TIPE
Problème de tournée des véhicules

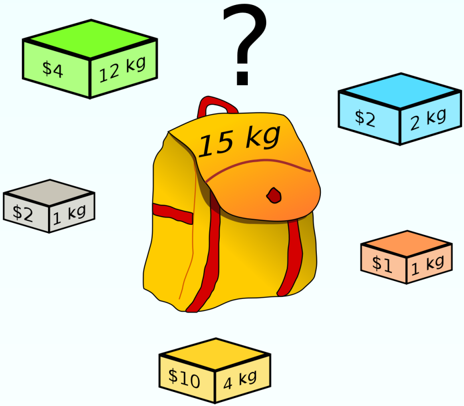
WILLEM Logan

Numéro de candidat: 1210

2020 / 2021

L'optimisation combinatoire — Qu'est-ce que c'est ?

Qu'est-ce que l'optimisation combinatoire ?



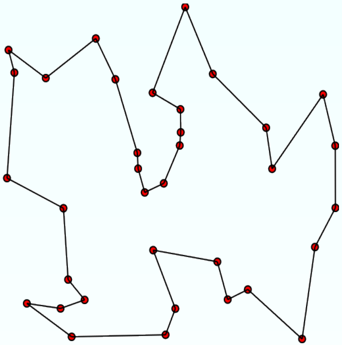
L'optimisation combinatoire — Quelques exemples

Quelques problèmes d'optimisation combinatoire :

L'optimisation combinatoire — Quelques exemples

Quelques problèmes d'optimisation combinatoire :

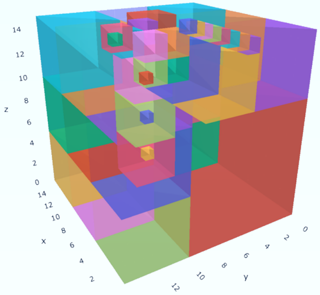
— **Le voyageur de commerce**



L'optimisation combinatoire — Quelques exemples

Quelques problèmes d'optimisation combinatoire :

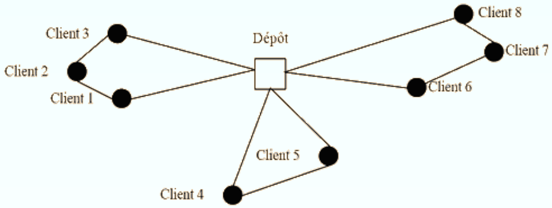
- Le voyageur de commerce
- ***Bin packing***



L'optimisation combinatoire — Quelques exemples

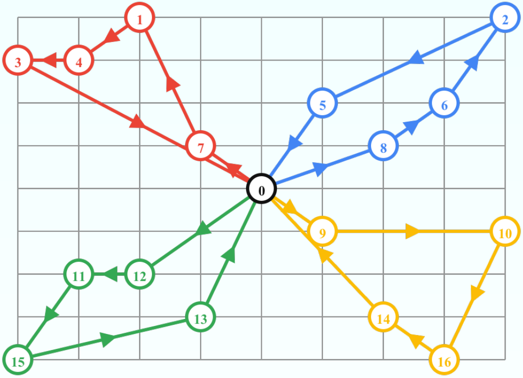
Quelques problèmes d'optimisation combinatoire :

- Le voyageur de commerce
- *Bin packing*
- **Tournée des véhicules**



Problème de tournée des véhicules — Contexte

Le problème de tournée des véhicules (VRP : *Vehicule Routing Problem*) consiste en la minimisation du coût total (en distance par exemple) de la tournée de tous les véhicules, ayant pour objectif de livrer un nombre défini de clients.

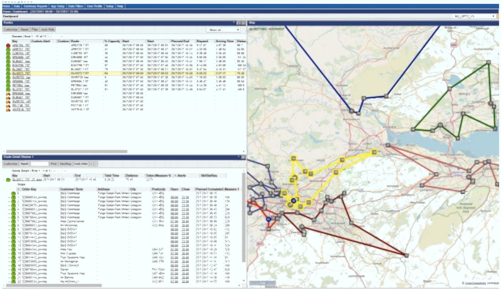


Problème de tournée des véhicules — Ancrage à la vie réelle

INTERNATIONAL

Ontex s'équipe de la solution d'optimisation du transport de Descartes

16.10.2018 • 11h00 | par Emilien VILLEROY



Fournisseur mondial de produits d'hygiène corporelle jetables, Ontex utilise désormais la solution Route Planner de l'éditeur Descartes pour optimiser ses tournées au Royaume-Uni.

