Sujet TD&P n°5 - Séances 7 & 8 Réplications d'un Algorithme Glouton Randomisé

- Q1.

Pour le problème (P1) :
Nous devons créer une structure ou une classe véhicule, qui va contenir la capacité du véhicule et notre structure qui contient nos objets. Pour la structure/classe objets, nous allons devoir stocker l'index, la consommation de l'objet ainsi que son bénéfice.
Cela se traduit par l'utilisation des objets "class", "struct" et "vector".

Pour le problème (P2) :
Nous devons créer une classe Ville, qui va contenir le nombre de villes, le noms des villes et la matrice des distances.
Il nous faut créer une donnée traduisant la matrice des distances entre chaque couple de villes. L'utilisation d'un vecteur de vecteur d'entiers semble le plus adéquat afin de traduire cette matrice. Comparé à un array, il ne faut pas définir à l'avance la taille d'un vecteur.
L'indice de la ville dans le vecteur nomVilles va donner son indice pour la matrice de distance. Exemples : indice Lille = 0, indice Lens = 1, distance entre Lille et Lens = matriceDistance[0][1].

- Q2.

Dans ce code, on vérifie tout d'abord que bons arguments ont été passés à l'exécution du script (les noms des fichiers à lire). Si ce n'est pas le cas, le programme affiche un message d'erreur et se termine.

```
   // Mauvais paramètres de lancement
   if (argc != 4 && argc != 3 && argc != 7)
   {
       cerr << "Formats corrects:" << endl;
       cout << "./TP5 graine fichier_colis.txt fichier_villes.txt" <<
endl;
       cout << "./TP5 -1 fichier_resultat.txt fichier_villes.txt" <<
endl;
       cout << "./TP5 -2 capacite nombre_obj p1.txt nombre_villes
p2.txt" << endl;
       return EXIT_FAILURE;
   }
```

-1 pour générer notre meilleure solution et -2 pour créer nous même des exemples.

Ensuite, on ouvre les deux fichiers en mode lecture, et on initialise les variables vides pour stocker leur contenu.

Pour (P1) :

```
string pathColis = argv[2];
Colis colisObj(pathColis);
```

Pour (P2) :

```
string pathVilles = argv[3];
Villes villesObj(pathVilles);
```

On lit ensuite chaque ligne des deux fichiers avec une boucle for, et on ajoute chaque ligne à la variables correspondante.
Enfin, on ferme les deux fichiers, et on affiche le contenu des listes à titre de vérification.

Voir le code des classes dans le dossier classes, colis.cpp (P1), ou villes.cpp (P2).

- Q3.

Pour (P1) : la structure la plus adaptée pour résoudre ce problème serait une boucle Pour Faire. On parcourt le vecteur d'objets et on calcul le bénéfice et la consommation en fonction de la place disponible.

Algorithme issu de la classe colis :

Fonction meilleurRangement (entrée: fonction d'une classe , sortie : une solution):
    solution = 0, consoTotal = 0, benefTotal = 0
    trier le vecteur en fonction du ratio
    Pour tous les éléments elem dans le vecteur **Faire** :
        obj1 = elem
        obj2 = elem + 1
        sol = obj1
        Si obj1.conso + consoTotal > capacité et obj2.conso + consoTotal > capacité **Faire** :
            continuer
        Fin si
        Sinon si obj1.conso + consoTotal > capacité **Faire** :
            sol = obj2
        Fin sinon si
        Sinon si obj1.conso + consoTotal <= capacité et obj2.conso + consoTotal <= capacité **Faire** :
            sol = choisir aléatoirement obj1 ou obj2
        Fin sinon si
        solution += sol
        consoTotal += sol.conso
        benefTotal += sol.benef

    Fin Pour
    Retourner solution

Pour (P2) : la structure la plus adaptée pour résoudre ce problème serait une boucle Tant que Faire. Tant qu'il que la taille du chemin est plus petite que le nombre de villes, on continue de parcourir les villes en cherchant le plus court chemin possible.

