

Projet java : Le jeu du plombier

L'objectif de ce projet est de réaliser en java une application répondant à un cahier des charges défini. Nous allons aborder point par point ces directives ainsi que les méthodes choisies pour les respecter.

Dans un premier temps le programme doit permettre de résoudre le problème. En l'occurrence il s'agit d'un plateau de 8 cases par 8 dans lesquelles nous pouvons placer une unique source, ainsi que des tuyaux directs et coudés. Le programme doit pouvoir prendre en compte de multiples tuyaux placés initialement et trouver le ou les chemins permettant de tous les alimenter tout en passant par un maximum de cases. Ainsi il a été fait le choix que cette résolution serait effectuée « back-end » avec un affichage en console pour la conception et le débogage dans un premier temps.

Dans un second temps, il est nécessaire de proposer une interface abordable, ergonomique et lisible. J'ai donc choisi de proposer une fenêtre composée d'un tableau de boutons ainsi que d'une barre d'outils qui proposera les différentes fonctions de manipulation du tableau et de l'algorithme.

Dans un but d'ergonomie évident, il sera proposé des fonctions démarrer, arrêter et réinitialiser pour lancer, stopper l'algorithme, et repartir à zéro à la guise de l'utilisateur et sans avoir à redémarrer le programme.

Il a été primordial dans un premier temps, et tout au long du projet de poser sur papier mes idées et de détailler les grandes lignes d'exécution de l'algorithme principal.

J'avais également toujours avec moi une fiche de documentation décrivant le codage numérique des tuyaux, des sorties, ou encore l'utilisation de certaines fonctions. Une documentation similaire est disponible en annexe de ce rapport, ainsi que lorsque cela était possible en commentaires dans le code comme dans la classe tuyau qui a été détaillée.

La conception a été répartie chronologiquement en trois phases :

- La phase préparatoire
- L'algorithme de résolution
- L'interface et les fonctions utilisateur

Dans un premier temps, j'ai donc cherché à concevoir différents outils qui me permettraient plus tard d'avoir les fonctions nécessaires à la résolution du problème. C'est ainsi que j'ai commencé par concevoir une classe Tuyau, qui contient les coordonnées d'un tuyau, son type (6 Possibles), le nombre d'occurrences de test/backtrack effectuées dessus, et sa

direction de sortie (Nord,Sud,Est,Ouest). La documentation d'utilisation de cet élément sera détaillée en annexe, et également en commentaire dans la classe Tuyau.

Pour l'empilement, la classe pile utilisée dans le programme Euler proposé en cours sera utilisée et permettra l'empilement des éléments tuyau les uns à la suite des autres.

En parallèle de cet empilement qui permet de retracer le chemin, il est nécessaire d'avoir un plateau, qui permet de sauvegarder l'état de chaque case et de retourner, connaissant la direction de sortie du précédent tuyau, l'état de la case à venir dans le chemin (Tuyau fixe empruntable, rien, ou case occupée). Pour ce faire nous avons donc la classe plateau, inspirée de la classe échiquier.

La classe plateau dispose aussi de fonctions pour lire une case de coordonnées précisées et pour la remettre à zéro.

Les bordures sont composées de nombres négatifs. $-2x$ le numéro de ligne en haut et en bas et le numéro de colonne à droite et à gauche. Cela permettra de résoudre les problèmes de coordonnées et de dépassement.

Enfin la classe PlombierApp contient le main exécutable du programme. C'est ici que se trouve l'algorithme de résolution et de backtracking.

J'ai d'abord programmé la résolution sans backtrack, qui consiste à empiler à la suite des tuyaux droits jusqu'à rencontrer un obstacle. Ici, les bordures en nombres négatifs identifiables ont été d'une grande aide pour résoudre certains problèmes de résolution de coordonnées dans le plateau.

Une fois ces problèmes résolus, le programme est capable, en lui ayant donné une source et des tuyaux fixes en dur, de compléter avec des tuyaux droits. Pour ce faire, il regarde simplement ce qui se trouve dans la case suivante, en utilisant la fonction crée dans le plateau plus tôt. Si il s'agit d'un vide ou d'un tuyau fixe empruntable, le chemin continue et est enregistré sur le plateau et sur la pile.

