1- Apply Savings algorithm for the VRP (note that this algorithm should be rewritten for the VRP and not TSP and the data is asymmetric)

Importing necessary libraries

In [7]:
```
import pandas as pd
import numpy as np
```

Read the distance matrix & demand from the excel file Exam-Data.xls, I used the following link to learn how to do it: https://stackoverflow.com/questions/26521266/using-pandas-to-pd-read-excel-for-multiple-worksheets-of-the-same-workbook#:~:text=xls%20%3D%20pd.ExcelFile(%27path_to_file.xls%27)%0Adf1%20%3D (https://stackoverflow.com/questions/26521266/using-pandas-to-pd-read-excel-for-multiple-worksheets-of-the-same-workbook#:~:text=xls%20%3D%20pd.ExcelFile(%27path_to_file.xls%27)%0Adf1%20%3D

In [8]:
```
#To read different sheets in an excel file we can you this block of
xls = pd.ExcelFile('real_distances_30_customers.xlsx')
df_distance_matrix = pd.read_excel(xls, 'distance', header=None) # h
df_demand = pd.read_excel(xls, 'demand', index_col=0) # col= 0 becau
```

In [9]:
```
d = df_distance_matrix.values.tolist() # to make pandas df a python'
```

"demand = demand_df.values.tolist()" is gives the demand list in lists to make it not lists in a list I used following link, https://stackoverflow.com/questions/716477/join-list-of-lists-in-python#:~:text=x%20%3D%20%5B%5B%22a%22%2C%22b%22%5D%2C%20%5B%2 (https://stackoverflow.com/questions/716477/join-list-of-lists-in-python#:~:text=x%20%3D%20%5B%5B%22a%22%2C%22b%22%5D%2C%20%5B%2

In [405]:
```
demand_list = sum(df_demand.values.tolist(), [])  # explained above
```

In [11]:
```
vehicle_capacity = 120 #given in the data
```

In [12]:
```
origin = 0 # depot node
```

In [13]:
```
n_veh = int(np.ceil(sum(demand_list)/vehicle_capacity)) # min number
```

In [14]:
```
n_veh
```

Out[14]: 3

In [410]:
```
nodes = []
for i in range(31):
    nodes.append(i)

def VRP_savings(nodes, origin, d, n_veh, demand_list, vehicle_capaci
    '''
    Constructs a VRP solution using the savings method for a given s
    nodes, their pairwise distances-d, the origin, number of vehicle
    '''
    ans = [] # Creating an empty list to store solution data
    for iteration in range(n_veh): # every iteration is for finding
            # Set of customer nodes (i.e. nodes other than the origi
            customers = {i for i in nodes if i != origin}

            # Initialize out-and-back tours from the origin to every
            tours = {(i,i): [origin, i, origin] for i in customers}

            # Compute savings
            savings = {(i, j): round(d[i][origin] + d[origin][j] - d
                        for i in customers for j in customers if j !=

            # Define a priority queue dictionary to get a pair of no
            # the maximum savings
            #pq = pqdict(savings, reverse = True)
            sorted_savings = sorted(savings.items(), key=lambda item
            # Merge subtours until obtaining a TSP tour
            break_while = False # to control while loop
            while len(tours) > 1 and not break_while: #if you have o
                A = sorted_savings.pop() # pops the maximum savings(
                # A = (i, j) --> biggest saving
                i = A[0][0] # i
                j = A[0][1] # j

                break_outer = False  # Outer loop

                for t1 in tours: # iterate all tours
                    for t2 in tours.keys()-{t1}: # iterate over all
                        sum_= 0
                        for x in (tours[t1][:-1] + tours[t2][1:]): #
                            sum_ += demand_list[x] # to find capacit
                        if sum_ > vehicle_capacity: # if used capaci
                            break_while = True
                            break
                        if t1[1] == i and t2[0] == j:  # checks the
                            tours[(t1[0], t2[1])] = tours[t1][:-1] +
                            del tours[t1], tours[t2] # delete the to
                            break_outer = True
                            break
                    if break_outer:
                        break

            x = tours.popitem() # delete the tour that achieved its

            ans.append(x[1]) # append the value of that tour to ans
            for i in ans[iteration][1:]: # delete the nodes that use
                for j in nodes:
                    if i == j:
                        nodes.remove(i)

    # compute the answer's length
    ans_len = 0
```

```
    for r in ans:
        for i in range(len(r)-1):
            ans_len += d[r[i]][r[i+1]]

    return ans, round(ans_len, 2) # return the ans and ans_len
```

