A0201613Y – Computational Assignment Report

1 Facility Location Problem

(a) We first note that there are m customers so $i \in \{1, 2, ..., m\}$. There are n facilities so $j \in \{1, 2, ..., n\}$. The linear inequalities listed under (FLP) are:

$$x_{i,j} \leq y_j \text{ for all } i, j \tag{1.1}$$
$$0 \leq x_{i,j} \leq 1 \tag{1.2}$$

For each facility j, there are m choices for customer i. With n facilities, there are mn possible choices of i, j. Hence, for (1.1), there are mn inequalities.

As for (1.2), this compound inequality consists of two separate inequalities, $0 \leq x_{i,j}$ and $x_{i,j} \leq 1$, so there are $2mn$ inequalities.

Under (FLP), we have $mn + 2mn = 3mn$ linear inequalities.

The linear inequalities listed under (AFL) are:

$$\sum_{i=1}^{m} x_{i,j} \leq my_j \text{ for all } j \tag{1.3}$$
$$0 \leq x_{i,j} \leq 1 \tag{1.4}$$

For (1.3), the number of inequalities only depends on the possible choices of j. As such, there are only n inequalities for (1.3).

For (1.4), the inequality is the same as (1.2). For the same reasons, there are $2mn$ inequalities.

Under (AFL), we have $n + 2mn = (2m + 1)n$ linear inequalities.

Thus, for $m \geq 1$, we have $(2m + 1)n \leq 3mn$. (AFL) has fewer inequalities for $m > 1$.

(b) For a solution to be feasible, all constraints must be satisfied. Comparing the two formulations, all the constraints are exactly the same except (1.1) and (1.3). I want to show that (1.1) and (1.3) are equivalent, i.e.

$$(1.1) \iff (1.3)$$

Proof: (1.1) $\implies$ (1.3): Assuming $x_{i,j} \leq y_j$ for all i, j, we sum up these inequalities over all i to get,

$$x_{1,j} + x_{2,j} + ... + x_{m,j} \leq y_j + y_j + ... + y_j \text{ for all } j$$
$$\sum_{i=1}^{m} x_{i,j} \leq my_j \text{ for all } j$$

(1.1) $\Longleftarrow$ (1.3): Assuming $\sum_{i=1}^{m} x_{i,j} \leq my_j$ for all j,

$$x_{1,j} + x_{2,j} + ... + x_{m,j} \leq y_j + y_j + ... + y_j \text{ for all } j \tag{1.5}$$

From (1.5), since $y_j \in \{0, 1\}$ and $0 \leq x_{i,j} \leq 1$, if $y_j = 0$, $x_{i,j} = 0$ for all i. We get that at $y_j = 0$, $x_{i,j} = y_j$ for all i.

At $y_j = 1$, since $0 \leq x_{i,j} \leq 1$, we have $x_{i,j} \leq y_j$ for all i. Taking the union of these two possible cases of $y_j \in \{0, 1\}$, we get

$$x_{i,j} \leq y_j \text{ for all } i, j$$

□

We have shown that constraints (1.1) and (1.3) are equivalent. Since all the constraints of (FLP) and (AFL) are equivalent, the set of feasible solutions of (FLP) and (AFL) are equal. With the same feasible region and also the same objective function, the two formulations are thus equivalent.

(c) , (d) , (e) After replacing the binary constraints with the box constraints for y_j, we arrive at the following relations:

$$\text{(AFL-Val)} = \text{(FLP-Val)} \tag{1.6}$$
$$\text{(FLP-LR-Val)} \leq \text{(FLP-Val)} \tag{1.7}$$
$$\text{(AFL-LR-Val)} \leq \text{(AFL-Val)} \tag{1.8}$$

For (1.6), since (FLP) and (AFL) are shown to be equivalent in part (b), this implies that their optimal values are equal.

Following that, we should note that their corresponding LRs may not necessarily have the same optimal values. When the binary constraint is dropped, (FLP-LR) and (AFL-LR) do not have the same set of feasible solutions and are thus not equivalent. In other words,

$$\text{(1.1)} \iff \text{(1.3) may not hold when the binary constraint is dropped.}$$

Proof with counter-example: (1.1) $\Longleftarrow$ (1.3): Suppose $\sum_{i=1}^{m} x_{i,j} \leq my_j$ for all j,

$$x_{1,j} + x_{2,j} + ... + x_{m,j} \leq y_j + y_j + ... + y_j \text{ for all } j \tag{1.9}$$

The relation holds trivially for $m = 1$. Hence, we find a counter-example at $m \geq 2$. For $m \geq 2$, we assume $y_j = \gamma$, where $\gamma \in \{y_j : 0 \leq y_j \leq 1\} \backslash \{0, 1\}$. We assign $x_{1,j} = \gamma + \varepsilon$ for $0 < \varepsilon \leq \gamma$ while $x_{i,j} = 0$ for all $i \neq 1$. Under (1.9), we have $\gamma + \varepsilon \leq 2\gamma \leq m\gamma$ but $x_{1,j} = \gamma + \varepsilon > \gamma = y_j$. We have that $x_{i,j} \leq y_j$ is not satisfied at $i = 1$.

□

For (1.7) and (1.8), the optimal values for the LRs are lower bounds of the corresponding optimal values of the MILPs. Since we are minimizing over a larger constraint set in the LRs, this means that the feasible region of the MILPs is contained in the feasible region of the corresponding LRs. Moreover, if the optimal solutions of the LRs are integral, these integral solutions are also feasible in the corresponding MILPs and their optimal values will be equal.

