Operational Research - Metaheuristics

June 29, 2019

BOUZAIEN Mokhles & HENIA Rami & WANG Steven

[292]:
```
# Imports
from pulp import *
import pandas as pd
import numpy as np
from itertools import *
import matplotlib.pyplot as plt
import random
from scipy import interpolate
from scipy.signal import savgol_filter
from scipy.optimize import leastsq
import scipy as sc
import seaborn as sns
```

[293]:
```
# File Name
InputData = 'InputDataTelecomLargeInstance.xlsx'
```

[294]:
```
# Input Data Preparation
def read_excel_data(filename, sheet_name):
  data = pd.read_excel(filename, sheet_name=sheet_name, header=None)
  values = data.values
  if min(values.shape) == 1:  # This If is to make the code insensitive to␣
↪column-wise or row-wise expression #
      if values.shape[0] == 1:
          values = values.tolist()
      else:
          values = values.transpose()
          values = values.tolist()
      return values[0]
  else:
      data_dict = {}
      if min(values.shape) == 2:  # For single-dimension parameters in Excel
          if values.shape[0] == 2:
              for i in range(values.shape[1]):
                  data_dict[i+1] = values[1][i]
          else:
```

```
            for i in range(values.shape[0]):
                data_dict[i+1] = values[i][1]

    else:  # For two-dimension (matrix) parameters in Excel
        for i in range(values.shape[0]):
            for j in range(values.shape[1]):
                data_dict[(i+1, j+1)] = values[i][j]
    return data_dict
```

[295]:
```
# Create parameters
param_C = read_excel_data(InputData, 'C')                                  # set␣
 ↪of customers
param_M = read_excel_data(InputData, 'M')                                  # set␣
 ↪of end offices
param_N = read_excel_data(InputData, 'N')                                  # set␣
 ↪of digital hubs
param_h = read_excel_data(InputData, 'CustToTargetAllocCost(hij)')         # cost␣
 ↪of allocating customer i to end office j
param_c = read_excel_data(InputData, 'TargetToSteinerAllocCost(cjk)')      # cost␣
 ↪of allocating end office j to digital hub k
param_g = read_excel_data(InputData, 'SteinerToSteinerConnctCost(gkm)') #␣
 ↪digital hub k to digital hub m
param_f = read_excel_data(InputData, 'SteinerFixedCost(fk)')               #␣
 ↪digital hub k fixed cost
param_alpha = read_excel_data(InputData, 'alpha')                          #␣
 ↪minimum percentage of surved customers
param_u = read_excel_data(InputData, 'TargetCapicity(Uj)')                 # end␣
 ↪office j capacity
param_v = read_excel_data(InputData, 'SteinerCapacity(Vk)')                #␣
 ↪digital hub k capacity
```

[296]:
```
# Create sets
set_C = [i for i in range(1,param_C[0]+1)]  #Customers
set_M = [j for j in range(1,param_M[0]+1)]  #End Offices
set_N = [k for k in range(1,param_N[0]+1)]  #Digital Hubs
```

[297]:
```
# Hubs' fixed cost
def fixedCost(SRlist):
    '''
    return the total digital hub fixed cost
    '''
    return sum(param_f[hub-1] for hub in SRlist)
```

[298]:
```
# Testing the fixed cost function
testListHH = [2,1,4,6,3,5]
fixedCost([3,4,1]) , fixedCost([4,3,1])
```

[298]: (3553, 3553)

[299]:
```
def cost(SRlist, param):
    '''
    SRlist : the solution representation list
    pram : can be CE (for Customer-End office), EH (for End office-Hub) or HH␣
↪(for Hub-Hub)
    return : the total cost
    '''
    if param == 'HH':
        s = param_g[(SRlist[0],SRlist[-1])]
        for k in range(len(SRlist)-1):
            s += param_g[(SRlist[k],SRlist[k+1])]
        return s
    d = {'CE': param_h, 'EH': param_c}
    return sum(d[param][(i+1,SRlist[i])] for i in range(len(SRlist)) if␣
↪SRlist[i] != 0)
```

[300]:
```
# Testing the cost function
testListCE = [2, 1, 0, 3, 2, 4, 3, 0]
testListEH = [3,1,4,5]
cost(testListCE, 'CE'), cost(testListEH, 'EH')
```

[300]: (457, 198)

[301]:
```
# Objective function
def objectiveFunction(CElist, EHlist, HHlist):
    '''
    This calculates the objective function of the given solution (CElist,␣
↪EHlist, HHlist)
    '''
    return cost(CElist, 'CE') + cost(EHlist, 'EH') + cost(HHlist, 'HH') +␣
↪fixedCost(HHlist)
```

[302]:
```
objectiveFunction(testListCE, testListEH, testListHH)
objectiveFunction([5,8,4,2,8,5,4,4,1,3,2,2,5,0,1], [1,3,3,3,4,4,3,1], [3, 1, 4])
```

[302]: 5690

[303]:
```
# Plotting function
def graphing(iterList, costList, iterMinList = None, costMinList = None, smooth␣
↪= True):
    '''
    This function is used to plot graphs
    smooth parameter allows smoothing the graph using interpolation
    '''
    plt.figure(num=None, figsize=(8, 6), dpi=100, facecolor='w', edgecolor='k')
```

```
    if iterMinList is not None:
        plt.plot(iterMinList, costMinList, '--bo', color = 'red', label =␣
 ↪'Local Minimums')
    if smooth:
        x_sm = np.array(iterList)
        y_sm = np.array(costList)
        tck = interpolate.splrep(x_sm, y_sm, s=0)
        xnew = np.linspace(x_sm.min(), x_sm.max(), 500)
        ynew = interpolate.splev(xnew, tck, der=0)
        plt.plot(xnew, ynew, 'blue', linewidth=1, label = 'Cost Variations␣
 ↪(interpolation)')
    else:
        plt.plot(iterList,costList, color = 'green', label = 'Cost Variations')
    plt.xlabel('itrations')
    plt.ylabel('cost')
    plt.legend()
    plt.show()
```

[304]:

```
def nearestHub(currentList):
    '''
    return the nearest hub to the last hub in currentList (nearest = least␣
 ↪cost)
    '''
    values = {key:param_g[key] for key in param_g if key[0] == currentList[-1]␣
 ↪and key[1] not in currentList}
    nearest = min(values, key = values.get)[1]
    return nearest
```

