Compléments d'informatique

Projet 2 : Aidez Saint-Nicolas !

Ce projet vise à aider Saint-Nicolas (Vecoven) à optimiser sa tournée des villes de Belgique. En terme de programmation, ce projet vous permettra d'exercer les compétences suivantes : programmation modulaire, types de données abstraits, liste liée, analyse de complexité, et lecture de fichiers. Vous en apprendrez aussi un peu sur le problème du voyageur de commerce, qui est un problème algorithmique important.

La section 1 décrit d'abord le problème qu'on vous demande de résoudre. La section 2 décrit ensuite la structure du code proposée et les différentes fonctions qu'il vous faudra implémenter. Ce projet est à réaliser par groupe de deux étudiants maximum. Les modalités de soumission du projet via la plateforme sont décrites dans la section 3.

1 Problème du voyage de commerce

Saint-Nicolas voudrait optimiser sa tournée des villes de Belgique. Le problème est modélisé de la manière suivante. Soit une liste $\{v_1, \ldots, v_N\}$ de N villes représentées par leurs coordonnées (x_j, y_j) (avec $j \in \{1, \ldots, N\}$). On aimerait trouver un ordre de visite de ces villes qui minimise la longueur du trajet total, en supposant que Saint-Nicolas peut se déplacer en ligne droite entre deux villes et aussi qu'il souhaite revenir à sa ville de départ une fois sa tournée terminée. Soit $(v_{i_1}, \ldots, v_{i_N})$ un tel ordre avec $(i_1, i_2, \ldots, i_N)$ une permutation de $(1, 2, \ldots, N)$. La longueur de la tournée sera :

$$\sum_{k=1}^{N-1} \sqrt{(x_{i_k} - x_{i_{k+1}})^2 + (y_{i_k} - y_{i_{k+1}})^2} + \sqrt{(x_{i_N} - x_{i_1})^2 + (y_{i_N} - y_{i_1})^2}, \tag{1}$$

où les deux derniers termes correspondent à la longueur du chemin pour revenir à la première ville.

Vu vos compétences en informatique, vous décidez d'aider Saint-Nicolas. Après vous être renseignés, vous vous rendez compte que le problème correspond au fameux problème

du voyageur de commerce ("Traveling Salesman Problem", TSP) mais vous apprenez malheureusement par la même occasion que personne n'a encore trouvé d'algorithme efficace (c'est-à-dire de complexité non exponentielle par rapport au nombre de villes) permettant de trouver la solution optimale à ce problème. Comme le 6 décembre approche, vous vous dites que vous gagneriez du temps en utilisant un algorithme heuristisque, c'est-à-dire un algorithme ne fournissant qu'une solution approchée au problème mais nettement plus efficace que l'algorithme optimal. Les deux algorithmes suivants vous semblent intéressants à tester :

- **Heuristique 1 :** On fixe arbitrairement un ordre des villes. On crée un premier tour qui ne contient que la première ville seule. On ajoute ensuite les villes une par une au tour courant en suivant l'ordre initial et en ajoutant chaque nouvelle ville dans le tour juste **après** la ville du tour dont elle est la plus proche (en terme de distance Euclidienne).
- **Heuristique 2 :** On procède exactement comme pour l'algorithme précédent si ce n'est qu'on ajoute la nouvelle ville à la position dans le tour qui minimise l'augmentation de la longueur du tour.

Dans ce projet, on vous demande d'implémenter ces deux heuristiques et de les utiliser pour trouver le tour le plus court possible de toutes les villes de Belgique dont les coordonnées vous sont fournies.

2 Implémentation

Votre implémentation demandera la création de trois modules :

- Un module constitué des fichiers `town.c` et `town.h` implémentant la structure et les fonctions nécessaires à représenter et manipuler des villes.
- Un module constitué des fichiers `tour.c` et `tour.h` implémentant un tour de villes.
- Un module constitué des fichiers `tsp.c` et `tsp.h` implémentant les heuristiques décrites ci-dessus visant à résoudre le problème du voyageur de commerce.

