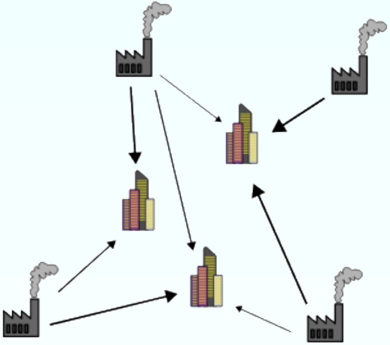
TIPE
Transport Optimal

ROCHER Kilian — WILLEM Logan

2020 / 2021

L'optimisation des transports

Qu'est-ce que l'optimisation des transports ?



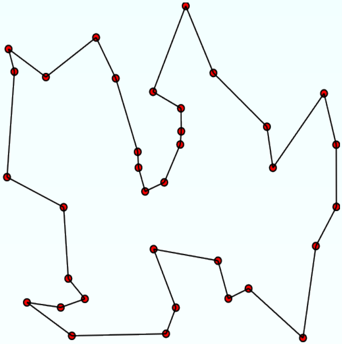
Quelques exemples

Quelques problèmes d'optimisation

Quelques exemples

Quelques problèmes d'optimisation

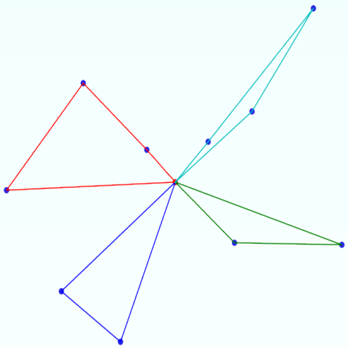
— **Le voyageur de commerce**



Quelques exemples

Quelques problèmes d'optimisation

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

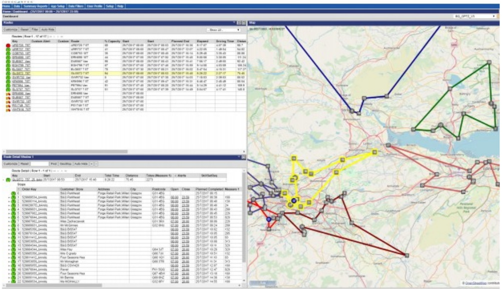


Applications réelles

INTERNATIONAL

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

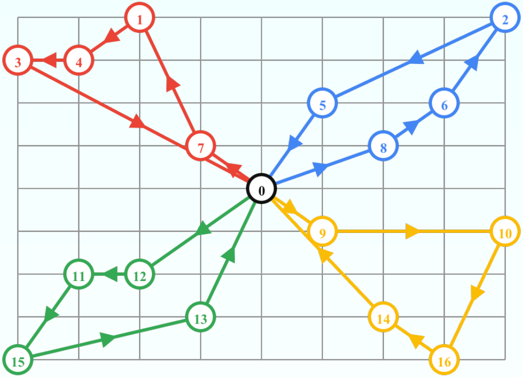
16.10.2018 • 11h00 | par Emilien VILLEROY



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.

Contexte

Dans ce type de problèmes, il s'agit de minimiser le coût total (en distance par exemple) de la tournée de tous les véhicules, ayant pour objectif de livrer à un nombre défini de clients.



Variantes

Différentes variantes du problème de tournée des véhicules

— Classique (VRP)

Variantes

Différentes variantes du problème de tournée des véhicules

— Classique (VRP)

— Contrainte de capacité (CVRP)

Variantes

Différentes variantes du problème de tournée des véhicules

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

Variantes

Différentes variantes du problème de tournée des véhicules

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

Variantes

Différentes variantes du problème de tournée des véhicules

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

On s'est intéressé à la version classique afin de s'approprier au mieux le problème.

Pourquoi utiliser une résolution approchée ?

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

Pourquoi utiliser une résolution approchée ?

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

2^n possibilités

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

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 :

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 :

complexité **exponentielle**

L'algorithme de Clarke & Wright

Données

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

L'algorithme de Clarke & Wright

Données

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

L'algorithme de Clarke & Wright

Données

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

L'algorithme de Clarke & Wright

Données

- Un point D de coordonnées $(0, 0)$: le dépôt
- Une famille de points $(i_1, \ldots, i_n) \in \left([-100, 100]^2\right)^n$ 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 & 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 $d(D,i)+d(i,j)+d(j,D)$

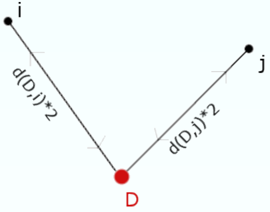
$$s(i,j) = 2d(D,i) + 2d(j,D) - [d(D,i) + d(i,j) + d(j,D)]$$
$$s(i,j) = d(D,i) + d(j,D) - d(i,j)$$

L'algorithme de Clarke & 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 $d(D,i) + d(i,j) + d(j,D)$

