IE 517 - Project: Solving the Traveling Salesman Problem with Profits

Instructor: Prof. Necati Aras

Students:

Gönül Aycı - 2016800003,

ÇağlaYaman - 2017800018

Tabu Search on Travelling Salesperson Problems with Profits

In this project, we tried to solve Travelling Salesperson Problems with Profits (TSPs with profits) with Tabu Search (TS).

Travelling Salesperson Problems with Profits:

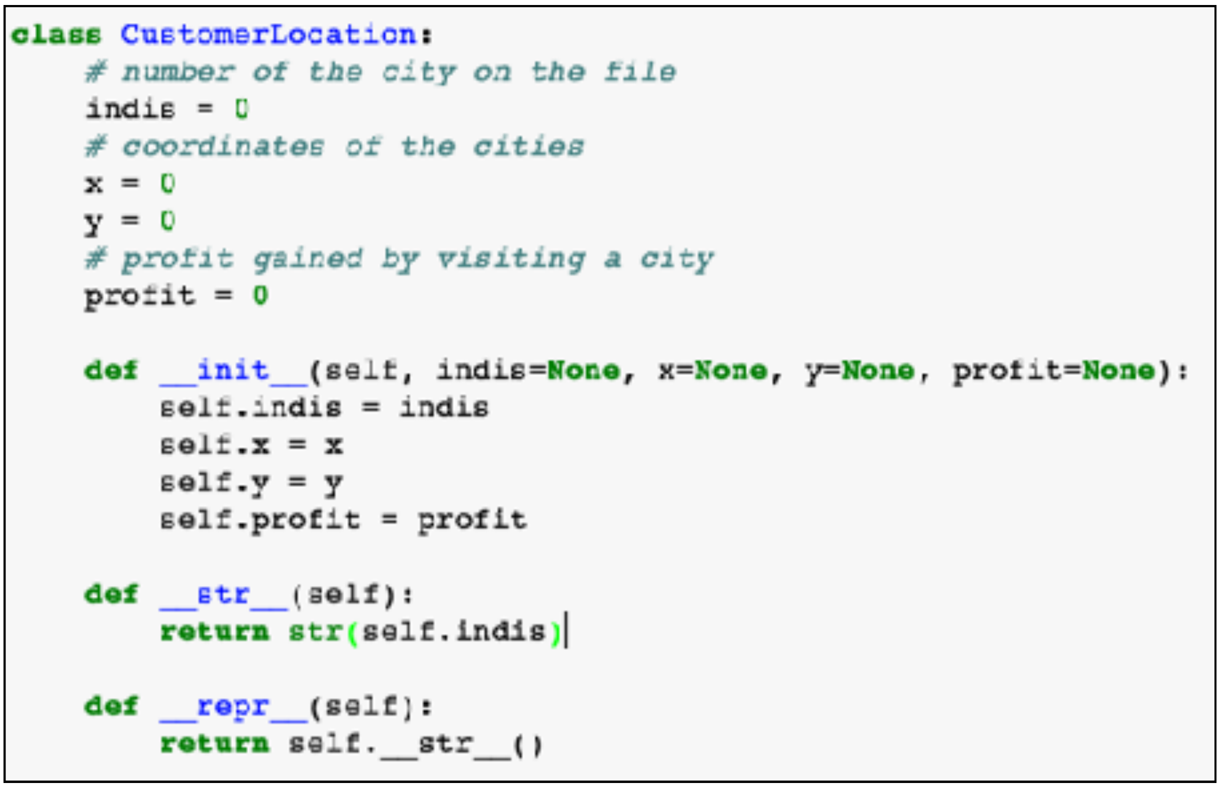
Traveling Salesperson problems with profits (TSPs with profits) are a generalization of the traveling salesman problem (TSP), where it is not necessary to visit all vertices. A profit is associated with each vertex. The overall goal is the simultaneous optimization of the collected profit and the travel costs.

Solution Representation:

Permutation representation is chosen as solution representation.