(f) Since all the random variables are defined uniformly over a unit interval or unit square, `np.random.rand()` is used to generate them. With the coordinates of the customers and the facilities, we compute the distance matix using the Euclidean distance. To generate 100 sets of the random variables, for-loops are used. Here, we set a seed value to reference back to the same dataset to use as a numerical example for part (h).

(g) For each of the four models, the for-loop is used when running the optimization, iterating through each set of random variables. At every iteration, we record the optimal value. As such, we will have 100 different optimal values recorded under each model.

(h) Due to rounding error from computer arithmetic, to compare the equality of two optimal values for every set of random variables, we choose to check that their difference is approximately 0.

We first examine how often (FLP-Val) is equal to (FLP-LR-Val). Iterating through the 100 objective values of each model, the differences are all of magnitude 10^{-15}, which is approximately 0. Hence, we conclude that (FLP-Val) is always equal to (FLP-LR-Val). This is an equality because we can check that (FLP-LR) always has an integral solution. This supports our answers in part (e), that this integral solution will also be feasible in (FLP), thus attaining the same optimal values.

For (AFL-Val) and (AFL-LR-Val), the smallest difference over the 100 pairs of objective values is approximately 0.668, which is significantly different. Hence, we conclude that (AFL-Val) is always different from (AFL-LR-Val), or specifically (AFL-Val) $>$ (AFL-LR-Val). The reason for the strict inequality is that the solutions of (AFL-LR) are always non-integers. Hence, this non-integral solution will not be feasible for (AFL).

Following the same method, we can also check that (FLP-Val) and (AFL-Val) are always equal. On the other hand, their corresponding (FLP-LR-Val) and (AFL-LR-Val) always have different optimal values, as proven in part (e) that the feasible regions of the two LRs are not equivalent.

Overall, the findings support the relations explained in part (e).

2 Traveling Salesman Problem

(a) To compute the total distance of a tour, a for-loop is used in the function to iterate through every consecutive pair of locations and sum up their distances. We then add the distance between the last and first location to indicate the return to the starting location.

(b) For the function that outputs a perturbed tour, since we have to pick two unique indices $i < j$, we use `random.sample()` to pick two indices without replacement. For this first perturb rule, we use list slicing to inverse all the cities between indices i and j, inclusive.

(c) We initialize a random tour using `random.shuffle()` to shuffle the indices of the cities. To compare results with the second perturb rule in part (f), we use `.copy()` to create the same initialized tour to use afterwards. The simulated annealing algorithm is set to run for 10000 loops, iterating through the sequence of steps given in the question.

$$\exp\left(-\frac{f_{\text{cand}} - f_{\text{curr}}}{T}\right) \geq u \tag{2.1}$$

Under step 3 of each iteration, if $f_{\text{cand}} \leq f_{\text{curr}}$, the condition in (2.1) always hold, i.e. the candidate tour of shorter distance gets accepted with 100% probability. However, when $f_{\text{cand}} > f_{\text{curr}}$, depending on the value of T and u, a bad candidate tour may be accepted even if it has a longer distance.

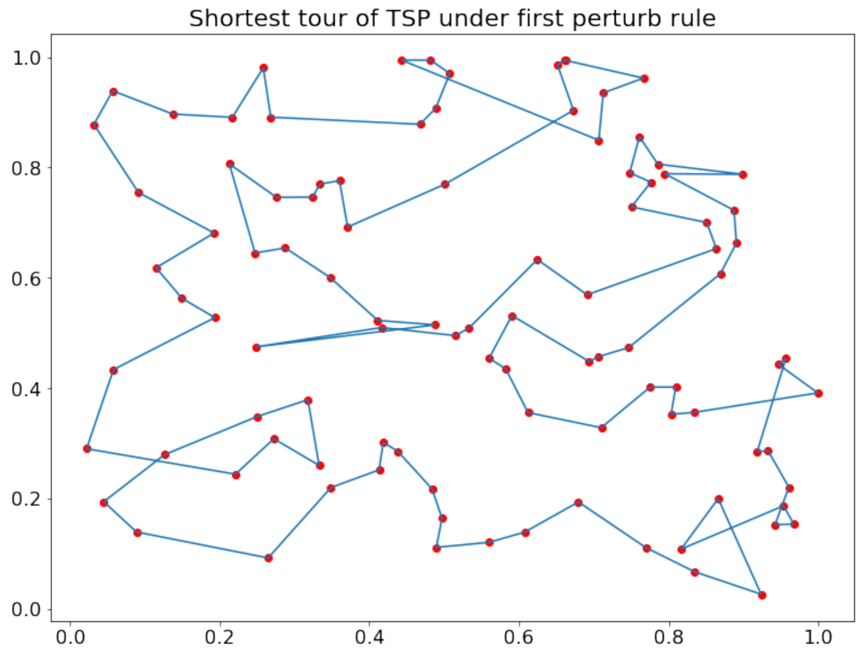
We infer that, at very small values of T and $f_{\text{cand}} > f_{\text{curr}}$, the probability of the bad candidate tour being accepted approaches 0. As such, if the initial value of T is set to be very small, we may be stuck with a local minimum solution.

This is because a large enough value of initial T allows the algorithm to search over the whole feasible region [1], even if a bad candidate tour is accepted in the process. This results in a higher probability of getting a global minimum solution. Likewise, the initial T should not be too large such that all the bad candidate tours are accepted throughout all iterations, making the algorithm inefficient.