In [411]:
```
vrp_solution_saving, vrp_solution_saving_len = VRP_savings(nodes, or
print(vrp_solution_saving, vrp_solution_saving_len)
```

```
[[0, 16, 13, 30, 6, 25, 7, 12, 21, 8, 18, 3, 0], [0, 22, 17, 5, 11,
10, 20, 15, 2, 29, 28, 26, 1, 0], [0, 27, 14, 4, 19, 24, 23, 9, 0]]
2542.2
```

2- Apply 2-exchange algorithms exhaustively (try all possible improvements) to improve the solution you obtained from 1.

I used list() otherwise in the swapList function it swaps all list that intended or unintended. I found the solution in that link: https://stackoverflow.com/questions/29785084/changing-one-list-unexpectedly-changes-another-too#:~:text=different%20arrays%20use%3A-,vec%20%3D%20list(v),-Share

In [359]:
```
def two_exchange(vrp_solution, d):
    '''
    Improves a given VRP solution using the 2-exchange algorithm.
    It is a general function which can solve asymmetric and symmetri
    '''

    def swapList(sl,pos1,pos2):
        '''
        Swaps 2 list element according to their index.
        The list, and indexes of 2 item that will be swapped must be
        '''
        n = len(sl) # n --> length of the list

        # Swapping
        temp = sl[pos1]
        sl[pos1] = sl[pos2]
        sl[pos2] = temp
        return sl

    def sol_leng(list, d):
        '''
        gives the route length of a list.
        a list and a distance matrix must be given
        '''
        sol_len= 0
        for i in range(len(list)-1):
            sol_len += d[list[i]][list[i+1]]
        return sol_len

    improved_sol = [] # the list that we will be store data of the s
    improved_sol_length = 0 # with that var we keep track of the len
    for r in range(len(vrp_solution)): # iterates the indexes in vrp
        sol = list(vrp_solution[r]) # creating the list to compare
        sol_len = sol_leng(sol, d) # giving the length value to comp

        for i in range(len(sol)): # iterating the indexes of the sol
            if sol[i] != 0: # if sol is not the origin node
                #print(sol)
                #print(sol_len)
                for j in range(len(sol)): # iterating the indexes of
                    if sol[i] != sol[j] and sol[j] != 0: # if they a
                        # Making the exchanged list
                        temp = list(sol)
                        temp = list(swapList(temp, i, j))
                        temp_len = sol_leng(temp, d)
                        #print("swapping", i," and ", j)
                        #print("temp: ", temp, "temp len: ", temp_le
                        # To control if we have a better solution af
                        if temp_len < sol_len: # If we have a better
                            #print("solution improved")
                            sol_len = temp_len
                            sol = list(temp)
        improved_sol.append(sol) # append the improved solution to i
        improved_sol_length += sol_len # add the length of the tour

    return improved_sol, improved_sol_length
```

In [402]:
```
improved_sol_2ex, improved_sol_length_2ex = two_exchange(vrp_solutio
print(improved_sol_2ex, round(improved_sol_length_2ex, 2))
```

```
[[0, 16, 13, 30, 6, 25, 7, 21, 8, 12, 18, 3, 0], [0, 22, 17, 5, 11,
10, 20, 2, 15, 29, 28, 26, 1, 0], [0, 27, 14, 4, 19, 24, 23, 9, 0]]
2522.9
```

3- Apply 2-opt algorithms greedily and stop when the first improvement is recognised to improve the solution you obtained from 2.

greedy_two_opt is the modificated two_opt TSPlib function. We showed the parts that we changed with #Greedy comment and explained

I added greedy_two_opt to TSPlib and I made a function called VRP_greedy_two_opt in VRPlib with using greedy_two_opt