Ces modules seront utilisés par le fichier `salesman.c` pour générer des propositions de tours avec les deux heuristiques proposées sur base d'une fichier reprenant les coordonnées d'un ensemble de villes.

Fichiers `town.h` et `town.c`

Ce module doit définir une structure abstraite de type `Town` qui servira à contenir les informations utiles pour représenter une ville (nom et coordonnées x-y). Les fonctions liées à cette structure seront les suivantes :

`Town *createTown(const char *name, double x, double y)` : crée une ville nommée `name` et situé à la position $(\texttt{x}, \texttt{y})$. La chaîne de caractère utilisée pour le nom devra être recopiée dans un nouvel espace mémoire.

`double getTownX(Town *town)` : renvoie la coordonnée x de la ville `town`.

`double getTownY(Town *town)` : renvoie la coordonnée y de la ville `town`.

`const char *getTownName(Town *town)` : renvoie le nom de la ville `town` (une chaîne de caractères)

`double distanceBetweenTowns(Town *town1, Town *town2)` : renvoie la distance euclidienne entre les villes `town1` et `town2`.

`void freeTown(Town *town)` : libère la mémoire prise par `town` (y compris la chaîne de caractères allouée pour le nom).

Fichiers `tour.h` et `tour.c`

Ce module définit le type de données abstrait utilisé pour définir un tour de villes. Un tour est séquence de villes telle que la première ville est similaire à la dernière et toutes les villes entre les deux sont uniques (et différentes de la première). L'ordre des villes dans le tour est important car il représente l'ordre dans lequel un voyageur traverse ces villes avant de revenir à son point de départ. Pour représenter un tour, vous devez définir deux structures :

- Une structure `Tour` représentant le tour,
- Une structure `TourPosition` représentant une position dans le tour et permettant de parcourir les villes du tour dans l'ordre.

Bien qu'il y ait plusieurs manières de représenter ces structures, nous vous conseillons d'utiliser une structure de type liste liée telle que vue au cours théorique. La structure `TourPosition` représentera un élément (ou nœud) de liste liée. Les fonctions de manipulation de la liste que vous devez implémenter sont les suivantes :

`Tour *createEmptyTour(void)` : crée un tour vide.

`Tour *createTourFromFile(char *filename)` : crée un tour contenant les villes présentes dans le fichier `filename`. Ce fichier sera toujours sous le format suivant :

nom ville 1, coord. x ville 1, coord. y ville 1
nom ville 2, coord. x ville 2, coord. y ville 2
nom ville 3, coord. x ville 3, coord. y ville 3
...

où chaque ligne fournit le nom et les coordonnées (x, y) d'une ville. Le tour créé devra contenir les villes dans l'ordre dans lequel elles apparaissent dans le fichier.

`void freeTour(Tour *tour, int freeTown)` : libère la mémoire occupée par le tour `tour`. Si `freeTown` est plus grand que 0, les villes sont également libérées en utilisant la fonction `freeTown`.

`void addTownAtTourEnd(Tour *tour, Town *town)` : Ajoute la ville `town` à la fin du tour `tour`.

`void addTownAfterTourPosition(Tour *tour, TourPosition *pos, Town *town)` : Ajoute la ville `town` juste après la position `pos` dans le tour `tour`.

`TourPosition *getTourStartPosition(Tour *tour)` : Renvoie la première position dans le tour `tour`.

`TourPosition *getNextTourPosition(Tour *tour, TourPosition *pos)` : Renvoie la position qui suit la position `pos` dans le tour `tour`. Si `pos` est la denière position dans la tour (avant de revenir à la ville de départ), la fonction renverra `NULL`.

`Town *getTownAtPosition(Tour *tour, TourPosition *pos)` : Renvoie la ville à la position `pos` dans le tour `tour`.

`int getTourSize(Tour *tour)` : Renvoie le nombre de villes (uniques) dans le tour.

`double getTourLength(Tour *tour)` : Renvoie la longueur du tour telle que calculée par 1.

