TP Programmation
des méthodes de résolution des problèmes
Travel salesman Problem
Vehicle routing Problem

Filière: SCM3

École Nationale des sciences appliquées de
Tétouan

Abdellah El Fallahi

5 juin 2020

Table des matières

1 Objectif

L'objectif principal de ce TP est de développer au moins deux algorithmes d'optimisation de type heuristique ou bien méta-heuristiques pour résoudre le problème TSP et le problème CVRP. Le langage à utiliser peut être C++ ou bien Java. L'algorithme heuristique pour le TSP peut être le plus proche voisin ou bien le recuit simulé. Pour avoir à la fin de ce TP les deux méthodes programmées une partie de la classe choisira le PPV et la deuxième partie le recuit simulé. Pour le problème CVRP les deux méthodes proposées sont Tabu search et une algorithme génétique. Pour la programmation des algorithmes pour le TSP il faut procéder de la manière suivante :

- Il faut déterminer le codage des solutions
- Déterminer les structures de données nécessaires
- Déterminer les fonctions à utiliser
- Lecture des données et leur stockage dans les structures précédemment citées
- programmer les fonctions nécessaires

2 Le codage des solutions

Le codage des solutions dans le cas du TSP consiste à représenter une solution par une permutation de données, ou bien ce qu'on appelle un tour géant qui commence par 0 le point de départ et détermine au même point. A titre d'exemple si vous avez 5 villes indexées de 1 à 5 alors une solution peut être (0,2,3,4,5,1,0).

3 structures de données

Il y a plusieurs manières pour structurer les données dans le ca d'un TSP. La première version et qui est la plus simple consiste à travailler uniquement avec des vecteurs qui peuvent être représentés comme suit :

1. Chaque élément de la permutation a un prédécesseur et un successeur.
2. le vecteur pred ayant une dimension est égale à n, où n est le nombre de villes à visiter donnera le prédécesseur par exemple pred[4] donnera le prédécesseur de la ville numéro 4, ça veut dire la ville à partir de laquelle on est arrivé à la ville 4. De même le succ[4] donnera le successeur de la ville 4.

4 les fonctions

les fonctions principales pour programmer l'algorithme PPV pour résoudre le TSP on peut citer :

1. Read-data(file, structure) : cette fonction permettra de lire les données du problèmes dans notre cas les coordonnées des points à visiter. Le points seront par leurs coordonnées cartésiennes (x,y), ces données doivent ensuite stocker dans la structure données choisie.

2. la fonction dist() Calculer le distancier un tableau de taille (n+1, n+1)qui donnera la distances entre les différents points à visiter
3. les distances les plus utilisées sont la distance de manhatan et la distance euclidienne
4. la fonction ppv() c'est la fonction principale qui permettra de chercher à chaque itération la ville la plus proche de la dernière ville visité
5. update(pred,succ) cette fonction se chargera de la mise à jour des vecteurs *pred* et *succ*
6. computetotaledistance() cette fonction donnera la distance totale parcourue à la fin de la recherche
7. writesol() cette fonction permettra d'écrire la solution finale dans fichier pour son ultérieure utilisation
8. improve(sol) cette fonction sera appliquer en cas d'application d'une amélioration de la dernière solution trouvée

Algorithm 1 : Algo nearest neighbor

```
n :=1
while nb − visited ≠ n do
    From the current vertex, choose the edge with the smallest cost and use that as the actual edge in your circuit
    Contnue in this manner, choosing among the edges taht connect from the current vertex to vertices you have not yet visited.
    When you have visited every vertex, return the vectors pred and succ.
end while
```

5 simulated annealing algorithm

Pour le travail à faire avec l'algorithme du recuit simulé alors il faut suivre les même étapes préparatives des données. Les étapes de l'algorithme sont montrée dans l'algorithme 2. pour le T_{max} il faut prendre une valeur assez élevé pour éviter la convergence prématurée de l'algorithme.