Problème de tournée des véhicules — Variantes

Différentes variantes du problème de tournée des véhicules

— Classique (VRP)

Problème de tournée des véhicules — Variantes

Différentes variantes du problème de tournée des véhicules

- Classique (VRP)
- Contrainte de capacité (CVRP)

Problème de tournée des véhicules — Variantes

Différentes variantes du problème de tournée des véhicules

- Classique (VRP)
- Contrainte de capacité (CVRP)
- Dépôts multiples (MDVRP)

Problème de tournée des véhicules — Variantes

Différentes variantes du problème de tournée des véhicules

- Classique (VRP)
- Contrainte de capacité (CVRP)
- Dépôts multiples (MDVRP)
- Retours des colis (VRPPD et VRPB)

Problème de tournée des véhicules — Variantes

Différentes variantes du problème de tournée des véhicules

- Classique (VRP)
- Contrainte de capacité (CVRP)
- Dépôts multiples (MDVRP)
- Retours des colis (VRPPD et VRPB)
- ...

Je me suis intéressé à la version classique afin de m'approprier au mieux le problème.

Problème de tournée des véhicules — Pourquoi utiliser une résolution approchée ?

Deux choix pour le premier camion pour chaque client : "OUI" ou "NON"

Problème de tournée des véhicules — Pourquoi utiliser une résolution approchée ?

Deux choix pour le premier camion pour chaque client : "OUI" ou "NON"

2^n possibilités

Problème de tournée des véhicules — Pourquoi utiliser une résolution approchée ?

Deux choix pour le premier camion pour chaque client : “OUI” ou “NON”

$$2^n \text{ possibilités}$$

Le sens importe peu :

$$2^{n-1} \text{ possibilités}$$

Problème de tournée des véhicules — Pourquoi utiliser une résolution approchée ?

Deux choix pour le premier camion pour chaque client : “OUI” ou “NON”

$$2^n \text{ possibilités}$$

Le sens importe peu :

$$2^{n-1} \text{ possibilités}$$

Pour tous les trajets, il existe au moins 2^{n-1} possibilités :

Problème de tournée des véhicules — Pourquoi utiliser une résolution approchée ?

Deux choix pour le premier camion pour chaque client : "OUI" ou "NON"

2^n possibilités

Le sens importe peu :

2^{n-1} possibilités

Pour tous les trajets, il existe au moins 2^{n-1} possibilités :

complexité **exponentielle**

L'algorithme de Clarke et Wright

Données

- Un point D de coordonnées $(0, 0)$: le dépôt

L'algorithme de Clarke et Wright

Données

- Un point D de coordonnées $(0, 0)$: le dépôt
- Une famille de points $(i_1, \ldots, i_n) \in ([-100, 100]^2)^n$ pour un certain $n \in [2, +\infty[$ représentant les clients

L'algorithme de Clarke et Wright

Données

- Un point D de coordonnées $(0, 0)$: le dépôt
- Une famille de points $(i_1, \ldots, i_n) \in ([-100, 100]^2)^n$ pour un certain $n \in [2, +\infty[$ représentant les clients
- Une fonction d qui calcule la distance entre deux points.

L'algorithme de Clarke et Wright

Données

- Un point D de coordonnées $(0,0)$: le dépôt
- Une famille de points $(i_1, \dots, i_n) \in ([-100, 100]^2)^n$ pour un certain $n \in [2, +\infty[$ représentant les clients
- Une fonction d qui calcule la distance entre deux points.

Remarque : Dans cette méthode de résolution, le nombre de véhicules n'est pas fixé. C'est l'algorithme qui décide du nombre optimal de véhicules à utiliser.

L'algorithme de Clarke et Wright

Definition (Fonction *gain*)

Fonction s : calcule le gain après raccord de deux routes.
Pour deux points i et j, s calcule la différence de distance entre le chemin $D-i-D+D-j-D$ qui vaut $2d(D,i)+2d(j,D)$ au chemin $D-i-j-D$ de distance $d(D,i)+d(i,j)+d(j,D)$

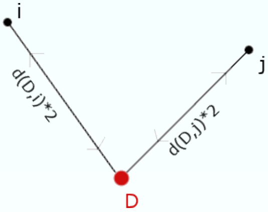
$$s(i,j) = 2d(D,i) + 2d(j,D) - [d(D,i) + d(i,j) + d(j,D)]$$
$$s(i,j) = d(D,i) + d(j,D) - d(i,j)$$

L'algorithme de Clarke et Wright

Definition (Fonction *gain*)

Fonction s : calcule le gain après raccord de deux routes.
Pour deux points i et j, s calcule la différence de distance entre le chemin $D-i-D+D-j-D$ qui vaut $2d(D,i)+2d(j,D)$ au chemin $D-i-j-D$ de distance $d(D,i)+d(i,j)+d(j,D)$

$$s(i,j) = 2d(D,i) + 2d(j,D) - [d(D,i) + d(i,j) + d(j,D)]$$
$$s(i,j) = d(D,i) + d(j,D) - d(i,j)$$