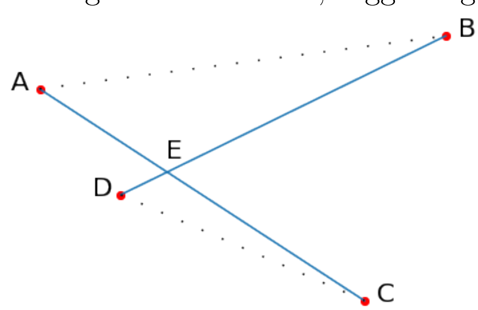
(d) Since the coordinates of the cities are defined uniformly over a unit square, we use `np.random.rand()` to generate a list of these coordinates. For reference, we name the cities by their indices in the list here. We can then compute the distance matrix using the Euclidean distance. As defined, the distance between any two cities in opposite directions will be the same. This problem can then be formulated as an undirected graph (as will be shown in the graph plot in part (e)).

We are now set to run the algorithm. After experimenting with a few choices of initial T (as mentioned in part (c)), the distance of the shortest tour generated is not very sensitive to the value of initial T. Hence, we just fix initial $T = 100$.

(e)

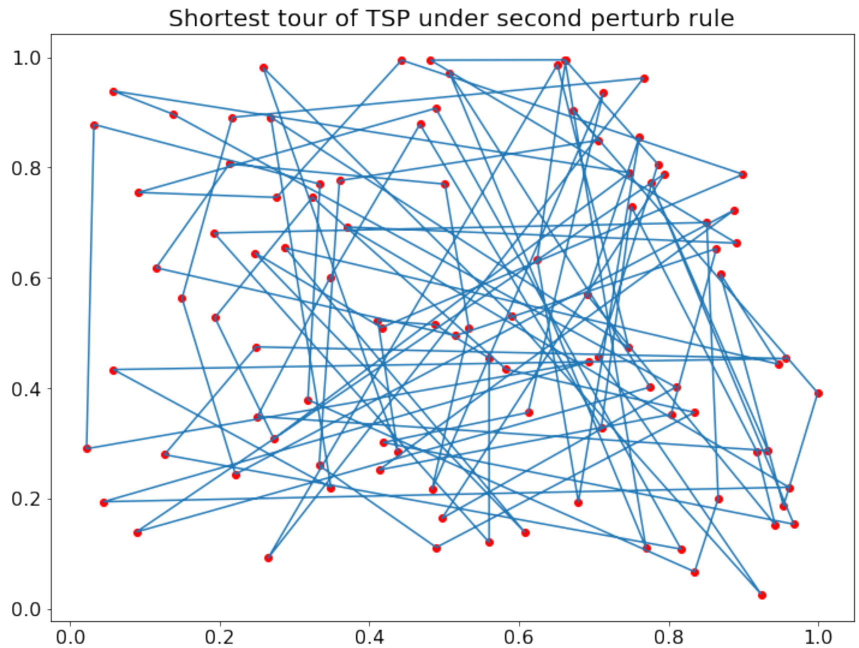


The above graph is one random instance of a solution. The red dots denote the location of the 100 cities. The blue line shows the shortest possible tour connecting every city. Evidently, the graph have edges that intersect, suggesting that the tour is not optimal.



We look at an intersection of edges formed by any four cities A, B, C and D as shown above, where E is the intersection point. By triangle inequality, $|AB| < |AE| + |BE|$ and $|CD| < |CE| + |DE|$. By replacing the blue lines with the dotted lines, we get a shorter tour. Hence, the above solution generated is not optimal.

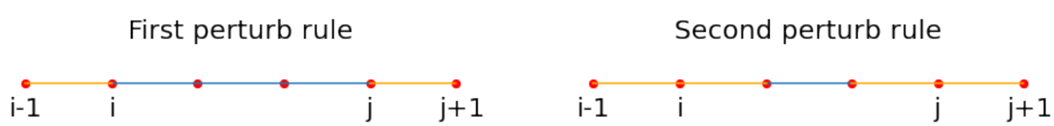
(f) Now, we create a new function that outputs a perturbed tour using a second perturb rule. This second perturb rule is now to swap only the city with index i and the city with index j. This can be done by reassigning the indices of the two cities in the list. We then run the algorithm again using the new function.



The above graph is one solution generated using the second perturb rule. Compared to the first graph, this graph has significantly more intersection of edges. This second perturb rule seems to be less efficient. To verify this, we compare their differences:

$$\text{Distance of initialized tour} = 50.5172279351468$$

	First perturb rule	Second perturb rule
Number of accepted candidate tours	889	5266
Distance of shortest tour generated	9.199323401302737	48.867230128219546



In general, the first perturb rule changes two edges, while the second perturb rule changes four edges for $j-i \geq 3$, as indicated by the orange edges in the above subgraph.

Since the first perturb rule changes less edges, it is more likely to pick solutions in its neighbourhood with similar total distance. It is thus able to approach a minimum steadily as bad candidate tours that get accepted do not increase the distance as much.

However, the second perturb rule changes more edges, so it may produce bad candidate tours that have relatively larger distance. The distance of the tour is more likely to increase than to decrease [1]. This results in a relatively large number of accepted candidate tours as the algorithm cannot approach a minimum efficiently. Hence, the distance of the shortest tour remains close to that of the initialized tour.

