Rapport de TP d’algorithmique avancée
Sujet TD&P n°5

Equipe 19

BAUM Matthieu / CACHEUX Nolan / HELLEBOID Hugo / LONGATTE Marc-Antoine

L’ensemble des réponses et des instructions d’exécutions pourront être aussi retrouvé dans les dossiers de chacune des questions.

JUNIA Grande école d’ingénieurs

Question 1 :

Pour enregistrer l'ensemble des données des deux problèmes (P1) et (P2), il faut définir des structures de données appropriées pour chaque problème.

Pour le problème(P1), il est nécessaire de stocker la capacité du véhicule, le nombre d'objets disponibles.
Pour chaque objet, son nom / index, sa consommation de capacité et son bénéfice.
On peut ainsi définir une structure de données pour stocker ces informations, par exemple :

```
struct P1Data {
    int capacity; // Capacité du sac à dos
    int num_objects; // Nombre d'objets disponibles
    std::vector<std::tuple<int, int, int>> objects; // (nom/index, consommation, bénéfice)
};
```

La capacité et le nombre d'objets sont stockés sous forme d'entiers simples.
les informations pour chaque objet sont ensuite stockées dans un vecteur de tuples, chaque tuple contenant le nom/index, la consommation et le bénéfice de l'objet.

Pour le problème (P2), il est nécessaire de stocker les informations suivantes :
le nombre de villes, les noms/index des villes et la matrice des distances/consommations d'énergie électrique associées aux différents déplacements possibles entre les villes.
On peut définir une structure de données pour stocker ces informations, par exemple :

```
struct P2Data {
    int num_cities; // Nombre de villes
    std::vector<std::string> city_names; // Noms des villes
    std::vector<std::vector<int>> distances; // Matrice des distances
};
```

Le nombre de villes est stocké sous forme d'entier simple, les noms/index des villes sont stockées dans un vecteur de chaînes de caractères, et la matrice des distances est stockée dans un vecteur de vecteurs d'entiers.

Question 2 :

Ici, nous ouvrons les fichiers de données et lisons les informations nécessaires pour remplir nos structures de données définies précédemment.
Nous vérifions également si les fichiers ont été correctement ouverts. Enfin, nous affichons les données pour vérifier si tout s'est bien passé.

Le programme prend les noms de fichiers en tant que paramètres d'exécution, de sorte qu'il peut être utilisé pour tout fichier ayant la même structure.
On exécute donc les fonctions via le terminal en tapant les instructions suivantes :

cd out/build/x64-debug/TP/Q2

./Q2.exe 5colis30capacite.txt 4villes.txt

Question 3 :

Les problèmes P1 et P2 peuvent être résolus à l'aide d'algorithmes gloutons, qui cherchent à optimiser la solution à chaque étape, sans prendre en compte les conséquences à long terme.

Dans le cas du problème P1, l'objectif est de remplir un camion avec des produits de manière à maximiser le bénéfice tout en respectant la capacité du camion.
Pour cela, on peut trier les produits restants par rapport bénéfice/capacité, puis insérer le produit ayant le meilleur rapport dans le camion.
Tant qu'il y a suffisamment de capacité. Répétez ce processus jusqu'à ce que tous les produits soient insérés. Il est important de noter que cette méthode ne garantit pas la solution optimale, mais elle permet de trouver une solution rapidement.

Dans le cas du problème P2, l'objectif est de trouver le chemin le plus court pour traverser toutes les villes en visitant chacune une seule fois.
Pour cela, on peut commencer par choisir une ville de départ, puis à chaque itération, ajouter la ville la plus proche de la dernière ville ajoutée.
On répète ensuite ce processus jusqu'à ce que toutes les villes soient visitées. Cette méthode est également une solution gloutonne qui ne garantit pas la solution optimale, mais elle est rapide et permet de trouver une solution acceptable.

Il est important de choisir des structures de données efficaces pour stocker les produits et les villes, afin de garantir des temps de calcul raisonnables.
Des tableaux ou des vecteurs peuvent être utilisés pour stocker les produits et les villes, ainsi que leur position.

