MATH EN INFO

TRAVAUX PRATIQUES N° 4

Distances et chemins

1 Objectifs

Maintenant que nous disposons d'une carte fonctionnelle, nous allons nous en servir pour déterminer le plus court chemin pour Abigail. Pour cela, il nous faut tout d'abord implémenter un algorithme de calcul de distances (un simple parcours en largeur dans un premier temps) puis interpréter le résultat de cet algorithme pour en déduire la série de déplacements à réaliser par Abigail.

2 Gestion de l'accessibilité des cases

Pour le moment, les cases de notre carte ne sont pas accessibles depuis l'extérieur de notre classe Carte. Nous avons besoin de changer cela.

1. **A FAIRE : Ajouter une méthode :**

   public Case getCase(Coordonnee coordonnee)

   à la classe "Carte" et l'implémenter pour qu'elle renvoie la case de la carte située aux coordonnées données en paramètre.

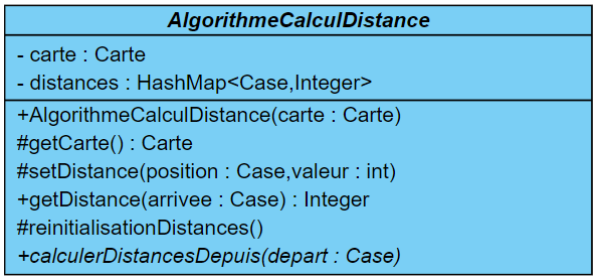
Il nous faut de plus différencier les cases "accessibles" aux joueurs et celles "inaccessibles". Si les cases d'eau sont toujours inaccessibles et les cases de terre toujours accessibles, les cases d'herbe sont plus problématiques. En effet, elles sont par défaut accessibles sauf si un objet "bloquant" est présent sur la case.

2. **A FAIRE : Ajouter une nouvelle méthode abstraite "public boolean estBloquant()" dans la classe "Objet" et implémentez-la. Cette méthode renvoie "true" pour les arbres et la maison et renvoie "false" pour les autres objets.**
3. **A FAIRE : Ajouter une nouvelle méthode abstraite "public boolean estAccessible()" dans la classe "Case" et implémentez-la dans les différentes classes filles. Cette méthode renvoie "true" si la case est une case de terre ou une case d'herbe ne contenant pas d'objet bloquant.**

3 Algorithme de calcul des distances

Nous allons commencer par mettre en place une classe abstraite représentant un algorithme (quelconque) de calcul de distances. Bien que durant ce TP nous n'utiliserons qu'un seul algorithme (un parcours en largeur) cette structure nous permettra d'en rajouter un autre plus pertinent par la suite sans devoir modifier notre code.

4. **A FAIRE : Créer un nouveau package "algorithmes" dans le package metier.**
5. **A FAIRE : Créer une nouvelle classe abstraite "AlgorithmeCalculDistance" dans le package "algorithmes" en respectant l'UML suivant :**



- **La méthode (protégée) getCarte renvoie simplement la carte.**
- **La méthode (protégée) setDistance insère le couple (position,valeur) dans la HashMap distances.**
- **La méthode (publique) getDistance renvoie la valeur stockée dans la HashMap pour la case donnée en paramètre. (Attention, le type de retour doit impérativement être Integer et non pas int. Si vous n'avez pas vu la différence entre Integer et int en POO, n'hésitez pas à demander à votre responsable de TP).**
- **La méthode (protégée) reinitialisationDistances vide la HashMap (clear).**
- **La méthode (publique) calculerDistancesDepuis est abstraite.**

Nous allons maintenant implémenter concrètement notre premier algorithme de calcul de distance : un parcours en largeur.