Intervient alors la programmation du backtrack. Le principe est simple, lorsque toutes les possibilités $n+1$ d'un tuyau ont été épuisées, on le remplace 2 fois par le type possible suivant.

Imaginons un tuyau sortant au nord, il est possible de le compléter avec un tuyau droit, coudé à droite ou coudé à gauche. Le programme de backtracking va tester toutes ces possibilités en enregistrant à chaque fois le nombre d'occurrences tentées (0 pour droit, 1 pour coudé droite, 2 pour coudé gauche) puis retourner d'un tuyau en arrière.

J'ai fait le choix de programmer l'interface à ce stade de la réalisation du projet. Cela me permettrait de mieux visualiser les tests, l'interface en console ayant ses limites. J'ai cependant gardé l'affichage en console, qui à l'énorme avantage de permettre d'afficher les variables au fur à mesure de la résolution, ce qui a beaucoup contribué au débogage.

L'interface se compose comme précisé précédemment d'un tableau de boutons et d'une

toolbar. Le tableau permettant d'affecter à chaque bouton le même évènement tout en renvoyant les coordonnées du bouton. Lors d'une saisie sur un bouton, une variable lui correspondant prend un état différent. Le programme principal vérifie à chaque répétition de la boucle principale si ces variables changent, c'est-à-dire si l'on a pressé le bouton de réinitialisation, d'arrêt ou de démarrage, ou encore si l'on a pressé un bouton pour changer un tuyau et le prends en compte dans le programme.

Pour donner suite à ces changements, et pour implémenter l'arrêt ainsi que le démarrage du programme, qui initialement se lançait instantanément, il a fallu le revoir un petit peu. Il a donc été nécessaire d'effectuer une boucle infinie à la suite de l'initialisation du programme dans laquelle nous avons deux états, en « veille », c'est-à-dire sans l'algorithme de résolution lancé, ou le programme est simplement en attente d'ordres comme par exemple placement de la source ou encore démarrage, et en « travail », état dans lequel on cherche le meilleur chemin avec les caractéristiques initiales données. Il a également fallu empêcher les erreurs, comme le démarrage du programme alors que la source n'est pas initialisée. C'est également à ce moment que j'ai dû effectuer la transition d'une source et de tuyau fixes codés en dur, et d'une source et de tuyaux fixes variables. Cela s'est accompagné d'un certain nombre de problèmes et de vérifications supplémentaires qui ont été nécessaires pour par exemple remettre à zéro la case après 8 appuis ou encore empêcher le placement de deux sources.

Pour palier à ce genre d'erreurs j'ai implémenté dans l'interface une fonction popup(message) qui permet d'afficher une popup contenant un message donné.

Par la suite, j'ai souhaité intégrer une sauvegarde du meilleur chemin trouvé. J'ai d'abord essayé de dupliquer la Pile Work et de l'assigner à une pile Save, mais cela n'a pas fonctionné car au final la Pile save pointait sur la pile Work dans laquelle le programme travaille et qui donc finit vide lorsque le programme effectue le dernier backtrack, sur la source. J'ai donc finalement répliqué case par case le plateau dans un nouveau plateau appelé platsave. Pour comparer les chemins, j'ai deux critères. Soit le nombre de tuyau initiaux parcourus est supérieur, auquel cas le chemin est plus intéressant, soit il traverse le même nombre de tuyaux initiaux mais aussi rempli plus de cases, auquel cas il est également meilleur. Dans ces deux cas on réplique le plateau dans platsave.

Lorsque le backtrack final est exécuté, ou que tous les tuyaux initiaux et toutes les cases sont remplies, on affiche le meilleur chemin et on stoppe l'exécution de l'algorithme pour repasser à l'état initial, en attente d'une réinitialisation et de la saisie éventuelle d'une nouvelle source.

Finalement, il a été nécessaire de tester le programme. Pour cela dans un premier temps j'ai veillé au passage de l'état veille à l'état travail, en vérifiant bien qu'il est possible de réinitialiser dans les deux cas. Il y a eu des problèmes avec la réinitialisation de la source dont il a fallu restaurer les valeurs initiales (négatives) attestant de sa désaffectation.

