MATH EN INFO

TRAVAUX PRATIQUES N° 6

Récolte néerlandaise

1 Présentation du problème et de la solution proposée

Dans les TPs précédents, nous avons mis en place une structure de graphe pour représenter notre carte. Assez logiquement, nous avons choisi d'utiliser les différentes cases de notre carte comme sommets de notre graphe et de relier par une arête deux cases situées l'une à côté de l'autre. Cette structure ne semble pas, à première vue, prendre en compte les différents types de cases et les objets. Cependant, dans un deuxième temps, nous avons décidé d'utiliser un parcours en largeur n'acceptant d'utiliser une arête que si celle-ci relie deux cases "accessibles" (case d'herbe sans objet ou case de terre). Nous avons donc en fait, indirectement, considéré un graphe excluant toutes les cases d'eau, la maison mais aussi les arbres ! Or les cases contenant un arbre ne sont pas réellement inaccessible, Abigail peut couper l'arbre et y accéder.

Une première idée pour permettre à Abigail d'accéder à ces cases, serait de simplement changer notre définition de case "accessible" en élargissant cette notion aux cases avec un arbre. Cette stratégie règle une partie des problèmes mais on constate alors que dans notre nouveau graphe, tous les mouvements ne "coûtent" pas autant à Abigail. En effet, si se déplacer sur une case d'herbe sans arbre ou une case de terre ne nécessite qu'une seule action pour Abigail (bouger), se déplacer sur une case d'herbe contenant un arbre nécessite de réaliser deux tours (deux tours pour couper l'arbre et un pour bouger). Or l'algorithme du parcours en largeur ne permet de calculer le plus court chemin entre deux sommets que si toutes les arêtes du graphe ont le même coût, ce qui n'est plus le cas. Nous allons donc devoir implémenter l'algorithme de Dijkstra.

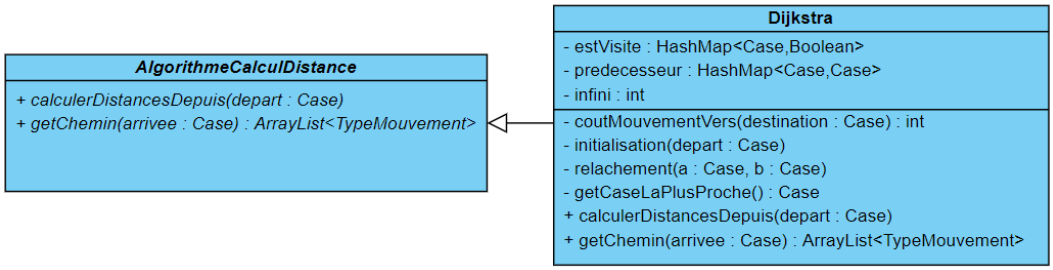
Notons au passage que dans notre cas, il n'est pas pertinent d'implémenter $A\star$. En effet, la plupart du temps, nous aurons besoin de calculer la distance d'Abigail à toutes les cases de la carte (afin de permettre à notre IA de choisir sa destination). Or dans cette situation, Dijkstra et $A\star$ sont strictement équivalents.

2 L'algorithme Dijkstra

1. **A FAIRE : Ajouter la méthode suivante à la classe "Carte" :**

   public Collection<Case> getCases()

   qui renvoie la liste des *valeurs* stockées dans la HashMap cases.
2. **A FAIRE : Ajouter une nouvelle classe "Dijkstra" dans le package "algorithmes" en respectant l'UML suivant :**



On laissera les méthodes vides ou renvoyant 0 ou null (si la méthode doit renvoyer quelque chose). Aucune erreur ne doit être détectée dans cette classe. (attention à la visibilité des différentes méthodes !)