Algorithme issu de la classe ville :

Fonction meilleurChemin (entrée: fonction d'une classe , sortie : une solution):

```
distanceTotale = 0, solDist = 0
meilleurDist = 999, oMeilleurDist = 999
meilleurVille, oMeilleurVille, solVille
début = nom de ville aléatoire

Tant que la taille de la solution < nombreVille Faire :
    Pour tous les éléments ville dans le vecteur de ville Faire :
        distance = avoir distance entre début et ville
        Si distance <= meilleurDist Faire:
            oMeilleurDist = meilleurDist
            oMeilleurVille = meilleurVille
            meilleurDist = distance
            meilleurVille = ville
        Fin si
    Fin Pour
    solVille = choisir aléatoirement entre oMeilleurVille ou meilleurVille
    solDist = distance de solVille choisie
    distanceTotale += avoir distance entre début et solVille
    solution += solVille
    début = solVille
    meilleurDistance = 999

Fin Tant que
Retourner solution
```

- Q4.

Voir le code dans le dossier classes, colis.cpp (P1), ou villes.cpp (P2).
Pour le problème (P1) voir la fonction getBestShipment().
Pour le problème (P2) voir les fonctions : getDistance() et getBestPath().

Nous utilisons les algorithmes de la Q3 pour développer les méthodes de résolution pour (P1) et (P2).
Pour (P1), le problème du sac à dos :

- Tri des éléments : Les éléments sont triés par ordre décroissant de leur rapport bénéfice/consommation.
- Sélection des éléments : L'algorithme sélectionne l'élément ayant le meilleur rapport bénéfice/consommation de la capacité restante du véhicule à chaque étape. Cela garantit que l'élément sélectionné est le meilleur choix parmi ceux disponibles à ce stade.

- Construction de la solution : La construction de la solution se poursuit jusqu'à ce que la capacité maximale du véhicule soit atteinte ou que tous les éléments aient été examinés.

Pour (P2), le plus proche voisin :
- Choix ville de départ : Le choix est effectué aléatoirement.
- Sélection ville suivante : à chaque étape, l'algorithme cherche la ville la plus proche de la ville actuelle qui n'a pas encore été visitée et la visite. Ce processus est répété jusqu'à ce que toutes les villes aient été visitées.

- Q5.

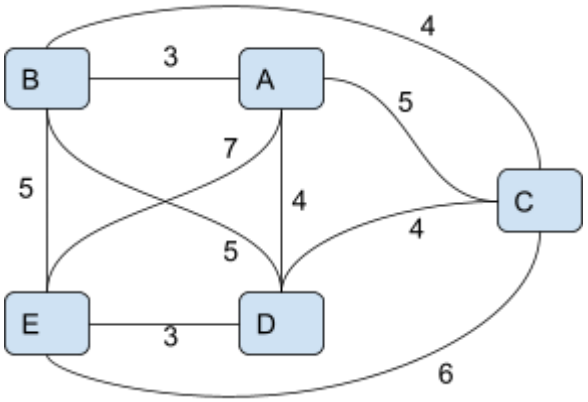
Voir les modifications de code dans le dossier classes.

En changeant la graine du Générateur, nous observons bien l'obtention de solutions différentes.
Par exemple :
seed : 5, bénéfice : 19, consommation : 35, objets n° : 8 4 2, villes : Lille Seclin Roubaix Lens, distance totale : 90
seed : 6, bénéfice : 20, consommation : 40, objets n°: 8 4 6 5, villes : Lille Roubaix Seclin Lens, distance totale : 75
seed : 7, bénéfice : 19, consommation : 35, objets n°: 8 4 2, villes : Seclin Lille Lens Roubaix, distance totale : 100

- Q6.

Nous allons utiliser l'algorithme "plus proche voisin" pour construire une tournée. Nous partons de la ville A et sélectionnons à chaque étape la ville la plus proche de la dernière ville visitée. Nous supposons que les distances sont données par les nombres sur les arcs.

Considérons l'exemple suivant :



Étape 1 : Nous commençons à la ville A. La ville la plus proche de A est la ville B. Nous nous rendons donc à B.
Étape 2: Nous sommes dans la ville B. La ville la plus proche de B est la ville C. Nous nous rendons donc à C.
Étape 3: Nous sommes dans la ville C. La ville la plus proche de C est la ville D. Nous nous rendons donc à D.

Étape 4: Nous sommes dans la ville D. La ville la plus proche de D est la ville E. Nous nous rendons donc à E.
Étape 5: Nous sommes dans la ville E. La ville la plus proche de E est la ville A. Nous nous rendons donc à A.
Étape 6: Nous sommes dans la ville A. Nous avons fini la tournée.

La longueur totale de la tournée est de 21.