[305]:

```
# Initial Solution Generation : Greedy (nearest neighbor)
def initialSolution():
    '''
    return an initial solution based on Greedy algorithm
    '''
    # Constraint 9 : Covering constraints
    if int(param_C[0] * param_alpha[0]) - param_C[0] * param_alpha[0] == 0:
        n = int(param_C[0] * param_alpha[0])
    else :
        n = int(param_C[0] * param_alpha[0]) + 1
    h0 = random.randint(1,len(set_N))
    hSolution = [h0]
    eSolution = []
    cSolution = []
    # Constarint 8 : A ring must have at least three digital hubs
    while len(hSolution) < 3:
        hSolution.append(nearestHub(hSolution))

    for j in range(1, len(set_M)+1):
```

```
        # Constraint 3 : Allocation of end offices to located digital hubs␣
↪(key[1] in hSolution)
        values = {key:param_c[key] for key in param_c if key[0] == j and key[1]␣
↪in hSolution}
        eSolution.append(min(values, key = values.get)[1])

    for i in range(1, len(set_C)+1):
        if i <= n:
            values = {key:param_h[key] for key in param_h if key[0] == i}
            cSolution.append(min(values, key = values.get)[1])
        else:
            cSolution.append(0)

    return cSolution, eSolution, hSolution
```

[306]:

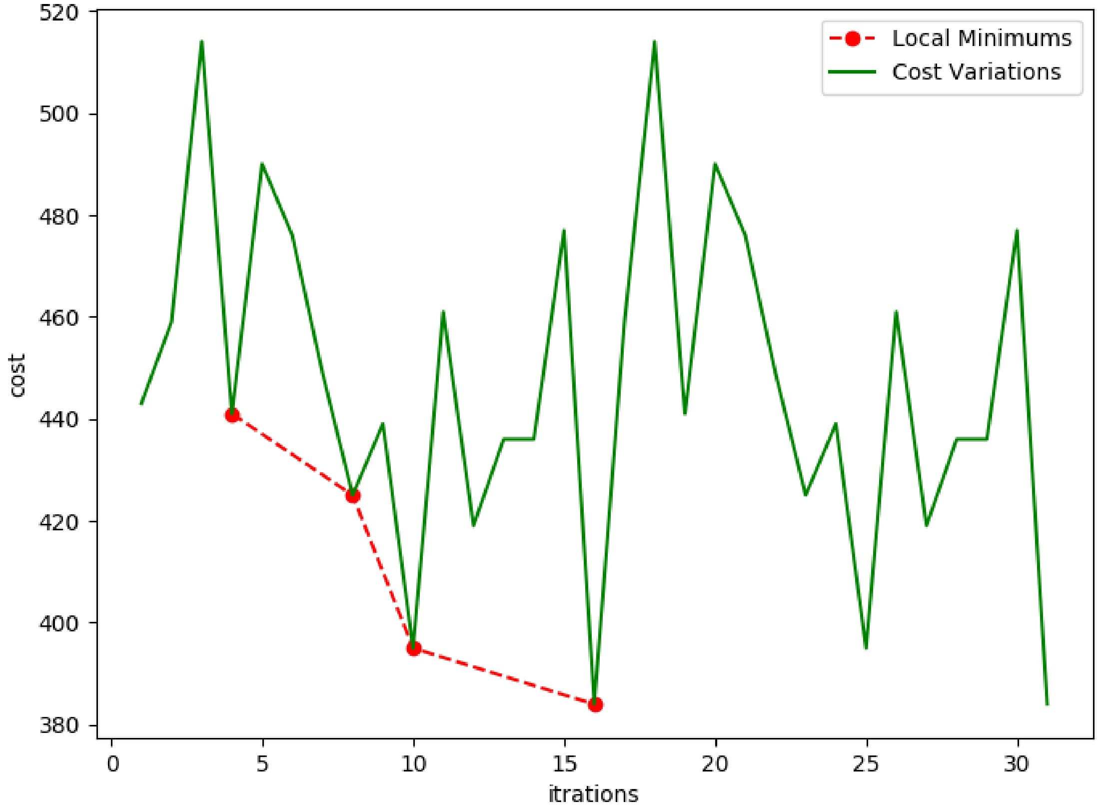
```
# Initial Solution Generation : Random
def initialSolutionRandom(hub = 3):
    '''
    return a random solution
    '''
    if int(param_C[0] * param_alpha[0]) - param_C[0] * param_alpha[0] == 0:
        n = int(param_C[0] * param_alpha[0])
    else :
        n = int(param_C[0] * param_alpha[0]) + 1
    hSolution = random.sample(range(1, len(set_N)+1), hub)
    eSolution = []
    for j in range(len(set_M)):
        eSolution.append(random.choice(hSolution))
    cSolution = []
    for i in range(len(set_C)):
        if cSolution.count(0) < len(set_C) - n:
            cSolution.append(random.choice([0]+set_M))
        else:
            cSolution.append(random.choice(set_M))
    return cSolution, eSolution, hSolution
```

[307]:

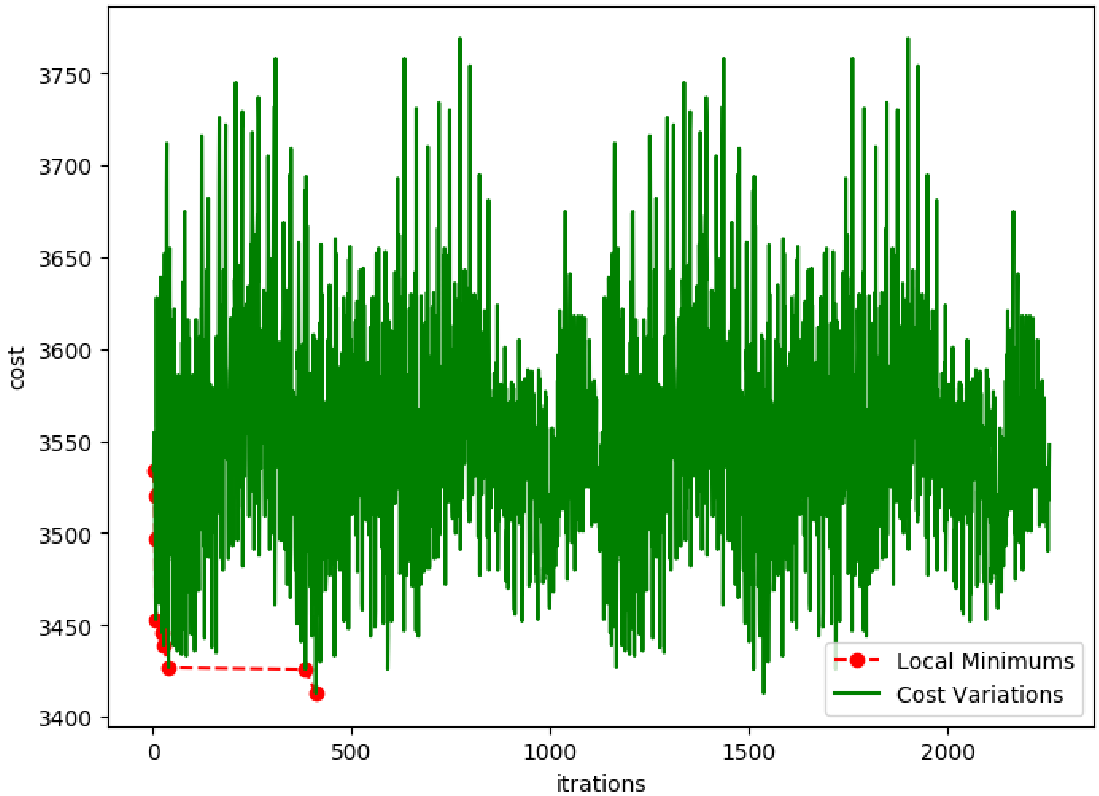
```
# Local Search : 2-opt
def two_opt(SRlist, param, graph = False):
    bestRoute = SRlist.copy()
    bestCost  = cost(SRlist, param)
    k = 1
    iterList, costList, iterMinList, costMinList = [k], [bestCost], list(),␣
↪list()
    improved = True
    while improved:
        improved = False
        for i in range(1, len(SRlist)-2):
```