With the shortest tour generated by the first perturb rule being significantly better, we conclude that the first perturb rule is more efficient.

References

[1] Wikipedia contributors. *Simulated annealing — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-October-2021]. 2021. URL: https://en.wikipedia.org/wiki/Simulated_annealing.

Appendix

The codes for the report are appended below. Appendix 1 is used for Question 1 while Appendix 2 is used for Question 2. Some outputs have been cleared as they are too long to be displayed.

Appendix 1

1. Facility Location Problem

In [1]:
```
import gurobipy as gp
from gurobipy import GRB
from itertools import product
import numpy as np
```

In [2]:
```
# Set number of customers and facilities

num_customers = 20    # m = 20
num_facilities = 15   # n = 15

# Define distances

def compute_distance(loc1, loc2):
    dx = loc1[0] - loc2[0]
    dy = loc1[1] - loc2[1]
    return np.sqrt(dx**2+dy**2)

# Compute key parameters of model formulation

cartesian_prod = list(product(range(num_customers), range(num_facilities)))
```

In [3]:
```
# Generating 100 realizations of all the random variables

customers = []
facilities = []
setup_cost = []
distance = []

np.random.seed(0) # Set the seed value to reference back to the same dataset

for j in range(100):
    customers.append([(np.random.rand(),np.random.rand()) for i in range(num_customers)])
    facilities.append([(np.random.rand(),np.random.rand()) for i in range(num_facilities)])
    setup_cost.append([np.random.rand() for i in range(num_facilities)])

for i in range(100):
    distance.append({(c,f) : compute_distance(customers[i][c], facilities[i][f]) for c, f in cartesian_prod})
```

FLP model

In [ ]:
```
# FLP model formulation

m_FLP = gp.Model('facility_location_FLP')

select_FLP = m_FLP.addVars(num_facilities, vtype=GRB.BINARY, name='Select_FLP')
assign_FLP = m_FLP.addVars(cartesian_prod, ub=1, vtype=GRB.CONTINUOUS, name='Assign_FLP')

m_FLP.addConstrs((assign_FLP[(c,f)] <= select_FLP[f] for c,f in cartesian_prod),
                 name='Setup2ship_FLP')
m_FLP.addConstrs((gp.quicksum(assign_FLP[(c,f)] for f in range(num_facilities)) ==
                  1 for c in range(num_customers)), name='Demand_FLP')

Opt_FLP_Val = [] # Record optimal values for FLP model

for i in range(100):
    m_FLP.setObjective(select_FLP.prod(setup_cost[i])+assign_FLP.prod(distance[i]), GRB.MINIMIZE)
    m_FLP.optimize()
    Opt_FLP_Val.append(m_FLP.objVal)
```

AFL model

In [ ]:
```
# AFL model formulation

m_AFL = gp.Model('facility_location_AFL')

select_AFL = m_AFL.addVars(num_facilities, vtype=GRB.BINARY, name='Select_AFL')
assign_AFL = m_AFL.addVars(cartesian_prod, ub=1, vtype=GRB.CONTINUOUS, name='Assign_AFL')

# Different constraint for the AFL model
m_AFL.addConstrs((gp.quicksum(assign_AFL[(c,f)] for c in range(num_customers)) <=
                  select_AFL[f]*num_customers for c,f in cartesian_prod), name='Setup2ship_AFL')

m_AFL.addConstrs((gp.quicksum(assign_AFL[(c,f)] for f in range(num_facilities)) ==
                  1 for c in range(num_customers)), name='Demand_AFL')

Opt_AFL_Val = [] # Record optimal values for AFL model

for i in range(100):
    m_AFL.setObjective(select_AFL.prod(setup_cost[i])+assign_AFL.prod(distance[i]), GRB.MINIMIZE)
    m_AFL.optimize()
    Opt_AFL_Val.append(m_AFL.objVal)
```

FLP_LR model

In [ ]:
```
# FLP_LR model formulation

m_FLP_LR = gp.Model('facility_location_FLP_LR')

# Replacing the binary constraint with the box constraint
select_FLP_LR = m_FLP_LR.addVars(num_facilities, ub=1, vtype=GRB.CONTINUOUS, name='Select_FLP_LR')

assign_FLP_LR = m_FLP_LR.addVars(cartesian_prod, ub=1, vtype=GRB.CONTINUOUS, name='Assign_FLP_LR')

m_FLP_LR.addConstrs((assign_FLP_LR[(c,f)] <= select_FLP_LR[f] for c,f in cartesian_prod),
                    name='Setup2ship_FLP_LR')
m_FLP_LR.addConstrs((gp.quicksum(assign_FLP_LR[(c,f)] for f in range(num_facilities)) ==
                     1 for c in range(num_customers)), name='Demand_FLP_LR')

Opt_FLP_LR_Val = [] # Record optimal values for FLP_LR model

for i in range(100):
    m_FLP_LR.setObjective(select_FLP_LR.prod(setup_cost[i])+assign_FLP_LR.prod(distance[i]), GRB.MINIMIZE)
    m_FLP_LR.optimize()
    Opt_FLP_LR_Val.append(m_FLP_LR.objVal)
```