In [390]:
```
def greedy_two_opt(tour, tour_length, d):
    '''
    Improves a given TSP solution using the Greedy 2-opt algorithm.
    It is a general function which can solve asymmetric and symmetri
    '''
    current_tour, current_tour_length = tour, tour_length
    best_tour, best_tour_length = current_tour, current_tour_length
    solution_improved = True #Greedy (If improved stop)
    improved = False
    while solution_improved and not improved:
        print()
        print('Attempting to improve the tour', current_tour,
              'with length', current_tour_length)
        solution_improved = False

        for i in range(1, len(current_tour)-2):
            for j in range(i+1, len(current_tour)-1):
                difference = round((d[current_tour[i-1]][current_tou
                                  + d[current_tour[i]][current_tour[
                                  - d[current_tour[i-1]][current_tou
                                  - d[current_tour[j]][current_tour[
                                  + d[current_tour[j]][current_tour[
                                  - d[current_tour[i]][current_tour[

                print('Cost difference due to swapping', current_tou
                      current_tour[j], 'is:', difference)

                if current_tour_length + difference < best_tour_leng
                    print('Found an improving move! Updating the bes

                    best_tour = current_tour[:i] + list(reversed(cur
                    best_tour_length = round(current_tour_length + d

                    print('Improved tour is:', best_tour, 'with leng
                          best_tour_length)
                    improved = True #Greedy (The tour is improved)
                    solution_improved = True
                if improved: #Greedy (breaks the inner for loop)
                    break
            if improved: #Greedy (breaks the outer for loop)
                break
        if improved: # Greedy (breaks the while loop)
            break
        current_tour, current_tour_length = best_tour, best_tour_len

    # Return the resulting tour and its length as a tuple
    return best_tour, best_tour_length
```

In [391]:
```
greedy_vrp_solution = [] # to store vrp route data
greedy_total_length = 0 # to count total length
for tour in improved_sol: # iterates over improved_sol that we found
    tour_length = sum(d[tour[i]][tour[i+1]] for i in range(len(tour)
    new_tour, new_tour_length = greedy_two_opt(tour, tour_length, d)
    greedy_vrp_solution.append(new_tour) # adds data to greedy_vrp_s
    greedy_total_length += new_tour_length # counts total length
```

```
Attempting to improve the tour [0, 16, 13, 30, 6, 25, 7, 21, 8, 1
2, 18, 3, 0] with length 1000.4000000000001
Cost difference due to swapping 16 and 13 is: 36.3
Cost difference due to swapping 16 and 30 is: 133.0
Cost difference due to swapping 16 and 6 is: 156.7
Cost difference due to swapping 16 and 25 is: 132.1
Cost difference due to swapping 16 and 7 is: 162.5
Cost difference due to swapping 16 and 21 is: 242.5
Cost difference due to swapping 16 and 8 is: 253.1
Cost difference due to swapping 16 and 12 is: 250.6
Cost difference due to swapping 16 and 18 is: 187.4
Cost difference due to swapping 16 and 3 is: 17.2
Cost difference due to swapping 13 and 30 is: 96.7
Cost difference due to swapping 13 and 6 is: 120.4
Cost difference due to swapping 13 and 25 is: 114.5
Cost difference due to swapping 13 and 7 is: 130.7
Cost difference due to swapping 13 and 21 is: 250.7
Cost difference due to swapping 13 and 8 is: 256.0
```

In [401]:
```
print(greedy_vrp_solution, round(greedy_total_length, 2))
```

```
[[0, 16, 13, 30, 6, 25, 7, 21, 8, 12, 18, 3, 0], [0, 22, 17, 5, 11,
10, 20, 2, 15, 29, 28, 26, 1, 0], [0, 23, 24, 19, 4, 14, 27, 9, 0]]
2521.1
```

5- Formulate the proposed VRP problem and solve its mathematical model using Gurobi and illustrate your solution.

In [1]:
```
import pandas as pd
import numpy as np
from gurobipy import *
```

In [2]:
```
#To read different sheets in an excel file we can you this block of
xls = pd.ExcelFile('real_distances_30_customers.xlsx')
df_distance_matrix = pd.read_excel(xls, 'distance', header=None) # h
df_demand = pd.read_excel(xls, 'demand', index_col=0) # col= 0 becau
```

find list of nodes

N set of Nodes $\{1, \ldots, n\}$

In [4]:
```
nodes = list(range(df_distance_matrix.shape[0]))
```

Build distance matrix

In [6]:
```
dist = [[round(float(df_distance_matrix[j][i]), 2) for j in nodes] f
```

