Halima Ksal

L3-A

Rapport Projet 12

Voyageur de commerce 2 :

introduction :

Ce rapport se concentre sur deux implémentations distinctes de l'heuristique du plus proche voisin pour résoudre le problème du voyageur de commerce (TSP), un problème classique d'optimisation consistant à trouver le chemin le plus court passant une seule fois par chaque ville d'un ensemble donné. Le voyageur de commerce est modélisé sur un graphe non orienté pondéré qui est non connexe, avec les villes représentées par les sommets du graphe. Les deux approches analysées utilisent des structures de données différentes: une basée sur des listes chaînées (liste.c) et l'autre sur des vecteurs (vecteur.c). L'objectif est d'évaluer ces méthodes en termes d'utilisation de la mémoire, de performances, et d'efficacité de l'itinéraire généré.

Méthodologie :

L'analyse se concentre sur la structure de données, l'utilisation de la mémoire, la performance en termes de temps d'exécution, et l'efficacité des itinéraires trouvés par l'heuristique du plus proche voisin dans chaque implémentation.

Au début j'ai commencé par tester l'heuristique du plus proche voisin sur un graphe complet où toutes les villes étaient connectées avec des valeurs réalistes, je me suis rendu compte qu'en dépassant les 15 villes le temps de calcule étaient énorme donc je n'avais pas réussi à avoir l'itinéraire. Par la suite, j'ai utilisée un graphe pondéré non orienté qui est non connexe avec un taux de connectivité de 20 % (plus le taux est faible plus il y a des trous) comme la consigne le demandait.

Choix heuristique : Heuristique du plus proche voisin

Avantages et Inconvénients de l'Heuristique du plus proche voisin :

- **Avantages** :
  - Facilité d'implémentation.
  - Résultats généralement bons pour des instances de petite à moyenne taille du voyageur de commerce.
- **Inconvénients** :
  - Absence de garantie de trouver la solution optimale.

Structures de Données :

- **Vecteur (vecteur.c) :**
  - Utilise un tableau dynamique pour stocker les arêtes (connexions) de chaque nœud (ville).
  - Les vecteurs permettent un accès rapide et direct aux données, mais peuvent nécessiter des réallocation lors de l'ajout de nouvelles arêtes.

```c
typedef unsigned short Shu;

typedef struct {
    int destination;
    Shu distance;
} Vecteur; // imaginer comme une route entre deux villes

typedef struct {
    int ville; // stock numero id de la ville
    bool vu; // pour indiquer si la ville a été visitée ou non (dfs)
    int vecteurCount; // nbr de connexions que la ville a avec d'autres villes
    Vecteur *vecteurs; // tableau vecteur pour stocker les info sur les connexions
} Node;
```

- **Liste (liste.c) :**
  - Utilise des listes chaînées pour stocker les arêtes.
  - Les listes chaînées offrent une flexibilité pour ajouter ou supprimer des éléments sans réallocation, mais l'accès aux données peut être plus lent car il nécessite un parcours de la liste.

```c
typedef unsigned short Shu;

typedef struct strand {
    int destination;
    Shu distance;
    struct strand *next; // pointeur vers la prochaine connexion
} strand;

typedef struct Node {
    int ville;
    strand *aretes; // pointeur vers le debut d'une liste d'aretes connectées à ce noeud
} Node;
```

Fonctions Générales :

Les deux fichiers contiennent des fonctions similaires, adaptées à leurs structures de données respectives :

Allocation de Memoire :

- Fonction pour allouer la mémoire et garder une trace de la mémoire allouée pour pouvoir comparer par la suite les deux programmes

Création de Graphes :

- Fonction pour construire un réseaux de villes en décidant aléatoirement quel chemin existe entre eux, il y a aussi un taux de connectivité de 20% pour créer un graphe peu dense (avec des trous).

Ajout d'Arêtes :

- Fonction pour ajouter des arêtes entre les nœuds. Cette fonction est appelé dans la fonction creerGraphe.

Affichage du Graphe :

- Fonction pour afficher les nœuds et leurs arêtes

Recherche de Composantes Connexes :

- Fonction pour identifier et lister les différentes composantes connexes du graphe.
- DFS : Parcourt le graphe pour voir quelles villes sont connectées les unes aux autres.