AFL_LR model

In [ ]:
```
# AFL_LR model formulation

m_AFL_LR = gp.Model('facility_location_AFL_LR')

# Replacing the binary constraint with the box constraint
select_AFL_LR = m_AFL_LR.addVars(num_facilities, ub=1, vtype=GRB.CONTINUOUS, name='Select_AFL_LR')

assign_AFL_LR = m_AFL_LR.addVars(cartesian_prod, ub=1, vtype=GRB.CONTINUOUS, name='Assign_AFL_LR')

# Different constraint for the AFL_LR model
m_AFL_LR.addConstrs((gp.quicksum(assign_AFL_LR[(c,f)] for c in range(num_customers)) <=
                     select_AFL_LR[f]*num_customers for c,f in cartesian_prod), name='Setup2ship_AFL_LR')

m_AFL_LR.addConstrs((gp.quicksum(assign_AFL_LR[(c,f)] for f in range(num_facilities)) ==
                     1 for c in range(num_customers)), name='Demand_AFL_LR')


Opt_AFL_LR_Val = [] # Record optimal values for AFL_LR model

for i in range(100):
    m_AFL_LR.setObjective(select_AFL_LR.prod(setup_cost[i])+assign_AFL_LR.prod(distance[i]), GRB.MINIMIZE)
    m_AFL_LR.optimize()
    Opt_AFL_LR_Val.append(m_AFL_LR.objVal)
```

Comparisons of results from the four models

In [8]:
```
# Comparing the last instance of the optimal values of the four models to get a general idea

print(Opt_FLP_Val[-1])
print(Opt_AFL_Val[-1])
print(Opt_FLP_LR_Val[-1])
print(Opt_AFL_LR_Val[-1])
```

```
4.776366365335146
4.776366365335146
4.776366365335145
3.4034990953486775
```

In [9]:
```
# Check the difference between FLP_Val and FLP_LR_Val

diff_FLP = []

for i in range (len(Opt_FLP_Val)):
    diff_FLP.append(Opt_FLP_Val[i] - Opt_FLP_LR_Val[i])

print(max(diff_FLP))
print(min(diff_FLP))
```

```
1.7763568394002505e-15
-2.6645352591003757e-15
```

In [10]:
```
# Check the difference between AFL_Val and AFL_LR_Val

diff_AFL = []

for i in range (len(Opt_AFL_Val)):
    diff_AFL.append(Opt_AFL_Val[i] - Opt_AFL_LR_Val[i])

print(max(diff_AFL))
print(min(diff_AFL))
```

```
2.8961983924675803
0.66761265347471
```

In [11]:
```
# Check the difference between FLP_Val and AFL_Val

diff_FLP_AFL = []
```

```
for i in range(len(Opt_FLP_Val)):
    diff_FLP_AFL.append(Opt_FLP_Val[i] - Opt_AFL_Val[i])

print(max(diff_FLP_AFL))
print(min(diff_FLP_AFL))
```

```
8.881784197001252e-16
0.0
```

In [12]:

```
# Check the difference between FLP_LR_Val and AFL_LR_Val

LR_diff_FLP_AFL = []

for i in range(len(Opt_FLP_LR_Val)):
    LR_diff_FLP_AFL.append(Opt_FLP_LR_Val[i] - Opt_AFL_LR_Val[i])

print(max(LR_diff_FLP_AFL))
print(min(LR_diff_FLP_AFL))
```

```
2.8961983924675803
0.66761265347471
```

Checking solutions of the last instance of the four models

In [ ]:

```
# Print solution for last instance of FLP model

for v in m_FLP.getVars():
     print('%s %g' % (v.varName, v.x))
```

In [ ]:

```
# Print solution for last instance of FLP_LR model

for v in m_FLP_LR.getVars():
     print('%s %g' % (v.varName, v.x))
```

In [ ]:

```
# Print solution for last instance of AFL model

for v in m_AFL.getVars():
     print('%s %g' % (v.varName, v.x))
```

In [ ]:

```
# Print solution for last instance of AFL_LR model

for v in m_AFL_LR.getVars():
     print('%s %g' % (v.varName, v.x))
```

Appendix 2

2. Traveling Salesman Problem

In [1]:
```
import random
import csv
import numpy as np
from itertools import product
import matplotlib.pyplot as plt
```

In [2]:
```
# Define distances

def compute_distance(loc1, loc2):
    dx = loc1[0] - loc2[0]
    dy = loc1[1] - loc2[1]
    return np.sqrt(dx**2+dy**2)

# Define function that takes in a tour and outputs the total distance

def tour_dist(tour):
    total_dist = 0
    for i in range(len(tour)-1):
        dist = distance[(tour[i], tour[i+1])]
        total_dist += dist
    total_dist += distance[(tour[-1], tour[0])] # distance returning to starting location
    return total_dist

# Define function that takes in a tour and outputs a perturbed tour, with first perturb function

def perturb(tour):
    indices = random.sample(range(0,len(tour)), 2) # random.sample ensures unique indices
    i = min(indices)
    j = max(indices)
    perturbed_tour = tour[:i] + tour[i:j+1][::-1] + tour[j+1:]
    return perturbed_tour
```

In [3]:
```
# Setting parameters

num_cities = 100 # n = 100
cart_prod = list(product(range(num_cities), range(num_cities))) # Label the cities by their indices
```

In [4]:
```
# Generate the coordinates of the cities

cities = [(np.random.rand(),np.random.rand()) for i in range(num_cities)]

# Compute distance matrix

distance = ({(c1,c2) : compute_distance(cities[c1], cities[c2]) for c1, c2 in cart_prod})
```