In [7]:
```
origin = 0 # Our origin is depo and the depo is node 0
```

build demand matrix

In [15]:
```
demand = df_demand.values.tolist() # this code gives demands as list
demand
```

Out[15]:
```
[[0],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10],
 [10]]
```

to make it not lists in a list I used following link,
https://stackoverflow.com/questions/716477/join-list-of-lists-in-python#:~:text=x%20%3D%20%5B%5B%22a%22%2C%22b%22%5D%2C%20%5B%2
(https://stackoverflow.com/questions/716477/join-list-of-lists-in-python#:~:text=x%20%3D%20%5B%5B%22a%22%2C%22b%22%5D%2C%20%5B%2

In [16]:
```
demand = sum(demand, []) # I implement the line of code that used in
print(demand)
```

```
[0, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
```

In [17]:
```
veh_cap = 120 #Vehicle Capacity
k = int(np.ceil(np.sum(demand)/veh_cap)) # number of vehicles
                                        # this line of code is findi
                                             # np.sum(demand) = 404,
                                            #vehicle # is integer an
                                            #If we use floor from di
```

In [18]:
```
k
```
Out[18]:
```
3
```

In [19]:
```
#Create a new model
vrp = Model("vrp_model")
```
```
Set parameter Username
```

Create Variables

Decision variables

$$x_{ij} = \begin{cases} 1, & \text{if customer } j \text{ is visited after customer } i \\ 0, & \text{otherwise} \end{cases}$$

u_i load of vehicle after visiting customer i

$$x_{ij} \in \{0,1\} \qquad \forall i,j \in N$$

$$u_i \geq 0 \qquad \forall i \in N$$

In [20]:
```
#Create variables
x = vrp.addVars(nodes, nodes, lb = 0,vtype = GRB.BINARY,name = 'x')
u = vrp.addVars(nodes, lb = 0,vtype = GRB.INTEGER,name = 'u')
```

Add constraints

$$\sum_{\substack{j \in N \\ j \neq i}} x_{ij} = 1 \qquad \forall i \in N, i \neq 0$$

$$\sum_{\substack{i \in N \\ i \neq j}} x_{ij} = 1 \qquad \forall j \in N, j \neq 0$$

$$\sum_{\substack{j \in N \\ j \neq 0}} x_{0j} = K$$

$$\sum_{\substack{i \in N \\ i \neq 0}} x_{i0} = K$$

$$u_i - u_j + Qx_{ij} \leq Q - d_j \qquad \forall i, j \in N, i \neq j, i, j \neq 0$$

$$d_i \leq u_i \leq Q \qquad \forall i \in N, i \neq 0$$

In [21]:
```
#Add constraint
vrp.addConstrs(((quicksum(x[i,j] for j in nodes if j != i) == 1) for
vrp.addConstrs(((quicksum(x[i,j] for i in nodes if i != j) == 1) for
vrp.addConstr(((quicksum(x[0,j] for j in nodes) == k)), name = '3');
vrp.addConstr(((quicksum(x[i,0] for i in nodes) == k)), name = '4');
vrp.addConstrs(((u[i] - u[j] + veh_cap*x[i,j] <= veh_cap - demand[j]
vrp.addConstrs(((u[i] <= veh_cap) for i in nodes), name = '6');
vrp.addConstrs(((u[i] >= demand[i]) for i in nodes), name = '7');
```

Set objective function

In [22]:
```
#set objective function
vrp.setObjective((quicksum(dist[i][j]*x[i,j] for i in nodes for j in
```

In [23]:
```
#vrp.setParam("TimeLimit", 1300)
vrp.update() # adds variables, constraints and the objective functio
vrp.optimize() # solves the model
```

```
Gurobi Optimizer version 11.0.0 build v11.0.0rc2 (mac64[x86] - Da
rwin 23.1.0 23B81)

CPU model: Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz
Thread count: 4 physical cores, 8 logical processors, using up to
8 threads

Optimize a model with 994 rows, 992 columns and 4534 nonzeros
Model fingerprint: 0xa1133188
Variable types: 0 continuous, 992 integer (961 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+02]
  Objective range  [6e+00, 4e+02]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+02]
Presolve removed 62 rows and 31 columns
Presolve time: 0.03s
Presolved: 932 rows, 961 columns, 4472 nonzeros
Variable types: 0 continuous, 961 integer (931 binary)
```

I runned the gurobi for 1510s but it still could not find the optimal solution so I terminated it, I could also give a time limit

status = by checking the status code from the website [https://www.gurobi.com/documentation/current/refman/optimization_status_codes.html (https://www.gurobi.com/documentation/current/refman/optimization_status_codes.html)](https://www.gurobi.com/documentation/current/refman/optimization_status_codes.html)

object_Value = objective value from the model

status 11: INTERRUPTED means Optimization was terminated by the user.

