Projet Alternant
Plus Court Chemin

Jocelyn Hauf
21/05/2021

Table des matières

Introduction

Le projet suivant s'est déroulé dans le cadre du 4ème semestre de licence en informatique, il traite différents algorithmes permettant de trouver le plus court chemin entre plusieurs points afin de comparer leurs efficacités.

Il traite deux algorithmes :

- Algorithme du voyageur de commerce version naïve (TSP naive) :
    - Exploration complète
    - Exploration élaguée
- Algorithme de colonies de fourmis

Le travail demandé était de rédiger ces derniers en utilisant le langage C, dans le but de les lancer avec une machine et d'observer leur comportement ainsi que leur vitesse d'exécution.

Dans un premier temp, j'ai effectué plusieurs recherches en plus du document informatif donné afin d'avoir des connaissances solides sur le principe de ces algorithmes. Puis, j'ai traduit le pseudo-code correspondant à la fonction TSP_Naive, ainsi que la fonction TSP_Elag_Naive qui était fourni dans le sujet. Par la suite j'ai réalisé le programme qui permet d'exécuter l'algorithme des fourmis. Enfin, j'ai mis chacune des fonctions précédentes en commun pour réaliser une application.

L'application utilise les librairies suivantes pour réaliser les différents calculs :

- <stdio.h>
- <stdlib.h>
- <time.h>
- <math.h>

Ainsi que la librairie suivante pour obtenir un affichage :

- <MLV/MLV_all.h>

Aussi, un affichage basique est proposé sur la console dans le cas où la librairie MLV ne serait pas disponible ou ne fonctionnerait pas.

Définitions de structures

Les algorithmes proposés doivent chacun indiquer le chemin final trouvé, l'idéal est de stocker ces derniers dans un tableau d'entier de taille n (nombre de ville), mais aussi, ils doivent donner le cout total du trajet trouvé. De plus il est intéressant de stocker le nombre de ville existante afin de permettre des affichages itératifs. De ce fait, j'ai créé une structure nommée ' Solution ' contenant les informations suivantes :

- Nombre de ville (int n) ;
- Tableau d'ordre de parcours (int * ordre) ;
- Cout total (int cout) ;

Aussi, pour que ces algorithmes fonctionnent, il est nécessaire de connaitre le coût (ou la distance) entre les villes. Il est possible de modéliser le problème comme un graphe :

- Les villes sont représentées par les sommets ;
- Les coûts sont représentés par les arêtes (ces dernières ont donc un poids) ;

Alors, on peut constater que pour *n* villes, chacune est reliée aux *n-1* autres. C'est-à-dire qu'il existe donc n^2-n coût à stocker. Dans ce cas, l'intérêt est de travailler avec une matrice carrée d'ordre *n* avec chaque élément de la diagonale qui est nul. Aussi il est intéressant de stocker le nombre de ville avec la structure dans le cas où nous devons construire une matrice à partir d'un fichier ou pour parcourir la matrice itérativement. La structure nommée ' mat_cout ' contient donc :

- Nombre de ville (int nb_ville) ;
- Matrice des coûts entre villes (int ** cv) ;

Jusqu'à maintenant, nous étions nécessairement obligés de connaitre le coût entre les villes, néanmoins, il est plus logique de connaitre leur emplacement pour ensuite calculer leur distance et remplir la matrice. De plus nous avons à disposition un fichier contenant des coordonnées de villes afin de tester nos fonctions. Alors j'ai défini une structure de donnée nommée 'point' contenant :

- Coordonnée de l'abscisse (int x) ;
- Coordonnée de l'ordonnée (int y) ;

Puis défini une structure ' tabpoints ' indiquant :

- Nombre de points (int nb_point) ;
- Tableau de la structure précédente (struct point * tabp) ;

TSP Naive

Afin de réaliser cet algorithme, j'ai traduit le pseudo-code proposé dans l'énoncé. Au départ je n'utilisais pas de pointeur lors de la gestion des tableaux 'ordre' et des matrices car je ne maitrisais pas correctement cette notion, ce qui engendrait de nombreux problèmes de mémoire.

Lorsque j'ai compris le fonctionnement et l'utilisation des pointeurs, j'ai pu mettre à jour ma fonction afin d'optimiser au mieux la gestion mémoire et ainsi me permettre de travailler sur un nombre de ville conséquent.