Simulated annealing algorithm with first perturb rule

In [5]:
```
# Initialize same random tour for both perturb rules for comparison

cities_index = [i for i in range(num_cities)]
random.shuffle(cities_index)
initial_tour = cities_index
```

In [6]:
```
# Initial tour for first perturb rule

current_tour = initial_tour.copy()

# Distance of initial tour

tour_dist(current_tour)
```

Out[6]:
```
50.5172279351468
```

In [7]:
```
# Initialize temperature parameter

T = 100
T_update = 0.99 # eta = 0.99

# Implement algorithm with first perturb rule

accepted_count = 0

for i in range(10000):
    f_curr = tour_dist(current_tour)
    cand_tour = perturb(current_tour)
    f_cand = tour_dist(cand_tour)
    u = np.random.rand()
    if np.exp(-((f_cand - f_curr) / T)) >= u:
        current_tour = cand_tour
```

```
        accepted_count += 1
    T = T_update * T
```

```
<ipython-input-7-d61a0cc8762c>:15: RuntimeWarning: overflow encountered in exp
  if np.exp(-((f_cand - f_curr) / T)) >= u:
```

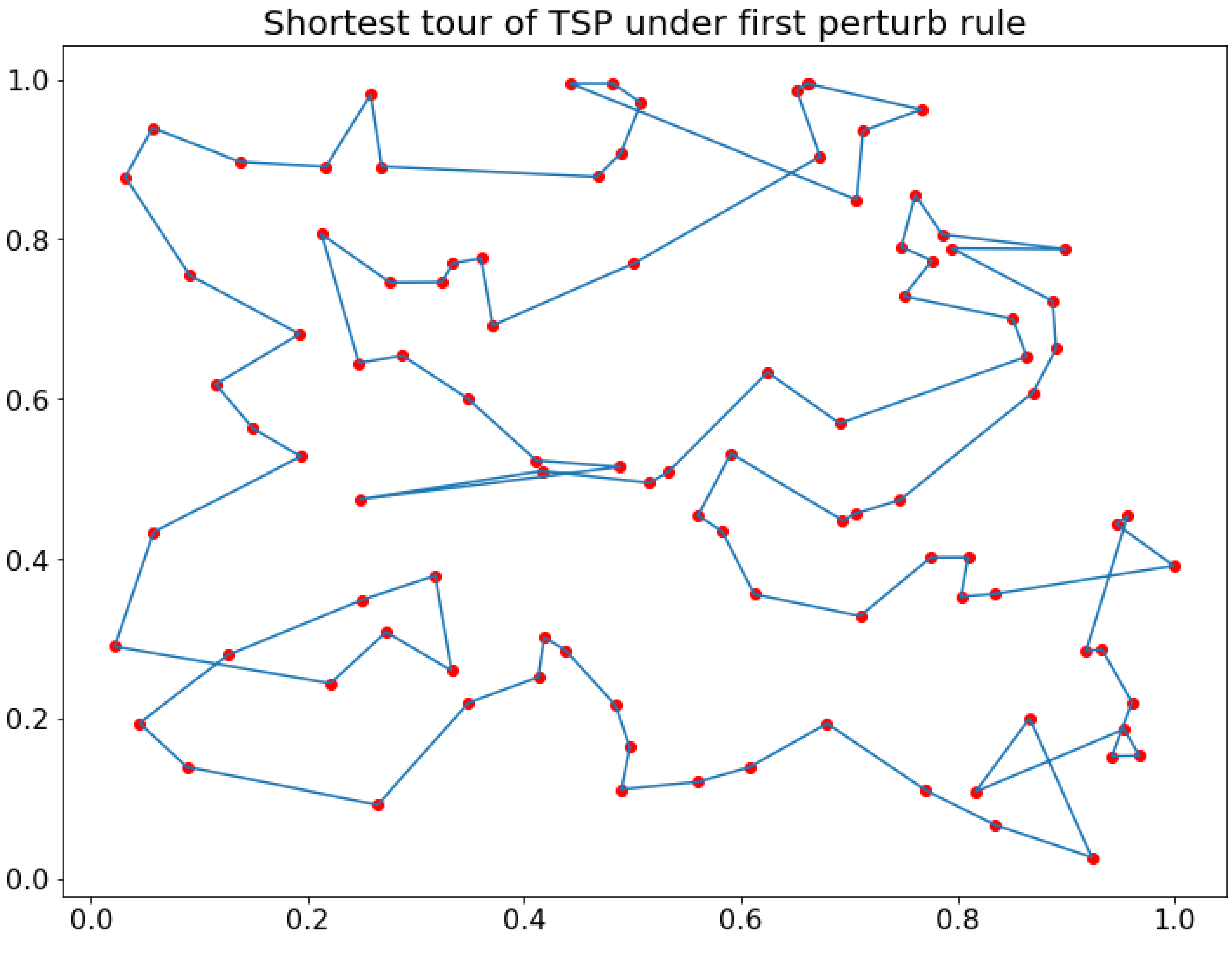
In [8]:

```
# Results for first perturb rule

# Number of candidate tours accepted
print(accepted_count)

# Distance of shortest tour
print(tour_dist(current_tour))
```

```
889
9.199323401302737
```