Cependant, cette tournée n'est pas la plus courte possible. En effet, la tournée ACDEBA a une longueur totale de 20, qui est inférieure à la tournée obtenue par l'algorithme "plus proche voisin". Ceci montre que cet algorithme ne garantit pas de trouver la tournée la plus courte dans tous les cas.

- Q7.

Les problèmes (P1) et (P2) sont indépendants l'un de l'autre. Les solutions de (P1) et (P2) suivent un objectif distinct, soit respectivement :

- maximiser le bénéfice alors que chaque produit consomme une certaine capacité du véhicule qui n'est pas proportionnelle à leur bénéfice,
- minimiser les coûts, i.e., la quantité d'électricité consommée par les véhicules.

Il est préférable de trier à l'exécution les solutions renvoyées à chaque réplication de (P1) et (P2) afin de construire une solution la plus optimale.

- Q8.

Voir le code dans le fichier "resultat.txt". La meilleure solution (qu'on a construite) est dans "output.txt".
Un script python se charge d'appeler notre programme avec différentes graines à chaque fois.

- Q9.

Voir le code dans le dossier classes, colis.cpp (P1), ou villes.cpp (P2).
On lit ligne par ligne le contenu de resultat.txt et on compare chaque solution avec celle qu'on avait précédemment. Si elle est meilleure, on remplace notre solution par celle-ci. Ceci permet de construire la solution au fur et à mesure de la lecture du fichier et nous permet d'avoir un très grand nombre de réplications pouvant être traitées. Pour (P1) une solution est meilleure si elle a un meilleure bénéfice et pour (P2) une solution est meilleure si la distance de la tournée est plus courte. Nous avons des structures pour contenir les solutions de (P1) et (P2):

```
typedef struct SolColis
{
   vector<objet> objets;
   int benef = INT32_MIN;
   int conso = INT32_MAX;
} solColis;
typedef struct SolVille
```

```
{
   vector<string> tournee;
   int distanceTotale = INT32_MAX;
} solVille;
```

Sorties de la forme: seed benef conso [objets] [villes] distance_totale
Une ligne = une réplication

contenu de resultat.txt:
1 19 36 8 9 5 2 15 8 6 12 6 Lens Lille Seclin Roubaix 80
2 20 39 1 8 5 4 11 6 6 12 6 5 8 3 Lens Seclin Lille Roubaix 55
3 19 34 1 8 5 4 11 6 2 15 8 Lens Seclin Lille Roubaix 55
[...]
contenu de output.txt:
0 **22 40** 1 8 5 8 9 5 4 11 6 6 12 6 **Lens Seclin Lille Roubaix 55**
-> On a donc un bénéfice de 22 pour une consommation de 40. La tournée idéale semble être Lens, Seclin, Lille, Roubaix et fait 55 km. Il s'agit de la meilleure solution de (P1) et la meilleure solution de (P2)

- Q10.

Voir le code dans le dossier classes, colis.cpp (P1), ou villes.cpp (P2), fonctions genVilles et genShipment.

Nous générons de nouveaux cas :
Pour le problème (P1), nous commençons par générer le nombre de colis et la capacité que nous allons utiliser. Ensuite, pour chaque colis on génère aléatoirement la capacité qu'il utilise et son bénéfice. Pour finir, on écrit toutes les informations dans un fichier txt sous la forme requise. Nous avons fixé comme limite :

- capacité max : capacité voiture, min : 1
- bénéfice max : 15, min : 3

Pour le problème (P2), nous commençons par générer aléatoirement les noms des villes. On génère le nombre de caractères pour le nom de la ville, puis on choisit les lettres à mettre dans le nom. La première lettre sera un caractère ascii en majuscule (choisi aléatoirement entre 65 et 91), et le reste seront des caractères ascii en minuscule (choisi aléatoirement entre 97 et 123). Après, il faut générer les distances entre les villes. Nous créons une matrice de taille le nombre de ville X le nombre de ville. Nous générons la moitié de la matrice que nous mettons pour l'autre moitié (en effet dist(villeA, villeB) = dist(villeB, villeA)).
Pour finir, on écrit les informations dans un fichier txt sous la forme requise.
Nous avons fixé comme limite :

- nombre de caractère par nom max : 9, min : 4
- distance entre les villes max : 50, min : 10

- Q11.