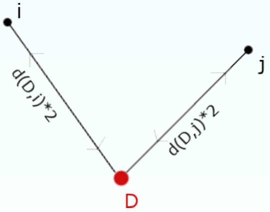


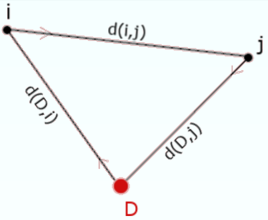
L'algorithme de Clarke et Wright

Definition (Fonction *gain*)

Fonction s : calcule le gain après raccord de deux routes.
Pour deux points i et j, s calcule la différence de distance entre le chemin $D - i - D + D - j - D$ qui vaut $2d(D,i) + 2d(j,D)$ au chemin $D - i - j - D$ de distance $d(D,i) + d(i,j) + d(j,D)$

$$s(i,j) = 2d(D,i) + 2d(j,D) - [d(D,i) + d(i,j) + d(j,D)]$$
$$s(i,j) = d(D,i) + d(j,D) - d(i,j)$$





L'algorithme de Clarke et Wright

L'algorithme comporte deux étapes majeures :

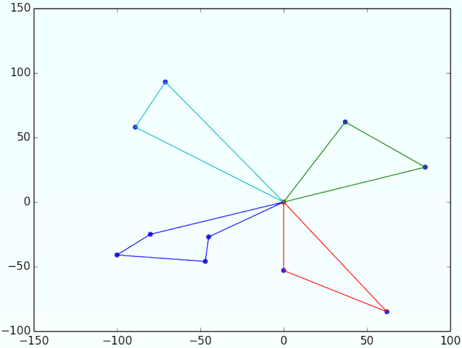
— **Étape 1 : Calcul des bénéfices**
Calcul la liste des bénéfices s_{ij} pour $i, j \in [\![1, n]\!]$ distincts

L'algorithme de Clarke et Wright

L'algorithme comporte deux étapes majeures :

- Étape 1 : Calcul des bénéfices
  Calcul la liste des bénéfices s_{ij} pour $i, j \in [\![1, n]\!]$ distincts
- **Étape 2 : Fusion des routes**
  Fusion des routes si celles-ci peuvent l'être, dans l'ordre donné par la liste précédente

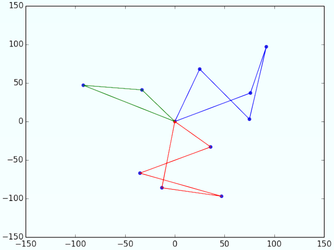
Résultats



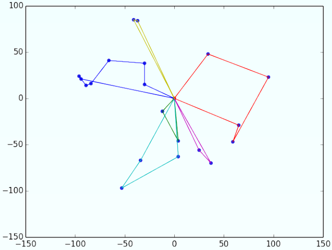
Solution satisfaisante d'un problème

Insuffisance de l'algorithme

Exemples de résultats insatisfaisants

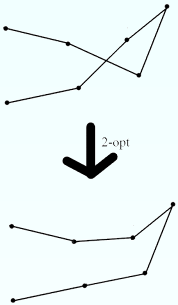


Pour $n = 10$



Pour $n = 20$

Principe du 2-opt



Principe du 2-opt
Suppression des liaisons sécantes

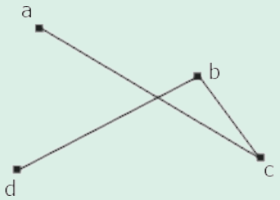
Principe du 2-opt

Exemple

Soit un chemin (a, c, b, d). Dans le graphe ci-dessous, on a :

$$d(a,c) + d(b,d) > d(a,b) + d(c,d)$$

Un changement va donc s'opérer : $(a, \mathbf{c}, \mathbf{b}, d) \to (a, \mathbf{b}, \mathbf{c}, d)$



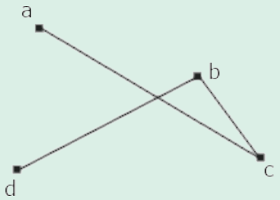
Principe du 2-opt

Exemple

Soit un chemin (a, c, b, d). Dans le graphe ci-dessous, on a :

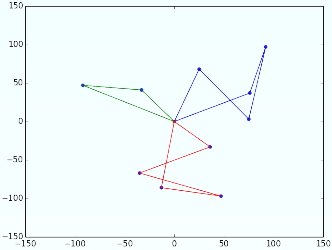
$$d(a, c) + d(b, d) > d(a, b) + d(c, d)$$

Un changement va donc s'opérer : $(a, \mathbf{c}, \mathbf{b}, d) \rightarrow (a, \mathbf{b}, \mathbf{c}, d)$