En ce qui concerne la complexité des algorithmes, la complexité de l'algorithme pour P1 dépend de l'efficacité du tri initial des produits.
Si les produits sont triés efficacement, la complexité de l'algorithme est O(nlogn) en raison du tri initial, où n est le nombre total de produits.

La complexité de l'algorithme pour P2 est de O(n^2) car il est nécessaire de parcourir la liste de villes restantes à chaque étape pour trouver la ville la plus proche.

Voici les algorithmes en pseudo-code pour les deux problèmes :

Algorithme pour P1 :

Entrée : un tableau de produits avec leur bénéfice et leur capacité, et la capacité du camion
Sortie : un tableau de produits à charger dans le camion

Trier les produits restants par rapport bénéfice/capacité

Tant que la capacité du camion n'est pas atteinte :
    Insérer le produit ayant le meilleur rapport bénéfice/capacité
Retourner le tableau des produits à charger dans le camion

Algorithme pour P2 :

Entrée : une liste de villes avec leurs coordonnées
Sortie : un tableau de villes dans l'ordre à visiter

Choisir une ville de départ

Ajouter cette ville au tableau de villes à visiter

Tant qu'il reste des villes à visiter :
    Trouver la ville la plus proche de la dernière ville ajoutée
    Ajouter cette ville au tableau de villes à visiter

Retourner le tableau de villes à visiter dans l'ordre

Pour P1, nous avons besoin de trier les produits par rapport au rapport bénéfice/capacité et insérer le produit avec le meilleur rapport jusqu'à ce que la capacité maximale du camion soit atteinte.
Nous pouvons stocker les produits dans un vecteur et trier le vecteur avec une fonction de comparaison personnalisée.

Pour P2, nous avons besoin de trouver la ville la plus proche de la dernière ville ajoutée à chaque itération.
Nous pouvons stocker les villes dans un tableau et utiliser une fonction de comparaison personnalisée pour trier les villes en fonction de leur distance par rapport à la dernière ville ajoutée.

Question 4 :

Pour P1, nous avons besoin de trier les produits par rapport au rapport bénéfice/capacité et insérer le produit avec le meilleur rapport jusqu'à ce que la capacité maximale du camion soit atteinte.
Nous pouvons stocker les produits dans un vecteur et trier le vecteur avec une fonction de comparaison personnalisée.

Pour P2, nous avons besoin de trouver la ville la plus proche de la dernière ville ajoutée à chaque itération.
Nous pouvons stocker les villes dans un tableau et utiliser une fonction de comparaison personnalisée pour trier les villes en fonction de leur distance par rapport à la dernière ville ajoutée.

Une des approches possibles pour le résoudre est d'utiliser la méthode de la recherche exhaustive (brute-force) qui consiste à tester toutes les permutations possibles des villes, puis de trouver la permutation qui donne la plus petite distance totale. Cette méthode est faisable pour des petits nombres de villes, mais pour un grand nombre de villes, elle est trop coûteuse en temps de calcul.

Une autre approche consiste à utiliser une heuristique, telle que l'algorithme du plus proche voisin, qui consiste à partir d'une ville de départ, puis à sélectionner à chaque étape la ville la plus proche, jusqu'à ce que toutes les villes soient visitées. Cette méthode donne une solution approximative et peut être implémentée en utilisant une liste de villes à visiter, une fonction de calcul de distance et une fonction de comparaison personnalisée pour trier les villes en fonction de leur distance par rapport à la dernière ville ajoutée.

Question 5 :

On peut voir au début du main dans le cpp (ligne 258) les deux lignes de codes suivantes :
int seed = 3; // valeur de la graine
std::default_random_engine generator(seed); // initialisation de la graine
Si l'on change la graine du générateur, les résultats seront différents.

Question 6 :

Considérons l'exemple suivant pour illustrer que prendre systématiquement la ville la plus proche à chaque étape ne garantit pas toujours la tournée la plus courte :
Supposons que nous ayons 5 villes (A, B, C, D, E) avec les distances suivantes :

	A	**B**	**C**	**D**	**E**
A	0	1	5	4	3
B	1	0	3	2	4
C	5	3	0	6	7
D	4	2	6	0	8
E	3	4	7	8	0

Si nous commençons à la ville A et prenons systématiquement la ville la plus proche à chaque étape, nous obtiendrons la tournée suivante :

A -> B -> D -> E -> C -> A
Ce qui donne une distance totale de 18.
Cependant, si nous prenons aléatoirement parmi les deux paires de villes les plus proches à chaque étape, nous pourrions obtenir une tournée plus courte. Par exemple, nous pourrions obtenir la tournée suivante :

A -> B -> C -> E -> D -> A
Ce qui donne une distance totale de 16.