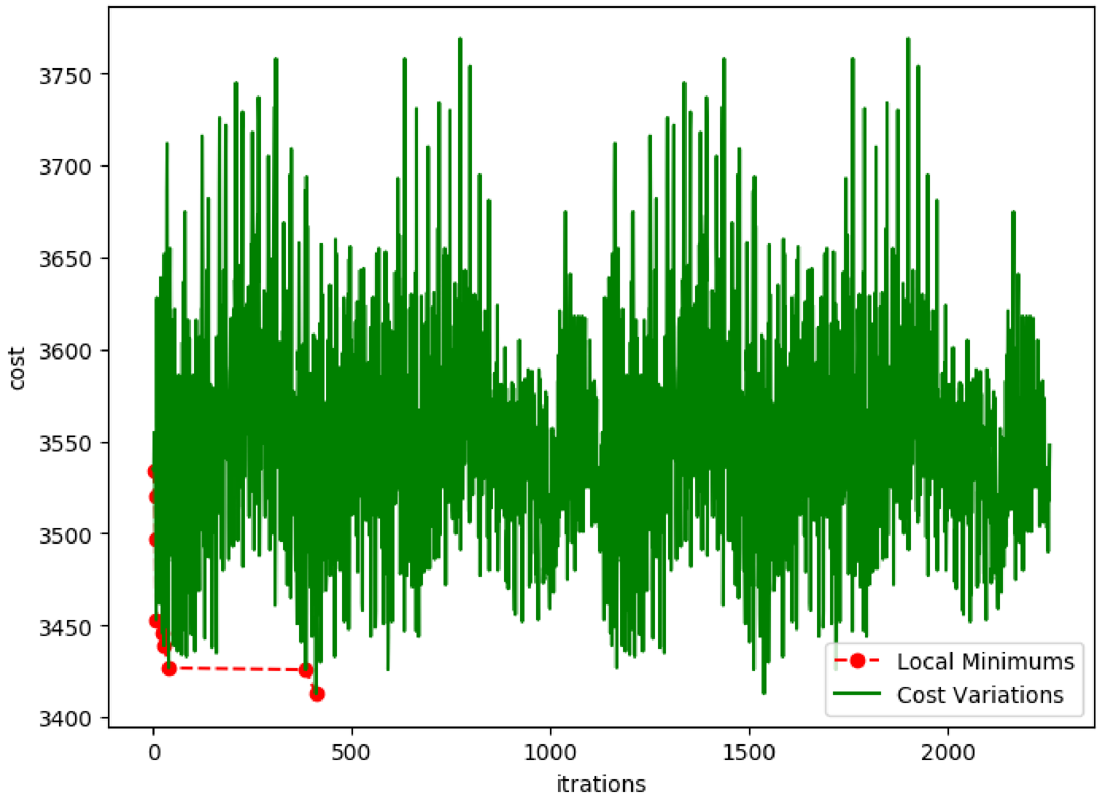
```
            for j in range(i+1, len(SRlist)):
                if j-i == 1:
                    continue # changes nothing, skip then
                newRoute = SRlist.copy()
                newRoute[i:j] = SRlist[j-1:i-1:-1] # this is the 2-opt Swap
                newCost = cost(newRoute, param)
                k += 1
                iterList.append(k)
                costList.append(newCost)
                if newCost < cost(bestRoute, param):
                    bestRoute = newRoute
                    bestCost  = newCost
                    iterMinList.append(k)
                    costMinList.append(bestCost)
                    improved  = True
    if graph:
        graphing(iterList, costList, iterMinList, costMinList, smooth = False)
    return bestRoute, bestCost
```

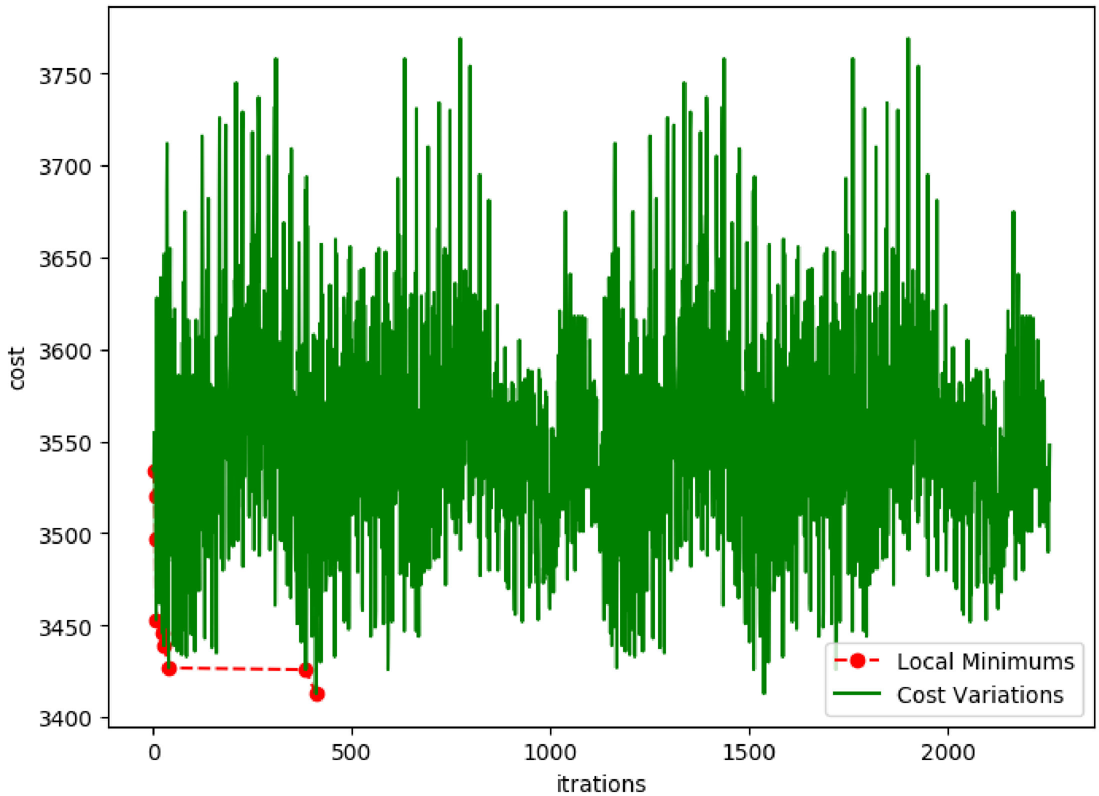
[308]:
```
# Testing 2-opt
testList = []
for i in range(len(set_C)):
    testList.append(random.randint(0,len(set_M)))
%timeit two_opt(testList, 'CE', graph = True)
```