The list [1, 2, 3, 4, 5] represents the route of the salesperson
n = number of cities
s = indis of a city s = [1,n]
x, y = place of a city in 2D coordinates

As our first homework, cities are represented as a class. A property “profit” is added to the class definition



```python
class CustomerLocation:
    # number of the city on the file
    indis = 0
    # coordinates of the cities
    x = 0
    y = 0
    # profit gained by visiting a city
    profit = 0

    def __init__(self, indis=None, x=None, y=None, profit=None):
        self.indis = indis
        self.x = x
        self.y = y
        self.profit = profit

    def __str__(self):
        return str(self.indis)

    def __repr__(self):
        return self.__str__()
```

Construction Heuristics:

At the initialization phase, a profit matrix is generated.

```
# profit of going from a to b = profit - distance of cities
customer_matrix = generate_customer_matrix(a)
```

Each entry in the matrix is the profit of moving from a current city to the next city. This is calculated by subtracting the Euclidean distance between cities from the profit of visiting the next city. The matrix is not symmetrical.

At the beginning of the construction algorithm, all cities are in the unvisited city list. As cities are added to the route, they are moved from unvisited city list. To construct a route, we choose the starting city which has the best profit among other cities. We delete it from unvisited city list. We add next city to the route which gives the best value according to the profit matrix until unvisited city list is empty.

Improvement Heuristics:

To improve our solution, we used multiple approaches (variable neighbourhood). Those are best deletion, best insertion, tabu search with random walk, best improvement with 4-opt and 3-opt, insertion mutation.

First delete/best insert :

Move operator : 1-delete, 1-insert

First approach is to improve the existing solution with the best deletion and insertion. The basic idea is that, if the route consists of only two cities and profit is less than zero, it is better not to visit any cities. If there are more than two cities in the route, check if removing a city improves the profit or not. If deletion of a city improves the route, that city is removed from route and inserted to the unvisited city list. Finally, if there are unvisited cities, insert those between consecutive cities with the best improvement in profit. We do these until there is no more improvement.

This improvement heuristic is applied after construction, and after tabu search.

Best Improvement Heuristics with Tabu Search:

Best improvement is applied to the initial solution which is already improved with best insert/delete. While there is improvement in the solution, variable neighborhood search is applied.

move operator : swap (4-opt), insertion mutation (3-opt)

Algorithm is as follows:

```
While there is improvement or 10 iterations without improvement:
    Current solution is the best solution so far
    For limited iterations(we choose as 5) with or without improvement:
        Apply tabu search with 4-opt
        If tabu search result is better than best result:
            Set tabu result to best result
    If solution is not improved change the neightborhood:
        For limited iterations(we choose as 5) with or without improvement:
            Apply tabu search with 3-opt
            If tabu search result is better than best result:
                Set tabu result to best result
```

Each tabu search is the same process except for the move operator: The best improvement among the random walks of neighborhood structure is chosen as the local optimum.

```
While there is improvement in local optimum (best solution of tabu):
    Current solution is the local best so far
    For a limited iteration do the random walk
        Move operator is applied to the current solution
        n neighbours are generated
        Best neighbor is set as current solution
        Swap operator used is put in the tabu list
```

With this approach optimum solution is modified and neighborhood is enlarged. By choosing the best candidate each time regardless of its comparison to local best

provides variety in the candidate selection. This helps avoiding the local best and getting close to global best.
For tabu list structure, a list of tuples is used. Tabu list is of fixed size. When the tabu list is full, the first element in the list is removed. Therefore the first move operator applied is removed from tabu and the same operator can be applied again. For swap operation (4-opt), each element is the index of the cities that are swapped. For 3-opt each element contains the index of the city that will be removed and the index of the removed city will be added to.

