UNIVERSITÉ DE LORRAINE | IUT Saint-Dié-des-Vosges

# SAE UE 1.1

## VOYAGEUR DE COMMERCE

RIGARD | RIVIERE–JOMBART

05/12/2022

# Démarche

## Structure du programme

Afin de traiter le problème du voyageur de commerce en C, nous avons décidé d'adopter une approche semi orientée objet(Utilisation des structs), Ceci nous permettra une implémentation épurée, efficace en mémoire et nous permet d'éviter des problèmes de désallocation en fournissant des fonctions adaptées pour désallouer correctement les structures créées.

Nos fonctions utilisant les structures fonctionnent toutes par références, nous passons à chaque fois un pointeur vers une structure et non la structure entière permettant une optimisation de la mémoire et du temps d'exécution en évitant des copies inutiles.

Afin d'améliorer la lisibilité de notre code et faciliter sa réutilisation, nous l'avons séparé en plusieurs fichiers. Chaque fichier contient l'implémentation de la structure et des fonctions lui étant liées.

L'arborescence du programme à été réfléchie afin de minimiser sa complexité et maximiser sa lisibilité, elle se compose comme suit : Les fichiers headers sont présent dans le dossier include tandis que les implémentation liées sont elles dans le dossier src, la séparation des headers et de l'implémentation permet d'améliorer la distinction entre les deux

## Fonctionnement du programme

Notre programme fonctionne d'une façon "orientée objet en C", si nous voulons aborder le problème, il est premièrement nécessaire de créer et d'allouer un board avec la fonction createBoard pernant une multitude de paramètre définissant le plateau, sa taille, sa largeur, son nombre de ville et le nom de ces dernières sont prises en compte.

Par la suite, il est nécessaire de générer des coordonnées aléatoires pour chacune des villes qui ont été allouées par createBoard, nous faisons donc appel à la fonction populateBoard, qui, à chaque ville va attribuer une coordonnée x et y aléatoires comprises respectivement entre la largeur et la hauteur du plateau. Chaque coordonnée attribuée à une ville est unique en conséquence, deux villes ne peuvent avoir la même position sur le plateau.

Nous pourrons par la suite afficher le tableau créé par la fonction createBoard, peuplée par populateBoard en appelant la fonction display board, qui par le biais du terminal représentera les villes par une croix.

Si nous voulons traiter le problème du voyageur de commerce, il est au préalable nécessaire de générer la matrice contenant distance entre toutes les villes afin d'optimiser l'algorithme en ne re-calculant pas à chaque fois la distance entre les villes. Pour réaliser cela, il faut faire appel à la fonction generateDistanceMatrix aprés avoir créé et peuplé le tableau, la matrice générée par cette dernière sera stockée dans la structure Board.

Nous arrivons à présent au cœur du problème, la recherche du chemin le plus court, pour calculer cela,  deux solutions sont possibles, la première consiste à calculer tous les chemins possibles et sélectionner le meilleur d'entre eux. C'est précisément ce que réalise notre algorithme récursif appelé par la fonction generatePossiblePaths, qui stockera dans le board la totalité des chemins possible, il faudra par la suite calculer la distance de tous ces chemins en appelant la fonction calculateAllPathsDistances qui stockera la distance de chaque chemin dans eux même. Nous pourrons ainsi par la suite récupérer le chemin le plus optimisé par la fonction getOptimizedRoute qui retourne le chemin le plus optimisé.

Une seconde approche à ce problème est réalisable, en partant de la ville de départ, on choisit à chaque fois la ville la plus proche. Avec cet algorithme nous ne recevons pas le chemin le plus optimisé, mais pas le chemin le moins optimisé non plus. Cependant même si cette approche n'est pas parfaite, elle reste bien moins gourmande en mémoire et en temps de calcul que la première solution. Nous avons implémenté cette solution et pour l'utiliser il suffit d'appeler la fonction getOptimizedRouteByDistance qui retournera le chemin le plus optimisé en suivant l'algorithme précédemment énoncé.

Pour afficher dans le terminal le résultat de ces deux solutions, il est possible, grâce à la fonction displayPath d'afficher sur un plateau donné un path. L'ordre de passage dans les villes est représenté par un numéro allant de zéro au nombre total de villes présentes dans le board.

Prenons à présent un exemple de cas d’utilisation, nous avons un voyageur de commerce devant se déplacer de la façon la plus optimisée entre huit villes, ce voyageur, muni de sa voiture électrique devra la recharger dans certaines villes car son autonomie est inférieure à la distance de la tournée complète. Pour optimiser le temps de recharge dans chaque ville afin que le voyageur puisse réaliser sa tournée, il est nécessaire de calculer le temps d'arrêt à chaque ville, cette optimisation est réalisé par la fonction optimizePathForVehicle qui prend en paramètre un véhicule caractérisé par son autonomie, sa recharge (en km/min) et son modèle, cette fonction prend également le temps maximal d'arrêt souhaité par ville. Cette fonction nous retourne alors un tableau dont l’indexage correspondra à celui des villes présentes dans le board, il ne suffit plus qu'à faire usage de ces valeurs.

## Difficultées rencontrées

La principale difficulté rencontré à la réalisation de ce programme à été la création de la fonction générant tous les chemins possibles, en effet son fonctionnement interne récursif nécessite un système gérant l’indexage dans la matrice des chemins, nous avons solutionné ce problème en utilisant une variable que nous avons nommé la divergence, gérant l’indexage, à chaque récursion, on recalcule la divergence en fonction de la précédente, cela nous permet d’obtenir in fine l’index du path dans lequel la fonction doit écrire la ville qu’il est en train de traiter.

La seconde difficulté que nous avons rencontré est l’importation circulaire, en effet certains header ont besoin de types définis dans un autre header et vice versa, cependant quand deux headers ou plus s'apportent mutuellement, une importation circulaire se crée, le programme ne peut donc pas compiler. Pour palier à ce problème, il est possible de créer des structures de “remplacement”, remplaçant les structures nécessaires, cela permet de créer la définition de fonction utilisant ces types sans avoir à faire à la directe implémentation de ces dernières.