GALBEZ Flavien

GHAHARIAN Alexandre

TD J

Problème noté n°1

Structure de donnée et algorithme

Commentaires importants :

- On a corrigé le tiret manquant à Le-Mans dans « connexions.csv » et retiré le dernier retour à la ligne dans les deux fichiers pour qu'il soit plus accord avec le format .csv (chaque ligne est une grille de valeur or la dernière n'en avait aucune)
- Le programme n'affiche que les trajets (ou les branches pour l'arbre) se terminant sur la ville de destination : elle n'affiche pas les branches non terminées à cause de la hauteur (voir comment dans explication de type)
  ⇨ On respecte l'intégralité de l'énoncé
- L'arbre n'a pas 2 fois la même ville présente dans une même branche comme demandé dans l'énoncé.
- Les allocations mémoires sont toujours testées et de taille adaptée.
- On prend soin de libérer la mémoire avant de créer un nouvel arbre.
- Toutes les saisies sont sécurisées
- Le réglage de la hauteur se fait en dur dans le .h sous forme de constante
- Les détails de fonctionnement de toutes les fonctions ont été commentés dans le programmes

Données en mémoire : La méthode choisit pour structurer les données en mémoire et la même que celle proposée dans le sujet à une différence près. En effet, les villes sont représentées dans la liste chaîné des destinations par un index correspondant à la position dans le tableau de ville correspondant au lieu d'être un char*, cela simplifiera un grand nombre d'opération et réduira l'espace mémoire nécessaire à cette structure.

Structure de l'arbre : Le type Nœud est un type récursif : il est composé d'un index_ville correspondant à l'indice de la ville correspondant dans le tableau de ville, d'un nombre de fils, et d'un pointeur de pointeur qui renvoyera sur des nœuds et aura pour espace alloué le nombre de fils. Il possèdera aussi un booléen valide qui permettra de savoir s'il mène à un chemin ayant pour ville final la ville de destination.

Création de l'arbre : Chaque nœuds sont initialisée grâce à creer_nœud. On garde en mémoire les villes parcourus grâce à un tableau d'index. Lors de la création d'un nœud, on exclut dans le comptage des fils les villes qu'on a déjà parcouru. On alloue les espaces mémoires du nœud correspondant et mets valide à false.

Ensuite nous utilisons la fonction creation_arbre_rec qui permet de créer l'arbre récursivement. Elle a deux cas d'arrêt : la fonction à atteint la hauteur max, la ville d'arrivée a été atteinte. Dans le deuxième cas, grâce à la variable validité, la valeur valide des nœuds menant à ce cas d'arrêt (feuille) deviendra true. A chaque fois qu'on crée un nœud, l'index de la ville correspondante et ajouté au tableau de villes visitées afin de ne pas repasser par cette dernière. On appelle ensuite récursivement cette fonction pour chacun des nœuds. (Pour plus de détails voire les commentaires dans le code)

Enfin nous créons la fonction creation_arbre qui permet d'appeler la fonction creation_arbre_rec avec les bons paramètres.

Utilisation de l'arbre : Son utilisation fonctionne généralement de la même façon : On ne prend en comptes que les nœuds avec valide = true et on appelle cette fonctionnalité de manière récursive avec come cas d'arrêt le fait d'être une feuille pour chacun de nœuds en retrouvant le chemin correspondant dans le tableau de ville pour avoir la durée et la distance. (Le détails de chacune des fonctions a été fait directement en commentaires dans le programme)

Le menu :

Dans un premier temps, nous utilisons la fonction Ville* intitialiser_donnees() qui compte le nombre de ville présent dans le fichier ville.csv puis un tableau est réalisé pour enregistrer les villes avec Ville* creation_liste_ville() et les connexions sont ensuite enregistrés dans le tableau pour avoir les connexions entre les villes avec int creation_liste_trajet(), cette fonction est u int pour vérifier si il y a des erreurs sur la lecture du fichier. Puis il est demandé si l'utilisateur veut voir les villes disponibles, en tapant 1 les villes du fichiers ville.csv sont affichés et en tapant 0, le programme rentre directement dans la fonction Noeud* saisie_arbre() qui demande la saisie de la ville de départ et de la ville d'arrivée. Puis le nom des villes rentrés par l'utilisateur seront transformés en index par la fonction int convertir_nom_ville_en_index(), pour vérifier si les villes sont bien présent dans le tableau réalisé et créer les trajets plus facilement.