Local Minimums
Cost Variations
cost
itrations
520
500
480
460
440
420
400
380
0
5
10
15
20
25
30


3750
3700
3650
3600
3550
3500
3450
3400
cost
0
500
1000
1500
2000
itrations
Local Minimums
Cost Variations


3750
3700
3650
3600
3550
3500
3450
3400
cost
0
500
1000
1500
2000
itrations
Local Minimums
Cost Variations



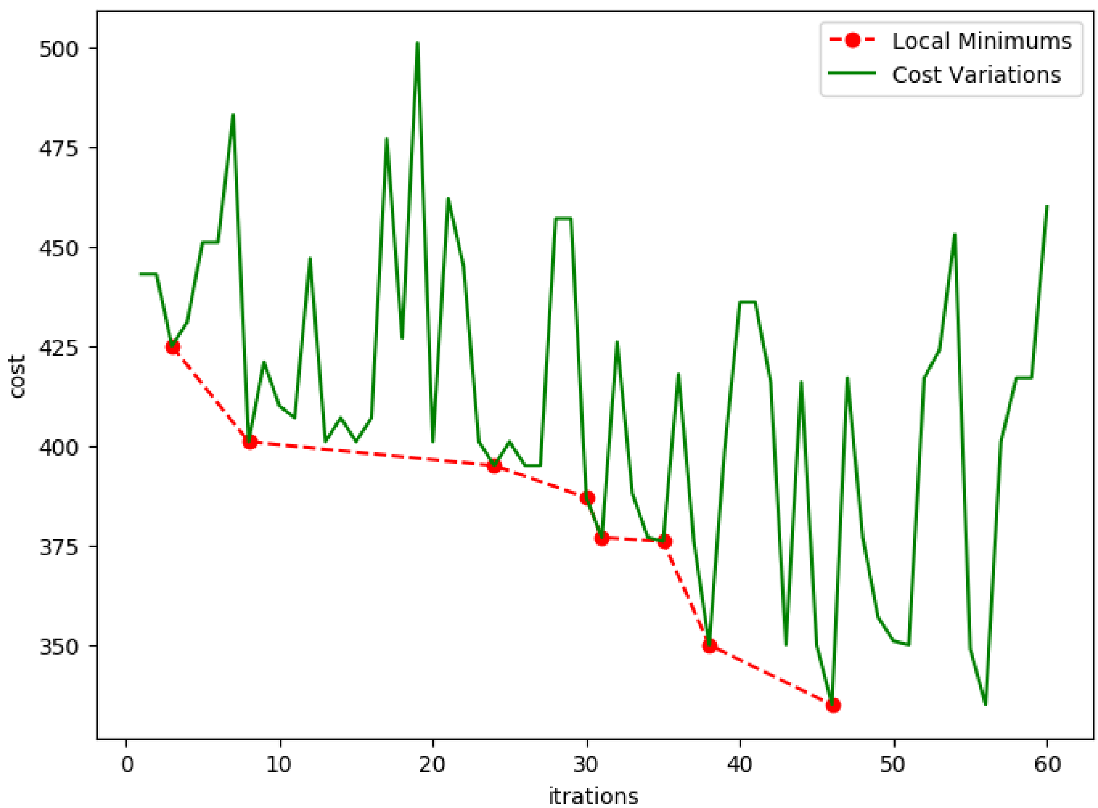

```
1 loop, best of 3: 366 ms per loop
```

Graphe 1 : two-opt : on remarque que la fonction two-opt effectue bien la fonction souhaitée. Le cost s'améliore quand le nombre d'itérations augmente. Mais bien que plus de 2000 itération sont faites, la solution optimale a été trouvé avant la 500ème iteration.

```
[309]: # Local Search : Swap
def swap(SRlist, param, threshold = 0, maxIter = 250, graph = False):
    newRoute = SRlist.copy()
    newCost  = cost(SRlist, param)
    n = len(SRlist)
    k = 1
    iterList, costList, iterMinList, costMinList = [k], [newCost], list(),␣
 ↪list()
    while newCost > threshold and k < maxIter:
        k += 1
        indexList = random.sample(range(0, n), 2)
        indexList.sort()
        index1, index2 = indexList
        tempRoute = newRoute[:index1] + newRoute[index1:index2][::-1] +␣
 ↪newRoute[index2:]
        tempCost = cost(tempRoute, param)
```

```
        if  tempCost < newCost:
            newRoute = tempRoute
            newCost = tempCost
            iterMinList.append(k)
            costMinList.append(newCost)
        iterList.append(k)
        costList.append(tempCost)
    if graph:
        graphing(iterList, costList, iterMinList, costMinList, smooth = False)
    return newRoute, newCost
```

[310]:
```
swap(testList, 'CE', threshold = 300, maxIter = 60, graph = True)
```



[310]:
```
([14,
  13,
  8,
  11,
  8,
  5,
  9,
  4,
```

```
  6,
  5,
  15,
  4,
  11,
  10,
  13,
  7,
  4,
  4,
  12,
  6,
  0,
  12,
  6,
  14,
  2,
  8,
  7,
  14,
  6,
  14,
  13,
  2,
  12,
  9,
  5,
  10,
  3,
  3,
  2,
  11,
  12,
  14,
  5,
  14,
  15,
  5,
  9,
  10,
  6,
  0],
 3206)
```

Graphe 2 : swap : on remarque que la fonction swap effectue bien la fonction souhaitée. Le cost s'améliore quand le nombre d'itérations augmente.