Ainsi, en prenant des choix aléatoires, nous avons la possibilité de trouver des tournées plus courtes que si nous avions pris systématiquement la ville la plus proche à chaque étape. Cela souligne l'importance de l'algorithme glouton randomisé pour le problème du voyageur de commerce.

Question 7 :

Il est préférable de directement trier les solutions de chaque réplication de l'algorithme glouton randomisé plutôt que de reconstituer une solution à partir des meilleurs résultats de (P1) et (P2) pris séparément pour chaque réplication.
En effet, la recherche de la meilleure solution en combinant les résultats de deux problèmes différents peut être complexe et potentiellement imprécise, tandis que le tri des solutions obtenues par l'algorithme glouton randomisé est simple et efficace.

Question 8 :

En changeant la graine, on constate que les résultats sont différents.
Question 9 :

Pour le problème P1, la meilleure solution parmi toutes celles générées que nous avons choisi est celle qui a le meilleur résultat pondéré. (coefficient de 0.5 pour le bénéfice et de 0.5 pour l'énergie consommée.)
On peut donc sélectionner la solution ayant le résultat pondéré le plus élevé parmi toutes les solutions générées et afficher ainsi le bénéfice total ,la consommation d'énergie et la composition de la solution en conséquence

Pour le problème P2, la meilleure solution est celle qui a la distance minimale.
On peut donc sélectionner la solution ayant la distance la plus faible parmi toutes les solutions générées et afficher le chemin en conséquence.
Q9 est un affichage en console de Q8 dans le fichier.txt afin de simplifier le travail lors de la récupération des solutions (P1) et (P2)

Instruction à faire pour tester le code :
- Ajoute les deux fichiers 4villes.txt et 5colis30capacites.txt dans le build puis dans le terminal :
- cd out/build/x64-debug/TP/Q9
- ./Q9.exe 5colis30capacite.txt 4villes.txt

Question 10 :
Le code fourni génère aléatoirement les données pour les problèmes P1 et P2, puis les écrit dans des fichiers.
Lors de la génération des données aléatoires pour les deux problèmes P1 et P2, il est important de réfléchir aux limites à fixer pour créer des situations relativement sensées.
Pour le problème P1, les limites à prendre en compte peuvent être les suivantes :
- Le nombre d'objets doit être suffisamment petit pour que le problème reste raisonnable à résoudre (par exemple, entre 5 et 20).
- Le profit maximum des objets doit être inférieur ou égal à la somme des énergies maximales des objets, afin d'assurer qu'il est possible de ramasser tous les objets.
- L'énergie maximale des objets ne doit pas être trop grande pour éviter des situations où un seul objet permet de remplir le sac à dos.
- Les profits et énergies des objets doivent être suffisamment variés pour que la solution ne soit pas triviale, mais pas trop dispersés pour éviter des cas extrêmes.

Pour le problème P2, les limites à prendre en compte peuvent être les suivantes :
- Le nombre de villes doit être suffisamment petit pour que le problème reste raisonnable à résoudre (par exemple, entre 5 et 20).

- Les distances entre les villes doivent être suffisamment petites pour que toutes les villes soient connectées, mais pas trop proches pour éviter des situations où la solution est triviale.

Dans le code fourni, les limites choisies pour la génération aléatoire sont assez simples :
Les profits des objets sont compris entre 1 et 10, les énergies entre 1 et 20 pour P1, et les distances entre 1 et 100 pour P2.
Ces limites peuvent être modifiées en fonction des besoins spécifiques de l'application considérée.
Il est également possible de fixer des limites sur la taille des instances générées pour éviter des temps de calcul trop longs.
Par exemple, pour le problème P2, il peut être judicieux de limiter le nombre de villes à quelques dizaines plutôt que quelques milliers.