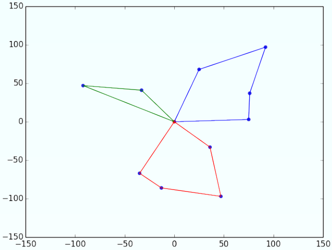


Comparaison avec et sans 2-opt

10 clients



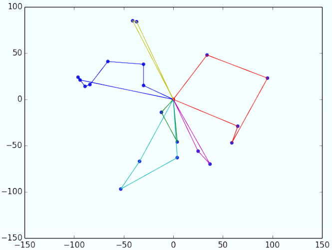
Avant



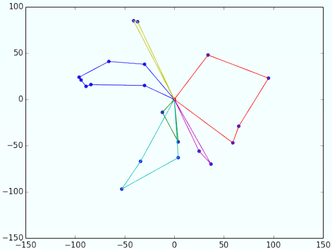
Après

Comparaison avec et sans 2-opt

20 clients



Distance : 1224km
Avant



Distance : 1193km
Après

Principe de l'opérateur

L'algorithme comporte trois étapes :

— **Étape 1 : Calcul des plus proches voisins pour chaque point**
Calcul de la liste
des plus proches voisins à chaque point pour chaque autre route déjà formée

Principe de l'opérateur

L'algorithme comporte trois étapes :

— Étape 1 : Calcul des plus proches voisins pour chaque point
Calcul de la liste des plus proches voisins à chaque point pour chaque autre route déjà formée

— **Étape 2 : Calcul de l'optimisation engendré**
Calcul de la liste des gains (et/ou des pertes) par migration temporaire d'un point sur chaque autre route en fonction de la liste de l'étape 1.

Principe de l'opérateur

L'algorithme comporte trois étapes :

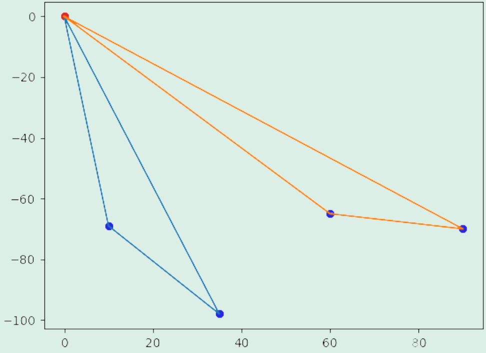
— Étape 1 : Calcul des plus proches voisins pour chaque point
Calcul de la liste des plus proches voisins à chaque point pour chaque autre route déjà formée

— Étape 2 : Calcul de l'optimisation engendré
Calcul de la liste des gains (et/ou des pertes) par migration temporaire d'un point sur chaque autre route en fonction de la liste de l'étape 1.

— **Étape 3 : Migrations des points**
La migration du point donnant le plus grand gain est faite définitivement. Puis on recommence à l'étape 1 tant que cette liste contient des gains.

Exemple d'utilisation

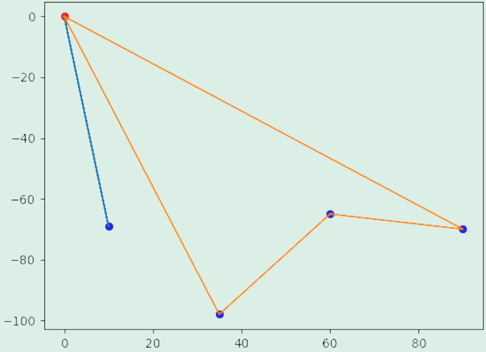
Exemple : Solution de départ



Distance : 445km

Exemple d'utilisation

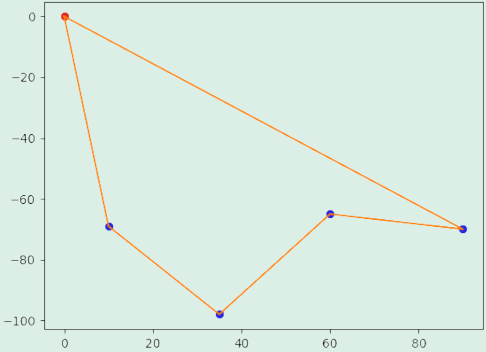
Exemple : Première exécution de l'opérateur



Distance : 429km

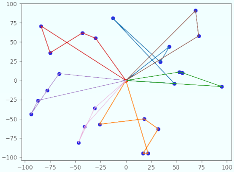
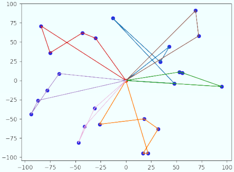
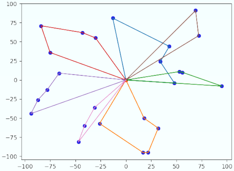
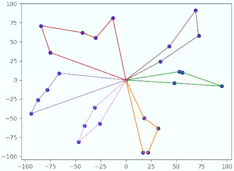
Exemple d'utilisation