[311]:
```
# Applying double-bridge move to perturbate a given SRlist
def perturbate(SRlist):
    n = len(SRlist)
    indexList = random.sample(range(0, n), 3) # generate three different random␣
 ↪integers
    indexList.sort()
    index1, index2, index3 = indexList
    return SRlist[:index1] + SRlist[index3:] + SRlist[index2:index3] +␣
 ↪SRlist[index1:index2]
```

[312]:
```
testListCE, perturbate(testListCE), list(set(testListCE)),␣
 ↪len(list(set(testListCE)))
```

[312]: ([2, 1, 0, 3, 2, 4, 3, 0], [2, 1, 0, 0, 3, 3, 2, 4], [0, 1, 2, 3, 4], 5)

[313]:
```
# Single swap perturbation
def singleSwap(SRlist):
    n = len(SRlist)
    tempList = SRlist.copy()
    indexList = random.sample(range(0, n), 2) # generate two different random␣
 ↪integers
    indexList.sort()
    index1, index2 = indexList
    tempList[index1], tempList[index2] = SRlist[index2], SRlist[index1]
    return tempList
```

[314]:
```
# Constraint 6 : End office capacity constraint
def capacityEO(SRlist):
    SRset = list(set(SRlist))
    check = True
    k = 0
    while check and k < len(SRset):
        if SRset[k] != 0:
            check = SRlist.count(SRset[k]) <= param_u[SRset[k]-1]
        k += 1
    return check
```

[315]:
```
# Constraint 7 : Digital hub capacity constraint
def capacityHub(CElist, EHlist, HHlist):
    capacityDict = {hub:0 for hub in EHlist}
    for eo in range(len(EHlist)):
        capacityDict[EHlist[eo]] += CElist.count(eo+1)
    k = 0
    check = True
    while check and k < len(HHlist):
        check = capacityDict[HHlist[k]] <= param_v[HHlist[k]-1]
        k += 1
```

```
    return check
```

[317]:
```
# Acceptance criterion
def acceptCriter(s, sPrime, threshold = 1):
    CElist, EHlist, HHlist = s
    CElistPrime, EHlistPrime, HHlistPrime = sPrime
    if objectiveFunction(CElist, EHlist, HHlist) * threshold >␣
↪objectiveFunction(CElistPrime, EHlistPrime, HHlistPrime) :
        '''and capacityEO(CElistPrime) and capacityHub(CElistPrime,␣
↪EHlistPrime, HHlistPrime)'''
        return sPrime, objectiveFunction(CElistPrime, EHlistPrime,␣
↪HHlistPrime), True
    return s, objectiveFunction(CElist, EHlist, HHlist), False
```

[326]:
```
def ILS(random = True, localSearch = '2-opt', perturbation = 'dbm', iteration =␣
↪500, threshold = 1, graph = False):
    iterList = [1,2]
    costList = []
    iterMinList, costMinList = list(), list()

    if random:
        s = initialSolutionRandom()
    else:
        s = initialSolution()

    SRlistCE, SRlistEH, SRlistHH = s
    initialCost = objectiveFunction(SRlistCE, SRlistEH, SRlistHH)
    costList.append(initialCost)

    if localSearch == '2-opt':
        s_Star = two_opt(SRlistCE, 'CE')[0], two_opt(SRlistEH, 'EH')[0],␣
↪two_opt(SRlistHH, 'HH')[0]
    elif localSearch == 'swap':
        s_Star = swap(SRlistCE, 'CE')[0], swap(SRlistEH, 'EH')[0],␣
↪swap(SRlistHH, 'HH')[0]

    SRlistCE_Star, SRlistHE_Star, SRlistHH_Star = s_Star
    bestCost = objectiveFunction(SRlistCE_Star, SRlistHE_Star, SRlistHH_Star)
    costList.append(bestCost)
    condition = True

    k = 2
    while condition and k < iteration:
        if perturbation == 'dbm':
            s_Prime = perturbate(SRlistCE_Star), perturbate(SRlistHE_Star),␣
↪perturbate(SRlistHH_Star)
```

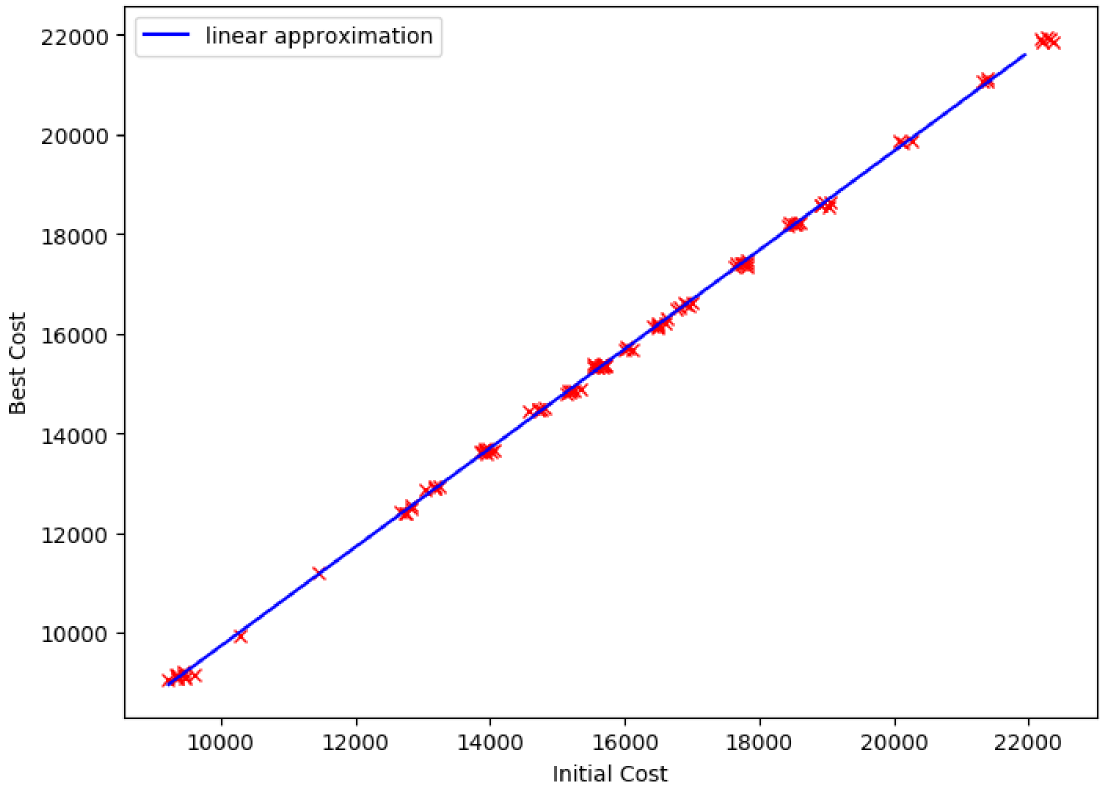
```
        else :
            s_Prime = singleSwap(SRlistCE_Star), singleSwap(SRlistHE_Star),␣
↪singleSwap(SRlistHH_Star)

        if localSearch == '2-opt':
            s_Prime = two_opt(s_Prime[0], 'CE', graph = False)[0],␣
↪two_opt(s_Prime[1], 'EH', graph = False)[0], two_opt(s_Prime[2], 'HH', graph␣
↪= False)[0]
        elif localSearch == 'swap':
            s_Prime = swap(s_Prime[0], 'CE', graph = False)[0],␣
↪swap(s_Prime[1], 'EH', graph = False)[0], swap(s_Prime[2], 'HH', graph =␣
↪False)[0]

        s_Star, cost, condition = acceptCriter(s_Star, s_Prime, threshold)
        bestCost = min(bestCost, cost)
        SRlistCE_Star, SRlistHE_Star, SRlistHH_Star = s_Star
        k += 1
        iterList.append(k)
        costList.append(cost)
    if graph:
        graphing(iterList, costList, smooth = False)
    return initialCost, bestCost
```