Algorithme du Plus Proche Voisin (PPV) :

- Fonction implémentant l'heuristique du plus proche voisin pour trouver un chemin approximatif à travers le graphe.
- On commence par une ville de départ disons la ville A
- On regarde toutes les villes connectées à A et on choisi la ville la plus proche par exemple : si c'est B alors on va à B
- On répéte l'opération avec les autres villes connectées avec B ainsi de suite jusqu'à visiter toutes les villes

Calcul du Coût Total de l'Itinéraire :

- Fonction pour calculer le coût total de l'itinéraire trouvé par l'heuristique du plus proche voisin.

Libération de la Mémoire :

- Fonction pour libérer la mémoire allouée au graphe, adaptées à la structure de données utilisée.

Étapes du Projet

Les étapes typiques du projet comprennent :

- Initialisation du graphe avec des nœuds et des arêtes.
- Application de l'heuristique pour trouver un itinéraire à travers le graphe.
- Évaluation de la performance de l'heuristique en termes de coût total de l'itinéraire et de temps d'exécution.

- Comparaison des performances entre les deux implémentations pour comprendre les avantages et les inconvénients de chaque structure de données.

Analyse des Implémentations :

Pour la version Vecteur (vecteur.c) :

- **Initialisation** : Chaque nœud est initialisé avec un identifiant, un marqueur non visité, et un tableau dynamique pour les arêtes.
- **Gestion de la Mémoire** : Utilise un tableau avec réallocation, adapté pour des graphes stables ou de grande taille.
- **Accès aux Arêtes** : Direct et rapide grâce au tableau de vecteurs.
- **Inconvénients** : Coûts potentiels liés aux réallocation fréquentes.

Pour la version Liste (liste.c) :

- **Initialisation** : Similaire à vecteur.c, mais les arêtes sont stockées dans une liste chaînée.
- **Gestion des Listes Chaînées** : Permet une insertion facile des nouvelles arêtes, efficace pour des graphes dynamiques.
- **Accès aux Arêtes** : Plus lent, nécessitant un parcours de la liste.
- **Avantages** : Flexibilité dans la gestion des arêtes et moindre fragmentation de mémoire.

Comparaison des Performances :

- **Temps d'Exécution** : La version liste s'est avérée légèrement plus rapide que la version vecteur, avec des moyennes respectives de 46.8 et 56.2 microsecondes par itération.

Pour le programme basé sur une LISTE :

Itération 1 : 0 secondes et 51 microsecondes

Itération 2 : 0 secondes et 49 microsecondes

Itération 3 : 0 secondes et 53 microsecondes

Itération 4 : 0 secondes et 43 microsecondes

Itération 5 : 0 secondes et 38 microsecondes

Somme des temps d'exécution pour la LISTE :

(51 + 49 + 53 + 43 + 38) = 234 microsecondes

Moyenne du temps d'exécution pour la LISTE :

234 / 5 = 46.8 microsecondes par itération

Pour le programme basé sur un VECTEUR :

Itération 1 : 0 secondes et 47 microsecondes

Itération 2 : 0 secondes et 49 microsecondes

Itération 3 : 0 secondes et 86 microsecondes

Itération 4 : 0 secondes et 52 microsecondes

Itération 5 : 0 secondes et 47 microsecondes

Somme des temps d'exécution pour le VECTEUR :

(47 + 49 + 86 + 52 + 47) = 281 microsecondes

Moyenne du temps d'exécution pour le VECTEUR :