Exemple : Seconde exécution de l'opérateur



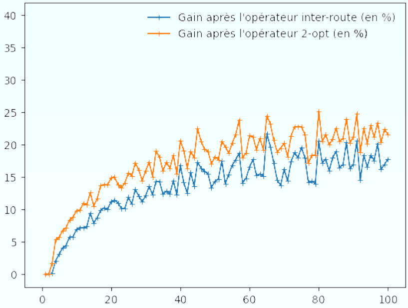
Distance : 382km

Comparaison avec et sans l'opérateur inter-route

2-opt	Opérateur inter-route + 2-opt
	
↓	↓
	
Distance : 1575km	Distance : 1429km

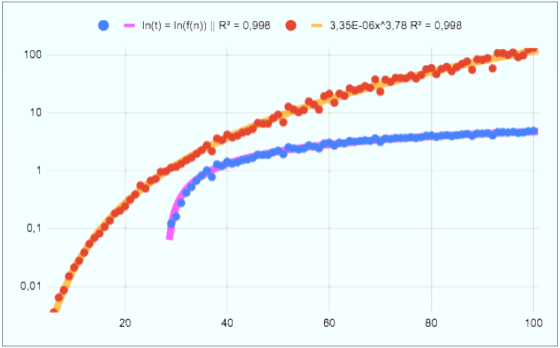
Résultats

Moyenne des gains en fonction du nombre n de clients



Moyenne avec l'opérateur inter-route seulement (à partir de $n = 40$) : 16.6 %
Moyenne avec les opérateurs inter et intra route (à partir de $n = 40$) : 20.6 %
Meilleure optimisation après les deux opérateurs : 35 % (pour $n = 88$)

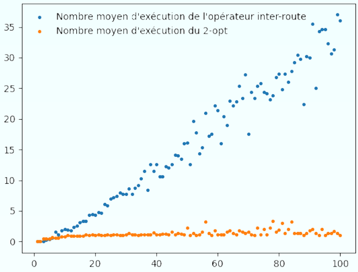
Complexité de l'algorithme



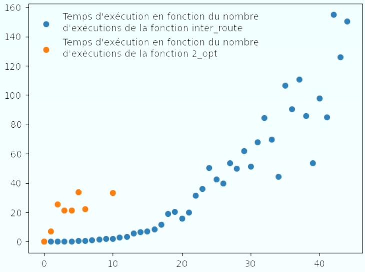
$$t = 3.35 \times 10^{-6} x^{3.78}$$

Complexité : **polynomiale**

Complexité de l'algorithme



Nombre d'appels des opérateurs en fonction de n



Temps d'exécution en fonction du nombre d'exécutions des opérateurs

Limites de l'algorithme

Limite n°1 : Ajout de contraintes

Algorithme de Clarke et Wright introduit pour le problème VRP.

Donc non adapté aux ajouts de contraintes.

Limites de l'algorithme

Limite n°1 : Ajout de contraintes

D'autres algorithmes permettent d'obtenir des solutions satisfaisantes :

Limites de l'algorithme

Limite n°1 : Ajout de contraintes

D'autres algorithmes permettent d'obtenir des solutions satisfaisantes :

- Recherche exacte :
  - Séparation et Évaluation (*Branch and Bound*)

Limites de l'algorithme

Limite n°1 : Ajout de contraintes

D'autres algorithmes permettent d'obtenir des solutions satisfaisantes :

- Recherche exacte :
  - Séparation et Évaluation (*Branch and Bound*)
- Heuristiques :
  - *Petal algorithm*
  - *Sweep algorithm*

Limites de l'algorithme

Limite n°1 : Ajout de contraintes

D'autres algorithmes permettent d'obtenir des solutions satisfaisantes :

- Recherche exacte :
  - Séparation et Évaluation (*Branch and Bound*)
- Heuristiques :
  - *Petal algorithm*
  - *Sweep algorithm*
- Métaheuristiques :
  - Recherche Tabou (*Tabu Search*)
  - Algorithme génétique (*Genetic algorithm*)
  - Colonie de fourmis (*Ant algorithm*)
  - GRASP (*Greedy Randomized Adaptive Search Procedure*)

Limites de l'algorithme

Limite n°2 : Algorithme glouton

L'algorithme de Clarke et Wright est glouton : pour chaque étape, seul le plus grand bénéfice est considéré
Aucune dégradation d'une solution n'est donc envisageable.

Limites de l'algorithme

Limite n°3 : Long temps d'exécution

Objectif : Trouver des solutions en évitant la complexité exponentielle de l'algorithme naïf.