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

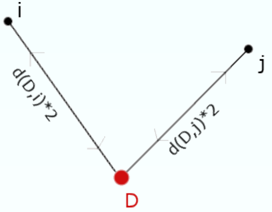


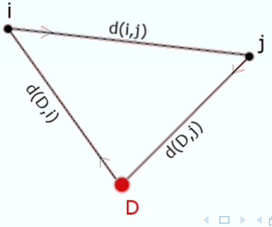
L'algorithme de Clarke & Wright

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 & Wright

L'algorithme comporte deux étapes majeures

— **Étape 1 : Calcul des bénéfices**

L'algorithme de Clarke & 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

```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)


# Créer 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()):⋯


# Trier les savings afin d'en retirer les bénéfices les plus élevés
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)
```

L'algorithme de Clarke & 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
- **Etape 2 : Fusion des routes**

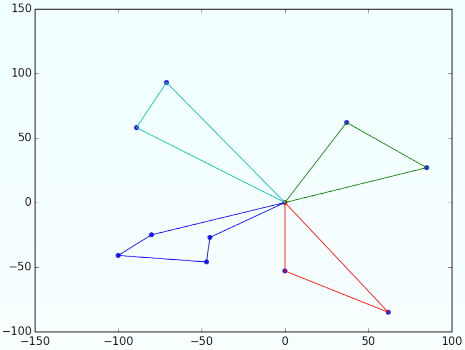
L'algorithme de Clarke & Wright

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
- **Etape 2 : Fusion des routes**
  Fusion des routes si celles-ci peuvent l'être, dans l'ordre donné par la liste précédente

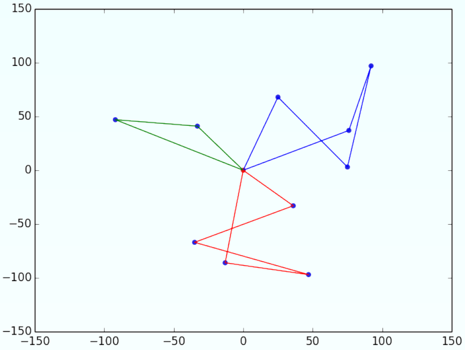
```python
# Forme les routes en prenant en compte les bénéfices
def merge_routes(ROUTE=ROUTES, list_savings=SAVINGS[0], client_savings=SAVINGS[1]):
    temp = 0
    for i in range(len(list_savings)):
        for j in range(len(ROUTE)):
            if ROUTE[j][0] == 0 and ROUTE[j][1] == client_savings[i][1]: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

Résultat



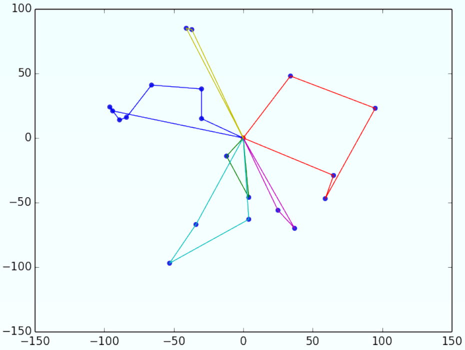
Solution satisfaisante d'un problème

Insuffisance de l'algorithme



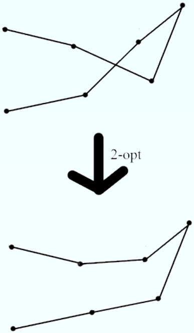
Résultat non satisfaisant pour 10 clients

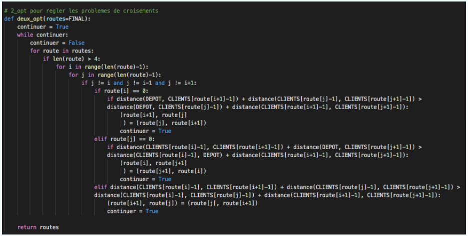
Insuffisance de l'algorithme



Résultat non satisfaisant pour 20 clients

Le 2-opt





```python
# 2_opt pour regler les problemes de croisements
def deux_opt(routes=FINAL):
    continuer = True
    while continuer:
        continuer = False
        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:
                                if distance(DEPOT, CLIENTS[route[i+1]-1]) + distance(CLIENTS[route[j]-1], CLIENTS[route[j+1]-1]) >
                                distance(DEPOT, CLIENTS[route[j]-1]) + distance(CLIENTS[route[i+1]-1], CLIENTS[route[j+1]-1]):
                                    (route[i+1], route[j]
                                     ) = (route[j], route[i+1])
                                    continuer = True
                            elif route[j] == 0:
                                if distance(CLIENTS[route[i]-1], CLIENTS[route[i+1]-1]) + distance(DEPOT, CLIENTS[route[j+1]-1]) >
                                distance(CLIENTS[route[i]-1], DEPOT) + distance(CLIENTS[route[i+1]-1], CLIENTS[route[j+1]-1]):
                                    (route[i], route[j+1]
                                     ) = (route[j+1], route[i])
                                    continuer = True
                            elif distance(CLIENTS[route[i]-1], CLIENTS[route[i+1]-1]) + distance(CLIENTS[route[j]-1], CLIENTS[route[j+1]-1]) >
                            distance(CLIENTS[route[i]-1], CLIENTS[route[j]-1]) + distance(CLIENTS[route[i+1]-1], CLIENTS[route[j+1]-1]):
                                (route[i+1], route[j]) = (route[j], route[i+1])
                                continuer = True

