PROJET JAVA *Colleen Callonego, Gabriel Couroux, Virgil Ribeyre*

Algorithme génétique Multi thread pour problème du voyageur de commerce

I. INTRODUCTION

Nous avons essayé de résoudre le problème du voyageur de commerce. Ce dernier est un problème qui vise à optimiser la distance d'un chemin. Le chemin en question doit être donc le plus petit possible, avec deux contraintes, passé par toutes les villes mise à sa disposition et ne pas passer deux fois pas la même ville, excepté la ville de départ et de retour qui peuvent être les même.

Version de java utilisée: Java 8

II. VILLES/DISTANCES

A. STRUCTURE FICHIER

Nous avons choisi d'utiliser un fichier JSON car il nous semblait être le plus approprié pour stocker les informations des villes et des distances. Voilà un exemple de la structure de notre fichier JSON pour une ville :

```
"Rennes": {
      "Distances": {
         "Rennes": "0.0",
         "Reims": "483.0",
         "Brest": "242.0",
         "Le Havre": "279.0",
         "Toulon": "1110.0",
         "Grenoble": "846.0",
         "Dijon": "617.0",
         "Angers": "129.0",
         "Nimes": "983.0",
         "Nancy": "644.0"
      }
},
```

Une ville doit donc contenir les distances qui la sépare des autres villes et d'elle-même. Mais les distances déjà entrées plus haut n'ont pas besoin d'être répété (*voir B. CRÉATION MATRICE DES DISTANCES*). Ainsi pour continuer d'illustrer la dernière ville du fichier ne contiendra que la distance qui la sépare d'elle même, i.e. 0 car toutes les autres distances ont déjà été écrites précédemment.

B. CRÉATION MATRICE DES DISTANCES

Pour lire le fichier JSON, qui doit être choisi par l'utilisateur, nous avons utilisé une librairie externe : org.json. Cette dernière nous a aidé pour lire notre fichier JSON et rentré toutes les distances dans une ArrayList d'ArrayList de Double, une sorte de "matrice", avec cette dernière nous pouvons accéder aux distances d'une ville à une autre. Nous avions pensé mettre des -1 dans cette "matrice" pour faire un graphe incomplet, mais nous n'avons pas été jusque là dans notre code. C'est donc uniquement avec un graphe complet que notre programme peut tourner.

Cette “matrice” est accompagnée d’une ArrayList de String, où va se trouver les villes qui se trouve dans la “matrice” dans l’ordre des indices de cette dernière. Ainsi si on veut savoir par exemple la distance de Nancy à Paris, il faut juste chercher l’indice de ces villes dans l’ArrayList pour ensuite prendre ces mêmes indices et trouver la distance entre les deux villes.

Et comme dit précédemment, dans le fichier il n’y a pas deux fois les mêmes informations car lors de la création de la “matrice” on met une distance à la place [i][j] mais aussi à la place [j][i].

III. LA POPULATION

A. CRÉATION DE LA POPULATION INITIALE

Nous avons choisi tout d’abord de représenter les individus de notre population par un Individu qui contient un Chemin et une Fitness. Un Chemin quand à lui est composé d’une ArrayList de villes et d’une longueur.

Les populations sont représentées par une ArrayList qui est modifiée à chaque génération.

Pour créer la première génération nous créons les individus à l’aide de threads. Un individu est créé en faisant un chemin aléatoire entre la ville de départ et la ville de retour. Nous n’avons pas besoin de nous préoccuper des chemins qui n’existent pas, car on considère que notre graphe est complet. La longueur du chemin est calculé au fur et à mesure de l’avancé, à chaque ville nous regardons, et ajoutons à la longueur du Chemin, sa distance par rapport à celle que l’on va rajouter. Ainsi à la fin de la création d’un Individu nous l’ajoutons à l’ArrayList de la Population. Nous continuons jusqu'à ce que la Taille de la Population voulue soit atteinte.

Une fois la Population créée, nous la trions pour avoir les meilleurs Individus en premier.

B. CRÉATION DES AUTRES POPULATIONS

Pour créer les autres populations, nous nous aidons toujours de la population d’avant. Nous avons différentes stratégies pour créer des nouveaux Individus.