In [24]:
```
status = vrp.status # shows the status of our code

object_Value = vrp.objVal # stores the objective value comes from mi

print()
print("model status is: ", status)
print()
print("Objective value is: ", object_Value)
```

```
model status is:  11

Objective value is:  2453.4999999999995
```

In [27]:
```
# printing routes
if status != 3 and status != 4:
    vis = []
    Sol_x = np.zeros([len(nodes), len(nodes)])
    for i in nodes:
        for j in nodes:
            if vrp.objVal < 1e+99:
                Sol_x[i,j] = x[i,j].getAttr("X")
            else:
                error_status = True
                ofvv = 1e+99
            if 1 - 0.00001 <= Sol_x[i,j] <= 1 + 0.00001:
                vis.append((i,j))

visited = np.array(vis)
solution = []
if visited[0][0] == 0:
    sol = [visited[0][0], visited[0][1]]
elif visited[0][0] !=0 and visited[0][1] == 0:
    sol = [visited[0][1], visited[0][0]]
else:
    print('First tuple should include depot 0')
visited = np.delete(visited,0, axis = 0)

solution = []
for i in visited:
    try:
        next_ind = int(np.where(visited[:,0] == sol[-1])[0])
        sol.append(visited[next_ind][1])
        visited = np.delete(visited,next_ind, axis = 0)
    except:
        next_ind = int(np.where(visited[:,1] == sol[-1])[0])
        sol.append(visited[next_ind][0])
        visited = np.delete(visited,next_ind, axis = 0)

    if sol[0] == sol[-1]:
        sol = np.asarray(sol)
        solution.append(sol)
        used = []
        for j in solution:
            for k in j:
                used.append(k)
        remain = list(set(nodes) - set(used))
        if remain == []:
            break
        sol=[visited[0][0], visited[0][1]]
        visited = np.delete(visited,0, axis = 0)
```

In [30]:
```
print(f"Gurobi's solution is {solution} with length {round(object_Va
```

```
Gurobi's solution is [array([ 0, 12, 21,  8, 18,  3, 10, 20,  2, 1
5, 29, 28, 11,  0]), array([ 0, 23, 26,  1, 24, 14,  9,  0]), array
([ 0, 27, 22, 16, 13, 30,  6, 25,  7,  5, 17,  4, 19,  0])] with le
ngth 2453.5
```

In [2]:
```
def nearest_neighbor(nodes, origin, d):
    '''
    Constructs a TSP solution using the nearest neighbor algorithm,
    for a given set of nodes, the associated pairwise distance matri
    and the origin.
    '''
    # Tour should start at the origin
    tour = [origin]

    # Initialize the tour length
    tour_length = 0

    # If the origin is not in nodes, add it to nodes
    if origin not in nodes:
        nodes.append(origin)

    # Nearest neighbor search until all nodes are visited
    while len(tour) < len(nodes):
        dist, next_node = min((d[tour[-1]][i], i) for i in nodes if
        tour_length += dist
        tour.append(next_node)
        print('Added', next_node, 'to the tour!')

    # Tour should end at the origin
    tour_length += d[tour[-1]][origin]
    tour.append(origin)

    # Round the result to 2 decimals to avoid floating point represe
    tour_length = round(tour_length, 2)

    # Return the resulting tour and its length as a tuple
    return tour, tour_length
```