281 / 5 = 56.2 microsecondes par itération

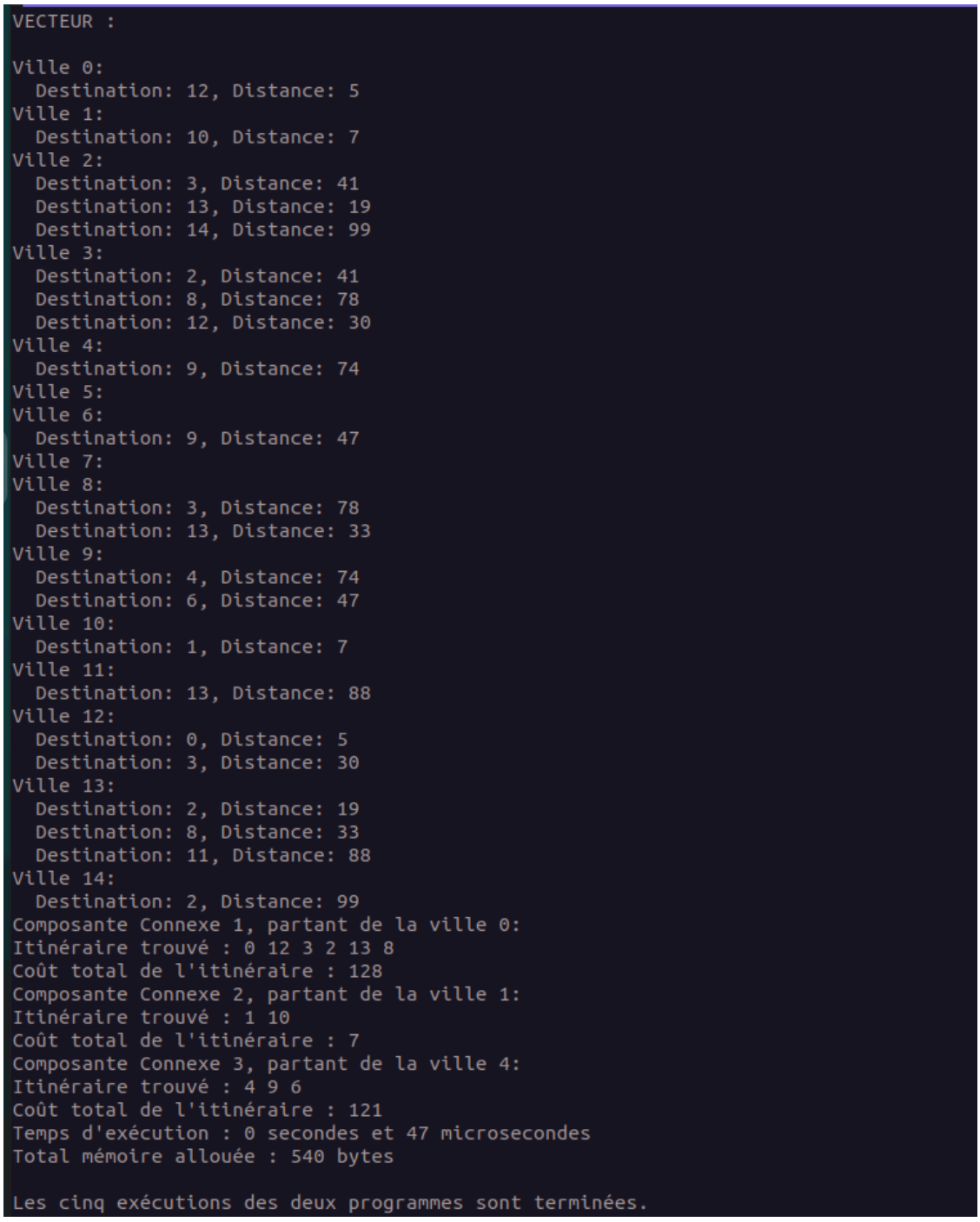
- **Utilisation de la Mémoire** : La version vecteur a montré une meilleure efficacité en termes d'utilisation de la mémoire.

Éxécution avec compare.c :

```
################################################################
Comparaison entre Liste et Vecteur pour le voyageur de commerce (Itération 5):

LISTE :

Ville 0:
  Destination: 12, Distance: 5
Ville 1:
  Destination: 10, Distance: 7
Ville 2:
  Destination: 14, Distance: 99
  Destination: 13, Distance: 19
  Destination: 3, Distance: 41
Ville 3:
  Destination: 12, Distance: 30
  Destination: 8, Distance: 78
  Destination: 2, Distance: 41
Ville 4:
  Destination: 9, Distance: 74
Ville 5:
Ville 6:
  Destination: 9, Distance: 47
Ville 7:
Ville 8:
  Destination: 13, Distance: 33
  Destination: 3, Distance: 78
Ville 9:
  Destination: 6, Distance: 47
  Destination: 4, Distance: 74
Ville 10:
  Destination: 1, Distance: 7
Ville 11:
  Destination: 13, Distance: 88
Ville 12:
  Destination: 3, Distance: 30
  Destination: 0, Distance: 5
Ville 13:
  Destination: 11, Distance: 88
  Destination: 8, Distance: 33
  Destination: 2, Distance: 19
Ville 14:
  Destination: 2, Distance: 99
Composante Connexe 1, partant de la ville 0:
Itinéraire trouvé : 0 12 3 2 13 8
Coût total de l'itinéraire : 128
Composante Connexe 2, partant de la ville 1:
Itinéraire trouvé : 1 10
Coût total de l'itinéraire : 7
Composante Connexe 3, partant de la ville 4:
Itinéraire trouvé : 4 9 6
Coût total de l'itinéraire : 121
Temps d'exécution : 0 secondes et 38 microsecondes
Total memoire alloué : 772 bytes
```