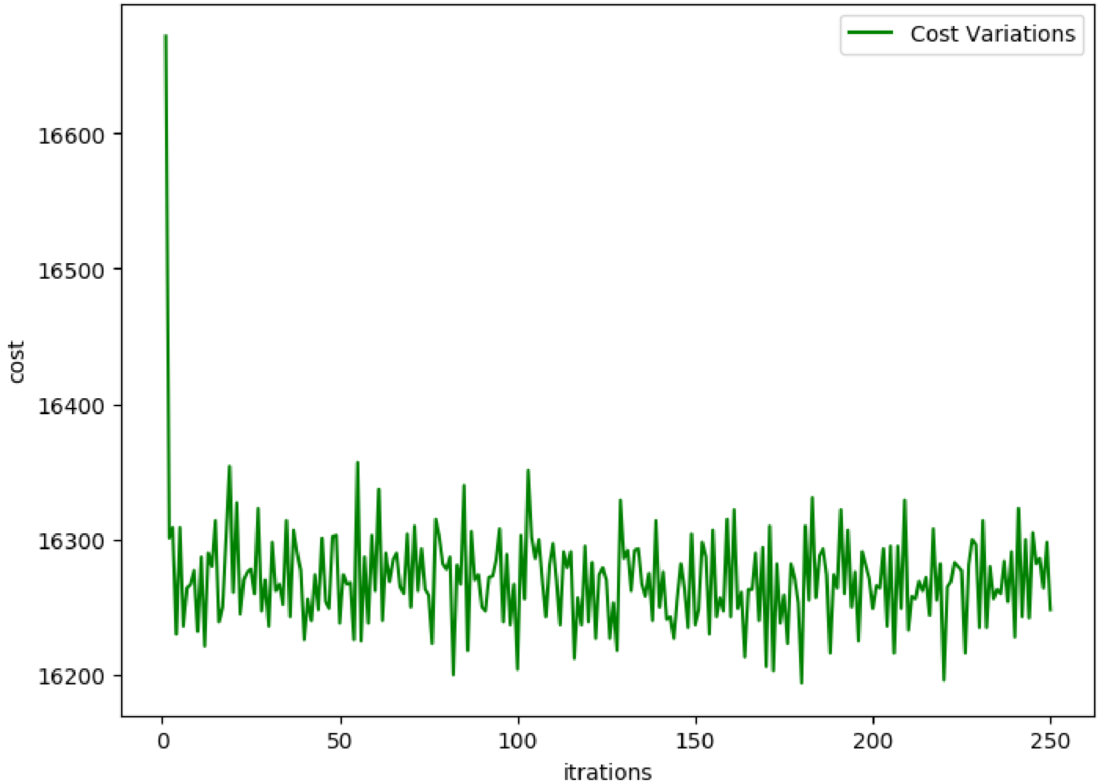
[327]:

```
ILS(random = True, localSearch = '2-opt', iteration = 250, threshold = 1.05,␣
↪graph = True)
```



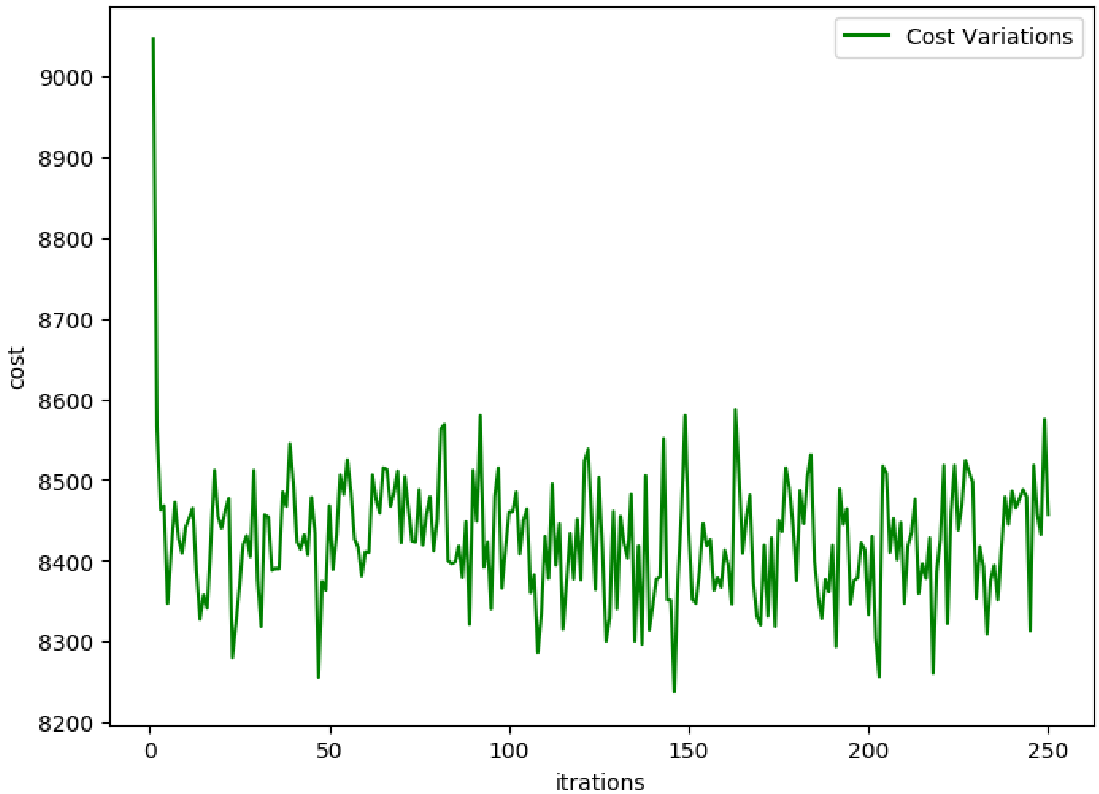
[327]: (9954, 9114)