In [3]:
```
def savings(nodes, origin, d):
    '''
    Constructs a TSP solution using the savings method for a given s
    nodes, their pairwise distances-d, and the origin.
    '''
    # Set of customer nodes (i.e. nodes other than the origin)
    customers = {i for i in nodes if i != origin}

    # Initialize out-and-back tours from the origin to every other n
    tours = {(i,i): [origin, i, origin] for i in customers}

    # Compute savings
    savings = {(i, j): round(d[i][origin] + d[origin][j] - d[i][j],
                  for i in customers for j in customers if j != i}

    # Define a priority queue dictionary to get a pair of nodes (i,j
    # the maximum savings
    #pq = pqdict(savings, reverse = True)
    sorted_savings = sorted(savings.items(), key=lambda item: item[1
    # Merge subtours until obtaining a TSP tour
    while len(tours) > 1:
        A = sorted_savings.pop()
        i = A[0][0]
        j = A[0][1]
        print((i, j))
        # Outer loop
        break_outer = False
        for t1 in tours:
            for t2 in tours.keys()-{t1}:
                if t1[1] == i and t2[0] == j:
                    print('Merging', tours[t1], 'and', tours[t2])
                    tours[(t1[0], t2[1])] = tours[t1][:-1] + tours[t
                    del tours[t1], tours[t2]
                    print(tours)
                    break_outer = True
                    break
            if break_outer:
                break
        else:
            print('No merging opportunities can be found for', (i,j)

    # Final tours dictionary (involves a single tour, which is the T
    print(tours)

    # Compute tour length
    tour_length = 0
    for tour in tours.values():
        for i in range(len(tour)-1):
            tour_length += d[tour[i]][tour[i+1]]

    # Round the result to 2 decimals to avoid floating point represe
    tour_length = round(tour_length, 2)

    # Return the resulting tour and its length as a tuple
    return tour, tour_length
```

In [4]:
```
def two_opt(tour, tour_length, d):
    '''
    Improves a given TSP solution using the 2-opt algorithm. Note: T
    applies 2opt correctly only when the distance matrix is symmetri
    of asymmetric distances, one needs to update the cost difference
    incurred by swapping.
    '''
    current_tour, current_tour_length = tour, tour_length
    best_tour, best_tour_length = current_tour, current_tour_length
    solution_improved = True

    while solution_improved:
        print()
        print('Attempting to improve the tour', current_tour,
              'with length', current_tour_length)
        solution_improved = False

        for i in range(1, len(current_tour)-2):
            for j in range(i+1, len(current_tour)-1):
                difference = round((d[current_tour[i-1]][current_tou
                                  + d[current_tour[i]][current_tour[
                                  - d[current_tour[i-1]][current_tou
                                  - d[current_tour[j]][current_tour[

                print('Cost difference due to swapping', current_tou
                      current_tour[j], 'is:', difference)

                if current_tour_length + difference < best_tour_leng
                    print('Found an improving move! Updating the bes

                    best_tour = current_tour[:i] + list(reversed(cur
                    best_tour_length = round(current_tour_length + d

                    print('Improved tour is:', best_tour, 'with leng
                          best_tour_length)

                    solution_improved = True

        current_tour, current_tour_length = best_tour, best_tour_len

    # Return the resulting tour and its length as a tuple
    return best_tour, best_tour_length
```

In [1]:
```
def greedy_two_opt(tour, tour_length, d):
    '''
    Improves a given TSP solution using the Greedy 2-opt algorithm.
    It is a general function which can solve both asymmetric and sym
    '''
    current_tour, current_tour_length = tour, tour_length
    best_tour, best_tour_length = current_tour, current_tour_length
    solution_improved = True #Greedy (If improved stop)
    improved = False
    while solution_improved and not improved:
        print()
        print('Attempting to improve the tour', current_tour,
              'with length', current_tour_length)
        solution_improved = False

        for i in range(1, len(current_tour)-2):
            for j in range(i+1, len(current_tour)-1):
                difference = round((d[current_tour[i-1]][current_tou
                                    + d[current_tour[i]][current_tour[
                                    - d[current_tour[i-1]][current_tou
                                    - d[current_tour[j]][current_tour[
                                    + d[current_tour[j]][current_tour[
                                    - d[current_tour[i]][current_tour[

                print('Cost difference due to swapping', current_tou
                      current_tour[j], 'is:', difference)

                if current_tour_length + difference < best_tour_leng
                    print('Found an improving move! Updating the bes

                    best_tour = current_tour[:i] + list(reversed(cur
                    best_tour_length = round(current_tour_length + d

                    print('Improved tour is:', best_tour, 'with leng
                          best_tour_length)
                    improved = True #Greedy (The tour is improved)
                    solution_improved = True
                if improved: #Greedy (breaks the inner for loop)
                    break
            if improved: #Greedy (breaks the outer for loop)
                break
        if improved: # Greedy (breaks the while loop)
            break
        current_tour, current_tour_length = best_tour, best_tour_len

    # Return the resulting tour and its length as a tuple
    return best_tour, best_tour_length
```