Nous allons commencer par implémenter le constructeur de notre classe. Ce constructeur initialise les différents attributs. Comme vu en CM, l'algorithme de Dijkstra manipule par moments des valeurs infinies, ce qui n'est pas possible en informatique (du moins avec des integers). Nous allons donc utiliser une valeur finie pour "représenter" l'infini. La seule contrainte que nous impose l'algorithme sur cette valeur est qu'elle soit toujours strictement plus grande que toutes les distances que nous pourrions rencontrer. Pour s'assurer de cela, nous allons utiliser le fait que, dans un graphe, le plus court chemin d'un sommet A à un sommet B passe, dans le pire des cas, exactement une et une seule fois par chaque arête. Ce résultat nous permet de poser :

infini = 1+ (nombre d'arêtes du graphe * coût maximum d'une arrête)

Or dans notre graphe :

- nombre d'arête < 4*getCarte().getTaille()*getCarte().getTaille()
- coût maximum = 4

Ce qui nous donne donc :

infini = 1+16*getCarte().getTaille()*getCarte().getTaille()

3. **A FAIRE : Implémenter le constructeur de notre classe.**

Comme nous l'avons fait remarquer dans la première partie, les différents "mouvements" n'ont pas tous le même coût. Cependant, on peut constater que le coût d'un déplacement ne dépend que de la case d'arrivée et plus précisément de la présence d'un arbre ou non sur la case d'arrivée. Vu que les arbres sont les seuls objets récoltables du jeu, nous pourrions les traiter de façon spécifique. Cependant, nous allons adopter une approche plus générique qui permettrait de gérer différents types d'objets récoltables.

4. **A FAIRE : Ajouter une méthode "public boolean estRecoltable()" à la classe "Objet" qui renvoie par défaut false et surchargez la classe ObjetArbre pour qu'elle renvoie true.**
5. **A FAIRE : Ajouter une méthode "public int coutRecolte()" à la classe "Objet" qui renvoie par défaut -1 et surchargez la dans la classe ObjetArbre pour qu'elle renvoie 2.**
6. **A FAIRE : Modifier la classe ObjetArbre pour qu'un arbre ne soit plus bloquant.**
7. **A FAIRE : Dans la classe Dijkstra, implémenter la méthode "coutMouvementVers" pour qu'elle renvoie le nombre d'actions nécessaires pour se rendre sur la case "destination" depuis une case voisine (récolte + déplacement). La méthode renverra "infini" si la case destination est inaccessible.**

Nous pouvons maintenant nous attaquer aux méthodes secondaires de notre algorithme.

8. **A FAIRE : Dans la classe Carte, ajouter une méthode "public Collection<Case> getCases()" qui renvoie la liste des cases de la carte (à récupérer dans la hashmap).**
9. **A FAIRE : En utilisant le code vu en CM, implémenter les méthodes "initialisation" et "relachement". Dans la méthode de relachement, on supposera que les deux cases sont voisines l'une de l'autre.**

A chaque étape, l'algorithme de Dijkstra choisit le sommet non encore visité ayant la plus petite distance déjà calculée. C'est l'objectif de la prochaine méthode.

10. **A FAIRE : Implémenter la méthode "getCaseLaPlusProche" en suivant le pseudo-code suivant :**

```
"distanceMin" <- "infini"
"res" <- null
pour toutes les cases "C" de la carte
  si "C" n a pas déjà été visitée et si "distance(C)" < "distanceMin"
    "distanceMin" <- "distance(C)"
    "res" <- "C"
renvoyer "res"
```

Nous disposons maintenant de tous les éléments pour pouvoir implémenter la méthode principale de l'algorithme. Le code proposé ici est une variante de celui vu en CM utilisant "getCaseLaPlusProche" pour détecter quand toutes les cases ont été visitées.

11. **A FAIRE : Implémenter la méthode "calculerDistancesDepuis" en suivant le pseudo-code suivant :**

```
initialisation
"caseLaPlusProche" <- la case non visitée de distance minimale
tant que "caseLaPlusProche" est non null
  caseLaPlusProche est maintenant visitée
  on effectue le relâchement entre "caseLaPlusProche" et chacun de
     ses voisins.
  "caseLaPlusProche" <- la case non visitée de distance minimale
```

Il ne nous reste plus qu'à implémenter la méthode "getChemin" pour qu'elle renvoie le plus court chemin de la case de départ à celle donnée en paramètre.