Éxécution avec 5 villes et 50 villes sur une composante connexe avec srand(45) et 20% en taux de connectivité pour les deux structures :

Vecteur :

```
Ville 0:
  Destination: 2, Distance: 69
Ville 1:
  Destination: 2, Distance: 8
Ville 2:
  Destination: 0, Distance: 69
  Destination: 1, Distance: 8
Ville 3:
Ville 4:
Composante Connexe 1, partant de la ville 0:
Itinéraire trouvé : 0 2 1
Coût total de l'itinéraire : 77
Temps d'exécution : 0 secondes et 17 microsecondes
Total mémoire allouée : 140 bytes
```

```
Ville 46:
  Destination: 18, Distance: 4
  Destination: 23, Distance: 8
  Destination: 27, Distance: 21
Ville 47:
  Destination: 18, Distance: 42
  Destination: 25, Distance: 77
  Destination: 28, Distance: 41
Ville 48:
  Destination: 3, Distance: 30
  Destination: 13, Distance: 88
  Destination: 30, Distance: 45
Ville 49:
  Destination: 17, Distance: 69
  Destination: 18, Distance: 72
  Destination: 23, Distance: 92
  Destination: 32, Distance: 20
Composante Connexe 1, partant de la ville 0:
Itinéraire trouvé : 0 5 20 30 48 3 4 24 7 38 12 22 37 40 32 21 41 15 39 2 33
Coût total de l'itinéraire : 622
Temps d'exécution : 0 secondes et 33 microsecondes
Total mémoire allouée : 1400 bytes
```

Liste :

```
Ville 0:
  Destination: 2, Distance: 69
Ville 1:
  Destination: 2, Distance: 8
Ville 2:
  Destination: 1, Distance: 8
  Destination: 0, Distance: 69
Ville 3:
Ville 4:
Composante Connexe 1, partant de la ville 0:
Itinéraire trouvé : 0 2 1
Coût total de l'itinéraire : 77
Temps d'exécution : 0 secondes et 31 microsecondes
Total memoire alloué : 184 bytes
```