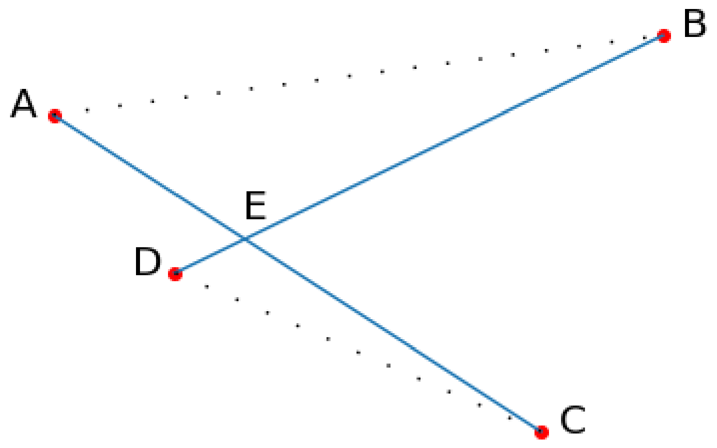
In [9]:

```
# Plot the cities with the tour under first perturb rule

fig, ax = plt.subplots(figsize=(12,9))


# Plot the coordinates of the cities
x_coord_of_all_cities = [coord[0] for coord in cities]
y_coord_of_all_cities = [coord[1] for coord in cities]
ax.scatter(x_coord_of_all_cities, y_coord_of_all_cities, color = 'red')

# Plot the shortest tour connecting every city
x_coord_of_tour = [cities[loc][0] for loc in current_tour]
y_coord_of_tour = [cities[loc][1] for loc in current_tour]

ax.plot(x_coord_of_tour, y_coord_of_tour, color = '#1f77b4')
ax.plot([x_coord_of_tour[-1], x_coord_of_tour[0]], [y_coord_of_tour[-1], y_coord_of_tour[0]],
        color = '#1f77b4')

ax.set_title('Shortest tour of TSP under first perturb rule', fontsize = 20)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)

plt.show()
```



In [10]:

```
# Plot the intersection of any two edges in general

fig, ax = plt.subplots()

ax.scatter([0.12,0.2,0.1,0.18],[0.14,0.2,0.18,0.1], color = 'red')

ax.plot([0.12,0.2],[0.14,0.2], color = '#1f77b4')
ax.plot([0.1,0.18],[0.18,0.1], color = '#1f77b4')
ax.plot([0.1,0.2],[0.18,0.2], linestyle = (0, (1, 10)), color = 'black')
ax.plot([0.12,0.18],[0.14,0.1], linestyle = (0, (1, 10)), color = 'black')

plt.text(0.093, 0.18, 'A', fontsize = 20)
plt.text(0.203, 0.2, 'B', fontsize = 20)
plt.text(0.183, 0.1, 'C', fontsize = 20)
plt.text(0.113, 0.14, 'D', fontsize = 20)
plt.text(0.131, 0.154, 'E', fontsize = 20)

plt.axis('off')
plt.show()
```



Simulated annealing algorithm with second perturb rule

In [11]:

```
# Define new perturbed tour function, with second perturb rule

def n_perturb(tour):
    indices = random.sample(range(0,len(tour)), 2) # random.sample ensures unique indices
    i = min(indices)
    j = max(indices)
    tour[i], tour[j] = tour[j], tour[i]
    return tour
```

In [12]:

```
# Initial tour for second perturb rule

n_current_tour = initial_tour.copy()

# Distance of initial tour

tour_dist(n_current_tour)
```

Out[12]:

```
50.5172279351468
```

In [13]:

```
# Initialize temperature parameter

T = 100
T_update = 0.99 # eta = 0.99

# Implement algorithm with second perturb rule, replace perturb with n_perturb

n_accepted_count = 0

for i in range(10000):
    f_curr = tour_dist(n_current_tour)
    cand_tour = n_perturb(n_current_tour)
    f_cand = tour_dist(cand_tour)
    u = np.random.rand()
    if np.exp(-((f_cand - f_curr) / T)) >= u:
        n_current_tour = cand_tour
        n_accepted_count += 1
    T = T_update * T
```

```
<ipython-input-13-3cdb247b8186>:15: RuntimeWarning: overflow encountered in exp
  if np.exp(-((f_cand - f_curr) / T)) >= u:
```