Result table:

Instance	Best Objective Value	No. of customers visited	Sequence of customers visited	CPU Time (s)
eil51-LP	40.52	29	[19, 41, 4, 18, 14, 6, 48, 23, 7, 26, 31, 28, 22, 32, 51, 46, 5, 38, 16, 9, 50, 34, 30, 10, 33, 45, 15, 44, 42]	130.95
eil51-HP	701.75	49	[21, 34, 30, 9, 50, 16, 38, 11, 32, 46, 51, 12, 47, 18, 4, 17, 37, 5, 49, 10, 39, 33, 45, 15, 44, 42, 19, 41, 13, 25, 14, 24, 43, 7, 23, 6, 27, 48, 8, 26, 31, 28, 22, 3, 36, 35, 20, 2, 29]	230.27
eil76-LP	137.2	55	[32, 44, 3, 6, 68, 75, 76, 67, 46, 8, 19, 14, 11, 53, 35, 7, 34, 52, 27, 13, 57, 15, 37, 20, 70, 60, 71, 69, 36, 47, 21, 74, 30, 2, 33, 73, 62, 22, 42, 41, 56, 23, 49, 16, 51, 17, 40, 12, 58, 10, 72, 39, 9, 25, 50]	38.13
eil76-HP	1244.56	74	[69, 71, 60, 70, 20, 37, 5, 15, 57, 13, 54, 19, 8, 35, 7, 53, 14, 59, 11, 66, 65, 38, 10, 58, 72, 39, 9, 40, 12, 17, 3, 44, 32, 50, 25, 55, 18, 24, 49, 23, 56, 41, 43, 42, 64, 22, 62, 73, 33, 63, 16, 51, 6, 68, 4, 75, 76, 26, 67, 34, 46, 52, 27, 45, 29, 48, 30, 2, 74, 28, 61, 21, 47, 36]	971.43
eil101-LP	237.58	73	[77, 3, 79, 33, 50, 69, 28, 101, 53, 6, 99, 93, 85, 91, 16, 86, 44, 100, 98, 37, 92, 95, 97, 87, 42, 43, 15, 57, 41, 22, 74, 4, 55, 25, 39, 23, 56, 75, 72, 73, 21, 40, 58, 13, 94, 96, 5, 84, 83, 18, 82, 7, 48, 47, 19, 11, 62, 31, 10, 90, 32, 20, 51, 9, 71, 35, 34, 78, 29, 24, 80, 68, 76]	49.08

eil101-HP	1623.38	95	[57, 41, 22, 74, 75, 56, 23, 67, 39, 4, 25, 55, 24, 29, 68, 80, 12, 26, 21, 73, 40, 58, 53, 101, 28, 27, 69, 31, 10, 62, 11, 19, 47, 48, 82, 7, 18, 60, 83, 5, 84, 17, 45, 8, 46, 36, 49, 64, 63, 90, 32, 66, 65, 71, 35, 34, 78, 79, 3, 77, 76, 50, 33, 81, 9, 51, 30, 70, 52, 89, 6, 96, 99, 93, 59, 95, 94, 13, 87, 97, 92, 98, 37, 100, 91, 85, 61, 16, 86, 38, 44, 14, 42, 43, 15]	274.64