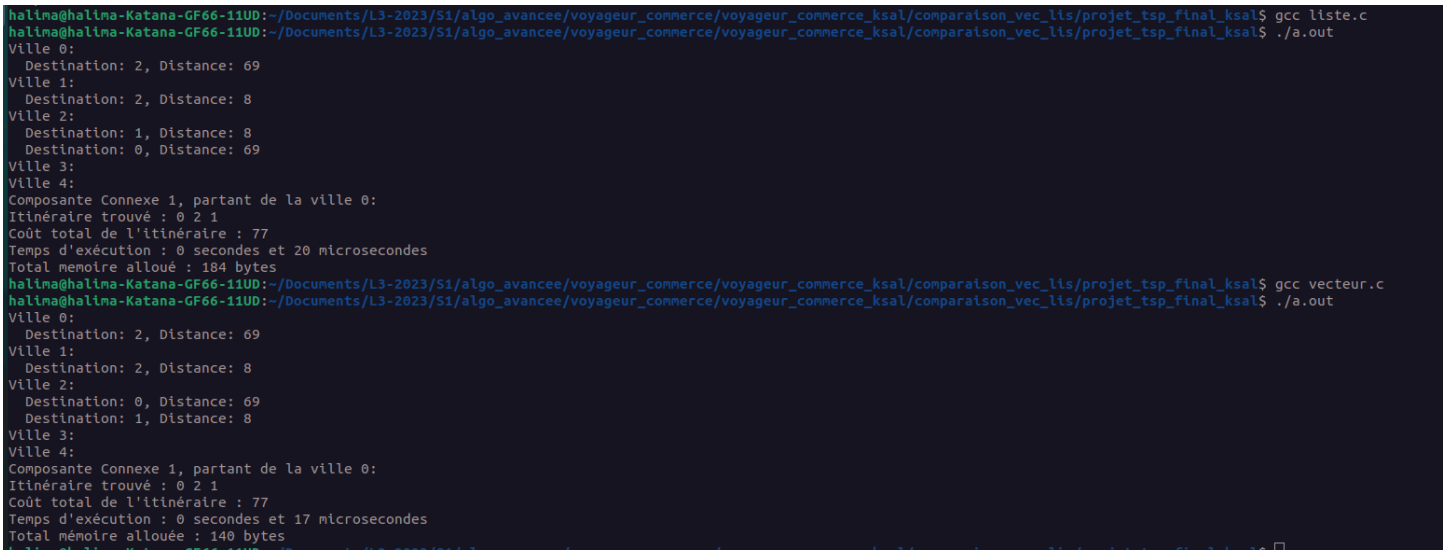
```
Ville 46:
  Destination: 27, Distance: 21
  Destination: 23, Distance: 8
  Destination: 18, Distance: 4
Ville 47:
  Destination: 28, Distance: 41
  Destination: 25, Distance: 77
  Destination: 18, Distance: 42
Ville 48:
  Destination: 30, Distance: 45
  Destination: 13, Distance: 88
  Destination: 3, Distance: 30
Ville 49:
  Destination: 32, Distance: 20
  Destination: 23, Distance: 92
  Destination: 18, Distance: 72
  Destination: 17, Distance: 69
Composante Connexe 1, partant de la ville 0:
Itinéraire trouvé : 0 5 20 30 48 3 4 24 7 38 12 22 37 40 32 21 41 15 39 2 33
Coût total de l'itinéraire : 622
Temps d'exécution : 0 secondes et 35 microsecondes
Total memoire alloué : 4432 bytes
```

Limitations et défis du projet :

- **Graphe "Gruyère"** : La faible connectivité (20% dans le programme) entre les villes crée des graphes avec des "trous", complique la résolution du voyageur de commerce.
- **Absence de Circuit Fermé** : L'heuristique plus proche voisin, bien qu'efficace pour trouver un chemin initial ne garantit pas de trouver la meilleur solution.

Test avec un plus petit nombre de sommets dans le graphe :

J'ai testé avec nbs = 5, le même taux = 20% et srand(45) pour garantir que le programme génère la même séquence de nombres aléatoires à chaque exécution pour les 2 programmes



Comparaison des deux programmes :

1. Complexité Algorithmique :

Liste (liste.c): Manipulation efficace grâce à la complexité O(1) pour l'ajout et la suppression, mais accès aux données en O(n) du fait de la nécessité de parcourir la liste.

Vecteur (vecteur.c): Accès direct aux données en O(1), bénéfique pour les opérations fréquentes d'accès, mais manipulation des données en O(n) en raison de potentiels décalages lors de l'ajout ou de la suppression.

2. Utilisation de la Mémoire :

Liste: Plus gourmande en mémoire en raison des pointeurs supplémentaires mais flexible, sans besoin de redimensionner.

Vecteur: Plus compact, mais peut nécessiter une réallocation coûteuse en cas d'extension.

3. Performance :

Liste: Moins performante pour l'accès aux données mais plus rapide pour les modifications tels que l'ajout et la supression.

Vecteur: Plus performant pour l'accès aux données, mais potentiellement plus lent pour les modifications structurelles.

Conclusion :

Ce projet m'a permis d'explorer en détail le problème du voyageur de commerce à travers deux approches distinctes utilisant l'heuristique du plus proche voisin. En comparant les implémentations basées sur des listes et des vecteurs, j'ai identifié des compromis clés entre l'utilisation de la mémoire, la performance et la flexibilité dans la manipulation des données. Tandis que ma version basée sur des listes offre une meilleure performance pour les modifications structurelles tels que l’ajout ou la suppression, ma version basée sur des vecteurs gère mieux l'accès rapide aux données ce qui me fait dire que la liste est plus adaptée pour les graphes non connexe avec des trous. Les limites rencontrées avec les graphes peu connectés soulignent la complexité du voyageur de commerce sur des graphes peu dense.