12. **A FAIRE : Implémenter la méthode "getChemin" pour qu'elle détermine le plus court chemin de la case départ à celle donnée en paramètre en utilisant les données stockées dans la HashMap "predecesseur".**

3 Il est temps de tester

L'algorithme de Dijkstra est un algorithme assez sensible à coder. Afin de nous assurer de sa bonne implémentation, nous allons réaliser quelques tests.

13. **A FAIRE : Créer un nouveau test unitaire pour la classe "Dijkstra" et recopier le contenu du fichier "DijkstraTest.java" présent dans le répertoire TP6 de TEAMS.**
14. **A FAIRE : Assurez-vous que le test passe.**

Maintenant que nous nous sommes assuré que notre algorithme fonctionnait correctement, nous allons pouvoir apprendre à notre IA à l'utiliser.

15. **A FAIRE : Modifier la méthode "seDeplacerEn" du module de décision pour qu'elle utilise l'algorithme de Dijkstra à la place du parcours en largeur.**
16. **A FAIRE : Testez votre IA. Que constatez-vous ?**

Votre IA doit se déplacer... parfois... et se heurter très régulièrement à des arbres, la maison... C'est logique, nous ne lui avons pas réellement appris à récolter les ressources par lesquelles elle veut passer.

17. **A FAIRE : Ajouter un élément "RECOLTE" à l'énumération TypeAction.**
18. **A FAIRE : Créer une nouvelle énumération TypeActionRecolte ne contenant actuellement qu'un seul élément : COUPERARBRE.**
19. **A FAIRE : Créer une nouvelle classe "ActionCouperArbre" héritant de la classe "Action". En s'inspirant de l'action de mouvement, l'implémenter.**
20. **A FAIRE : Dans la fabrique des actions, ajouter une nouvelle méthode "public static Action creerActionRecolte(TypeActionRecolte type, TypeMouvement direction)" et l'implémenter.**
21. **A FAIRE : Après avoir compris son code, remplacer votre module de décision par celui présent dans le répertoire TP6 du commun. Tester l'IA.**

Tout ce passe bien jusqu'au premier arbre. Après l'avoir coupé, Abigail se bloque et continue d'essayer de couper l'arbre qui n'existe plus. Nous n'avons en effet pas indiqué à l'IA qu'en le coupant, Abigail faisait "disparaitre" l'arbre.
Pour que notre IA puisse gérer la disparition des arbres, il nous faut modifier la méthode "effectuerAction" du module mémoire. Pour gérer le fait que les arbres nécessitant plusieurs coups de hache pour être coupés, nous pourrions mettre en place un système de "points de vie" (notre structure le permettrait facilement). Cependant, cette notion sera très vite rendue obsolète par certaines contraintes qui apparaitront sous peu. Nous allons donc faire en sorte que le module mémoire considère que la récolte a lieu dès le premier coup de hache. Ce choix simplificateur nous imposera cependant de nous assurer que par la suite notre IA ne change jamais d'idée au milieu d'un procédé de récolte.

22. **A FAIRE : Remplacer le code de la méthode "effectuerAction" du module mémoire par :**

```
if(action.getType() == TypeAction.MOUVEMENT) {
        this.joueur.deplacer(action.getDirection());
}
else if(action.getType() == TypeAction.RECOLTE) {
        if(action.getDirection() != null) {
        Case caseDestination = this.carte.getCase(this.getCaseJoueur().
            getCoordonnee().getVoisin(action.getDirection()));
        caseDestination.setObjet(null);
    }
}
```

23. **A FAIRE : Testez votre IA.**

Comme vous pouvez le constater, votre IA parvient maintenant à couper des arbres, tous les arbres de la carte... de façon complètement non optimisée...

24. **A FAIRE SI VOUS ETES EN AVANCE : Modifier la méthode determinerNouvellesActions du module de décision pour qu'Abigail aille couper l'arbre le plus proche d'elle.**