Question 11 :
En utilisant le fichier p2_data.txt généré à la Q10 pour 10 villes :
On génère plusieurs fois Q10.exe afin de créer de nouvelles instances du fichier p2_data.txt
Pourcentages de différence entre la solution optimale (BrutForce) et l'heuristique (solveP2) après de multiples exécutions : 30.30%, 11.97%, 18.23%, 15.23%
Ces résultats montrent que l'heuristique donne des solutions qui sont généralement proches de la solution optimale obtenue par la méthode de BrutForce,mais qu'il existe toujours une certaine marge d'erreur.
Plus le nombre de villes à visiter est élevé, plus cette marge d'erreur a tendance à augmenter, ce qui est normal car la complexité de l'algorithme BrutForce augmente de manière exponentielle avec le nombre de villes.
L'utilisation d'une heuristique peut donc être très utile pour résoudre des problèmes complexes avec un temps d'exécution raisonnable.

Question 12 :
En utilisant le fichier p2_data.txt généré à la Q12_FICHIERS_EXE pour générer un grand nombre de villes et un grand nombre d'éléments potentiellement transportable :
Ouvrir un terminal depuis Examples-main ; cd out/build/x64-debug/TP/Q12_FICHIERS_EXE
./Q12_FICHIERS_EXE.exe
Le fichier que nous allons désormais utiliser pour Q12 a maintenant été généré.

Un benchmark est le fait de mesurer la performance et la capacité d'un algorithme en comparant ses performances à d'autres.
Dans le contexte de cette question, la génération de situations à grande échelle pour les problèmes P1 et P2 est utilisée pour tester la capacité de l'ordinateur à résoudre ces problèmes en temps raisonnable.

Pour réaliser un benchmark, nous devons d'abord générer des instances de problèmes P1 et P2 de grande taille, en utilisant des données aléatoires pour simuler des situations réelles.
Ensuite, nous devons exécuter les algorithmes de résolution (BrutForce et heuristique) pour chaque instance générée et mesurer le temps d'exécution pour chacun.

Les résultats obtenus peuvent être présentés sous forme de tableau ou de graphique, où nous pouvons voir les temps d'exécution pour chaque instance de problème P1 et P2, ainsi que la taille de chaque instance. Cela nous permettra de voir comment les temps d'exécution varient en fonction de la taille des instances de problème, ce qui nous donnera une idée de la puissance de l'ordinateur.

En plus des temps d'exécution, nous pouvons également mesurer la précision de l'algorithme heuristique en calculant la différence entre sa solution et la solution optimale trouvée par l'algorithme BrutForce. Cela nous permettra de voir si l'heuristique fournit des solutions précises même pour les instances de problème de grande taille.

En résumé, le benchmark nous permet de tester la capacité de l'ordinateur à résoudre des problèmes P1 et P2 à grande échelle,

BONUS :

N et M candidats :

Pour répondre à cette question, nous allons ajouter des paramètres N et M aux fonctions de résolution des problèmes P1 et P2, respectivement, pour leur permettre de prendre en compte le nombre de candidats à considérer lors de la construction des solutions.

Tout d'abord, nous allons modifier la fonction solveP1 pour prendre en compte le nombre de candidats N. Au lieu de considérer uniquement les deux objets les plus bénéfiques à chaque étape, nous allons maintenant considérer les N objets les plus bénéfiques.

Nous allons ensuite sélectionner un sous-ensemble de ces N objets qui peuvent être inclus dans le sac à dos en fonction de leur consommation.
Nous allons répéter cette étape jusqu'à ce que le sac à dos soit plein ou qu'il n'y ait plus d'objets disponibles.

