IE517

Homework #1

Abdullah Hanefi Önaldı

Solving the TSP using Construction Heuristics and 2-Opt Improvement Heuristic

Problem Description

In this homework, you are going to solve the TSP for three data sets. They are called eil51.dat, eil76.dat, and eil101.dat, and consist of 51, 76, and 101 customer locations, respectively. Each data set includes the x-coordinates and y-coordinates of customers. The distances between customer locations are measured via Euclidean distance rounded to two digits after the decimal point. You can also compute the optimal tour length by considering the sequence given in the xxxopt.dat files.

1. Solve each instance using the one-sided nearest neighbor heuristic starting at cities 10, 20, and 30. This means that you will obtain nine tours. Provide the tour length of each one using the table below.
2. Solve each instance using the two-sided nearest neighbor heuristic starting at cities 10, 20, and 30. This means that you will obtain nine tours. Provide the tour length of each one using the table below.
3. Solve each instance using the nearest insertion heuristic starting at cities 10, 20, and 30. This means that you will obtain nine tours. Provide the tour length of each one using the table below.
4. Solve each instance using the farthest insertion heuristic starting at cities 10, 20, and 30. This means that you will obtain nine tours. Provide the tour length of each one using the table below.
5. For each tour obtained so far, apply the 2-opt improvement heuristic, and give the tour length using the table below.

I would like to remind you the following points which you should consider when you submit your homework. It will consist of two parts: your code and report. First, your code must be clear and you should define the following using comment lines in the code: variables names and their purpose, function names and their purpose. For example, you should write "X is the location variable", "CompObj calculates the objective value", etc. Or, you can use a function name that is self explanatory e.g., ApplyMove.

In the report part, you have to mention which solution representation and neighborhood structure you used as well as other pertinent and tiny details worth pointing out. You can use the following table for the output of your solutions.

	method	**1-Sided_NN**		**2-Sided_NN**		**Nearest_Insert**		**Furthest_Insert**	
	stage	**Initial**	**After_2-opt**	**Initial**	**After_2-opt**	**Initial**	**After_2-opt**	**Initial**	**After_2-opt**
dataset	**initial_customer**								
eil76	**10**								
	20								
	30								
eil101	**10**								
	20								
	30								
eil51	**10**								
	20								
	30								

Solution

Solution Representation

I used simple lists of customers to represent solutions to the TSP problem. The first customer in the list occurs at the end, and all other customers occur exactly once.

These solutions can be viewed in the excel file provided along with this report.

Data Structures

I used :

- `paths` : pandas dataframe with MultiIndex'es to store all paths
- `df` : pandas dataframe with MultiIndex'es to store the lengths of the paths
- `INSTANCES` : dictionary that contains all the following for each problem instance
  - contents of input files
  - contents of optimal solution files
  - `file` : the path of the dataset
  - `file_opt` : the path of the file containing optimal solution
  - `optimal_path` : the list of nodes visited in the optimal path
  - `distances` : pairwise distances of customers in a matrix

Code

Let's start by importing functions/modules and defining several helper functions:

- `pandas` : library for managing tabular data
- `numpy` : numeric operations, linear algebra modules etc
- `squareform` : used for creating square distance matrices
- `pdist` : pairwise distance calculations
- `partial` : creation of partial functions
- `combinations` : given a number of lists, generates all possible combinations of elements by taking one from each list

- `calc_total_length` : calculates the total path length, given the pairwise distances of all customers, and the order they are visited
- `insertion_cost` : given the pairwise distances between customers, calculates the cost of inserting customer k between i and j
- `find_method` : given a method name, construct the partial functions that will solve the problem using the aforementioned method

In [1]:
```
import pandas as pd
import numpy as np
from scipy.spatial.distance import squareform, pdist
from functools import partial
from itertools import combinations


def calc_total_length(path, distances):
    return distances.lookup(path[:-1], path[1:]).sum()


def insertion_cost(distances, i, j, k):
    return distances.loc[i, k] + distances.loc[k, j] - distances.loc[i, j]


def find_method(method_name):
    method_dict = {
        '1-Sided_NN': partial(nearest_neighbor, num_sides=1),
        '2-Sided_NN': partial(nearest_neighbor, num_sides=2),
        'Nearest_Insert': partial(insertion, kind='nearest'),
        'Furthest_Insert': partial(insertion, kind='farthest'),
    }
    return method_dict[method_name]
```

The static variables storing problem instances, initial customer indices, methods, and stages as described in the problem description