Pour la résolution, le programme est beaucoup plus rapide lorsqu'il n'y a pas de tuyaux initiaux, il va prendre quelques secondes. En fonction du nombre de tuyaux initiaux et de leur position, et s'il n'est pas possible de remplir tout le plateau en passant par tous les tuyaux initiaux, et donc si il n'existe pas de solution parfaite, l'algorithme va devoir exécuter toutes les possibilités, là ou il n'aurait du en temps normal trouver qu'une seule des solutions idéales possibles. Dans le cas ou tous les cas sont testés, cela prend approximativement 40 à 50 minutes.

La principale limite de ce projet est sa manière de calculer le meilleur chemin, par tests successifs. Il gagnerait énormément en rapidité si le programme était capable de lui-même de déterminer les tests « idiots » qui sont évitables, et en omettant de les calculer il gagnerait un temps précieux. Il serait également possible d'imaginer complexifier le programme en prenant par exemple en compte la pression de l'eau dans le tuyau et de considérer qu'à partir d'un certain moment, les tuyaux ne transmettent plus l'eau. Il faudrait alors déterminer le chemin le plus court passant par le plus de cases. Ou encore il serait possible d'imaginer des tuyaux à double sortie. On pourrait aussi imaginer des sources multiples.

Pour conclure, ce projet m'a aidé à comprendre énormément de subtilités de java et de la programmation objet, et sur la gestion back-end/front-end différée. Ainsi la notion de buffer de mémoire pour enregistrer et faire passer l'information d'une saisie utilisateur dans un main à a part qui était nouvelle pour moi a pu être bien assimilée.

La conception bien plus complexe que lors du précédent projet de GIF, m'a imposé une plus longue période de conception sur le papier et de réflexion sur le fonctionnement même de l'algorithme.








Cependant, sur le coté purement technique, j'ai trouvé le projet exigeant, challengeant. Le fait que le main exécutable soit composé d'un seul gros algorithme à eu pour conséquences une certaine frustration lorsqu'un ajout de fonctionnalités au programme venait perturber le reste du fonctionnement. Le projet m'a également pris énormément de temps et l'échéance était un peu juste. Il n'empêche que je suis satisfait de mon rendu même s'il aurait mérité quelques approfondissements. J'ai apprécié la liberté qui nous a été donnée quant à l'interface.

Pour finir, je pense que ce projet m'a été bénéfique et que sa difficulté a été formatrice. J'ai le sentiment d'avoir progressé, non seulement dans mes compétences en java, mais également plus globalement dans la conception d'algorithmes.

DOCUMENTATION

Grille de traduction de la variable « type tuyau » de la console et des fonctions :

Tableau 1

| Icone | Type |
|---|----------------------|
|  | 0 : source |
|  | 1 : droit horizontal |
|  | 2 : droit vertical |
|  | 3 : coudé ouest-sud |
|  | 4 : coudé sud-est |
|  | 5 : coudé ouest-nord |
|  | 6 : coudé nord-est |












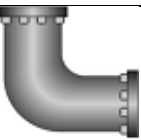
Grille de traduction de la variable « output »

Tableau 2

| Sortie | Type |
|--------|------|
| Nord | 1 |
| Est | 2 |
| Sud | 3 |
| Ouest | 4 |

Grille du type tuyau suivant à tester lors du backtrack en fonction de l'itération de backtrack et de l'output :

Tableau 3

| Output \ Itérations (typetestes) | 0 | 1 | 2 |
|----------------------------------|---|--|---|
| 1 : Nord |  |  |  |
| 2 : Est |  |  |  |
| 3 : Sud |  |  |  |
| 4 : Ouest |  |  |  |

Fonction `Tuyau(int i, int j, int output, int tests, int type)` :

Avec i, j les coordonnées du tuyau, outuput sa direction de sortie selon le tableau 3, tests le nombre d'itérations de backtrack, et type le type du tuyau selon le tableau 1.