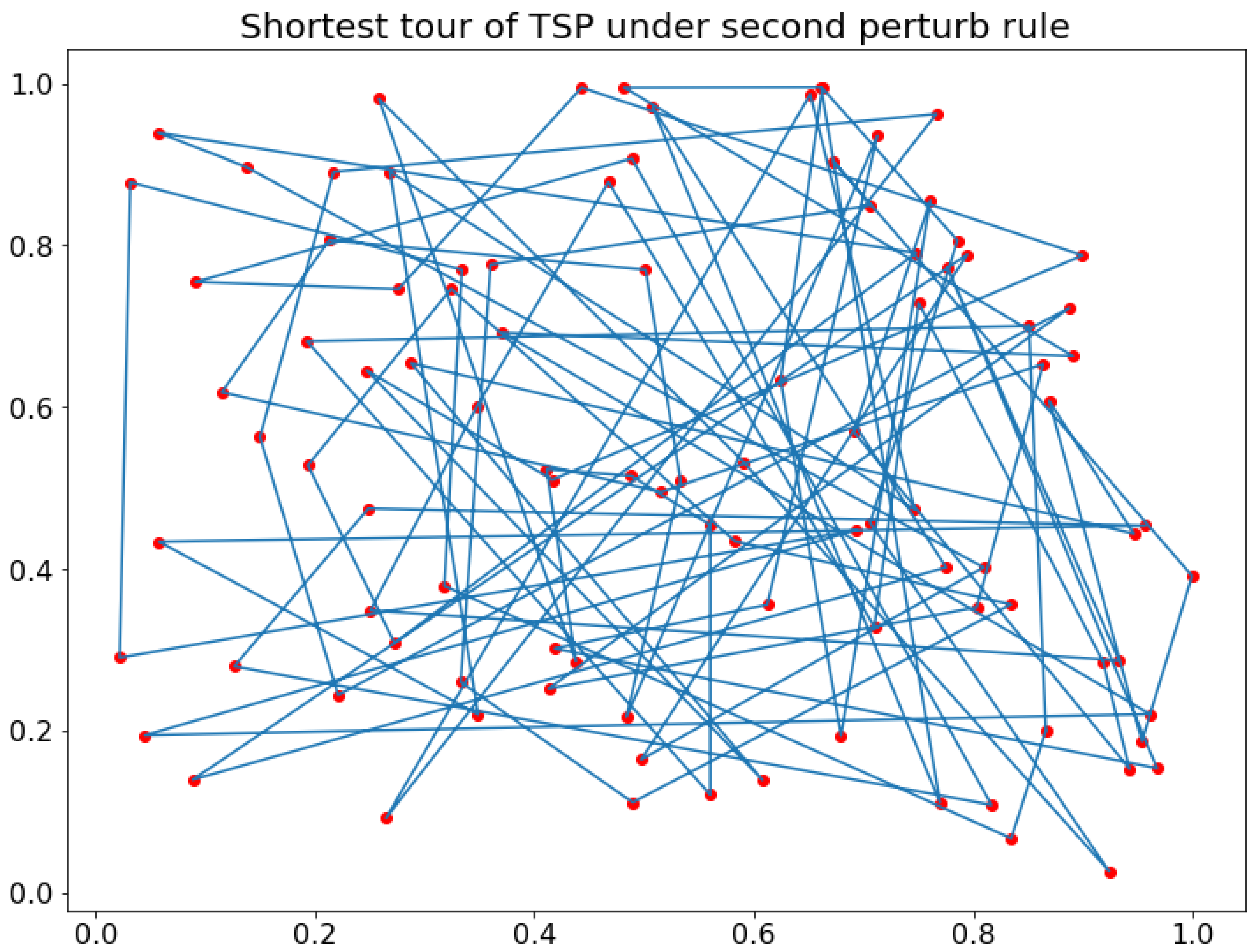
In [14]:

```
# Results for second perturb rule

# Number of candidate tours accepted
print(n_accepted_count)

# Distance of shortest tour
print(tour_dist(n_current_tour))
```

```
5266
48.867230128219546
```

In [15]:

```
# Plot the cities with the tour under second perturb rule

fig, ax = plt.subplots(figsize=(12,9))

# Plot the coordinates of the cities
x_coord_of_all_cities = [coord[0] for coord in cities]
y_coord_of_all_cities = [coord[1] for coord in cities]
ax.scatter(x_coord_of_all_cities, y_coord_of_all_cities, color = 'red')

# Plot the shortest tour connecting every city
x_coord_of_tour = [cities[loc][0] for loc in n_current_tour]
y_coord_of_tour = [cities[loc][1] for loc in n_current_tour]

ax.plot(x_coord_of_tour, y_coord_of_tour, color = '#1f77b4' )
ax.plot([x_coord_of_tour[-1], x_coord_of_tour[0]], [y_coord_of_tour[-1], y_coord_of_tour[0]],
        color = '#1f77b4')

ax.set_title('Shortest tour of TSP under second perturb rule', fontsize = 20)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)

plt.show()
```



In [16]:

```
# Difference between first perturb rule and second perturb rule in general

fig, ax = plt.subplots(1,2,figsize=(14,1))

ax[0].scatter([0.05,0.1,0.15,0.2,0.25,0.3], [0.5,0.5,0.5,0.5,0.5,0.5], color = 'red')
ax[0].plot([0.1,0.15,0.2,0.25], [0.5,0.5,0.5,0.5], color = '#1f77b4')
ax[0].plot([0.05,0.1],[0.5,0.5], color = 'orange')
ax[0].plot([0.25,0.3],[0.5,0.5], color = 'orange')


ax[1].scatter([0.05,0.1,0.15,0.2,0.25,0.3], [0.5,0.5,0.5,0.5,0.5,0.5], color = 'red')
ax[1].plot([0.15,0.2], [0.5,0.5], color = '#1f77b4')
ax[1].plot([0.05,0.1,0.15],[0.5,0.5,0.5], color = 'orange')
ax[1].plot([0.2,0.25,0.3],[0.5,0.5,0.5], color = 'orange')

ax[0].text(0.0975,0.475,'i',fontsize = 20)
ax[0].text(0.2475,0.475,'j',fontsize = 20)
ax[0].text(0.04,0.475,'i-1',fontsize = 20)
ax[0].text(0.29,0.475,'j+1',fontsize = 20)

ax[1].text(0.0975,0.475,'i',fontsize = 20)
ax[1].text(0.2475,0.475,'j',fontsize = 20)
ax[1].text(0.04,0.475,'i-1',fontsize = 20)
ax[1].text(0.29,0.475,'j+1',fontsize = 20)

ax[0].set_title('First perturb rule', fontsize = 20)
ax[1].set_title('Second perturb rule', fontsize = 20)


ax[0].axis('off')
ax[1].axis('off')
plt.show()
```