TP4 Outils d'Aide à a décision
Heuristiques pour le PDPTW

Maxime ESCOURBIAC Jean-Christophe SEPTIER
ISIMA • ZZ2F2 • 19 janvier 2011

Table des matières

I Présentation du problème

1 - Enoncé du problème

Le problème de livraisons et collectes consiste à définir un ensemble de routes de manière à satisfaire un ensemble de requêtes de transport. Une flotte de véhicules est disponible pour effectuer ces transports. Chaque véhicule a une capacité connue, un point de départ et un point d'arrivée (généralement le dépôt central). Une requête de transport est caractérisée par une quantité à transporter, un point de collecte (pickup) et un point de livraison (delivery). La requête ne doit être traitée que par un seul véhicule et la collecte doit être nécessairement réalisée avant la livraison. L'objectif est d'abord de minimiser le nombre de véhicules utilisés et, de manière secondaire, la distance totale parcourue par les véhicules. Une extension classique ajoute des contraintes temporelles sur les requêtes : à chaque point de livraison / collecte on associe une fenêtre de temps [A,B]. La première date correspond à l'heure d'ouverture et la seconde à l'heure de fermeture. Ainsi, le véhicule peut arriver avant l'heure A, mais il doit alors attendre l'ouverture. Par contre, il ne peut arriver après l'heure B. De plus, chaque collecte / livraison induit un certain temps de chargement / déchargement. Le PDP devient alors le problème de livraison et collecte avec fenêtre de temps (Pickup and Delivery Problem with Time Windows – PDPTW). Ce problème est un des problèmes central de l'industrie du transport, il est NP-difficile.

2 - Hypothèses effectuées

- Le nombre de véhicules est supposé illimité.

- On effectue la livraison juste après la collecte pour une requête.

- Le critère de choix de la meilleure tournée est faite sur le nombre de véhicules déployés, et à nombre de véhicules égal, la plus faible distance sera choisi.