Sur le graphe représentant le t en fonction de n : $n = 100, \quad t \approx 135\text{s}$

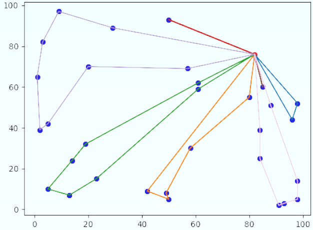
Il serait donc difficile de résoudre des instances trop grandes (jusqu'à $n = 30000$)

L'algorithme convient cependant a une utilisation "normale" (livraisons dans une zone géographique restreinte)

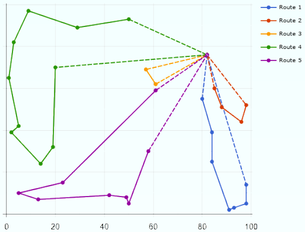
Limites de l'algorithme

Limite n°4 : Résultats non optimaux

Instance A-n32-k5



Distance : 945km

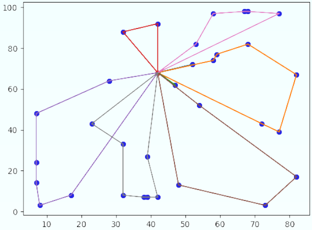


Distance : 784km

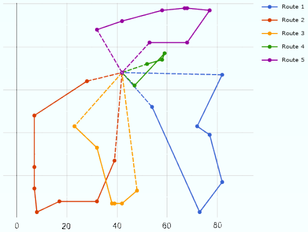
Limites de l'algorithme

Limite n°4 : Résultats non optimaux

Instance A-n33-k5



Distance : 784km



Distance : 661km

Conclusion

Nous avons vu :

— Les enjeux du problème

Conclusion

Nous avons vu :

— Les enjeux du problème

— Une première heuristique : celle de Clarke et Wright

Conclusion

Nous avons vu :

— Les enjeux du problème

— Une première heuristique : celle de Clarke et Wright

— Une heuristique de recherche locale : le 2-opt

Conclusion

Nous avons vu :

— Les enjeux du problème

— Une première heuristique : celle de Clarke et Wright

— Une heuristique de recherche locale : le 2-opt

— Une heuristique d'optimisation inter-route

Conclusion

Nous avons vu :

— Les enjeux du problème

— Une première heuristique : celle de Clarke et Wright

— Une heuristique de recherche locale : le 2-opt

— Une heuristique d'optimisation inter-route

— Les résultats

Conclusion

Nous avons vu :

- Les enjeux du problème
- Une première heuristique : celle de Clarke et Wright
- Une heuristique de recherche locale : le 2-opt
- Une heuristique d'optimisation inter-route
- Les résultats
- Les limites

Conclusion

Nous avons vu :

— Les enjeux du problème

— Une première heuristique : celle de Clarke et Wright

— Une heuristique de recherche locale : le 2-opt

— Une heuristique d'optimisation inter-route

— Les résultats

— Les limites

Il s'agirait désormais d'étendre la résolution du problème en y ajoutant des contraintes telles qu'une contrainte de capacité, ou de retour des marchandises.

Remerciements

Merci aux membres du jury pour leur écoute.

Merci à M. Bertault et M. Bouverot pour leur aide durant ces deux années.

Merci à M. Legrand-Lixon, agrégé de maths, pour l'orientation qu'il a donné à mon TIPE.

Merci enfin à M. Massoteau, ingénieur développeur chez Fastercom (et à son PDG, Romain Bonnifet), pour leur soutien et les différentes explications et aides fournies par appel visio.

