DEMANGE Alessi

NICOL Benoît

S3C

**Algorithmique Avancée – Solo Noble**

**SOMMAIRE :**

**PAGE 2 :** Algorithme de conception

**PAGE 3-4 :** Description générale du code

**PAGE 5 :** Conception du projet et problèmes rencontrés

**PAGE 6 :** Tests de rapidité d’exécution et d’efficacité de mouvements

**1 – Algorithme de conception**

algorithme :

fonction resoudreSoloNoble(tab InOut : tableau chaîne[0..li-1, 0..co-1], li : entier, co : entier, billes : entier) : booléen

début

réussi <- faux

si billes = 1 alors

valide <- vrai

***On ecrit toutes les solutions***

sinon

i <- 0

j <- 0

tant que non réussi et i < li faire

j <- 0

tant que non réussi et j < co faire

si tab[i][j] = "." alors

***On vérifie si un déplacement valide est possible***

***Si oui, on l'effectue***

tant que non réussi et ***On n’est pas encore arrivés à la fin des déplacements possibles*** faire

réussi <- resoudreSoloNoble(billes - 1)

si non réussi alors

***On effectue un retour arrière***

***On vérifie si un nouveau déplacement valide est possible pour ce même trou***

***Si oui, on l'effectue***

fsi

ftant

fsi

j <- j+1

ftant

i <- i+1

ftant

si réussi alors

***On enregistre la solution***

fsi

retourne réussi

fsi

fin

lexique :

- li : entier, le nombre de lignes de la grille

- co : entier, le nombre de colonnes de la grille

- tab InOut : tableau chaîne[0..li-1, 0..co-1], la grille du Solo Noble

- billes : entier, le nombre de billes contenues actuellement dans la grille

- réussi : booléen, vrai si on a résolu le Solo Noble, faux sinon

- i : entier, indice donnant la ligne actuelle du tableau

- j : entier, indice donnant la colonne actuelle du tableau

**2 – Description générale du code**

**Classe Grille :**

Attributs de la classe Grille :

* nbLignes, nbColonnes : int | Ces attributs correspondent au nombre de lignes et de colonnes de la grille.
* grille : String[][] | Cet attribut correspond à la grille en question, celle ou vont évoluer les billes.
* deplacement : String | Cet attribut correspond au dernier mouvement effectué par la méthode deplacerBille. De plus, il vaut « début » si l’on a pas encore testé de déplacement pour ce trou et « fin » si plus aucun déplacement n’est possible pour ce trou.

Constructeurs de la classe Grille :

* Avec paramètre String : Le constructeur prend un objet de type « String » en paramètre qui correspond au nom du tablier que l’on veut résoudre. Il ouvre le fichier et il convertit les caractères en String dans un tableau à double dimension. De plus, il initialise les autres attributs.
* Sans paramètres : Même chose que pour le constructeur avec attributs mais la grille est initialisée sous la première forme (celle de base du tablier1).

Méthode calculerNombreBilles :

* Permet de calculer le nombre de bille dans la grille.

Méthode deplacementValide :

* Méthode qui renvoie un booléen, si on peut déplacer une bille à des coordonnées données et à une direction donnée alors on renvoie vrai sinon on renvoie faux .

Méthode deplacerBille :

* Cette méthode possède deux attributs i et j qui correspondent aux coordonnées du trou trouvé.
* Elle possède une variable « deplacementEffectue » qui vaut vrai si on a déjà effectué un déplacement lors de cet appel.
* De plus, cette méthode vérifie si un déplacement est valide et l’effectue ensuite, et ce dans un ordre précis.

Méthode retourArriere :

* Méthode qui effectue le mouvement inverse (BackTracking) du dernier mouvement effectué.