[329]:
```
ILS(random = True, localSearch = '2-opt', perturbation = 'singleSwap',␣
 ↪iteration = 250, threshold = 1.05, graph = True)
```



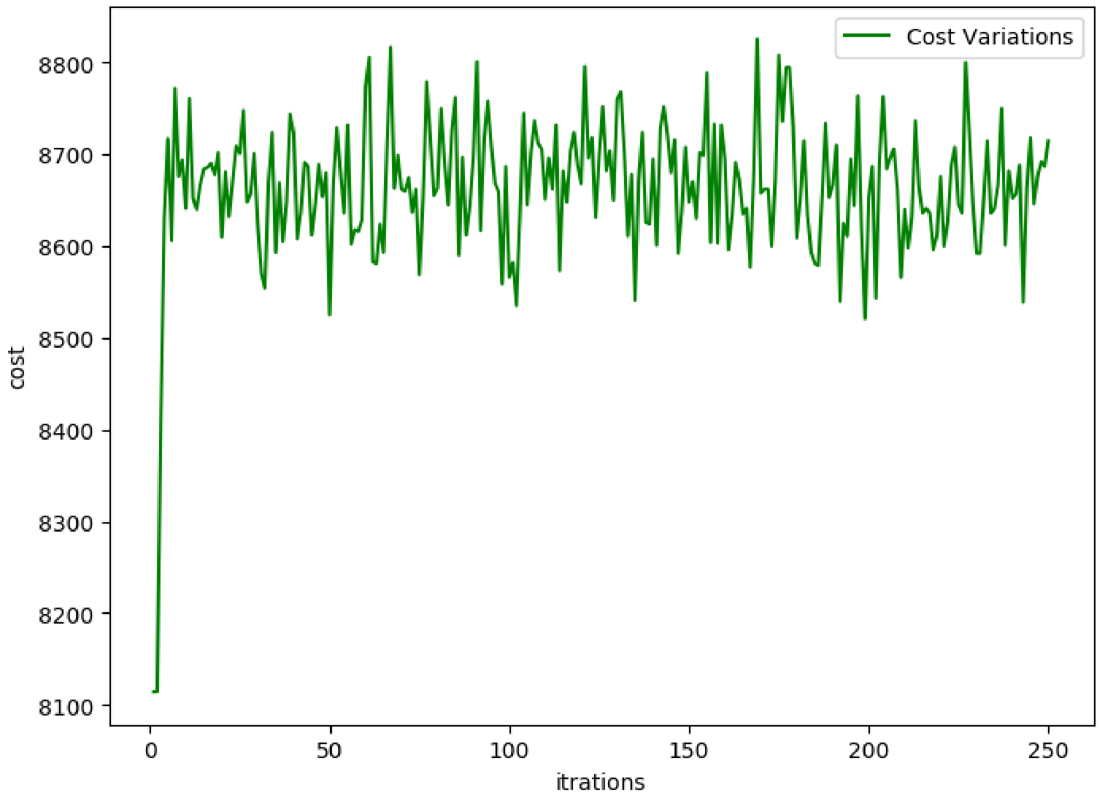
[329]: (8674, 7875)

[321]:
```
ILS(random = True, localSearch = 'swap', iteration = 250, threshold = 1.05,␣
 ↪graph = True)
```



[321]: (9047, 8237)

[322]:
```
ILS(random = False, localSearch = 'swap', iteration = 250, threshold = 1.05,␣
 ↪graph = True)
```



[322]: (8114, 8114)

Ces quatre derniers graphes illustrent l'importance du choix de la solution iniale pour le fonctionnement de l'ILS. On remarque que la solution initiale aléatoire s'améliore significativement dès les premières itérations, puis on accepte une solution qui est moins bien que le premier optimum local mais qui vérifie le critère d'acceptance (obtenue par une perturbation puis une autre recherche locale) et ainsi de suite. Pour une solution iniale bien choisi, la fonction objective augmente globalement et on s'eloigne de la solution optimale bien qu'au cours de la recherche on quitte les minimums locaux pour chercher un peu 'loin'.

[323]:
```
# Finding a link between initial cost and best cost
X, Y, T, Z = list(), list(), list(), np.linspace(1, 1.5, 50)
for i in Z:
    x,y = ILS(random = True, localSearch = 'swap', iteration = 250, threshold =␣
 ↪i, graph = False)
    X.append(x)
    Y.append(y)
    T.append(abs(x-y))
```