Annexe
Code complet

```python
from matplotlib.pyplot import axis, grid, show, plot, scatter
from numpy import sqrt, array
from random import randint
from copy import deepcopy

DEPOT = [0, 0]
NB_INTER = 0
NB_2OPT = 0

def creer_client_alea(n=10):
    CLIENTS = []
    for _ in range(n):
        CLIENTS.append([randint(-100, 100), randint(-100, 100)])
    return CLIENTS

CLIENTS = creer_client_alea(20)
```

```python
def copie(liste):
    return deepcopy(liste)

def distance(i, j):
    return ((j[1]-i[1])**2+(j[0]-i[0])**2)**(1/2)

# Utile pour l'implementation d'un operateur inter-routes
def distance_matrix(CLIENTS=CLIENTS):
    res = [[0 for i in range(len(CLIENTS))] for j in range(len(CLIENTS))]
    for i in range(len(CLIENTS)-1):
        for j in range(i+1, len(CLIENTS)):
            res[i][j] = round(distance(CLIENTS[i], CLIENTS[j]), 2)
    return res

distance_clients = distance_matrix()
```

```python
# Construction de la liste des "savings" pour chaque couple de points
def savings(DEPOT=DEPOT, CLIENTS=CLIENTS):
    list_savings = []
    client_savings = []
    temp = []
    temp2 = []
    for i in range(0, len(CLIENTS)-1):
        for j in range(i+1, len(CLIENTS)):
            temp.append((round(distance(CLIENTS[i], DEPOT) +
                distance(CLIENTS[j],DEPOT)-distance(CLIENTS[i],CLIENTS[j]),2)))
            temp2.append((i+1, j+1))
        list_savings.append(temp)
        client_savings.append(temp2)
        temp = []
        temp2 = []
    return (list_savings, client_savings)
```

```python
# Creer les routes D-i-D
def create_routes(DEPOT=DEPOT, CLIENTS=CLIENTS):
    list_route = []
    for i in range(len(CLIENTS)):
        list_route.append([0, i+1, 0])
    return list_route

def joindre_tableaux(res=savings()):
    list_savings = res[0]
    client_savings = res[1]
    new_l_s = []
    new_c_s = []
    for i in range(len(list_savings)):
        for j in range(len(list_savings[i])):
            new_l_s.append(list_savings[i][j])
            new_c_s.append(client_savings[i][j])
    return (new_l_s, new_c_s)
```

```python
# Trier les savings afin d'en retirer les benefices les plus eleves
def order_list(x=joindre_tableaux()):
    list_savings = x[0]
    client_savings = x[1]
    for i in range(len(list_savings)):
        for j in range(i, len(list_savings)):
            if list_savings[j] > list_savings[i]:
                (list_savings[i], list_savings[j]) = (
                    list_savings[j], list_savings[i])
                (client_savings[i], client_savings[j]) = (
                    client_savings[j], client_savings[i])
    return (list_savings, client_savings)

ROUTES = create_routes()
SAVINGS = order_list()
```

```python
# Forme les routes en prenant en compte les benefices
def merge_routes(ROUTE=ROUTES,l_savings=SAVINGS[0],c_savings=SAVINGS[1]):
    temp = 0
    for i in range(len(l_savings)):
        for j in range(len(ROUTE)):
            if ROUTE[j][0] == 0 and ROUTE[j][1] == c_savings[i][1]:
                for k in range(len(ROUTE)):
                    if ROUTE[k][1]==c_savings[i][0] and ROUTE[j][2]==0:
                        _ = ROUTE[k].pop()
                        ROUTE[k].append(c_savings[i][1])
                        ROUTE[k].append(0)
                        ROUTE.remove(ROUTE[j])
                        temp = 1
                        break
            if temp == 1:
                temp = 0
                break
    return ROUTE

ROUTES = merge_routes()
APRES_INTRA_ROUTE = copie(ROUTES)
```

```python
# 2_opt pour regler les problemes de croisements intra-routes
def deux_opt(D=DEPOT, C=CLIENTS, routes=APRES_INTRA_ROUTE):
    for route in routes:
        if len(route) > 4:
            for i in range(len(route)-1):
                for j in range(len(route)-1):
                    if j != i and j != i-1 and j != i+1:
                        if route[i] == 0 and route[j+1] == 0:
                            if distance(D, C[route[i+1]-1]) + distance(C[route[j]-1], D) >
                                distance(D, C[route[j]-1]) + distance(C[route[i+1]-1], D):
                                (route[i+1], route[j]) = (route[j], route[i+1])
                        elif route[i+1] == 0 and route[j] == 0:
                            if distance(C[route[i]-1], D) + distance(D, C[route[j+1]-1]) >
                                distance(C[route[i]-1], D) + distance(D, C[route[j+1]-1]):
                                (route[i], route[j+1]) = (route[j+1], route[i])
                        elif route[i] == 0:
                            if distance(D, C[route[i+1]-1]) + distance(C[route[j]-1], C[route[j+1]-1]) >
                                distance(D, C[route[j]-1]) + distance(C[route[i+1]-1], C[route[j+1]-1]):
                                (route[i+1], route[j]) = (route[j], route[i+1])
                        elif route[j] == 0:
                            if distance(C[route[i]-1], C[route[i+1]-1]) + distance(D, C[route[j+1]-1]) >
                                distance(C[route[i]-1], D) + distance(C[route[i+1]-1], C[route[j+1]-1]):