Source code:

```
# ## Import required libraries

import random
import math
from operator import attrgetter
import time
import xlrd



# ## Setup for reading data



# eil51, eil76, eil101
# dataset-LowProfit.xls, dataset-HighProfit.xls
dataset_name = 'eil51'
file_name = 'dataset-LowProfit.xls'



# ## Functions
```

```
def euclidean_distance(ca, cb):
    dist = math.sqrt((ca.x - cb.x) ** 2 + (ca.y - cb.y) ** 2)
    return round(dist, 2)



def generate_customer_matrix(a):
    customer_matrix = [[0 for x in range(len(a))] for y in range(len(a))]
    for r in range(len(a)):
        for c in range(len(a)):
            if r == c:
                customer_matrix[r][c] = - math.inf
            else:
                customer_matrix[r][c] = a[c].profit - round(euclidean_distance(a[r],
a[c]), 2)
    return customer_matrix;



class CustomerLocation:
    # number of the city on the file
    indis = 0
    # coordinates of the cities
    x = 0
    y = 0
    # profit gained by visiting a city
    profit = 0

    def __init__(self, indis=None, x=None, y=None, profit=None):
        self.indis = indis
        self.x = x
        self.y = y
        self.profit = profit

    def __str__(self):
        return str(self.indis)

    def __repr__(self):
        return self.__str__()



def init(file_name, result_file):
    # Read Excel file with file name and sheet name
    xl = xlrd.open_workbook(file_name)
    points = xl.sheet_by_name(result_file)
    print(points)
    for point in points.get_rows():
                a.append(CustomerLocation(int(point[0].value),  int(point[1].value),
int(point[2].value), int(point[3].value)))
    print(a)



# In[16]:


def construct_route():
    # keep a list of unvisited cities
    unvisited_cities = a.copy()
    print('unvisited: ', unvisited_cities)
    # choose the city with best profit
    starting_city = max(a, key=attrgetter('profit'))
    route = [starting_city]
    last_added = starting_city
    print('start with city : ', starting_city)
    unvisited_cities.remove(starting_city)

    while unvisited_cities:
        candidate_city = None
        candidate_profit = -math.inf
        for next_city in unvisited_cities:
            next_profit = customer_matrix[last_added.indis - 1][next_city.indis - 1]
            if next_profit > candidate_profit:
                candidate_city = next_city
                candidate_profit = next_profit
        if candidate_city is not None:
            route.append(candidate_city)
            last_added = candidate_city
            unvisited_cities.remove(candidate_city)
    return route


def improve_solution(unvisited_cities, route):
      # at first there is no unvisited cities, removed cities from the route will be
added here
    improvement = True
    while improvement:
        improvement = False


         # if route consists of only 2 cities and profit is less than 0, it is better not to
visit any cities
        if len(route) == 2:
            current_profit = customer_matrix[route[0].indis - 1][route[1].indis - 1]

            if current_profit < 0:

                route.remove(route[1])

              # if there are more than 2 cities in the route, check if removing a city
improves the profit

        if len(route) > 2:

            for c in range(len(route)):

                next_indis = c + 1

                if next_indis >= len(route):

                    next_indis = 0

                  current_profit = customer_matrix[route[c - 1].indis - 1][route[c].indis -
1] + customer_matrix[route[c].indis - 1]
[route[next_indis].indis - 1]

                possible_profit = customer_matrix[route[c - 1].indis - 1]
[route[next_indis].indis - 1]

                if possible_profit > current_profit:

                    improvement = True

                    unvisited_cities.append(route[c])

                    route.remove(route[c])

                    break;

        else:

            improvement = False

          # If there are unvisited cities, insert those between consecutive cities when
there is improvement

        if len(unvisited_cities) != 0:

            inserted_cities = unvisited_cities.copy()

            for u in range(len(unvisited_cities)):

                best_improvement = 0

                indis_r = -1

                for r in range(len(route)):

                    possible_profit = customer_matrix[route[r - 1].indis - 1]
[unvisited_cities[u].indis - 1] +
customer_matrix[unvisited_cities[u].indis - 1][route[r].indis - 1]
                current_profit = customer_matrix[route[r - 1].indis - 1][route[r].indis
- 1]