6. **A FAIRE : Créer une nouvelle classe "ParcoursLargeur" dans le package "algorithmes" héritant de la classe AlgorithmeCalculDistance.**
7. **A FAIRE : Implémentez la méthode "calculerDistanceDepuis" dont le pseudo-code est le suivant :**

```
//Remise à zéro
initialiser une variable "aTraiter" de type ArrayList<Case>
réinitialiser la HashMap distances

//Initialisation
mettre "depart" dans la liste "aTraiter"
mettre "depart" à distance 0 dans "distances"

//Calcul
tant que "aTraiter" n est pas vide
        "caseEnCours" <— tête de la liste "aTraiter" (élément en
            position 0)
        on enlève la tête de la liste "aTraiter"
        pour tout voisin "v" de "caseEnCours"
                si aucune distance n a déjà été calculée pour "v"
                        si "v" est accessible
                                mettre "v" à une distance 1 de plus que
                                    celle de "caseEnCours"
                                ajouter "v" à la liste "aTraiter"
```

8. **A FAIRE : Créez un nouveau test unitaire pour la classe "Parcours-Largeur" et recopiez le contenu du fichier "ParcoursLargeurTest.java" présent dans le répertoire TP4 du commun.**
9. **A FAIRE Assurez-vous que le test passe.**

4 Déterminer le plus court chemin

La méthode "calculerDistanceDepuis" de la classe "ParcoursLargeur" nous permet d'obtenir la distance séparant la case de départ donnée en paramètre à toutes les autres cases. Afin de pouvoir diriger Abigail, nous avons non seulement besoin de cette distance mais aussi (et surtout) de la suite de mouvements qu'Abigail doit réaliser pour se rendre à la case désirée. Comme vu en cours, nous pouvons déduire ce chemin du résultat des calculs de distance.

Tout d'abord, nous allons rajouter une méthode privée déterminant le mouvement à faire pour passer d'une case à une autre **(en supposant que ces deux cases sont voisines l'une de l'autre)**. Dans les séances précédentes, nous avions codé une méthode similaire dans la classe coordonnée que nous allons pouvoir utiliser.

10. **A FAIRE : Ajouter à la classe "Case" la méthode**

    public TypeMouvement getMouvementPourAller(Case arrivee)

11. **A FAIRE : Implémenter cette méthode en utilisant la méthode de même nom présente dans la classe "Coordonnee" pour qu'elle renvoie le mouvement à faire pour passer de la case en cours à la case "arrivee" (on supposera ces deux cases voisines l'une de l'autre).**

Ensuite, nous allons rajouter une méthode publique déterminant la séquence de mouvements à réaliser pour passer d'une case à une autre (voisines ou non).

12. **A FAIRE : Ajouter la méthode abstraite suivante dans la classe "AlgorithmeCalculDistance" :**

    public abstract ArrayList<TypeMouvement> getChemin(Case arrivee)

13. **A FAIRE : Implémenter cette méthode dans la classe "ParcoursLargeur" pour qu'elle renvoie la séquence de mouvements à réaliser pour se rendre à la case "arrivee" en partant de la case qui a servi de point de départ pour le calcul. Voici le pseudo-code de la méthode :**

```
//Initialisation
ArrayList<TypeMouvement> resultat = new ArrayList<>();
Case caseEnCours = arrivee

//Calcul
Si distance de "caseEnCours" est non null
        Tant que distance de "caseEnCours" > 0
                On trouve "casePrecedente" une case voisine de "
                    caseEnCours" dont la distance est non null et est
                    égale à la distance de "caseEnCours" - 1.
                On utilise la méthode "getMouvementPourAller" pour
                    déterminer le mouvement à faire pour passer de "
                    casePrecedente" à "caseEnCours".
                On met ce mouvement dans la liste "resultat".
                caseEnCours = casePrecedente.

        On retourne la liste "resultat" (en utilisant Collections.
            reverse()).
On renvoie "resultat".
```

14. **A FAIRE : Expliquer pourquoi on est sûr que la boucle "while" présente dans cette méthode terminera toujours.**

De nouveau, pour tester nos fonctions, nous allons utiliser des tests unitaires :

15. **A FAIRE : Copier la fonction testGetChemin présente dans le fichier "ParcoursLargeurCheminIT" sur le commun et l'ajouter à la classe de test existante ParcoursLargeurTest.**
16. **A FAIRE : S'assurer que ce test unitaire passe.**
17. **A FAIRE : Quelles sont les limites de cet algorithme ? Discutez-en avec votre responsable de TP.**
18. **A FAIRE : Déduisez de ces limites le sujet du prochain TP.**
19. **BONUS : Modifiez votre IA pour qu'elle se rende d'elle même sur une case fixée (en dur) dans l'IA.**