On importe le code du TP précédent pour comparer les résultats (méthode brute force). Nous obtenons:

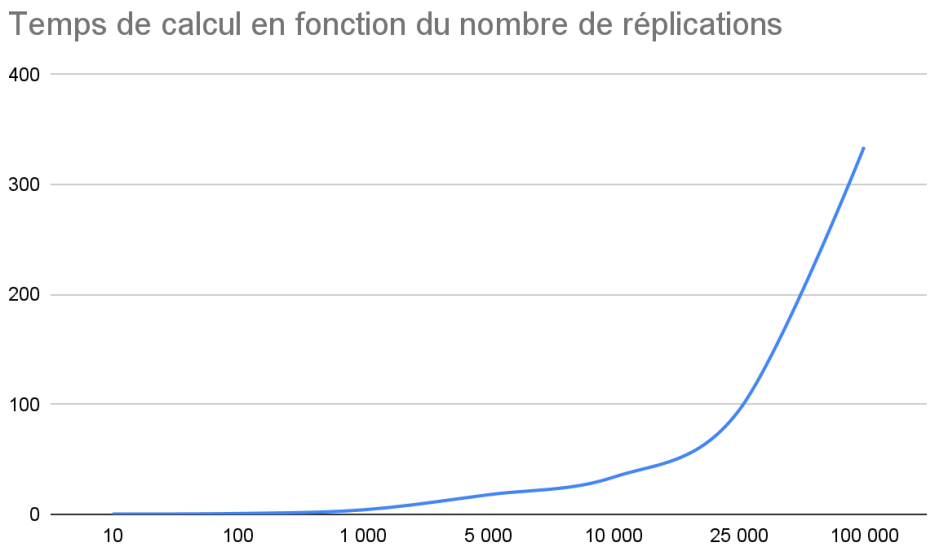
Nombre de villes	distance heuristique	distance extacte	écart relatif (%)
4	75	75	0
5	80	80	0
6	116	116	0
7	119	102	14,28571429
8	133	121	9,022556391
9	182	164	9,89010989
		Écart moyen:	5,533063428

On constate un écart moyen de 5,5% entre la méthode heuristique et celle exacte. Cela se traduit par une tournée plus longue de 5,5% en moyenne pour notre méthode heuristique nous fournissant la solution combinée de 10 réplications.

- Q12.

On commence par rebuild le programme en release afin d'en tirer les meilleures performances. Ensuite, on va générer 10, 100, 1000, 100000 réplications pour 9 villes et enregistrer le temps que cela prend. Notez que l'on calcule la meilleure distance heuristique ainsi que la distance exacte dans ce test afin d'ajouter en complexité.

Nombre de villes	nombre de réplications	temps (s)
9	10	0,294
9	100	0,779
9	1 000	4,087
9	5 000	17,826
9	10 000	33,815
9	25 000	94,383
9	100 000	334,147



On peut voir dans ce graphique que notre temps de calcul augmente drastiquement en fonction du nombre de réplications.

Bonus :

Solution du sac à dos déterministe.

Il n’est pas systématique d’obtenir un chargement maximisant le bénéfice en prenant uniquement le meilleur candidat à chaque étape de la construction d’une solution au problème P1. Preuve :

Supposons que nous ayons les produits suivants avec leurs bénéfices et consommations respectives :

Produit	bénéfice	consommation	bénéfice/consommation
P1	13	5	2,6
P2	8	3	2.667
P3	15	4	3.75
P4	4	2	2

Supposons également que la capacité du véhicule soit de 10.
En utilisant la stratégie gloutonne, nous sélectionnons d'abord le produit P3 car il a le meilleur rapport bénéfice/consommation de 3.75. La capacité restante est alors de 6 (10-4).
Nous passons ensuite au produit P2 avec un ratio de 2.67. On peut l'ajouter car sa consommation est de 3 et il reste 6 de capacité. La capacité restante est alors de 3 (6-3).
Ensuite, nous sélectionnons le produit P1 avec un rapport bénéfice/consommation de 2.6. Nous ne pouvons pas l’ajouter car il a une consommation de 5 et il ne reste que 3.
Nous passons donc au dernier produit P4 avec un ratio de 2. Nous pouvons l'ajouter car sa consommation est de 2.
La solution optimale selon cet algorithme, est de prendre le produit avec le meilleur ratio bénéfice/poids est donc d'ajouter les produits P2, P3 et P4 pour un bénéfice total de 27 et de consommation 9.

Cependant, si on avait choisi la stratégie de prendre le produit avec le meilleur bénéfice sans tenir compte du poids, la solution optimale serait d'ajouter les produits P3 et P1 pour un bénéfice total de 28 et de consommation 9.