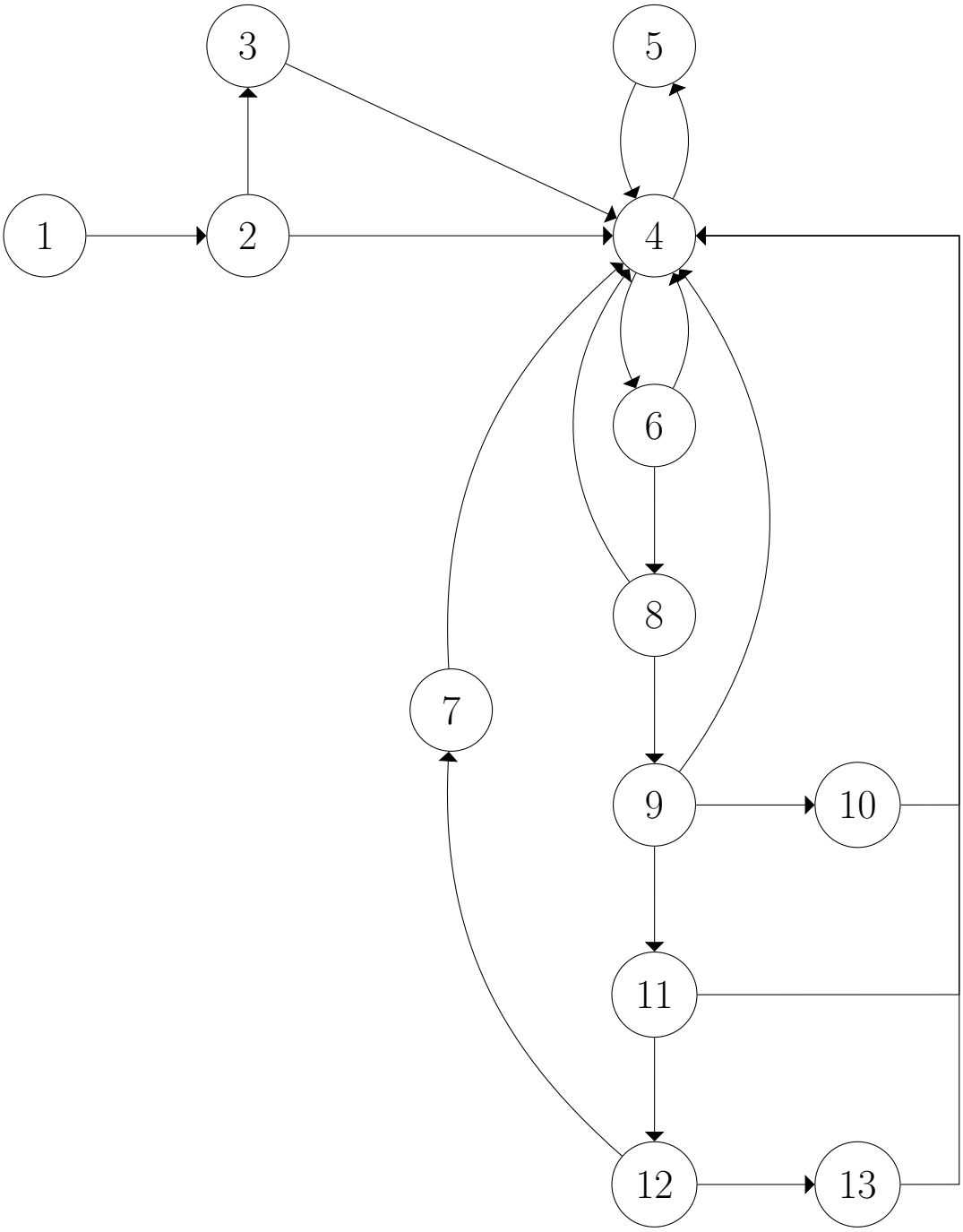
Maintenant qu'Abigail se fait de l'argent, pourquoi ne pas investir dans les poules ?

54. **A FAIRE : En s'inspirant des différents TPs, apprendre à Abigail à acheter une poule dès qu'elle possède de quoi l'acheter et à ramasser les oeufs chaque matin.**

Les oeufs c'est bien, la mayonnaise c'est mieux !

55. **A FAIRE : En s'inspirant des différents TPs, apprendre à Abigail à construire une machine à faire de la mayonnaise et à utiliser les oeufs pour en faire.**
56. **A FAIRE : Contempler l'IA jouer seule...**

6 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	EtatAllerRecolter	Détermine la plante la plus proche
7	EtatAllerDormir	Détermine comment rentrer pour dormir
8	EtatVendre	Va vendre ses légumes
9	EtatAcheter	Va acheter des graines
10	EtatDemandeMagasin	Demande le stock du magasin
11	EtatAllerPlanter	Va planter ses graines
12	EtatAllerArroser	Va arroser ses plantes
13	EtatAllerRemplir	Va remplir l'arrosoir