In [2]:
```
from math import ceil
from scipy.cluster.vq import kmeans2, whiten
from ipynb.fs.full.TSPlib import nearest_neighbor, savings, greedy_t
```

In [3]:
```
def CFRS(k, coordinates, d, plt):
    '''
    Constructs a VRP solution using cluster-first route-second (CFRS
    heuristic.
    '''
    # CLUSTERING PHASE - Clusters the nodes based on their proximity
    # each other via k-means method, which uses Euclidean distance a
    # the proximity measure).
    x, y = kmeans2(whiten(coordinates), k, iter = 100)
    plt.scatter(coordinates[:,0], coordinates[:,1], c = y)
    plt.axis('off')
    #plt.show()
    plt.savefig("clusters.png")

    depot = 0 # the depot node
    clusters = [{depot} for i in range(k)] # add the depot to all cl

    # Assign the nodes to their respective clusters
    for i, label in enumerate(y):
        if i != depot:
            clusters[label].add(i)
    print(clusters)

    # ROUTING PHASE - Constructs a list of TSP tours based on the cl
    vrp_solution = [] # a list of TSP tours
    total_length = 0 # total length of the TSP tours

    # Iterate over the clusters, and construct a TSP tour for each c
    for cluster in clusters:
        # Construct a TSP tour with your choice of construction meth
        tour, tour_length = nearest_neighbor(cluster, depot, d)
        # Add the new tour to vrp_solution
        vrp_solution.append(tour)
        # Add the length of the new TSP tour to total_length
        total_length += tour_length

    # Round the result to 2 decimals to avoid floating point
    # representation errors
    total_length = round(total_length, 2)

    # Return the resulting VRP solution and its total length as a tu
    return vrp_solution, total_length
```

In [4]:
```
def RFCS(k, nodes, d):
    '''
    Constructs a VRP solution using route-first cluster-second
    (RFCS) heuristic.
    '''
    depot = 0 # the depot node
    n_max = ceil((len(nodes)-1)/k) # vehicle capacity

    # ROUTING PHASE - Constructs a TSP tour over all nodes with
    # your choice of construction method
    #tour, tour_length = nearest_neighbor(nodes, depot, d)
    tour, tour_length = savings(nodes, depot, d)
    # CLUSTERING PHASE - Iterates over the nodes in the TSP tour and
    # splits it into smaller tours each containing at most n_max nod
    vrp_solution = []
    index = 1
    for i in range(k):
        current_tour = [depot]
        current_nodes = 0
        while current_nodes < n_max and tour[index] != depot:
            current_tour.append(tour[index])
            current_nodes += 1
            index += 1
        current_tour.append(depot)
        vrp_solution.append(current_tour)

    # Calculate the total length of the resulting tours
    total_length = 0
    for route in vrp_solution:
        for i in range(len(route)-1):
            total_length += d[route[i]][route[i+1]]

    # Round the result to 2 decimals to avoid floating point
    # representation errors
    total_length = round(total_length, 2)

    # Return the resulting VRP solution and its total length as a tu
    return vrp_solution, total_length
```

In [5]:
```
def VRP_greedy_two_opt(vrp_solution, d):
    '''
    Greedily improves a given VRP solution using the 2-opt algorithm
    It is a general function which can find solution for both asymme
    for a given VRP solution and a distance matrix.
    '''
    greedy_vrp_solution = [] # to store vrp route data
    greedy_total_length = 0 # to count total length
    for tour in vrp_solution: # iterates over improved_sol that we f
        tour_length = sum(d[tour[i]][tour[i+1]] for i in range(len(t
        new_tour, new_tour_length = greedy_two_opt(tour, tour_length
        greedy_vrp_solution.append(new_tour) # adds data to greedy_v
        greedy_total_length += new_tour_length # counts total length
    return greedy_vrp_solution, round(greedy_total_length, 2)
```