Mis à part ceci, je n'ai pas eu de problème pour la réalisation de cette fonction.

Pour vérifier que l'algorithme traduis était correct, j'ai utilisé un jeu de donnée avec une réponse connue, que j'ai trouvé sur le site « *Wikipedia.fr »,* avec les valeurs suivantes :

	1	**2**	**3**	**4**
1	0	4	3	1
2	4	0	1	2
3	3	1	0	5
4	1	2	5	0

Le résultat attendu était :

- [1,3,2,4] avec un coût total de 7 ;

Ma fonction me renvoyait le même résultat, afin d'être sûr j'ai réalisé une seconde fonction créant une matrice aléatoirement, et ainsi vérifier manuellement si les résultats finaux étaient corrects. Les tests ont été concluant, alors j'ai étudié la capacité de cette fonction avec un nombre plus élevé de ville. Cette dernière fonctionne très bien jusqu'à une dizaine de villes, ensuite elle devient trop longue. Ceci s'explique par le fait qu'elle explore chaque chemin possible, et qu'il en existe ! $(n-1)$, ce qui est exponentiel.

TSP Elag Naive

Afin de limiter le nombre de chemin à explorer, il est possible d'améliorer la fonction précédente en observant si le coût du chemin partiel est supérieur au meilleur coût trouvé.

Grace à cela, le nombre de ville maximum prise en compte par la fonction est passé à 18 avec ma machine, ce programme me permettra de vérifier mes futurs résultats avec le prochain algorithme

Algorithme des fourmis

Pour organiser cet algorithme, j'ai créé une matrice de phéromones initialisée à 0 et qui sera mise à jour à chaque itération, puis j'initialise un tableau qui contiendra la chance de visite des prochaines villes restantes. Afin de remplir ce tableau, j'utilise le calcul donné dans l'énoncé. Enfin, pour choisir la prochaine ville de la solution explorée, je tire une valeur aléatoire entre 0 et 1 qui sera dans un intervalle du tableau. Une fois un chemin trouvé, nous ajoutons des phéromones au tableau pour rendre les futurs choix plus précis.

De ce fait, cette fonction n'explore à chaque itération qu'un seul et unique chemin, alors elle est rapide d'exécution. Aussi le nombre d'itération dépend du nombre de fourmi renseignée.

Cela nous permet de rechercher des chemins avec un nombre conséquent de villes étant donné la rapidité du programme.

Cet algorithme prend en compte plusieurs paramètres :

- Nombre de fourmis (**entier [0 – inf[**);
- Importance des coûts (pondere a), (**entier [0 – inf[**) ;
- Importance des taux de phéromones (*pondere b*), (**entier [0 – inf[**) ;
- Proportion de phéromones qui s'évapore (*p*) (**flottant [0-1]**);

Ce nombre de paramètre implique la nécessité de faire plusieurs réglages pour obtenir un résultat correct.

Avant de réussir cette fonction, j'ai eu du mal à comprendre le calcul et son utilisation, aussi, je ne savais pas comment organiser les différentes matrices et je ne comprenais pas de quelle manière stocker la chance de choix de la prochaine ville. Après un peu de réflexion et de recherches j'ai pu néanmoins commencer à entreprendre le code de l'algorithme.

L'utilisation d'un tableau pour stocker les chances de visite m'a paru plus intéressant qu'une matrice, en effet avec cette dernière, il est nécessaire de parcourir entièrement les n^2 éléments alors qu'avec un tableau seulement les n chances de choix de chemin sont calculées, puis lors de la sélection c'est aussi un parcours de n éléments. Ainsi le temp de l'exécution est fortement réduit.

De plus, j'ai fait le choix d'utiliser des tableaux dynamiques afin de ne pas être limité par le nombre de villes entrées tout en optimisant la mémoire utilisée par le programme et ainsi alléger le temp d'exécution. Les risques étaient les possibles erreurs d'allocation ou de libération mémoire qui survenaient relativement souvent et qui n'étaient pas évidentes à déboguer, et qui peuvent encore possiblement intervenir.

Affichage

Pour gérer l'affichage j'ai choisi une bibliothèque que je maitrise, c'est-à-dire MLV.

