TP 5 Algo Glouton Random

Utilisé notre programme :

Il faut tout d'abord clone notre code depuis notre github, puis le compile à l'aide de makefile. L'executable prends quelque arguments : `<ville_file> <colis_file> <export_file>` qui sont les emplacement des fichiers de input et export des resultats.

Question 1 :

Nous avons créé une classe nommée `DataBase` pour stocker toutes les données nécessaires à la résolution des problèmes `P1` et `P2`.

- Pour `P1`, nous stockons la capacité de la voiture (un entier) dans la variable `capacityCar`, le nombre de paquets disponibles (un entier) dans la variable `nbPackages`. Nous avons également inclus une `map` nommée `packages` qui prend en clé l'identifiant de chaque paquet et en valeur un tuple contenant la taille du paquet et son bénéfice de vente. Les données relatives aux solutions pour le problème `P1` sont stockées dans un vecteur nommé `solutionsColis` contenant des objets de la classe `Colis`.
- Pour `P2`, nous stockons le nombre de villes à visiter (un entier) dans la variable `nbCity`, un vecteur nommé `nameCity` contenant tous les noms de villes à visiter et une matrice nommée `dist` qui répertorie les distances entre chaque ville. Les données relatives aux solutions pour le problème `P2` sont stockées dans un vecteur nommé `solutionsTrajet` contenant des objets de la classe `Trajet`.

Toutes les variables de cette classe sont privées pour garantir une bonne encapsulation des données et les vecteurs et les map sont passés par référence pour éviter de copier de grandes quantités de données.

```
class DataBase {
    private:
        // P1
        int capacityCar;
        int nbPackages;
        map<int,tuple<int,int>>& packages;
        vector<Colis> &solutionsColis;
        //P2
        int nbCity;
        vector<string>& nameCity;
        vector<vector<int>>& dist;
        vector<Trajet> &solutionsTrajet;
};
```

Question 2:

Pour la méthode `import_ville()` on lit le fichier ligne par ligne à l'aide de la fonction `getline()`. La première ligne contient le nombre total de villes dans la base de données, qui est stocké dans la variable `nbCity`. Les lignes suivantes contiennent le nom de chaque ville, qui est ajouté à un vecteur `nameCity` à l'aide de la fonction `push_back()`. Les lignes suivantes contiennent les distances entre chaque ville, qui sont stockées dans une matrice `dist`.

La méthode `import_package` permet de lire un fichier contenant des informations sur les colis, et de les stocker dans une base de données de colis représentée par une structure de données de type `map`. La première ligne du fichier contient deux informations : la capacité totale du camion (`capacityCar`) et le nombre total de colis (`nbPackages`). Ces informations sont stockées dans des variables séparées.

Les lignes suivantes contiennent des informations sur chaque colis, y compris son identifiant (`id`), son poids (`weight`) et son volume (`volume`). Ces informations sont stockées dans une structure de données de type map nommée packages, où chaque paire clé-valeur représente un colis.

Lorsqu'une ligne contient plusieurs informations séparées par des espaces, la méthode utilise un objet `istringstream` pour extraire chaque information une par une, à l'aide de la fonction `>>`, qui permet de lire des données à partir d'un flux d'entrée.

Question 3:

- Pour le premier problème des colis, nous avons créé une classe nommée `Colis` qui stocke les informations nécessaires pour résoudre l'algorithme et stocker le résultat. Cette classe contient la capacité du colis (`int`), le bénéfice (`int`), la liste des packages à inclure dans la solution (`vector<int>&`) et une référence à une map contenant les informations détaillées de chaque package (`map<int,tuple<int,int>>&`).

```
class Colis {
private:
    int capacity;
    int benefice;
    vector<int>& solusPackages;
    map<int, tuple<int,int>>& packages;
};
```

- Pour le deuxième problème des trajets, nous avons également créé une classe nommée `Trajet` qui stocke les informations nécessaires pour résoudre l'algorithme et stocker le résultat. Cette classe contient la distance totale du trajet (`int`), la liste ordonnée des villes visitées (`vector<string>`), une liste des noms de toutes les villes disponibles (`vector<string>`) et une référence à une matrice contenant les distances entre chaque paire de villes (`vector<vector<int>>&`).

```
class Trajet {
private:
    int distance;
    vector<string> path;
    vector<string> villes;
    vector<vector<int>>& dist;
};
```

Question 4:

Algo glouton colis :