Voici la fonction solveP1 modifiée avec le paramètre N :

```
int solveP1(P1Data& data, int N) {
    int capacity = data.capacity;
    int num_objects = data.num_objects;
    vector<pair<int, int>> objects = data.objects;
    int total_benefit = 0;

    // Tri des objets par rapport à leur ratio bénéfice/consommation
    sort(objects.begin(), objects.end(), [](auto& left, auto& right) {
        return ((double)left.second / left.first) > ((double)right.second / right.first);
    });

    // Ajout des objets dans le sac à dos
    while (capacity > 0 && !objects.empty()) {
        // Sélection des N objets les plus bénéfiques
        vector<pair<int, int>> candidates(objects.begin(), objects.begin() + min(N, num_objects));

        // Tri des candidats par ordre de consommation croissante
        sort(candidates.begin(), candidates.end(), [](auto& left, auto& right) {
            return left.first < right.first;
        });
```

```
        // Sélection d'un sous-ensemble de candidats pouvant être inclus dans le sac à dos
        int i = 0;
        while (capacity >= candidates[i].first && i < candidates.size()) {
            total_benefit += candidates[i].second;
            capacity -= candidates[i].first;
            i++;
        }

        // Suppression des objets déjà ajoutés
        objects.erase(objects.begin(), objects.begin() + min(i, (int)objects.size()));
        num_objects -= i;
    }

    return total_benefit;
}
```

Maintenant, nous allons modifier la fonction solveP2 pour prendre en compte le paramètre M.
Nous allons utiliser une approche similaire à celle de solveP1 en considérant les M villes les plus proches à chaque étape et en sélectionnant un sous-ensemble de ces M villes pour les ajouter à la tournée.
Nous allons répéter cette étape jusqu'à ce que toutes les villes aient été visitées.

Voici le code modifié pour la fonction solveP2 avec le paramètre M :

```
vector<int> solveP2(P2Data& data, int M) {
    int num_cities = data.num_cities;
    vector<vector<int>> distances = data.distances;

    // Création de la tournée
    vector<int> tour(num_cities, -1);
    tour[0] = 0;

    // Visite des villes
    for (int i = 1; i < num_cities; i++) {
        // Sélection des M villes les plus proches
        vector<int> candidates;
```

```
        for (int j = 0; j < num_cities; j++) {
        if (find(tour.begin(), tour.end(), j) == tour.end()) {
        candidates.push_back(j);
        }
    }
    sort(candidates.begin(), candidates.end(), [&distances, &tour](int& left, int& right) {
    int left_dist = numeric_limits<int>::max();
    int right_dist = numeric_limits<int>::max();
    for (int i = 0; i < tour.size() - 1; i++) {
        int dist = distances[tour[i]][left] + distances[left][tour[i+1]] - distances[tour[i]][tour[i+1]];
        left_dist = min(left_dist, dist);
        dist = distances[tour[i]][right] + distances[right][tour[i+1]] - distances[tour[i]][tour[i+1]];
        right_dist = min(right_dist, dist);
    }
    return left_dist < right_dist;
    });
    // Ajout des villes dans la tournée
    for (int j = 0; j < min(M, (int)candidates.size()); j++) {
        int city = candidates[j];
        int insert_pos = find(tour.begin(), tour.end(), -1) - tour.begin();
        tour[insert_pos] = city;
    }
}


return tour;
}
```

Maintenant, nous allons tester différentes combinaisons de valeurs de N et M pour évaluer la qualité des résultats pour chaque problème.

Nous allons générer aléatoirement des instances de chaque problème avec un nombre de villes ou d'objets fixe, puis nous allons exécuter chaque algorithme avec différentes valeurs de N et M pour chaque instance. Nous allons mesurer le temps d'exécution et la qualité de la solution pour chaque combinaison de N et M, puis nous allons comparer les résultats.

Voici le code pour tester différentes valeurs de N et M pour chaque problème :

```
int main() {
srand(time(NULL));

// Nombre de villes ou d'objets
int num_elements = 20;

// Génération aléatoire des instances de chaque problème
P1Data data1 = generateP1Data(num_elements, num_elements * 10);
P2Data data2 = generateP2Data(num_elements, num_elements * 5);

// Valeurs de N et M à tester
vector<int> N_values = {2, 3, 4, 5};
vector<int> M_values = {2, 3, 4, 5};

// Exécution des algorithmes pour chaque combinaison de N et M
for (int N : N_values) {
   for (int M : M_values) {
      // Exécution de solveP1 avec N
      auto start_time = chrono::high_resolution_clock::now();
      int result1 = solveP1(data1, N);
      auto end_time = chrono::high_resolution_clock::now();
      double exec_time1 = chrono::duration_cast<chrono::microseconds>(end_time - start_time).count() /
1000.0;

      // Exécution de solveP2 avec M
      start_time = chrono::high_resolution_clock::now();
      vector<int> result2 = solveP2(data2, M);
      end_time = chrono::high_resolution_clock::now();
      double exec_time2 = chrono::duration_cast<chrono::microseconds>(end_time - start_time).count() /
1000.0;
      // Affichage des résultats pour chaque combinaison de N et M
      cout << "N=" << N << ", M=" << M << endl;
```

```
    cout << "P1: " << result1 << " (execution time: " << exec_time1 << " ms)" << endl;
    cout << "P2: ";
    for (int city : result2) {
        cout << data2.city_names[city] << " ";
    }
    cout << "(execution time: " << exec_time2 << " ms)" << endl;
    cout << endl;
  }
  }

  return 0;
  }
```