+ Getter + Setter

**Classe SoloNoble :**

Attributs de la classe SoloNoble :

* tablier : Grille | Cet attribut correspond à l’objet Grille.
* solutions : HashMap<Integer, String[][]> | Cet attribut correspond au stockage des solutions et permet ensuite d’afficher le tout, ainsi que le nombre de déplacements et le nombre d’appels à la méthode « resoudreSoloNoble ».
* nombreDeplacements : entier | Cet attribut correspond au nombre de déplacement total lors de la résolution du Solo Noble.
* nombreAppelsResoudreSoloNoble : entier | Cet attribut correspond au nombre de fois où la méthode « resoudreSoloNoble » a été appelée.

Constructeurs de la classe SoloNoble :

* Avec paramètre String : Le constructeur permet d’initialiser l’attribut grille avec le constructeur Grille(String) où ce paramètre correspond au nom du fichier à charger, contenant la grille. De plus, on affiche un message de départ et
* Sans paramètres : Même chose mais en utilisant le constructeur Grille() créant par défaut la grille du tablier 1.

Méthode resoudreSoloNoble :

* Méthode de résolution qui utilise la méthode « deplacerBilles ». Elle permet, en utilisant le « BackTracking », de trouver la solution parfaite pour résoudre le tablier qui est inscrit.

Méthode ecrireToutesSolutions :

* Méthode qui affiche toutes les solutions les unes après les autres et qui affiche également le nombre de déplacements et le nombre d’appels à la méthode « resoudreSoloNoble ».

Méthode Main :

* Méthode qui nous permet d’exécuter la méthode de résolution, d’afficher le nombre de billes et le temps de résolution du Solo Noble.

+ Guetter

**3 –** **Conception du projet et problèmes rencontrés**

Au commencement du projet Solo Noble, il a fallu préparer un algorithme papier à l’aide du « BackTracking » pour pouvoir résoudre un tablier donné. Nous avons fait, tous les deux, un algorithme différent pour pouvoir mettre en commun notre vision d’un algorithme permettant de résoudre le problème. Ce choix nous a beaucoup aidé car nous avons obtenu beaucoup d’éléments pour commencer notre algorithme java.

La programmation en Java est devenue plus compliquée que l’algorithme papier. Notre plus grand problème a été le bon fonctionnement du retour arrière et l’affichage des grilles.

En ce qui concerne le retour arrière, nous avons décidé de travailler avec une boucle tant que qui, lorsque l’on a trouvé un trou aux coordonnées (i, j) de la table, effectue un déplacement tant que cela est possible pour ce trou ou que l’on a pas résolu le Solo Noble (« !grilleValide). On appelle avant cela la méthode « deplacerBille » qui va modifier l’attribut « tablier » du Solo Noble et donc l’attribut « deplacement » de la classe Grille. Si cet attribut vaut « fin », alors on ne rentre même pas dans la boucle tant que. Sinon, on incrémente le nombre de déplacements, on récupère la valeur actuelle de l’attribut « deplacement » et on actualise la valeur de la variable « grilleValide » avec l’appel récursif à la méthode « resoudreSoloNoble(**billes - 1**) ». Si la grille n’est pas valide, alors actualise l’attribut « deplacement » à la valeur que l’on avait enregistrée, on effectue le retour arrière, on incrément le nombre de déplacements et on effectue de nouveau un appel à la méthode « deplacerBille(i, j) ». Si cette méthode actualise l’attribut « deplacement » à « fin », alors on sortira de la boucle.

Le premier de nos principaux problèmes a été de gérer effectivement ce retour arrière. Nous avons dû changer plusieurs fois de conception pour l’implémentation de la méthode de résolution avant d’opter pour celle-ci. Nous avons considérer que gérer le déplacement et le retour arrière autour de l’attribut « deplacement » de la classe Grille constituait un choix judicieux.

Nous avons également dû faire face à un autre problème détectable qu’avec des ajouts d’affichage (avec la méthode « System.out.println() ») au bout de nombreux essais. C’est la cause principale de notre retard et de notre demande de délai.

Il s’est avéré que nous ne pouvions pas stocker directement la grille dans notre table « solutions » puisque, la grille changeant constamment lors de l’exécution de la fonction, nous obtenions à la fin une table contenant un nombre de grilles égal au nombre de billes initiales, mais elles étaient toutes identiques à la grille finale à une seule bille.

