X5I0040
Projet "Jeu Connexion"

Sidney Falhun Corentin Chédotal

9 Décembre 2016

Table des matières

1 Introduction

Dans le cadre de l'Unité d'Enseignement **X5I0040 "Algorithmique et Structures de Données 3"**nous avons été amené à réaliser le Projet "Jeu Connexion". Il consiste en l'implémentation d'un jeu tout en suivant un cahier des charges particulier. Le jeu consiste en la capture de diverses cases d'un plateau carré par deux joueurs dans le but d'intégrer dans ses zones de contrôle le plus de cases étoilées possible. Nous devions programmer notre implémentation du jeu en Java et en utilisant les structures de données vues en cours de notre choix. Enfin concernant l'interface graphique nous n'avions pas de consignes particulières mais il était précisé de ne pas perdre trop de temps dessus, celle-ci n'étant pas notée.

Dans ce rapport, comme demandé, nous expliciterons le fonctionnement de notre vision du projet. En commençant par sa compilation et son exécution, puis en montrant son implémentation et enfin en donnant des jeux de données types permettant de tester le bon fonctionnement du programme.

2 Exécution et compilation

Ce projet utilise un fichier `Makefile` pour faciliter la compilation pour l'Utilisateur. Nous allons donc expliciter ci-après les commandes importantes de ce fichier.

Pour compiler et exécuter le projet l'Utilisateur n'a qu'à faire `make`. Le `Makefile` s'occupera du reste. Si l'Utilisateur souhaite uniquement compiler le projet il a à sa disposition la commande `make compile`. Enfin, sont implémentées dans le fichier les commandes `make clean` et `make mrproper` afin de supprimer les fichiers intermédiaires et tout les fichiers résultant de la compilation respectivement.

3 Implémentation

Dans cette section nous allons rentrer dans le détail de l'implémentation du projet en expliquant notre choix de structures de données parmi toutes celles vues en cours. Puis on indiquera en détail comment nous avons répondu aux questions concernant les méthodes à implémenter obligatoirement.

3.1 Structures de données choisies

La structure de données vues en cours que nous avons décidé d'utiliser est la *Classe-Union*. Initialement on a donc une Classe (au sens de la *Classe-Union* pas de la programmation orientée objet) par case du plateau de jeu. Puis par les utilisations d'Union (par la méthode du même nom dans `Plateau.java`) ce sont les Composantes qui vont être les Classes. De plus afin d'optimiser les appels nous avons implémenter une compression des chemins.

L'implémentation exacte choisie a été celle par tableau de pères. En effet dans ce cas précis il faut remonter souvent de fils aux pères afin de savoir à quelle Composante les Cases appartiennent or l'utilisation d'un arbre est optimisée pour le sens inverse. Pour descendre du père au fils. Nous avons donc fait ce choix et nous avons choisi de faire un tableau de pères à deux dimensions. Le Plateau étant à deux dimensions il paraissait logique de stocker ainsi les données concernant les pères des Cases dans une représentation tabulaire précise du Plateau.

Ainsi on peut facilement ajouter des cases au fur et à mesure du déroulement du jeu au différentes Composantes et fusionner les Composantes entre elles (si elles sont de la bonne couleur bien sur).

3.2 Implémentation des méthodes obligatoires

Le cahier des charges du projet requérait l'implémentation de dix méthodes spécifiques. Dans cette partie nous allons montrer comme demandé notre façon de les intégrer au programme.

3.2.1 `colorerCase()`

Entrées : String col
Sorties : Un booléen indiquant le succès ou non de l'opération
Résultat : La case est coloriée de la couleur donnée par col
si *col = "blanc"* **alors**
  Case.col ⟵ col;
  retourner *vrai*;
sinon
  afficher "Cette case est déjà en couleur.";
  retourner *faux*;
fin

Algorithme 1 : La méthode `colorerCase()`

La méthode `colorerCase()` prend une chaîne de caractère représentant une couleur et l'appliquera à la Case à condition que la Case soit blanche. En effet on vérifie si la Case a déja recu une couleur. Si oui il n'est pas possible de la recolorer (empêchant ainsi un joueur de prendre une Case à son adversaire). Enfin la méthode retourne un booléen indiquant la réussite ou non du coloriage à des fin de contrôles.