Dans cet exemple, nous testons toutes les combinaisons de valeurs de N et M pour des instances de problèmes avec 20 villes ou objets.

Nous mesurons le temps d'exécution et la qualité de la solution pour chaque combinaison de N et M, puis nous affichons les résultats à la fin.

On peut ajuuster les valeurs de num_elements, N_values et M_values en fonction la puissance de l'ordinateur et du temps disponible pour l'exécution.

Un sac à dos brut de décoffrage :

Pour résoudre le problème du sac à dos de manière exacte, nous pouvons utiliser un algorithme de recherche exhaustive.

L'idée est de générer toutes les combinaisons possibles d'objets et de calculer leur bénéfice total, puis de sélectionner la combinaison qui maximise le bénéfice tout en respectant la capacité maximale du sac à dos. Voici l'algorithme de recherche exhaustive :

```
// Fonction pour résoudre le problème P1 avec un algorithme de recherche exhaustive
int solveP1Bruteforce(P1Data& data) {
int capacity = data.capacity;
int num_objects = data.num_objects;
vector<pair<int, int>> objects = data.objects;
int max_benefit = 0;
// Génération de toutes les combinaisons possibles d'objets
for (int i = 0; i < pow(2, num_objects); i++) {
```

```
    int benefit = 0;
    int consumption = 0;

    // Calcul du bénéfice et de la consommation pour la combinaison courante
    for (int j = 0; j < num_objects; j++) {
        if (i & (1 << j)) {
            benefit += objects[j].second;
            consumption += objects[j].first;
        }
    }

    // Mise à jour du bénéfice maximal si la consommation est inférieure à la capacité maximale du sac à dos
    if (consumption <= capacity && benefit > max_benefit) {
        max_benefit = benefit;
    }
}

return max_benefit;
}
```

Nous pouvons maintenant comparer les résultats de l'algorithme de recherche exhaustive avec ceux de l'algorithme glouton et de l'heuristique précédemment implémentés.

Comme l'algorithme de recherche exhaustive est beaucoup plus coûteux en temps de calcul que les autres algorithmes, nous ne l'utiliserons que pour de petites instances.
Pour comparer les résultats de chaque algorithme sur une petite instance du problème P1 :

```
int main() {
    srand(time(NULL));

    // Instance de petite taille
    P1Data data = generateP1Data(5, 50);

    // Résolution avec l'algorithme glouton
    int result1 = solveP1(data);
```

```
    cout << "Resultat avec l'algorithme glouton : " << result1 << endl;

    // Résolution avec l'heuristique
    int result2 = solveP1(data, 2);
    cout << "Resultat avec l'heuristique : " << result2 << endl;

    // Résolution avec l'algorithme de recherche exhaustive
    int result3 = solveP1Bruteforce(data);
    cout << "Resultat avec l'algorithme de recherche exhaustive : " << result3 << endl;

    return 0;
}
```

En comparant les résultats, nous pouvons constater que l'algorithme de recherche exhaustive donne la solution optimale pour cette petite instance, mais qu'il est beaucoup plus coûteux en temps de calcul que les autres algorithmes.

L'algorithme glouton et l'heuristique donnent tous deux des résultats proches de la solution optimale en un temps de calcul beaucoup plus court.

Solution du sac à dos déterministe.

Lorsqu'on cherche à résoudre le problème du sac à dos de manière 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.