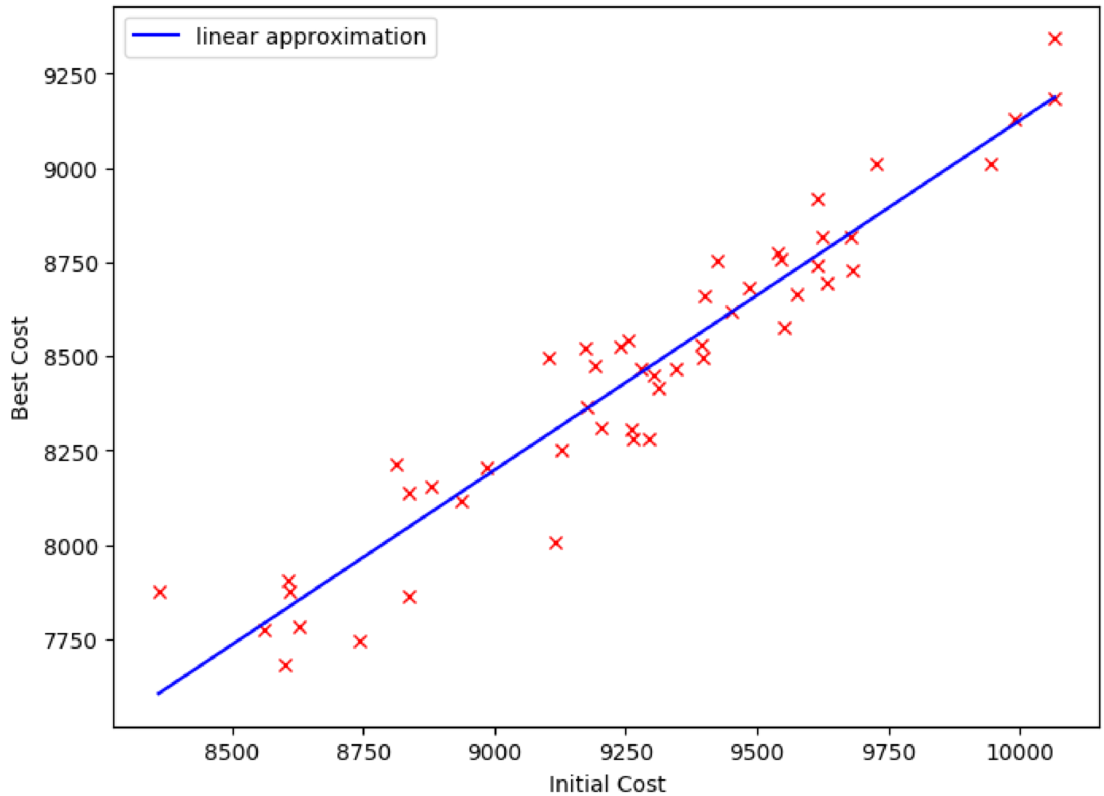
[324]:
```
# Modeling functions
def funcLine(x, a, b):
```

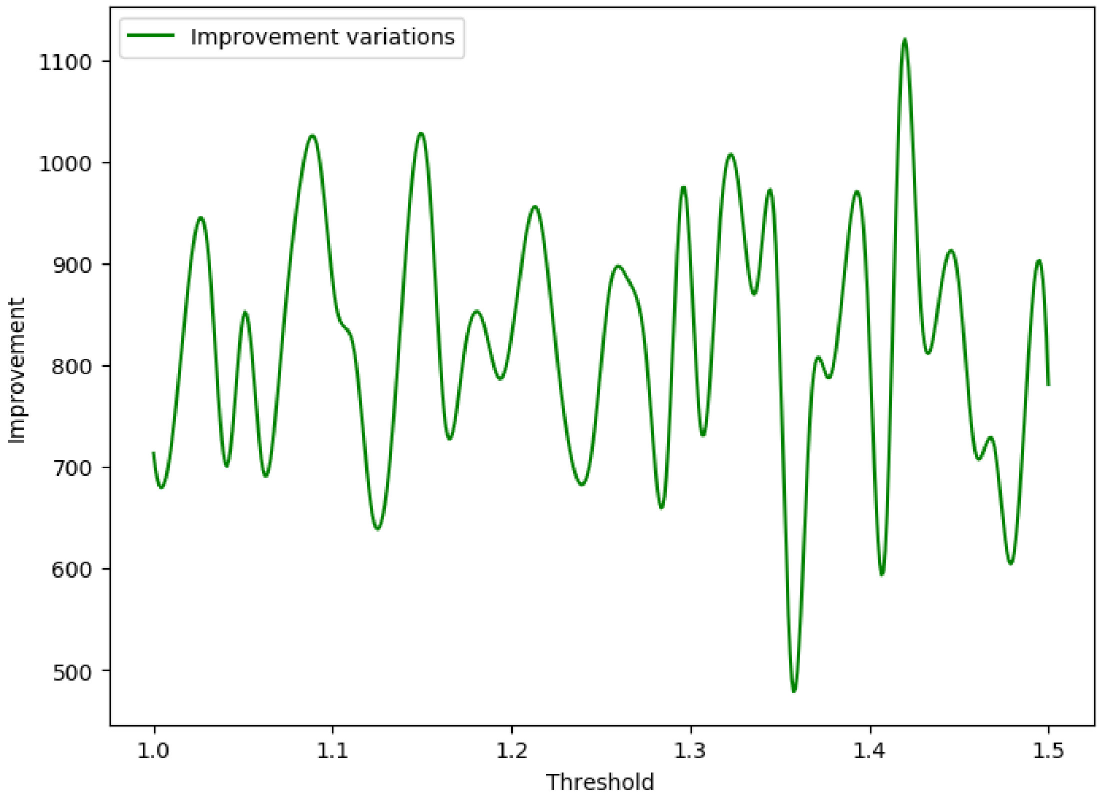
```
    return a*x+b

# Optimize constants for the linear function
constantsLine, _ = sc.optimize.curve_fit (funcLine, X, Y)

Xlin = np.linspace(min(X),max(X),100)
Ylin = funcLine(Xlin, *constantsLine)

fig = plt.figure(num=None, figsize=(8, 6), dpi=100, facecolor='w',␣
 ↪edgecolor='k')
plt.plot(X, Y, 'rx')
plt.plot(Xlin, Ylin, 'r-', label = 'linear approximation', color = 'blue')
plt.xlabel('Initial Cost')
plt.ylabel('Best Cost')
plt.legend()
plt.show()
fig = plt.figure(num=None, figsize=(8, 6), dpi=100, facecolor='w',␣
 ↪edgecolor='k')
z_sm = np.array(Z)
t_sm = np.array(T)
tck = interpolate.splrep(z_sm, t_sm, s=0)
znew = np.linspace(z_sm.min(), z_sm.max(), 500)
tnew = interpolate.splev(znew, tck, der=0)
plt.plot(znew, tnew, color = 'green', label = 'Improvement variations')
plt.xlabel('Threshold')
plt.ylabel('Improvement')
plt.legend()
plt.show()
```


linear approximation
9250
9000
8750
8500
8250
8000
7750
Best Cost
8500
8750
9000
9250
9500
9750
10000
Initial Cost



Graphe 5 : Ici on trace le meilleur cout obtenu en fonction du cout initial. On trouve que le meilleur cout varie presque linéairement en fonction du cout initial. Cela prouve donc l'importance de la solution initiale et son impact sur la solution finale. On cherche donc à trouver des solutions initiales qui s'approchent de la solution exacte pour améliorer la solution finale. Cela confirme le choix de l'algorithme glouton comme solution initiale.

Graphe 6 : On cherche à étudier l'impact du treshold sur l'amélioration de la solution. Cette figure illustre qu'on ne peut pas déduire une relation claire entre l'amélioration et le treshold. Cependant, pour certaines valeurs de treshold choisies judicieusement, l'amélioration obtenue est signficative.

Remarque : d'autres tests peuvent être effectués en agissant sur les paramètres random, localSearch, iteration, perturbation et threshhold de la fonction ILS().

[ ]:

```
```