Fichiers `tsp.h` et `tsp.c`

Ce module sert à l'implémentation des fonctions de calculs de chemins, plus particulièrement des heuristiques proposées ci-dessus. Les fonctions suivantes devront donc être implémentées :

`Tour *heuristic1(Tour *tour)` : Renvoie un **nouveau** tour qui contient l'ensemble des villes de `tour` ordonnées grâce à la première heuristique.

`Tour *heuristic2(Tour *tour)` : Renvoie un **nouveau** tour qui contient l'ensemble des villes de `tour` ordonnées grâce à la seconde heuristique.

Note. Nous insistons sur le fait que le tour renvoyé doit être un nouveau tour. Il ne s'agit donc pas de modifier le tour passé en argument aux fonctions. De plus, nous attirons votre attention sur le fait que si le tour `tour` passé en argument est détruit, le tour renvoyé par la fonction devra toujours être complètement fonctionnel.

Fichier `salesman.c`

Ce fichier devra implémenter une fonction `main` qui effectuera les étapes suivantes :

1. Création d'un tour au moyen de la fonction `createTourFromFile` du fichier `tour.c` à partir d'un fichier passé en argument.
2. Application des deux heuristiques demandées pour calculer le plus court chemin passant par toutes les villes définies par le fichier.
3. Création de trois fichiers `PPM`, `tour-default.ppm`, `tour-heuristic1.ppm` et `tour-heuristic1.ppm`, représentant respectivement le tour suivant l'ordre des villes telles qu'elles apparaissent dans le fichier fourni en argument, le tour tel qu'obtenu avec la première heuristique et le tour tel qu'obtenu avec la seconde heuristique.

Pour cette dernière étape, nous vous fournissons dans le fichier `salesman.h` une fonction `void ppmTour(Tour *tour, const char *ppmName, int size)` qui crée le fichier `ppm`, nommé selon `ppmName`, à partir du tour `tour`. L'argument `size` de cette fonction est la taille de l'image générée que vous pouvez fixer à 1000 dans votre fichier `salesman.c`. Cette fonction se base sur l'interface définie dans les fichiers `town.h` et `tour.h`. Un fichier `xy-belgium-towns.csv` vous est fourni contenant les coordonnées x et y des 2756 communes belges (en kilomètres par rapport à Bruxelles supposée située en $(0, 0)$), vous permettant de tester vos fonctions.

3 Soumission

Le projet doit être soumis via la plateforme de soumission sous la forme d'une archive au format `zip` ou `tar.gz` contenant les fichiers suivants :

— `town.h` et `town.c`
— `tour.h` et `tour.c`
— `tsp.h` et `tsp.c`
— `salesman.c`
— Le fichier `questions.txt` complété avec vos réponses à différentes questions, relatives notamment à la complexité de vos implémentations de différentes fonctions.

Vos fichiers seront compilés et testés sur la plateforme de soumission en utilisant le fichier `Makefile` fourni via la commande `make all` ou de manière équivalente en utilisant la commande suivante :

```
gcc -o salesman salesman.c town.c tsp.c tour.c easyppm.c --std=c99 -lm
```

Votre programme devra alors s'utiliser comme suite :

```
./salesman nom_de_fichier.csv
```

En outre, nous utiliserons les flags de compilation habituels (`-pedantic -Wall -Wextra -Wmissing-prototypes`), qui ne devront déclencher aucun avertissement lors de la compilation, sous peine d'affecter négativement la cote. Toutes les soumissions seront soumises à un programme de détection de plagiat. En cas de plagiat avéré, l'étudiant (ou le groupe) se verra affecter une cote nulle à l'ensemble des projets.

Note importante. Au vu des soucis de compilation sur la plateforme rencontrés pour le premier projet, un feedback plus complet relatif à la compilation sera rendu lors de la soumission. Néamoins pour vous décourager d'utiliser la plateforme pour débugger votre code, le nombre de soumissions par jour sera limité à 5.

Bon travail !