Il y a aussi la fonction void menu(), qui donne le choix à l'utilisateur sur ce qu'il veut que le programme affiche, en tapant 1, le programme affichera 1 trajet aléatoirement, grâce à void afficher_un_trajet(), qui alloue dynamiquement l'espace pour écrire le nom de la ville en prenant en compte le nom le plus long. Puis cette fonction appelle la fonction void afficher_un_trajet_rec() pour mettre en forme le trajet en écrivant le nom de la ville, une flèche, pour signifier les étapes entre chaque ville visitée et la distance et la durée après la dernière ville pour estimer le trajet, pour écrire le nom de la ville et une flèche nous utilisons

l'appel récursif. Puis il y a, l'appel de la fonction choix suivant qui demande si l'utilisateur veut continuer la simulation, si non taper 0, le programme se fermera et si oui taper 1, il sera demandé si l'utilisateur veut changer de ville ou pas, si oui taper 1 et entrer les nouvelles villes si non taper 0 et les 6 choix seront de nouveaux demandés. En tapant 2, la fonction void afficher_tout_trajet() sera effectué, qui a le même fonctionnement que void afficher_un_trajet(), et fera appelle à void afficher_tout_trajet_rec(), qui a le même fonctionnement que void afficher_un_trajet_rec(), mais il parcourt toute les possibilités de trajet entre les 2 villes. En tapant sur 3, la fonction void afficher_plus_court_trajet() sera effectué, qui rentre dans la fonction void trouver_plus_efficace_trajet_rec(), qui permet de trouver le trajet le plus court ou le plus rapide en fonction de si l'utilisateur a tapé sur 3 ou 4, puis la fonction void afficher_plus_efficace_trajet_rec() est effectué pour afficher le résultat et fonctionne comme void afficher_un_trajet_rec(). En tapant sur 4, c'est le même principe que si l'utilisateur tape sur 3, mais à la différence que void afficher_plus_rapide_trajet() pour que la fonction void trouver_plus_efficace_trajet_rec() donne le trajet le plus rapide. En tapant 5, la fonction void afficher_arbre() est effectué, ce qui lance en récursif void afficher_arbre_rec(), qui fonctionne comme void afficher_un_trajet_rec() mais à la place de d'écrire une flèche entre les villes, cette fonction positionne les ville de façon à avoir différents étage pour distinguer les différentes étapes et possibilités pour les trajets sans écrire plusieurs fois le même nom de ville. Pour finir en tapant 6, le programme se ferme

Exemple d'affichage :

Pour une hauteur_max égale à 5

Avec le trajet Paris à Lille

```
Veuillez choisir l'affichage souhaite (1 to 6):
 1 - afficher un trajet
 2 - afficher tous les trajets
 3 - afficher le trajet le plus court
 4 - afficher le trajet le plus rapide
 5 - afficher l'arbre des trajets
 6 - quitter
```

```
Le trajet trouve est :
Paris -> Reims -> Lille -> 3h 38min / 350km
```

```
Les trajets trouves sont :
Paris -> Reims -> Lille -> 3h 38min / 350km
Paris -> Le-Mans -> Rennes -> Caen -> Amiens -> Lille -> 9h 55min / 946km
Paris -> Le-Mans -> Caen -> Amiens -> Lille -> 8h 10min / 773km
Paris -> Dijon -> Strasbourg -> Metz -> Reims -> Lille -> 12h 8min / 1208km
Paris -> Amiens -> Lille -> 3h 30min / 288km
```

```
Le trajet le plus court est :
Paris -> Amiens -> Lille -> 3h 30min / 288km
```

```
Le trajet le plus rapide est :
Paris -> Amiens -> Lille -> 3h 30min / 288km
```

```
L'arbre des trajets est :
Paris
|-> Reims
|   |-> Lille (3h 38min / 350km)
|-> Le-Mans
|   |-> Rennes
|   |   |-> Caen
|   |   |   |-> Amiens
|   |   |   |   |-> Lille (9h 55min / 946km)
|   |-> Caen
|   |   |-> Amiens
|   |   |   |-> Lille (8h 10min / 773km)
|-> Dijon
|   |-> Strasbourg
|   |   |-> Metz
|   |   |   |-> Reims
|   |   |   |   |-> Lille (12h 8min / 1208km)
|-> Amiens
|   |-> Lille (3h 30min / 288km)
```