3 - Modèle retenue

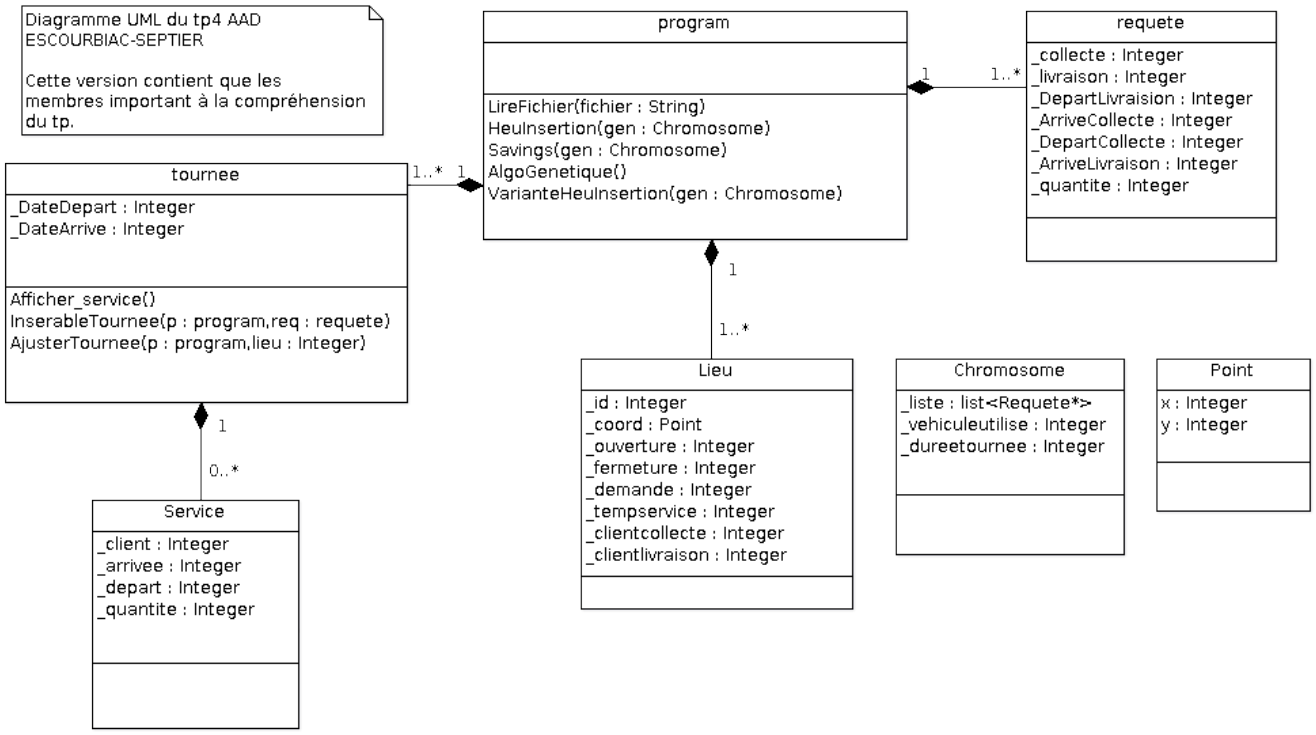

fig 1 : Diagramme UML réalisé avec argoUML

Description des classes/structure:

Program:

Classe regroupant toutes les méthodes générales, les heuristiques, ainsi que l'algorithme génétique.

Requete:

Classe ayant toutes les informations sur une requête donnée.
Les attributs _Depart***** et _Arrive***** sont des attributs permettant des calculs lors de l'heuristique d'insertion.

Lieu:

Classe contenant toutes les informations d'un client comme les coordonnées , les dates d'ouverture et de fermeture, ainsi que la relation collecte/livraison associé à ce lieu.

Tournee:

Représente la tournée d'un véhicule.

Service:

Représente une marque de passage d'un véhicule à un lieu donné.

Chromosome:

Cette structure permet d'effectuer l'algo génétique. Elle contient la liste de requête qui lui est associée ainsi que les valeurs de son évaluation.

Point:

Structure représentant les coordonnées d'un point.

II Algorithmes essentiels

1 - Algorithme de stockages des données

Cet algorithme permet de passer des données en forme de données exploitables dans le programme.

```
LireFichier(fichier)
|
|  ouverture du fichier
|  si ouverture non réussi
|  |  quitter programme
|  fsi
|
|  lecture de la première ligne (inutile avec nos hypotheses
|  tant non fin de fichier
|  |  lire ligne
|  |  stocker un lieu
|  |  si le lieu est un point de collecte
|  |  |  Stocker la requete
|  |  fsi
|  fait
|  fermer le fichier
fin
```

2 - Algorithme de l'heuristique d'insertion

Cet algorithme permet d'évaluer le nombre de véhicules à utiliser et la distance parcourue par une liste de requêtes donnée en paramètre.
Cependant cet algorithme est le moins efficace des trois présentés

```
HeuInsertion(Chromosome * gene)
|  faire une copie de la liste du chromosome
|  Tant qu'il reste des requêtes dans la copie de la liste
|  |  faire une copie de la copie de la liste
|  |  Rajouter en fin une nouvelle tournée dans la liste des tournées
|  |  Tant que la copie de la copie de liste n'est pas vide
|  |  |  On supprime les requêtes non insérable en fin de la tournée de la copie
|  |  |  de copie de liste
|  |  |  On tri selon l'ordre croissant la copie de la copie de liste selon la date
|  |  |  d'arrive au dépôt.
|  |  |  On ajoute la meilleure requête dans tournée courante
|  |  |  On supprime la meilleure requête de la copie de la liste
|  |  |  On met a jour les variables temporelles
|  |  fait
|  |  on finalise la tournée
|  fait
|  On récupère le nombre de véhicule et la distance parcourue
fait
```

3 - Algorithme de l'heuristique d'insertion amélioré

Cet algorithme est basé sur le même principe que le précèdent , mais l'insertion peut se faire à n'importe quel endroit de la tournée.

L'algorithme principal:

```
Pour toute les requêtes du chromosome faire
|  Tant que on a pas parcouru toutes les tournées et qu'on a pas insérer la requête faire
|  |  Test si on peut l'insérer dans la tournée courante
|  fait
|  Si requête n'a pas été inséré alors
|  |  Créer une nouvelle tournée et l'insérer en fin de liste.
|  fsi
fait
```

L'algorithme pour tester l'insertion d'une requête dans une tournée existante

```
meilleur_temp = MAX_VALUE
Pour chaque inter-requête faire
|  Test si on peut insérer la requête dans l'inter-requête
|  Si on peut l'inserer alors
|  |  on affecte le temps de retour au dépôt  à meilleur_temp si il est meilleur
|  fsi
fait
Si il y a eu insertion alors
|  l'inserer au meilleur endroit
|  return true
Sinon
|  return false
fsi
```

4 - Algorithme de Savings de Clarke & Wright:

Cet algorithme permet de donner toujours le même résultat, quelque soit l'ordre de la requête. On crée des arcs(i,j) où i est la livraison d'une requête et j la collecte d'une autre. On calcul ensuite le gain de temps si on fusionne ses deux requêtes, plutôt que de les faire séparément. Ces gains sont ensuite triés pour savoir quelle est la meilleure fusion possible, si elle est possible.

```
Savings(Chromosome * gene)
|  Pour toutes les requêtes
|  |  Créer une tournée correspondante à celle-ci
|  Pour toutes les requêtes
|  |  Créer un arc (i,j) entre celle ci et les autres requêtes
|  |  Calculer les gains des arcs
|  Trier les arcs selon le gain
|  Pour tout les arcs
|  |  si la fusion est possible
|  |  |  fusionner les deux tournées
|  |  fsi
|  |  retirer l'arc de la liste des arcs
|  fait
|  finaliser les tournées
fait
```

Si O est l'origine, alors le gain Gij = CiO + COj - Cij

5 - Algorithme génétique:

```
//initialisation de la population
tant que population.taille() < TAILLE_POP faire
|  générer  chromosome aléatoire;
|  évaluer le chromosome;
|  Si vecteur est diffèrent de chaque vecteur de la population alors
|  |  insérer vecteur;
|  fsi;
fait;
trier la population selon le nombre de véhicules, puis la distance;

//phase d'amélioration itérative
tant que iter < itermax faire
|  tant que population Fils.taille() < TAILLE_POP faire
|  |  choisir 2 chromosomes parmi la population (choix élitiste && les chromosomes doivent
|  |                                                être différents)
|  |  faire un croisement avec ces 2 chromosomes pour obtenir 2 chromosomes fils;
|  |  effectuer une opération de mutation pour le fils 1;
|  |  Si nb véhicule du fils1 muté > nb véhicule du fils 1  alors
|  |  |  chromosome_to_insert = fils1_muté;
|  |  Sinon
|  |  |  chromosome_to_insert = fils1;
|  |  fsi;
|  |  inserer dans la populationFils chromosome_to_insert;
|  |  effectuer une opération de mutation pour le fils 2;
|  |  Si le nb véhicule du fils2 muté > nb véhicule du fils 2  alors
|  |  |  chromosome_to_insert = fils2_muté;
|  |  Sinon
|  |  |  chromosome_to_insert = fils2;
|  |  fsi;
|  |  inserer dans la population chromosome_to_insert;
|  fait;
|  trier la population Fils selon le nombre de véhicules, puis la distance;
|  concaténer la population avec la populationFils
|  Supprimer les doublons de la population
|  Supprimer les plus mauvais chromosomes pour réatteindre la TAILLE_POP
fait;
```

Note:

Les populations sont stockées dans des listes pour pouvoir la trier plus rapidement grâce à la méthode sort() et supprimer les doublons grâce à la méthode unique().

III Résultats

1 - Heuristiques de Savings

Les tests ont été effectués sur un système Linux, avec un processeur AMD 6000x2 et 3Go de Ram.

instance	Distance	Véhicule
lc101	20653	27
lc102	19716	26
lc103	17292	21
lc104	16482	16
lc105	19784	27

Après quelques essais, on se rend compte que les résultats de Savings donne de très mauvais résultats. Ce n'est donc pas un bon algorithme pour résoudre ce problème.

2 - Algorithmes génétique avec heuristiques d'insertion amélioré

Les conditions de tests sont les mêmes que pour Savings. La taille de la population a été définie à 50 et le nombre d'itérations à 50 également.

instance	Distance	Véhicule	référence web
lc101	12675	13	10
lc206	15134	7	3
lc202	15517	7	3
lc102	11963	12	10
lc103	11851	11	9
lc208	998	7	3
lr105	3056	16	14
lr106	2432	15	12
lr107	2131	14	10
lrc105	3207	16	13
lrc201	2140	16	4
lr107	2530	13	10
lc108	1934	12	10
lc1_4_1	36612	48	40

Conclusion:

Les résultats obtenues sont en dessous des résultats du benchmark.

Cela peut s'expliquer par la limite de l'algorithme génétique, en appliquant l'algorithme mémétique on aurait eu un meilleur résultat. On peut aussi ajouter l'impact de nos hypothèses simplificatrices notamment celle où on ne doit pas faire 2 collectes de suite.

Les résultats sont toutefois meilleurs que pour Savings.