1. On commence par récupérer le nombre de colis à traiter et initialiser un tableau packing vide qui va stocker les couples (numéro du colis, ratio bénéfice/consommation de capacité).
2. Ensuite, on itère sur chaque colis et on calcule le ratio bénéfice/consommation de capacité pour chaque colis en divisant le bénéfice par la consommation de capacité. Ce ratio est stocké dans une variable temporaire tmp.
3. On ajoute le couple (numéro du colis, ratio) dans le tableau packing.
4. On trie le tableau packing en ordre croissant de ratio.
5. On itère sur chaque colis trié par ordre croissant de ratio et on ajoute les colis dans le colis courant tant que la capacité maximale n'est pas atteinte.
6. On incrémente le compteur i à chaque fois qu'on ajoute un colis au colis courant.
7. On continue à ajouter les colis jusqu'à atteindre la capacité maximale ou jusqu'à ce qu'il ne reste plus de colis.
8. À la fin de l'algorithme, on obtient la solution du problème qui est constituée des numéros des colis qui ont été ajoutés au colis courant, ainsi que du bénéfice total obtenu.

Algo glouton chemin :

L'algorithme glouton pour trouver le chemin fonctionne de la manière suivante :

1. On récupère l'indice de la ville de départ et les informations nécessaires à l'algorithme (dist, nbCity et nameCity).
2. On initialise un vecteur d'entiers contenant les index des villes non visitées.
3. On ajoute la ville de départ au chemin et on l'enlève de la liste des villes non visitées.
4. On répète les étapes suivantes pour chaque ville restante :
    - On parcourt toutes les villes non visitées pour trouver celle la plus proche de la ville précédente.
    - On ajoute la ville trouvée au chemin et on ajoute la distance entre cette ville et la précédente. On supprime également l'index de la ville visitée de la liste des villes non visitées.

5. On ajoute la ville de départ pour finir le chemin ainsi que la distance qui va avec.
6. On renvoie le chemin trouvé et la distance totale de ce chemin.

Question 5:

Algo glouton randomisé colis :

1. On prend en entrée un seed pour initialiser le générateur de nombres aléatoires.
2. On crée un tableau "packing" qui contiendra des couples de la forme (numéro du paquet, valeur de bénéfice/consommation de capacité).
3. Pour chaque paquet, on calcule la valeur bénéfice/consommation de capacité et on l'ajoute à "packing".
4. On trie le tableau "packing" en fonction de la valeur bénéfice/consommation de capacité, du plus petit au plus grand.
5. On mélange le tableau "packing" de manière aléatoire en échangeant deux éléments choisis au hasard plusieurs fois. Cela introduit une randomisation dans l'ordre de traitement des paquets.
6. On parcourt le tableau trié et aléatoirement mélangé, et on ajoute les paquets dans le colis tant que la capacité du colis n'est pas dépassée.
7. On calcule le bénéfice total et on ajoute les numéros des paquets sélectionnés dans le vecteur "solusPackages".

Algo glouton randomisé chemin :

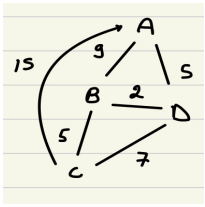
L'algorithme fonctionne de la manière suivant :

1. Il stocke dans un vecteur d'entier la liste des index des villes non visité.
2. Il va ensuite parcourir toutes les villes non visité afin de trouver les deux villes les plus proche de la ville précédente.
3. Un fois ces villes trouvés, il choisie l'une des deux villes de manière aléatoire..
4. Il ajoute la ville choisie au chemin et ajoute la distance entre cette ville et la précédente. Il supprime également l'index de la ville visité de la liste des ville non visité.
5. Une fois que toutes les villes ont été visité en répétant les étapes 2), 3) et 4), il ajoute la ville de départ afin de finir le chemin ainsi que la distance qui va avec.

Question 6:

Prenons l'exemple d'un voyageur de commerce qui doit visiter quatre villes A,B,C et D. La distance entre chaque ville est la suivante avec :

- A à B : 9km
- A à D : 5km

- B à C : 5km
- B à D : 2km
- C à D : 7km
- C à A : 15km 

Si nous commençons par la ville A et choisissons à chaque étape la ville la plus proche de la dernière ville visitée, nous obiendrons la tournée suivante :

- étape 1 : A->B (9km)
- étape 2 : B->C (5km)
- étape 3 : C->D (7km)
- étape 4 : D->A (5km) la longueur totale de cette tournée est de 26 km. Cependant, si nous avons commencé par la ville A et que nous avons réfléchi davantage à la meilleure tournée possible, nous pourrions obtenir une tournée plus courte. Par exemple :
- étape 1 : A->D (5km)
- étape 2 : D->B (2km)
- étape 3 : B->C (5km)
- étape 4 : C->A (15km) L longueur totale de cette tournée est de 27 km soit un kilomètre de plus que la première tournée. Cela montre que choisir la ville la plus proche à chaque étape ne garantit pas toujours la tournée la plus courte possible.