En effet, si on considère un ensemble d'objets avec des valeurs de consommation et de bénéfice différentes, il est possible qu'un objet moins bénéfique mais moins consommateur puisse être ajouté dans le sac à dos au lieu d'un objet plus bénéfique mais plus consommateur. Si la consommation des objets est telle que l'on ne peut pas inclure tous les objets, alors l'algorithme peut être amené à faire des compromis pour maximiser le bénéfice.

Par exemple, supposons qu'on ait un sac à dos d'une capacité de 50 kg et 4 objets disponibles avec les caractéristiques suivantes :

Objet A : consommation 20 kg, bénéfice 100 euros

Objet B : consommation 25 kg, bénéfice 120 euros

Objet C : consommation 10 kg, bénéfice 50 euros

Objet D : consommation 15 kg, bénéfice 70 euros

Si on choisit de prendre uniquement le meilleur candidat à chaque étape de la construction de la solution, on ajouterait d'abord l'objet B, qui est le plus bénéfique. Mais ensuite, on ne pourrait pas ajouter l'objet A ou l'objet D car ils ne rentreraient pas dans le sac à dos, et on serait donc obligé de choisir l'objet C, qui a un rapport bénéfice/consommation moins bon que les autres objets. Cela donnerait une solution avec un bénéfice total de 170 euros.

En revanche, si on choisit une approche déterministe qui prend en compte la consommation des objets, on pourrait d'abord ajouter l'objet C, qui a une consommation faible et un bénéfice acceptable. Ensuite, on pourrait ajouter l'objet D, qui a une consommation intermédiaire et un bénéfice intéressant. Enfin, on pourrait ajouter l'objet A, qui a une consommation plus élevée mais un bénéfice important. Cela donnerait une solution avec un bénéfice total de 270 euros.

Opérateurs locaux.

Pour mettre en place des opérateurs locaux dans la résolution des problèmes P1 et P2, nous allons utiliser une approche itérative pour perturber les solutions existantes et essayer de les améliorer.

Pour le problème P1, nous pouvons implémenter des opérateurs locaux tels que l'échange d'objets sélectionnés et non sélectionnés, ou l'échange de deux objets sélectionnés contre un seul non sélectionné. Nous pouvons également utiliser des opérateurs qui ajoutent ou suppriment des objets de la solution.

Pour le problème P2, nous pouvons implémenter des opérateurs locaux tels que l'échange d'ordre de deux villes dans la tournée, ou l'insertion ou la suppression de villes de la tournée.

Après avoir appliqué un opérateur local, nous devons évaluer la qualité de la nouvelle solution et comparer avec la solution précédente. Si la nouvelle solution est meilleure, nous la remplaçons, sinon nous conservons la solution précédente. Nous répétons ce processus jusqu'à ce qu'un critère d'arrêt soit atteint, tel qu'un nombre maximum d'itérations ou un temps d'exécution maximum.

```
// Fonction pour appliquer des opérateurs locaux à la solution du problème P1

void applyLocalOperatorsP1(P1Data& data, vector<bool>& solution) {

int num_objects = data.num_objects;

vector<pair<int, int>> objects = data.objects;

int capacity = data.capacity;


// Échange d'objets sélectionnés et non sélectionnés

for (int i = 0; i < num_objects; i++) {

   if (solution[i]) {
```

```
        for (int j = num_objects; j < objects.size(); j++) {
            if (objects[j].first <= capacity && !solution[j]) {
                solution[i] = false;
                solution[j] = true;
                capacity += objects[i].first - objects[j].first;
            }
        }
    }
}

// Échange de deux objets sélectionnés contre un seul non sélectionné
for (int i = 0; i < num_objects; i++) {
    if (solution[i]) {
        for (int j = i+1; j < num_objects; j++) {
            if (solution[j]) {
                for (int k = num_objects; k < objects.size(); k++) {
                    if (objects[k].first <= capacity && !solution[k] && objects[i].first + objects[j].first - objects[k].first <=
capacity) {
                        solution[i] = false;
                        solution[j] = false;
                        solution[k] = true;
                        capacity += objects[i].first + objects[j].first - objects[k].first;
                    }
                }
            }
        }
    }
}

// Ajout d'objets non sélectionnés
for (int i = num_objects; i < objects.size(); i++) {
    if (objects[i].first <= capacity && !solution[i]) {
        solution[i] = true;
```

```
        capacity -= objects[i].first;
    }
}

// Suppression d'objets sélectionnés
for (int i = 0; i < num_objects; i++) {
    if (solution[i]) {
        solution[i] =true;
capacity -= objects[i].first;
}
}

// Suppression d'objets sélectionnés
for (int i = 0; i < num_objects; i++) {
if (solution[i]) {
solution[i] = false;
capacity += objects[i].first;
}
}

// Ajout d'objets non sélectionnés
for (int i = num_objects; i < objects.size(); i++) {
if (objects[i].first <= capacity && !solution[i]) {
solution[i] = true;
capacity -= objects[i].first;
}
}

return solution;
}
```