Tout d’abord il faut choisir la stratégie de sélection, nous avons le choix entre une stratégie élitiste, garder les K meilleurs Individus, ou bien une stratégie selon des tournois, où on fait K tournois et il faut choisir le nombre d’Individu par tournoi (attention ce nombre ne doit pas être supérieur à (Taille de la Population / K)).

Ensuite il faut choisir la stratégie de remplacement : si l’on souhaite garder la K meilleurs de la génération précédente ou non. Dans tous les cas nous compléterons la nouvelle Population par les “enfants” de ces K meilleurs Individus, et si besoin nous rajouterons des nouveaux Individus.

Nous pouvons aussi choisir une stratégie de recombinaison mais nous n'en proposons d'une seule : la recombinaison simple, on coupe à un endroit aléatoire deux Individus et on les combine ensemble. Nous voulions aussi proposé une recombinaison par enjambement mais nous ne l'avons pas fait. Une fois les enfants créer nous regardons s'il ne manque pas des villes et qu'il 'y ai pas de doublon. Une fois cette vérification faite, un enfant peut muter ou non (échange de position de deux villes), et nous pouvons choisir le taux de mutation.

On peut aussi choisir si l'on souhaite un temps de calcul basé sur un nombre de génération précis, un temps précis (en secondes) ou même avoir un temps illimité et arrêter lorsqu'on le souhaite.

Tous les paramètres peuvent être choisis. Et nous voyons l'évolution du meilleur Individu de chaque Population à l'aide d'un graphique.

IV. MULTITHREADING

A. MISE EN PLACE

Notre programme contient plusieurs threads. Nous en utilisons pour créer la première Population. Nous utilisons quatre threads, pour ce faire nous créons dans chaque threads (Taille de la Population / 4) Individus. Et nous créons quelques individus en plus si il en manque, dans le cas où la Taille de la Population n'est pas un multiple de quatre.

Nous utilisons aussi un thread pour lancer l'algorithme génétique afin qu'il ne bloque pas l'interface graphique.

B. RÉSULTAT

L'utilisation de threads nous aide à aller plus vite dans la création de la Population malgré le fait que nous devons stopper pendant un temps notre programme pour que le résultat des threads aille dans la variable Population.

Notre algorithme génétique et notre interface graphique se partagent le temps, et grâce à cela nous pouvons voir évoluer le meilleur Individu. Sans thread à cette endroit notre courbe ne pourrait pas être affiché au fur et à mesure et donc nous ne pourrions pas avoir d'évolution en continu, on ne verrai pas la courbe au fur et à mesure du temps.

V. DIFFICULTÉS

La mise en place de l'utilisation de threads était assez compliqué à mettre en oeuvre, déjà pour savoir où en utiliser, et gérer le fait qu'ils doivent être fini pour pouvoir continuer notre programme (dans le cas de la création de la Population).

La création de l'interface graphique a été longue et difficile. Ce n'est pas les quelques séances de TD qui ont permis de maîtriser les classes liées aux interfaces graphiques. Le thème est donc resté méconnu. La plus grosse difficulté étant lors de l'utilisation de LayoutManager pour les deux panels contenus dans la Jframe. Aucun Layout ne nous a permis de garder la taille que l'on définissait à nos deux panels. Nous avons été obligé, dans la majorité des cas, de nous en passer. Nous avons donc disposé nos

différents panels manuellement en définissant les coordonnées et la taille de chaque panel. La deuxième difficulté rencontré pour l'interface graphique a été l'utilisation d'une nouvelle librairie (JFreeChart) pour représenter la courbe de Fitness. Nous avons trouvé très peu de documentation ou d'exemple de code pour comprendre comment faire une courbe.

VI. CONCLUSION

Pour conclure, ce projet a été une bonne introduction aux algorithmes génétiques, nous avons bien compris comment marche ces derniers. Nous arrivons mieux à utiliser les threads. Avec l'utilisation de Git, que nous avons appris pour certain d'autres nous, notre travail en groupe s'est amélioré. Avec ce projet nous avons bien travailler en groupe et nous avons aussi amélioré notre capacité à travail en groupe.