L'algorithme proposé ici est correct puisqu'on ne fait qu'ajouter une couleur à la Case avec la vérification pour empêcher le "vol" de Cases. De plus dans le pire des cas une Case aura toujours une couleur puisqu'elles sont construites avec la couleur blanche affectée de base.

La complexité de cet algorithme est en $O(1)$.

Il n'est pas possible de faire plus efficace qu'$O(1)$.

3.2.2 `afficheComposante()`

```
Entrées : int x, int y, String col
Résultat : Affiche la Composante formée par case1 et case2
entier limite = Plateau.longueur - 1;
pour i allant de 0 à limite faire
    pour j allant de 0 à limite faire
        si existeCheminCases(tableauPeres[x][y],tableauPeres[i][j], col) alors
            afficher "La case [" + x + ", " + y + "] contient la composante suivante :";
            afficher i + " :" + j;
        fin
    fin
fin
```

Algorithme 2 : La méthode `afficheComposante()`

La méthode `afficheComposante()` va chercher dans tout le tableau de pères les Cases ayant le même père que la Case localisée aux coordonnées reçues en entrées. Elle va ensuite les afficher au fur et à mesure qu'elle en trouve.

Cet algorithme est correct car la définition même de la Composante dans notre cas de *Classe-Union* est qu'il s'agit du groupe de Cases partageant le même père. Ainsi afficher la Composante d'une Case consiste bien en l'affichage de toutes les Cases partageant le même père que la Case donnée en entrée (par le biais de ses coordonnées).

Cette méthode a une complexité temporelle de l'ordre de $O(limite^2)$. En effet la présence d'une boucle imbriquée dans une autre donne cette complexité qui est liée à la taille du tableau à parcourir et donc aussi à la taille du Plateau.

Il ne nous apparaît pas possible d'être beaucoup plus efficace que ceci.

3.2.3 `existeCheminCases()`

```
Entrées : Case case1, Case case2
Sorties : Un booléen indiquant si case1 et case2 ont le même père
retourner tableauPeres[case1.x][case1.y] = tableauPeres[case2.x][case2.y]
```

Algorithme 3 : La méthode `rechercheMemePere()` utilisée par `existeCheminCases()`

```
Entrées : Case case1, Case case2, String col
Sorties : Un booléen indiquant si case1 et case2 bien un chemin de la même couleur les
          reliant
retourner rechercheMemePere(case1, case2) ET case1.col = case2.col
```

Algorithme 4 : La méthode `existeCheminCases()` en elle même

La méthode `existeCheminCases()` recherche d'abord si les deux Case ont le même père dans le tableau de pères. On vérifie donc leur appartenance à la même Composante. Se faisant on en profite pour appliquer la compression de chemin afin d'optimiser les futures recherches dans celui-ci. Une fois que l'on a confirmation que les deux Cases ont le même père il suffit de s'assurer que celles-ci sont bien de la même couleur. Si c'est le cas alors il existe bien un chemin d'une couleur entre les deux Cases.

L'appartenance à une même Composante est donnée par le tableau de pères. Cette appartenance montre qu'il y a bien un chemin reliant nos deux Cases. La vérification de la couleur portée par

celles-ci est une dernière étape afin de s'assurer que l'on respecte bien la partie "de la même couleur" de la consigne. Ainsi l'algorithme est donc correct.

La complexité de celui-ci est en $O(1)$. En effet `existeCheminCases()` fait uniquement appel à `rechercheMemePere()` qui est lui même en $O(1)$.

Il n'est pas possible d'être plus efficace qu'$O(1)$.

3.2.4 `relierCasesMin()`

Malheureusement nous n'avons pas réussi à implémenter une méthode fonctionnelle pour `relierCasesMin()` dans le temps imparti. Cependant le code de celle-ci est toujours présent dans le fichier `Plateau.java`. En effet nous avons tenter d'implémenter une version de l'algorithme de DIJKSTRA mais cette implémentation ne fonctionne pas. Elle ne crashe pas et ne cause pas de problèmes à la compilation donc il semblerait que cela soit notre approche algorithmique qui soit incorrecte. Ainsi celle-ci ne risquant pas l'intégrité fonctionnelle du programme nous l'avons conservée dans le code afin de montrer notre tentative de résolution de cette partie du cahier des charges du sujet.