                                (route[i], route[j+1]) = (route[j+1], route[i])
                        elif route[i+1] == 0:
                            if distance(C[route[i]-1], D) + distance(C[route[j]-1], C[route[j+1]-1]) >
                                distance(C[route[i]-1], C[route[j]-1]) + distance(D, C[route[j+1]-1]):
                                (route[i], route[j+1]) = (route[j+1], route[i])
                        elif route[j+1] == 0:
                            if distance(C[route[i]-1], C[route[i+1]-1]) + distance(C[route[j]-1], D) >
                                distance(C[route[i]-1], C[route[j]-1]) + distance(C[route[i+1]-1], D):
                                (route[i+1], route[j]) = (route[j], route[i+1])
                        elif distance(C[route[i]-1], C[route[i+1]-1]) + distance(C[route[j]-1], C[route[j+1]-1]) >
                            distance(C[route[i]-1], C[route[j]-1]) + distance(C[route[i+1]-1], C[route[j+1]-1]):
                            (route[i+1], route[j]) = (route[j], route[i+1])
```

```python
def distance_comparaison(D=DEPOT,C=CLIENTS,R_AV=ROUTES,R_AP=APRES_INTRA_ROUTE):
    d1 = 0
    d2 = 0
    for j in R_AV:
        for k in range(len(j)-1):
            if j[k] == 0:
                d1 += distance(D, C[j[k+1]-1])
            elif j[k+1] == 0:
                d1 += distance(C[j[k]-1], D)
            else:
                d1 += distance(C[j[k]-1], C[j[k+1]-1])
    for j in R_AP:
        for k in range(len(j)-1):
            if j[k] == 0:
                d2 += distance(D, C[j[k+1]-1])
            elif j[k+1] == 0:
                d2 += distance(C[j[k]-1], D)
            else:
                d2 += distance(C[j[k]-1], C[j[k+1]-1])
    return (round(d1, 2), round(d2, 2))
```

```python
def inter_route(CLIENTS=CLIENTS, routes=APRES_INTRA_ROUTE, distance_matrix=distance_clients, NB_INTER = NB_INTER):
    def plus_proche_par_route(num_client, route, CLIENTS=CLIENTS):
        res = route[1]
        for i in range(2, len(route)-1):
            if distance(CLIENTS[num_client-1], CLIENTS[res-1]) >= distance(CLIENTS[route[i]-1], CLIENTS[res-1]):
                res = route[i]
        return res
    while 1:
        res_intermediaire = []
        for route in routes:
            for point in route[1:-1]:
                dist_par_routes = []
                for route2 in routes:
                    dist_par_routes.append((plus_proche_par_route(point, route2), routes.index(route2)))
                res_intermediaire.append(((point, routes.index(route)), dist_par_routes))
        res_distance = [] # Pour stocker (point, route) le plus proche de (point2, route2) et donne la valeur de distance
        for optim in res_intermediaire:
            for i in optim[1]:
                if optim[0] != i:
                    res_distance.append((optim[0], i))
        dist_optimised = []
        for i in res_distance:
            routes_temp = copie(routes)
            d1 = distance_comparaison()[1]
            routes_temp[i[1][1]].insert(routes_temp[i[1][1]].index(i[1][0])+1, i[0][0])
            routes_temp[i[0][1]].remove(i[0][0])
            d2 = distance_comparaison(DEPOT, CLIENTS, ROUTES, routes_temp)[1]
            dist_optimised.append((res_distance.index(i), round(d1-d2, 2)))
        if dist_optimised != []:
            maxi = dist_optimised[0]
        for i in range(1, len(dist_optimised)): # Tri par valeur d'optimisation
            if dist_optimised[i][1] > maxi[1]:
                maxi = dist_optimised[i]
        if len(dist_optimised) > 0 and maxi[1] > 0:
            NB_INTER += 1
            routes[res_distance[maxi[0]][1][1]].insert(
                routes[res_distance[maxi[0]][1][1]].index(res_distance[maxi[0]][1][0])+1, res_distance[maxi[0]][0][0])
            routes[res_distance[maxi[0]][0][1]].remove(res_distance[maxi[0]][0][0])
            # dessin(routes)
        else:
            break
    return NB_INTER
```

```python
def dessin(chemin):
    X = [CLIENTS[i][0] for i in range(len(CLIENTS))]
    Y = [CLIENTS[i][1] for i in range(len(CLIENTS))]
    scatter(X, Y, color=(0.15, 0.15, 0.9))
    scatter(DEPOT[0], DEPOT[1], color=(0.9, 0.15, 0.15))
    for i in chemin:
        X = [DEPOT[0]]
        Y = [DEPOT[1]]
        for j in range(1, len(i)-1):
            X.append(CLIENTS[i[j]-1][0])
            Y.append(CLIENTS[i[j]-1][1])
        X.append(DEPOT[0])
        Y.append(DEPOT[1])
        plot(X, Y)
    show()
```

```python
# Appel de l operateur inter-routes
NB_INTER = inter_route()
dessin(APRES_INTRA_ROUTE)
distance1 = distance_comparaison(DEPOT, CLIENTS, ROUTES, APRES_INTRA_ROUTE)
print("Apres inter-route: " + str(distance1))

# Appel de l'operateur intra-routes...
FINAL2 = copie(APRES_INTRA_ROUTE)
deux_opt(FINAL2)

# ... autant de fois qu il est necessaire
temp = distance_comparaison(DEPOT, CLIENTS, APRES_INTRA_ROUTE, FINAL2)
while temp[0] != temp[1]:
    NB_2OPT += 1
    APRES_INTRA_ROUTE = copie(FINAL2)
    deux_opt(FINAL2)
    temp = distance_comparaison(DEPOT, CLIENTS, APRES_INTRA_ROUTE, FINAL2)
    dessin(APRES_INTRA_ROUTE)
```