J'ai décidé d'utiliser une fonction de cette librairie permettant de récupérer la taille du bureau de la machine qui exécute le programme afin d'afficher une fenêtre ne dépassant pas de l'écran pour chaque type de résolution. Pour ce faire, j'utilise pour chacune des fonctions de MLV des valeurs en pourcentages que j'ai calculée au préalable sur une feuille. Cela m'a permis aussi d'organiser les différents éléments présents sur la fenêtre.

J'ai tout d'abord créé un menu permettant de charger la matrice de ville de quatre manières :

- Le test de Wikipédia (4 villes) ;
- Symétrique aléatoire ;
- Non symétrique aléatoire ;
- Sauvegardé dans un fichier .txt ;

Les matrices aléatoires requièrent l'entrée d'un nombre n de villes.

Une fois chargés, les points non aléatoires s'affichent pour avoir une représentation plus parlante. Pour les points chargés aléatoirement, une matrice les représentant s'affiche sur la console, il doit aussi être possible de pouvoir les afficher dans la fenêtre.

Aussi je comptais permettre l'affichage de quatre types d'exécutions :

- Algorithme des fourmis seulement
- Algorithme TSP Elag seulement
- Comparaison entre TSP Elag et TSP
- Comparaison entre TSP_Elag et Fourmis

Chacune de ces exécutions devaient être affichée dans une nouvelle fenêtre mais je n'ai pu traiter que les deux premières.

Pour aller plus loin, il serait intéressant d'afficher le chemin final sur la fenêtre sous forme de tableau ou même de façon visuelle avec des liens entre les points, ainsi que de montrer le coût final trouvé et le temp total d'exécution.

Tests et réglages

Les tests effectués sur les algorithmes TSP étaient correct et n'avait aucun réglage à effectuer. Cependant des réglages sont obligatoire en utilisant l'algorithme des fourmis. J'en ai réalisé de nombreux avec quatre villes et notamment avec la valeur test tirée de Wikipédia. En constatant que surtout le nombre de fourmis était importante pour ce nombre de ville mais qu'il fallait tout de même avoir un taux d'évaporation faible pour de meilleurs résultats. Pour un nombre plus élevé de ville, il faut aussi donner une meilleure importance aux coûts des chemins. Parfois le résultat n'est pas correct mais se rapproche du plus court chemin.

Les valeurs des réglages avec lesquels j'ai trouvé les résultats les plus fiables sont les suivant :

- Nombre de fourmis : 100 ;
- Importance des phéromones : 1 ;
- Importance des couts : 4 ;
- Taux d'évaporation : 0.4

Si je place 1 en taux d'évaporation, mon programme disfonctionne.

Lorsque j'ai testé avec les valeurs du fichier « defi250 », j'ai malheureusement une erreur lors d'un calcul entre la ville n°104 ainsi que la ville n°114, en ciblant ces valeurs tout en créant un nouveau fichier de 25 villes (« defi25 ») j'ai pu vérifier si l'erreur ne venait pas de ces valeurs, avec ce nouveau fichier mon programme fonctionnait correctement, alors je suppose que ce disfonctionnement est dû à une erreur de segmentation, donc à une mauvaise gestion de la mémoire.

De plus j'ai pu vérifier le maximum de ville que je pouvais traiter grâce aux initialisations de matrices aléatoire. Je suis arrivé à la conclusion que mon programme fonctionne pour l'instant jusqu'à 80-90 villes. Une fois ce problème de mémoire réglé, nous pouvons supposer que le plus court chemin de ce fichier test devrait s'afficher beaucoup plus rapidement qu'avec TSP Elagué mais possiblement avec des couts plus élevés.

Conclusion

À la suite de ce projet, on peut facilement démontrer que les algorithmes TSP sont très précis mais très lent et gourmand, alors qu'à l'inverse l'algorithme des fourmis n'est pas très précis lorsque les réglages ne sont pas connus mais est très rapide tant que le nombre d'itération n'est pas lourd.

On peut néanmoins supposer que lorsque les paramètres sont correctement réglés, le résultat final doit être relativement précis.

Une solution pour trouver les meilleurs réglages est d'explorer chacun des réglages possibles, mais aussi de comparer avec d'autre programme les paramètres donnant les résultats les moins coûteux trouvés. Pour accélérer ces recherches il est possible d'exécuter le programme avec différents paramètres sur plusieurs machines.

Aussi nous pouvons nous demander si l'utilisation d'une recherche par arbre binaire serait possible et si elle serait meilleure pour ce genre de recherche.