3.2.5 `getNbEtoiles()`

Entrées : entier x, entier y, String col
Sorties : Le nombre d'étoiles contenues dans la Composante de la Case aux coordonnées [x][y]
retourner *tableauPeres[x][y].nbEtoile()*

Algorithme 5 : La méthode `getNbEtoiles()`

La méthode `getNbEtoiles()` va chercher dans le tableau de pères le pères de la Case au coordonnées [x][y] et va retourner la variable du nombre d'étoiles que le père contient. En effet à chaque Union c'est le père qui récupère les étoiles de ses nouveaux fils.

La correction de cette méthode est assez triviale. En effet on ne fait qu'appel à la variable donnant le nombre d'étoiles que contient une Composante. Celle-ci se trouve dans le père de chaque Composante et c'est donc pourquoi on va d'abord chercher le père de la Case aux coordonnées données en entrée.

La complexité de cette méthode est en $O(1)$. En effet on ne fait qu'accéder à un emplacement spécifique dans un tableau et on retourne simplement une variable.

Il n'est pas possible de faire plus efficace qu'$O(1)$.

3.2.6 `afficheScores()`

```
Entrées : String col
Résultat : Affiche le score du joueur à la couleur col
si col = "bleu" alors
    afficher "Vous avez réussi à connecter " + Plateau.scoreJ2 + " étoiles.";
sinon
    si col = "rouge" alors
        afficher ""Vous avez réussi à connecter " + Plateau.scoreJ1 + " étoiles.";
    sinon
        afficher "Couleur fausse";
    fin
fin
```

Algorithme 6 : La méthode `afficheScores()`

La méthode `afficheScores()` prend la couleur du joueur dont on veut afficher les scores puis va chercher dans le Plateau la variable correspondante puis va l'afficher.

Une fois de plus cette méthode est assez triviale. En effet elle ne fait que chercher la variable du joueur correspondant à la couleur donnée.

Cette méthode a une complexité temporelle en $O(1)$ puisqu'il ne s'agit que d'un affichage d'une variable une fois un simple test effectué.

Il n'est pas possible de faire plus efficace que $O(1)$ en terme de complexité temporelle.

3.2.7 `relieComposante()`

```
Entrées : entier x, entier y, String col
Sorties : un booléen indiquant si la Case au coordonnées [x][y] relie une ou plusieurs
          Composantes avec la Case d'origine
si getLesVoisins(x,y,col).x = x ET getLesVoisins(x,y,col).y = y alors
    retourner faux;
sinon
    retourner getLesVoisins(x,y,col).col = tableauPeres[x][y].col
fin
```

Algorithme 7 : La méthode `relieComposante()`

La méthode `relieComposante()` regarde initialement que la Case donnée par les coordonnées [x][y] n'est pas déja la Case d'origine. Si c'est le cas alors la méthode retourne forcément faux. Si ce n'est pas le cas il suffit alors de regarder si les voisins de la Case en [x][y] sont de la même couleur que la Composante de la Case [x][y].

L'algorithme est correct car il vérifie d'abord le cas où l'utilisateur donne les coordonnées de la Case d'origine. Ensuite on vérifie bien que les voisins de la Case sont de la même couleur que la Composante de la Case en elle même, condition sine qua non pour que la Case puisse relier une ou plusieurs Composantes.

La méthode a une complexité de $O(1)$ en effet elle ne comprend pas de boucle et le seul appel à une fonction est à `getLesVoisins()` qui elle-même a une complexité temporelle en $O(1)$.

Il n'est pas possible de faire plus efficace qu'$O(1)$.

3.2.8 `joueDeuxHumains()`

On utilise et intialise les variables suivantes dans l'algorithme ci-après :
booléen fin ⟵ faux
booléen result ⟵ faux
entier etoiles, choix
entier i ⟵ 0
entier x ⟵ 0
entier y ⟵ 0
entier x2 ⟵ 0
entier y2 ⟵ 0
String couleur

```
etoiles ⟵ initialiser();
tant que lnotfin faire
    si i mod 2 = 0 alors
        Plateau.couleur ⟵ "bleu";
    sinon
        Plateau.couleur ⟵ "rouge";
    fin
    afficher affichage du menu;
    lire choix;
    afficher tableau graphique du jeu;
    suivant choix faire
        cas où 1 faire
            cf Cas 1
        fin
        cas où 2 faire
            cf Cas 2
        fin
        cas où 3 faire
            cf Cas 3
        fin
        cas où 4 faire
            cf Cas 4
        fin
        cas où 5 faire
            cf Cas 5
        fin
        cas où 6 faire
            fin ⟵ vrai;
        fin
    fin
    si scoreJ2 = etoiles alors
        afficher "Le joueur bleu a gagné !";
        fin ⟵ vrai;
    sinon
        si scoreJ1 = etoiles alors
            afficher "Le joueur rouge a gagné !";
            fin ⟵ vrai;
        fin
    fin
fin
```