Question 7:

Les deux problèmes sont complètement indépendants l'un de l'autre donc les solutions optimales de P1 ne dépendent pas des solutions optimales de P2, et vice versa. Ce qui implique que pour obtenir le meilleur résultat possible, il est préférable de prendre la réplication pour chacun qui a donné le meilleur résultat.

Question 8:

Pour pouvoir lancer les résolutions des algorithmes en parallèle, on utilise les threads et dans chaque instance, ils ont accès à la classe `Database` qui a une méthode qui instancie la classe `Colis` ou `Trajet` avec les données nécessaires pour les méthodes d'algorithmes. On passe également une variable `i` qui est incrémentée et utilisée comme seed pour le random des solveurs. Après avoir résolu un problème avec une certaine seed, on met la solution dans la database. Avant de continuer le code, on attend que tous nos threads aient fini leurs calculs.

Nos resultat ne sont donc jamais lié entre eux.

Question 9 :

Afin de construire une nouvelle solution combinant les meilleures solutions obtenues pour chaque problème, nous avons utilisé deux fonctions `findBestColis()` et `findBestTrajet()` qui renvoient respectivement les meilleurs colis et trajets obtenus lors des réplications.

Nous avons ensuite utilisé la fonction `export_best_results(string namefile)` pour exporter les deux meilleures solutions dans un fichier texte. Cette fonction crée un fichier s'il n'existe pas déjà, puis y écrit les résultats de `findBestColis()` et `findBestTrajet()` séparés par une ligne grace à une implementation de l'operateur << qui peux écrire dans n'importe quelle std::output dont ofstream en fait partie.

Question 10 :

Pour exporter de la donnée dans deux fichiers textes, il est nécessaire d'utiliser la librairie `fstream` permettant de créer des fichiers texte et d'y insérer du texte. Chaque valeur de ces 2 fichiers est calculée dans la fonction `exportData()` avant le l'insérer dans le document. Cette fonction fait appelle à plein de petites fonctions faisant les calculs random :

- `randomCityNumber()` génère un nombre de villes aléatoires entre 3 et 6 (pas plus pour rester dans le raisonnable).
- `randomVecteurVille(int)` permet de créer le nom des villes aléatoirement comme fait dans le TDP précédent.
- `randomDist(int)` permet de créer la matrice des distances aléatoirement avec des distances ne dépassant pas 100 km (choix personnel).
- `randomCarCapacity()` génère la capacité du véhicule allant de 20 à 100.
- `randomNbPackages(int,int)` génère aléatoirement le nombre de paquets disponibles dans l'entrepôt. Nous avons décidé de prendre un nombre supérieur de colis que de villes. Ce nombre peut aller jusqu'à un maximum. Dans notre cas nous avons mis 10 à but indicatif. Mais il est totalement possible de le modifier à la hausse.
- Enfin , `randomMapPackage(int)` remplit la map avec le numéro du colis, la place qu'il prend (entre 2 et 17) et le bénéfice qu'il nous apporte (entre 2 et 15).

Il suffit juste d'appeler la fonction `exportData()` dans le `main()` pour effectuer cette tâche.

Question 11 :

Nombre de villes	Plus courte distance (Brute force)	Plus courte distance en 3 essais (Heuristique)	Gap
4	103	103	0%
6	235	288	22.553%

Nombre de villes	Plus courte distance (Brute force)	Plus courte distance en 3 essais (Heuristique)	Gap
10	300	345	15%

Nous pouvons observer une différence entre les deux sorties de programmes. En revanche, le temps d’exécution du programme de Brute force pour un nombre de villes importantes (comme pour 10) augmente énormément. À l'inverse, il n'y a pratiquement pas de différence au niveau du temps d'exécution pour l'algorithme heuristique. De plus, nous pouvons exécuter plusieurs fois l'algorithme heuristique afin de possiblement obtenir une solution plus précise.

Question 12 :

Déjà avec les petits input mon PC throw des erreur comme `terminate called after throwing an instance of 'std::bad_alloc` et `terminate called after throwing an instance of 'std::bad_array_new_length'` ce qui indique qu'il n'arrive pas à alloué de la mémoire. Ca peux venir du fait que notre code est extremement mal optimisé avec des alocation de mémoire innutile ou alors que l'ISEN nous a founie des PC pas super. On ne le saura surement jamais.