Maintenant, nous allons implémenter la fonction d'opérateur pour le problème P2. Nous allons utiliser un opérateur qui échange deux villes dans la tournée et vérifie si la nouvelle tournée est valide en termes de capacité. Si la nouvelle tournée est valide, nous la renvoyons ; sinon, nous renvoyons la tournée d'origine.

```
vector<int> operatorP2(P2Data& data, vector<int>& tour) {
int num_cities = data.num_cities;
vector<vector<int>> distances = data.distances;

// Sélection aléatoire de deux villes à échanger
int i = rand() % (num_cities - 1) + 1;
int j = rand() % (num_cities - 1) + 1;

// Échange des villes
swap(tour[i], tour[j]);

// Vérification de la validité de la nouvelle tournée
int capacity = numeric_limits<int>::max();
for (int k = 0; k < num_cities; k++) {
   if (capacity < distances[tour[k - 1]][tour[k]]) {
      swap(tour[i], tour[j]);
      return tour;
   }
   capacity -= distances[tour[k - 1]][tour[k]];
}

return tour;
}
```

Multi-véhicules.

en faitt cela est une extension des problèmes P1 et P2 où plusieurs véhicules de capacité différente sont utilisés pour effectuer les livraisons.

Le but est de minimiser le coût total de la tournée en satisfaisant toutes les demandes de livraison tout en respectant les contraintes de capacité de chaque véhicule.

Pour résoudre le CVRP, il existe de nombreuses approches heuristiques et exactes, notamment la méthode de Clarke and Wright, la recherche tabou, les algorithmes génétiques et les méthodes de programmation linéaire en nombres entiers.

Une approche possible pour résoudre le CVRP est de combiner les approches développées pour les problèmes P1 et P2.

On peut d'abord utiliser une heuristique pour déterminer les tournées de chaque véhicule en prenant en compte la capacité de chaque véhicule.

Ensuite, on peut utiliser une métaheuristique pour optimiser la planification des tournées en prenant en compte les interactions entre les tournées de

différents véhicules.

De meilleurs algorithmes.

Pour le problème du voyageur de commerce, l'algorithme le plus performant actuellement connu pour les grandes instances est le Concorde TSP Solver, qui utilise une combinaison d'heuristiques et d'algorithmes de branch and cut. Cependant, cet algorithme peut être très lent pour les instances de taille moyenne ou petite.

Pour les instances de taille moyenne ou petite, l'algorithme 2-opt peut être une bonne option. Il s'agit d'un algorithme d'optimisation locale qui améliore une solution initiale en échangeant deux arêtes dans le circuit. L'algorithme 3-opt est une extension de l'algorithme 2-opt qui échange trois arêtes au lieu de deux et peut être encore plus efficace pour de petites instances.

Pour le problème du sac à dos, l'algorithme le plus simple et le plus couramment utilisé est l'algorithme glouton, qui sélectionne les objets les plus bénéfiques en fonction de leur rapport bénéfice/poids et les ajoute au sac à dos tant qu'il y a de la place disponible. Cependant, cet algorithme ne garantit pas la solution optimale et peut donner des résultats suboptimaux.

Pour les instances de taille moyenne ou grande, l'algorithme de branch and bound peut être une meilleure option. Cet algorithme explore toutes les solutions possibles en utilisant une stratégie de parcours en profondeur de l'arbre des solutions, en éliminant les sous-arbres qui ne peuvent pas conduire à la solution optimale. Cet algorithme peut être très efficace pour les instances avec des contraintes de capacité très strictes.