In [2]:
```
INSTANCES = {
    'eil76': {
        'file': 'data/eil76.dat',
        'file_opt': 'data/eil76opt.dat'
    },
    'eil101': {
        'file': 'data/eil101.dat',
        'file_opt': 'data/eil101opt.dat'
    },
    'eil51': {
        'file': 'data/eil51.dat',
        'file_opt': 'data/eil51opt.dat'
    },
}
INITIAL_CUSTOMERS = [10, 20, 30]
METHODS = ['1-Sided_NN', '2-Sided_NN', 'Nearest_Insert', 'Furthest_Insert']
STAGES = ['Initial', 'After_2-opt']
```

Create the Pandas DataFrame that will hold all the solutions

In [3]:
```
def create_df(instances=INSTANCES,
              initial_customers=INITIAL_CUSTOMERS,
              methods=METHODS,
              stages=STAGES):
    indexes = [instances.keys(), initial_customers]
    row_index = pd.MultiIndex.from_product(
        indexes, names=['dataset', 'initial_customer'])

    indexes = [methods, stages]
    column_index = pd.MultiIndex.from_product(
        indexes, names=['method', 'stage'])

    df = pd.DataFrame(index=row_index, columns=column_index)

    return df


df = create_df()
paths = create_df()
```

Read the files of the given instances

In [4]:
```
def read_files(instances=INSTANCES):
    for instance in instances.values():
        coords = pd.read_csv(
            instance['file'], header=None, index_col=0, delim_whitespace=True)

        instance['optimal_path'] = pd.read_csv(
            instance['file_opt'], header=None, squeeze=True)

        instance['distances'] = pd.DataFrame(
            squareform(pdist(coords)),
            columns=coords.index,
            index=coords.index)

        instance['optimal_length'] = calc_total_length(
            path=instance['optimal_path'], distances=instance['distances'])


read_files()
```

Create the path using the nearest neighbor heuristic given the number of sides to search, the pairwise distances, and the initial customer to start the search

In [5]:
```
def nearest_neighbor(num_sides, distances, initial_node):
    distances = distances.copy()
    np.fill_diagonal(distances.values, np.nan)
    path = [initial_node]

    if num_sides is 1:
        current = initial_node
        for _ in range(distances.shape[0] - 1):
            next_ = distances[current].idxmin()
            path.append(next_)
            distances.loc[current, :] = np.nan
            current = next_

    elif num_sides is 2:
        head, tail = initial_node, distances[initial_node].idxmin()
        path.append(tail)
        distances.loc[:, 'head'] = np.nan
        distances.loc[:, 'tail'] = np.nan

        for _ in range(distances.shape[0] - 2):
            next_head, next_tail = distances[[head, tail]].idxmin()
            if distances.loc[head, next_head] > distances.loc[next_tail, tail]:
                path.insert(0, next_tail)
                distances.loc[tail, :] = np.inf
                tail = next_tail
            else:
                path.append(next_head)
                distances.loc[head, :] = np.inf
                head = next_head
    else:
        raise ValueError('nearest_neighbor is either one or two sided')

    path.append(path[0])
    return path
```

Create the path using the insertion heuristic given the kind (farthest or nearest), the pairwise distances, and the initial customer to start the search

In [6]:
```
def insertion(kind, distances, initial_node):
    distances = distances.copy()
    np.fill_diagonal(distances.values, np.nan)

    if kind is 'nearest':
        closest = distances[initial_node].idxmin()
        path = [initial_node, closest, initial_node]

        distances['subtour'] = distances[[closest, initial_node]].min(axis=1)
        for _ in range(distances.shape[0] - 2):
            distances['subtour'].loc[path] = np.nan
            closest = distances['subtour'].idxmin()
            costs = [
                insertion_cost(distances, i, j, closest)
                for i, j in zip(path, path[1:])
            ]
            min_cost = np.argmin(costs) + 1
            path.insert(min_cost, closest)
            distances['subtour'] = distances[[closest, 'subtour']].min(axis=1)

    elif kind is 'farthest':
        fartest = distances[initial_node].idxmax()
        path = [initial_node, fartest, initial_node]

        distances['subtour'] = distances[[fartest, initial_node]].min(axis=1)
        for _ in range(distances.shape[0] - 2):
            distances['subtour'].loc[path] = np.nan
            fartest = distances['subtour'].idxmax()
            costs = [
                insertion_cost(distances, i, j, fartest)
                for i, j in zip(path, path[1:])
            ]
            min_cost = np.argmin(costs) + 1
            path.insert(min_cost, fartest)
            distances['subtour'] = distances[[fartest, 'subtour']].min(axis=1)
    else:
        ValueError('insertion is either nearest or farthest')

    return path
```