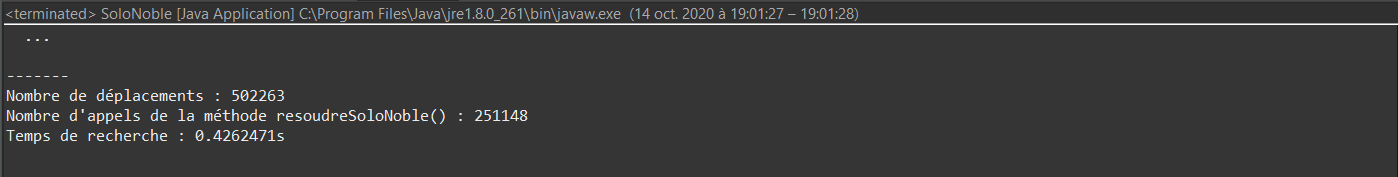
Afin de contourner ce problème, nous avons décidé de créer une grille temporaire dans laquelle nous avons stocké, caractère par caractère, la grille actuelle. Nous avons pu ainsi mettre la grille temporaire dans la table « solutions » et obtenir les résultats attendus.

Nous avons stocké nos solutions dans une variable de type HashMap pour pouvoir afficher nos solutions les unes après les autres.

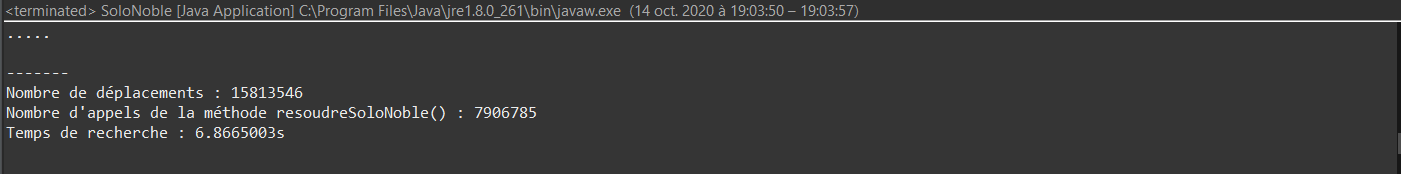
**4 - Tests de rapidité d’exécution et d’efficacité de mouvements:**

Pour le tablier1.txt, on a testé de nombreuses combinaisons de déplacements et seule une poignée nous permet d’obtenir une solution (en moins d’une seconde), le reste ne donnant pas de solutions en plus d’une minute... Cependant nous ne savons pas si les autres combinaisons ne permettent pas de trouver de solutions (ce qui serait étrange) ou sont seulement extrêmement inefficaces. Par ailleurs, nous avons remarqué que pour les combinaisons acceptables, la dernière bille finit toujours au milieu de la grille !

La meilleure combinaison que nous ayons trouvée pour tablier1.txt est « 1.haut 2.gauche 3.droite 4.bas » :



Pour le tablier2.txt, nous avons testé certaines combinaisons et à priori toutes les combinaisons sont fonctionnelles. Cependant, il s’est avéré que certaines combinaisons étaient bien plus efficaces que d’autres. Par exemple, avec la meilleure combinaison pour le tablier1.txt que nous ayons trouvée (« 1.haut 2.gauche 3.droite 4.bas » ), le temps pour trouver une solution est de 9 secondes. Cependant nous avons remarqué que la combinaison (« 1.droite 2.bas 3.haut 4.gauche » ) était plus efficace que d’autres. Voici une exécution que nous avons réalisé pour le tablier2.txt :



On remarque que sa durée de recherche est inférieure à celle pour le tablier1.txt mais elle effectue plus de déplacements… Peut-être y en a-t-il une meilleure ?

Enfin, pour le tablier3.txt, nous avons testé quelques combinaisons seulement et elles fonctionnent à priori toutes et sont également très rapides (moins de 0,1 seconde). La meilleure que nous ayons trouvée est la suivante :