6 Résolution du problème CVRP

Présentation du problème : le problème de tournées de véhicules consiste à définir un ensemble de routes de manière à satisfaire la demande d'un ensemble de n clients. Une flotte de véhicules homogènes est disponible en un dépôt central noté 0 pour livrer l'ensemble des clients. L'ensemble fdes véhicules ont la même capcité donnée par Cap. La demande de chaque client i est donnée par Req_i et il doit doit être traitée en une seule fois et par un seul véhicule. La distance entre un client i et un autre j est donnée par d_{ij}. On fait souvent l'hypothèse qu'elle est symétrique et satisfait l'inégalité triangulaire. L'objectif principal est de minimiser le nombre de véhicules puis, en critère secondaire, la distance totale parcourue. Dans ce travail on suppose que la longueur de la tournée est limité à $Tlong_{max}$ qu'il ne faut pas dépasser. Pour résoudre se problème les données des

Algorithm 2 : Algo Simulated annealing

```
1: T ← T_max
2: best ← INIT()
3: while T > T_min do
4:   next ← NEIGHBOUR(T,best)
5:   ΔE ← ENERGY(next) − ENERGY(best)
6:   if ΔE < 0 then
7:     best ← next
8:   else
9:     if random< ACCEPT(T,ΔE) then
10:      best ← next
11:    end if
12:  end if
13:  T ← COOLING(T,best)
14: end while
15: return best
```

clinets (coordonnées ainsi que les demandes) seront récupérer des problèmes de solomon. les étapes de résolution peuvent se résumer en :

- Récupération des données : pour cela il faut visiter le site (http ://w.cba.neu.edu/ msolomon/problems.htm ou à http ://www.sintef.no/Projectweb/TOP/VRPTW/Solomon benchmark/100 customers/customers/)
- Structures des données, chaqu'un a le choix entre une représentation sous la forme des objets de classes ou bien on peut travailler uniquement avec les tableaux. De préférence et pour booster votre niveau de programmation il est très souhaité d'utiliser les classes en C++ ou bien Java.
- Pour faciliter la manipulation des solutions il faut penser à un codage que vous comprenez parfaitement
- Pour construire une solution de départ on peut utiliser les heuristiques vues en cours comme la méthode de clack & wright ou bien la méthode de gillet & miller.
- une autre manière est de construire des tours géants et on utilisera la méthode de Besaley pour décomposer les tours géants en routes
- Comme méthode d'amélioration on utilisera la méthode $2-opt$

Pour lés méthodes méta-heuristiques on travaillera en deux groupes le premier groupe programmera la une méthode génétique et le deuxième groupe travaillera avec la méthode de tabu search. dans ce qui suit on va rappeler par des algorithmes simples la structure des deux méthodes.

7 Méthode Génétique

la structure de m'importe quelle méthode génétique peut être présenté comme le montre l'algorithme 3.

8 Méthode Tau serach

les éléments principaux de la méthode tabu search sont montrés dans l'algorithme 4.

Algorithm 3 : Algo Gentic algorithm

```
1: BEGIN /* genetic algorithm */
2: generate initial population
3: compute fitness of each individual
4: while NOT finished do
5:    BEGIN /* produce new generation */
6:    for population_size/2 do
7:       BEGIN /* reproductive cycle */
8:       select two individuals from old generation for mating
9:       recombine the two individuals to give two offspring
10:      compute fitness of the two offspring
11:      Select the best offspring
12:      with a probability P_r apply a local search ( 2-opt) to the best offspring
13:      insert offspring in new generation and reorder the population
14:   end for
15:   if population has converged then
16:      finished := TRUE
17:   end if
18: end while
19: return first solution in the population
```

— M ← Number of runs/iterations
— N ← Size of neighborhood
— L ← Lenght of Tabu list
— σ ← Move operator

Notes : to avoid cycling and prelature congergence to local optima, we can introduce the asperation criteria. Diversify to explore the serach area

Algorithm 4 : Algo Tabu serach

1: Set M, L, and σ
2: N is based on the σ
3: Randomly generate an initial solution Sol_0
4: Set $Best_{sol} = Sol_0$
5: M := 1
6: **for** i in neighborhood of Sol_0 **do**
7: Evaluate all neighborhood solutions
8: Sort from best to worst OF Value
9: **end for**
10: **for** j in sorted evaluations **do**
11: **if** solution j is not tabu **then**
12: move to solution j and add j to tabulist
13: **end if**
14: **if** $TabuList \geq L$ **then**
15: Remove the oldest solution from the tabu list
16: **end if**
17: Repeat steps 10 to 16 M times
18: **end for**
19: **return** $best_{sol}$