Improve a solution using two opt given pairwise distances and the path found in the solution

In [7]:
```
def two_opt(distances, path):
    path = path.copy()
    while True:
        no_gain = True
        for start, end in combinations(range(1, len(path) - 2), r=2):
            if end - start is 1:
                continue
            c1 = path[start - 1]
            c2 = path[start]
            c3 = path[end]
            c4 = path[end + 1]

            gain = + distances[c1][c2] + distances[c3][c4] \
                   - distances[c1][c3] - distances[c2][c4]

            if gain > 1e-10:
                no_gain = False
                path[start:end + 1] = path[end:start - 1:-1]

        if no_gain:
            return path
```

Iterate over all the datasets, initial customers, and methods. Construct solutions and then improve them using two opt heuristic

In [8]:
```
for instance in INSTANCES:
    df.loc[instance, 'optimal'] = INSTANCES[instance]['optimal_length']
    distances = INSTANCES[instance]['distances']
    for initial in INITIAL_CUSTOMERS:
        for method_name in METHODS:
            method = find_method(method_name)

            path = method(distances=distances, initial_node=initial)
            length = calc_total_length(path, distances)

            paths.loc[(instance, initial), (method_name, 'Initial')] = path
            df.loc[(instance, initial), (method_name, 'Initial')] = length

            better_path = two_opt(distances, path)
            length = calc_total_length(better_path, distances)

            df.loc[(instance, initial), (method_name, 'After_2-opt')] = length
            paths.loc[(instance, initial), (method_name,
                                            'After_2-opt')] = better_path

df
```

Out[8]:

	method	1-Sided_NN		2-Sided_NN		Nearest_Insert		Furthest_Insert		optimal
	stage	Initial	After_2-opt	Initial	After_2-opt	Initial	After_2-opt	Initial	After_2-opt	
dataset	initial_customer									
eil76	10	640.533	599.838	704.103	652.116	636.201	610.281	599.215	596.102	545.387552
	20	735.983	650.89	708.861	663.681	614.819	608.437	580.563	580.563	545.387552
	30	730.285	642.509	711.812	635.205	626.494	609.627	579.946	579.946	545.387552
eil101	10	796.041	712.081	720.585	696.957	728.333	714.068	684.778	684.033	642.309536
	20	800.708	735.65	800.014	710.235	735.845	710.125	692.276	692.276	642.309536
	30	776.518	699.699	784.347	742	735.845	717.476	688.832	682.889	642.309536
eil51	10	558.849	511.2	496.688	432.482	490.181	473.476	444.555	444.555	429.983312
	20	567.304	526.656	554.273	506.732	514.379	478.556	454.656	454.656	429.983312
	30	520.018	479.436	527.628	459.112	490.181	471.406	458.272	458.272	429.983312

Divide the path lengths by the optimal length to better see the performance

In [12]:
```
df = df.div(df['optimal'], axis='rows')
df.drop(columns=['optimal'], inplace=True)
df
```

Out[12]:

	method	1-Sided_NN		2-Sided_NN		Nearest_Insert		Furthest_Insert	
	stage	Initial	After_2-opt	Initial	After_2-opt	Initial	After_2-opt	Initial	After_2-opt
dataset	initial_customer								
eil76	10	1.17445	1.09984	1.29101	1.19569	1.16651	1.11899	1.0987	1.09299
	20	1.34947	1.19345	1.29974	1.2169	1.12731	1.1156	1.0645	1.0645
	30	1.33902	1.17808	1.30515	1.16469	1.14871	1.11779	1.06336	1.06336
eil101	10	1.23934	1.10863	1.12187	1.08508	1.13393	1.11172	1.06612	1.06496
	20	1.24661	1.14532	1.24553	1.10575	1.14562	1.10558	1.07779	1.07779
	30	1.20895	1.08935	1.22114	1.15521	1.14562	1.11702	1.07243	1.06318
eil51	10	1.2997	1.18888	1.15513	1.00581	1.14	1.10115	1.03389	1.03389
	20	1.31936	1.22483	1.28906	1.17849	1.19628	1.11297	1.05738	1.05738
	30	1.20939	1.11501	1.22709	1.06774	1.14	1.09634	1.06579	1.06579

Write the paths and performances to an excel file

In [10]:
```
writer = pd.ExcelWriter('results.xlsx')
df.to_excel(writer, sheet_name='performance')
paths.to_excel(writer, sheet_name='paths')
writer.save()
```