In [1]:
```
def VRP_savings(nodes, origin, d, n_veh, demand_list, vehicle_capaci
    '''
    Constructs a VRP solution using the savings method for a given s
    nodes, their pairwise distances-d, the origin, number of vehicle
    '''
    ans = [] # Creating an empty list to store solution data
    for iteration in range(n_veh): # every iteration is for finding
            # Set of customer nodes (i.e. nodes other than the origi
            customers = {i for i in nodes if i != origin}

            # Initialize out-and-back tours from the origin to every
            tours = {(i,i): [origin, i, origin] for i in customers}

            # Compute savings
            savings = {(i, j): round(d[i][origin] + d[origin][j] - d
                       for i in customers for j in customers if j !=

            # Define a priority queue dictionary to get a pair of no
            # the maximum savings
            #pq = pqdict(savings, reverse = True)
            sorted_savings = sorted(savings.items(), key=lambda item
            # Merge subtours until obtaining a TSP tour
            break_while = False # to control while loop
            while len(tours) > 1 and not break_while: #if you have o
                A = sorted_savings.pop() # pops the maximum savings(
                # A = (i, j) --> biggest saving
                i = A[0][0] # i
                j = A[0][1] # j

                break_outer = False  # Outer loop

                for t1 in tours: # iterate all tours
                    for t2 in tours.keys()-{t1}: # iterate over all
                        sum_= 0
                        for x in (tours[t1][:-1] + tours[t2][1:]): #
                            sum_ += demand_list[x] # to find capacit
                        if sum_ > vehicle_capacity: # if used capaci
                            break_while = True
                            break
                        if t1[1] == i and t2[0] == j:  # checks the
                            tours[(t1[0], t2[1])] = tours[t1][:-1] +
                            del tours[t1], tours[t2] # delete the to
                            break_outer = True
                            break
                    if break_outer:
                        break

            x = tours.popitem() # delete the tour that achieved its

            ans.append(x[1]) # append the value of that tour to ans
            for i in ans[iteration][1:]: # delete the nodes that use
                for j in nodes:
                    if i == j:
                        nodes.remove(i)

    # compute the answer's length
    ans_len = 0
    for r in ans:
        for i in range(len(r)-1):
            ans_len += d[r[i]][r[i+1]]
```

```
    return ans, round(ans_len, 2) # return the ans and ans_len
```

In [6]:
```python
def VRP_two_exchange(vrp_solution, d):
    '''
    Improves a given VRP solution using the 2-exchange algorithm.
    It is a general function which can solve asymmetric and symmetri
    for a given VRP solution and distance matrix.
    '''

    def swapList(sl,pos1,pos2):
        '''
        Swaps 2 list element according to their index.
        The list, and indexes of 2 item that will be swapped must be
        '''
        n = len(sl) # n --> length of the list

        # Swapping
        temp = sl[pos1]
        sl[pos1] = sl[pos2]
        sl[pos2] = temp
        return sl

    def sol_leng(list, d):
        '''
        gives the route length of a list.
        a list and a distance matrix must be given
        '''
        sol_len= 0
        for i in range(len(list)-1):
            sol_len += d[list[i]][list[i+1]]
        return sol_len

    improved_sol = [] # the list that we will be store data of the s
    improved_sol_length = 0 # with that var we keep track of the len
    for r in range(len(vrp_solution)): # iterates the indexes in vrp
        sol = list(vrp_solution[r]) # creating the list to compare
        sol_len = sol_leng(sol, d) # giving the length value to comp

        for i in range(len(sol)): # iterating the indexes of the sol
            if sol[i] != 0: # if sol is not the origin node
                #print(sol)
                #print(sol_len)
                for j in range(len(sol)): # iterating the indexes of
                    if sol[i] != sol[j] and sol[j] != 0: # if they a
                        # Making the exchanged list
                        temp = list(sol)
                        temp = list(swapList(temp, i, j))
                        temp_len = sol_leng(temp, d)
                        #print("swapping", i," and ", j)
                        #print("temp: ", temp, "temp len: ", temp_le
                        # To control if we have a better solution af
                        if temp_len < sol_len: # If we have a better
                            #print("solution improved")
                            sol_len = temp_len
                            sol = list(temp)
        improved_sol.append(sol) # append the improved solution to i
        improved_sol_length += sol_len # add the length of the tour

    return improved_sol, improved_sol_length
```