Algorithme 8 : La méthode `joueDeuxHumains()`

afficher demande d'ajout de case;
lire x, y;
result ⟵ tableauPeres[x][y].colorerCase(couleur);
si *getNbEtoiles(x,y,couleur) < getNbEtoiles(getLesVoisins(x,y,couleur).x, getLesVoisins(x,y,couleur).y,couleur)* **alors**
  union(getLesVoisins(x,y,couleur).x,getLesVoisins(x,y,couleur).y,x,y);
sinon
  union(x,y,getLesVoisins(x,y,couleur).x, getLesVoisins(x,y,couleur).y);
fin
preparerScores(x,y,couleur);
afficheScores(couleur);
nombresEtoiles(x,y,couleur);
i ⟵ i + 1;

Algorithme 9 : Le Cas 1 de la méthode `joueDeuxHumains()`

afficher demande de quelle Composante afficher;
lire x,y;
Compression de Chemin;
afficheComposante(x,y,couleur);

Algorithme 10 : Le Cas 2 de la méthode `joueDeuxHumains()`

afficher demande de case à tester pour relier une Composante;
lire x;
Compression de Chemin;
si *lnotrelieComposantes(x,y,couleur)* **alors**
  afficher "Cette case ne relie aucune composante";
sinon
  afficher "Cette case relie une ou plusieurs composantes";
fin

Algorithme 11 : Le Cas 3 de la méthode `joueDeuxHumains()`

afficher demande des deux cases dont on doit tester si un chemin existe entre les deux;
lire x,y,x2,y2;
Compression de Chemin;
si *existeCheminCases(tableauPeres[x][y],tableauPeres[x2][y2],couleur)* **alors**
  afficher "Il existe un chemin entre les deux cases.";
sinon
  afficher "Il n'existe pas de chemin entre les deux cases.";
fin

Algorithme 12 : Le Cas 4 de la méthode `joueDeuxHumains()`

afficher demande des deux cases dont on veut savoir la distance minimum les séparant;
lire x,y,x2,y2;
Compression de Chemin **afficher** relierCasesMin(x,y,x2,y2,couleur);

Algorithme 13 : Le Cas 5 de la méthode `joueDeuxHumains()`

3.3 *(optionnel)* Méthodes d'évaluation

Malheureusement, la méthode `relierCasesMin()` ayant pris beaucoup plus de temps que prévu à implémenter il ne nous a pas été possible de nous pencher sur les méthodes dont l'ajout était optionnel.

4 Jeux de données

La position initiale des cases étoilées était choisie aléatoirement il ne nous est pas possible de donner un jeu de données exact qui reproduirait forcément le même scénario de test. Cependant nous pouvons conseiller un départ avec deux Cases étoilées par joueurs. A la suite de ce choix, le jeu se lance avec un menu, il ne reste plus qu'à jouer pour afficher le scénario (agencement des Cases étoilées) et le tester.

5 Conclusion

Ce projet fut assez intéressant car malgré la petite envergure de celui-ci le cahier des charges était très précis et le résultat final, avoir un jeu qui marche et est immédiatement utilisable, était très gratifiant. De plus cela permettait de réflechir directement sur les différentes structures de données vues en cours et de mettre en place une structure de *Classe-Union* dans un cas concret, structure qui nous était totalement étrangère avant cette Unité d'Enseignement.

En ce qui concerne les quelques difficultés rencontrées au cours de la réalisation de ce projet on notera surtout la mise en place des tests afin de récupérer les voisins d'une Case. En effet nous avons difficilement trouvé une solution qui marche mais qui est surement optimisable et qui pourrait être rendue plus lisible. De plus la méthode `relierCasesMin()` a été très difficile à mettre en place et nous avons eu de réelles difficultés à implémenter une version un tant soit peu fonctionnelle etn'avons pas réussi à obtenir celle-ci avant de devoir rendre le projet.