                    if (possible_profit - current_profit) > best_improvement:
                        indis_r = r
                        best_improvement = (possible_profit - current_profit)
                if indis_r > -1:
                    improvement = True
                    route.insert(indis_r, unvisited_cities[u])
                    inserted_cities.remove(unvisited_cities[u])
            unvisited_cities = inserted_cities
        return route, unvisited_cities



def swap(current, indis1, indis2):
        difference = 0
        new_route = current.copy()
        next1 = indis1 + 1
        if next1 == len(current):
        next1 = 0
        next2 = indis2 + 1
        if next2 == len(current):
        next2 = 0


                difference -= customer_matrix[new_route[indis1 - 1].indis - 1]
[new_route[indis1].indis - 1]
                difference -= customer_matrix[new_route[indis1].indis - 1]
[new_route[next1].indis - 1]
                difference -= customer_matrix[new_route[indis2 - 1].indis - 1]
[new_route[indis2].indis - 1]
                difference -= customer_matrix[new_route[indis2].indis - 1]
```

```
[new_route[next2].indis - 1]

        difference += customer_matrix[new_route[indis1 - 1].indis - 1]
[new_route[indis2].indis - 1]
        difference += customer_matrix[new_route[indis2].indis - 1]
[new_route[next1].indis - 1]
        difference += customer_matrix[new_route[indis2 - 1].indis - 1]
[new_route[indis1].indis - 1]
        difference += customer_matrix[new_route[indis1].indis - 1]
[new_route[next2].indis - 1]

  first_city = new_route[indis1]
  second_city = new_route[indis2]

  new_route[indis1] = second_city
  new_route[indis2] = first_city
  return difference, new_route



def flip(current, indis1, indis2):
  difference = 0
  new_route = current.copy()
  next1 = indis1 + 1
  if next1 == len(current):
    next1 = 0
  next2 = indis2 + 1
  if next2 == len(current):
    next2 = 0

        difference -= customer_matrix[new_route[indis1 - 1].indis - 1]
[new_route[indis1].indis - 1]
```

```
        difference -= customer_matrix[new_route[indis1].indis - 1]
[new_route[next1].indis - 1]
        difference -= customer_matrix[new_route[indis2].indis - 1]
[new_route[next2].indis - 1]


        difference += customer_matrix[new_route[indis1 - 1].indis - 1]
[new_route[next1].indis - 1]
        difference += customer_matrix[new_route[indis2].indis - 1]
[new_route[indis1].indis - 1]
        difference += customer_matrix[new_route[indis1].indis - 1]
[new_route[next2].indis - 1]


    first_city = new_route[indis1]
    new_route.pop(indis1)


    new_route.insert(indis2+1, first_city)
    return difference, new_route



def shuffle_swap(current):
    while True:
        indis1 = random.randint(0, len(current) - 1)
        indis2 = random.randint(0, len(current) - 1)
        if indis1 != indis2:
            difference, new_route = swap(current, indis1, indis2)
            return (indis1, indis2), difference, new_route


def shuffle_flip(current):
    while True:
        indis1 = random.randint(0, len(current) - 1)
        indis2 = random.randint(0, len(current) - 1)
        if indis1 != indis2:
            difference, new_route = flip(current, indis1, indis2)
            return (indis1, indis2), difference, new_route


def calculate_profit(solution):
    total_profit = 0
    if len(solution) == 1:
        return solution[0].profit
    for i in range(-1, len(solution) - 1):
        total_profit += customer_matrix[solution[i].indis - 1][solution[i + 1].indis - 1]
    return total_profit


def tabu_search(best):
    route_len = len(best)
    tabu_list = []
    tabu_list_size = route_len
    candidate_size = route_len*3
    current = best.copy()
    tabu_best = best.copy()
    is_improved = True
    while is_improved:
        is_improved = False
        tabu_list = []
        for i in range(route_len*10):
            candidate_list = []
            for j in range(candidate_size):
                candidate_chosen = False
                t = 0
                while t < route_len and not candidate_chosen:
                    t += 1
                    swap_operator, difference, candidate = shuffle_swap(current)
                    if (swap_operator[0], swap_operator[1]) not in tabu_list and (swap_operator[1], swap_operator[0]) not in tabu_list:
                        candidate_list.append((swap_operator, difference, candidate))
                        candidate_chosen = True

            candidate_list.sort(key=lambda x: x[1], reverse=True)
            if not candidate_list:
                continue
            best_candidate = candidate_list[0]
            #if best_candidate[1] > 0:
            current = best_candidate[2]
            tabu_list.append(best_candidate[0])
            if len(tabu_list) > tabu_list_size:
                tabu_list.pop(0)
            #print('inner tabu current:', calculate_profit(current))
            if calculate_profit(current) > calculate_profit(tabu_best):
                is_improved = True
                tabu_best = current.copy()

        current = tabu_best.copy()
    return tabu_best

def tabu_search_3opt(best):
    route_len = len(best)
    tabu_list = []
    tabu_list_size = route_len
    candidate_size = route_len*3
    current = best.copy()
    tabu_best = best.copy()
    is_improved = True
    while is_improved:
        is_improved = False
        tabu_list = []
        for i in range(route_len*10):
            candidate_list = []
            for j in range(candidate_size):
                candidate_chosen = False
                t = 0
                while t < route_len and not candidate_chosen:
                    t += 1
                    swap_operator, difference, candidate = shuffle_flip(current)
                    if (swap_operator[0],swap_operator[1]) not in tabu_list and
(swap_operator[1],swap_operator[0]) not in tabu_list:
                        candidate_list.append((swap_operator, difference, candidate))
                        candidate_chosen = True


            candidate_list.sort(key=lambda x: x[1], reverse=True)
            if not candidate_list:
                continue
            best_candidate = candidate_list[0]
            #if best_candidate[1] > 0:
            current = best_candidate[2]
            tabu_list.append(best_candidate[0])
            if len(tabu_list) > tabu_list_size:
                tabu_list.pop(0)
            #print('inner tabu current:', calculate_profit(current))
            if calculate_profit(current) > calculate_profit(tabu_best):
                is_improved = True
                tabu_best = current.copy()

        current = tabu_best.copy()
    return tabu_best



def solve_tsp():
    route = construct_route()
    print(route)
    print(calculate_profit(route))
    print("initial solution")
    print(route)
    print(calculate_profit(route))
    best = route
    unvisited_cities = []
    best, unvisited_cities = improve_solution([], route)
    print("improved initial solution")
    print(best)
    print(calculate_profit(best))

    for t in range(1):
        print("t=", t)
        if t == 0:
            tabu_best = best.copy()
        else:
            tabu_best = random.sample(a, len(a))
            unvisited_cities = []
        is_improved = True
        i = 0

        while is_improved and i < 100:
            is_improved = False
            i += 1
            for j in range(2):
                tabu_best = tabu_search(tabu_best)
                print("swap tabu search reasult : ", len(tabu_best), " ", calculate_profit(tabu_best))
                tabu_best, unvisited_cities = improve_solution(unvisited_cities, tabu_best)
                print("improved tabu reasult : ", len(tabu_best), " ", calculate_profit(tabu_best))
                if calculate_profit(tabu_best) > calculate_profit(best):
                    print("swap tabu search improved reasult : ", len(tabu_best), " ", calculate_profit(tabu_best))
                    best = tabu_best.copy()
                    is_improved = True
            if not is_improved:
                for j in range(2):
                    tabu_best = tabu_search_3opt(tabu_best)
                    print("tabu search reasult : ", len(tabu_best), " ", calculate_profit(tabu_best))
                    tabu_best, unvisited_cities = improve_solution(unvisited_cities, tabu_best)
                    print("improved tabu reasult : ", len(tabu_best), " ", calculate_profit(tabu_best))
                    if calculate_profit(tabu_best) > calculate_profit(best):
                        print("flip tabu search improved reasult : ", len(tabu_best), " ", calculate_profit(tabu_best))
                        best = tabu_best.copy()
                    is_improved = True
            tabu_best = best.copy()
            print("=====best===== ", len(best), calculate_profit(best))
            print(best)


# ## Main Solver

# list of customer locations (with coordinates and profit value)
a = []
init(file_name, dataset_name)
# matrix of the utilities gained by going from a city to a city
# profit of going from a to b = profit - distance of cities
customer_matrix = generate_customer_matrix(a)
start_time = time.time()
solve_tsp()
end_time = time.time()
print("CPU time : ", (end_time - start_time))
```