    return routes
```

Principe du 2-opt
Suppression des liaisons sécantes

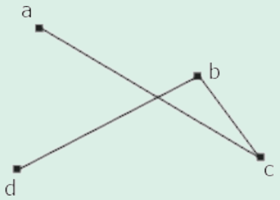
Le 2-opt

Exemple

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 : $(a, \mathbf{c}, \mathbf{b}, d) \rightarrow (a, \mathbf{b}, \mathbf{c}, d)$



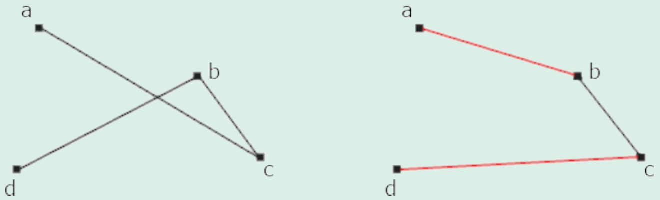
Le 2-opt

Exemple

Dans le graphe ci-dessous, on a :

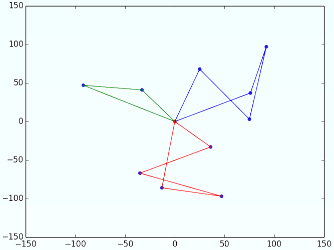
$$d(a,c) + d(b,d) \; > \; d(a,b) + d(c,d)$$

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

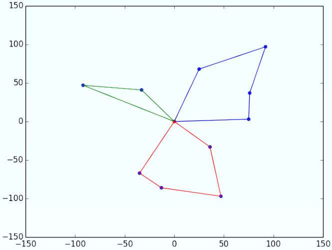


Résultats avec 2-opt

10 clients



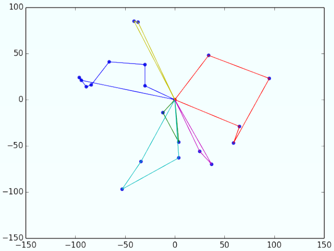
Avant



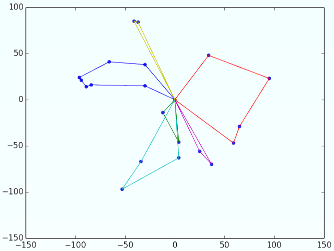
Après

Résultats avec 2-opt

20 clients



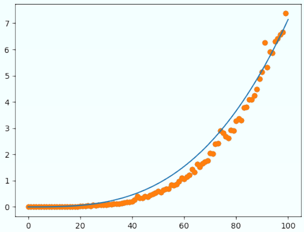
Distance : 1224km
Avant



Distance : 1193km
Après

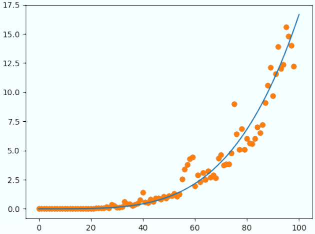
Complexité des algorithmes

Sans 2-opt



$$f(x) = x^3/14000$$

Avec 2-opt



$$f(x) = x^4/6000000$$

Limites de l'algorithme ?

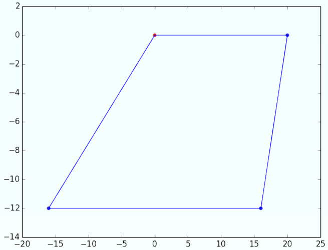
Mauvaise résolution pour les points suivants :

$$i_1 = (20, 0),\ i_2 = (-16, -12),\ i_3 = (16, -12)$$

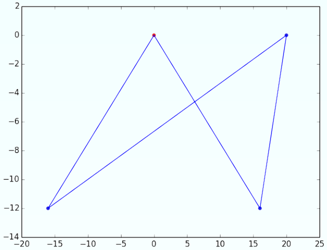
Limites de l'algorithme ?

Mauvaise résolution pour les points suivants :

$$i_1 = (20, 0),\ i_2 = (-16, -12),\ i_3 = (16, -12)$$



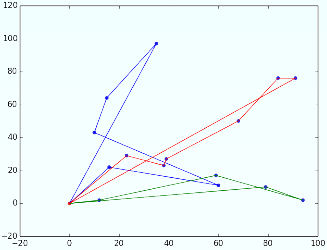
Distance : 85km
Avant



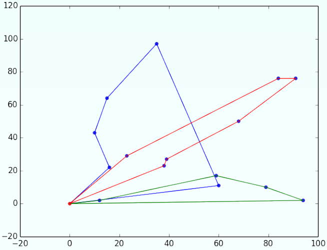
Distance : 91km
Après

Limites de l'algorithme ?

Résolution moins satisfisante lorsqu'il s'agit de quarts de plan



Distance : 742km
Avant



Distance : 695km
Après