Discrete Optimization Computational Assignment

Huang Rui

October 29, 2023

1 Facility Location Problem

1.1 Solution to Question(a)

In (FLP), there are mn constraints of the form $x_{i,j} \le y_j$.There are mn constraints of the form $0 \le x_{i,j} \le 1$. So total inequalities is $2mn$.
In (AFL), there are n constraints of the form $\sum\limits_{i=1}^{m} x_{i,j} \le my_j$. There are mn constraints of the form $0 \le x_{i,j} \le 1$. So total inequalities is $mn + n$.
Comparing $2mn$ versus $mn + n$, (AFL) has fewer inequalities.

1.2 Solution to Question(b)

To show feasible regions are equal:
Take any feasible solution (x, y) to (FLP). Since it satisfies $x_{i,j} \le y_j$ for all i, j, summing over i gives $\sum\limits_{i=1}^{m} x_{i,j} \le \sum\limits_{i=1}^{m} y_j = my_j$. So it satisfies the $\sum\limits_{i=1}^{m} x_{i,j} \le my_j$ constraints. It also satisfies the $\sum\limits_{j=1}^{n} x_{i,j} = 1$constraints. So it is feasible for (AFL).
Conversely, take any feasible solution (x, y) to (AFL). Since it satisfies $0 \le x_{i,j} \le 1$ and $\sum\limits_{i=1}^{m} x_{i,j} \le my_j$, it must satisfy $x_{i,j} \le y_j$ for all i, j. It also satisfies $\sum\limits_{j=1}^{n} x_{i,j} = 1$. So it is feasible for (FLP).
Since the feasible sets contain each other, they must be equal.
To conclude the two formulations are equivalent:
Since we have shown:
Any feasible solution to (FLP) is also feasible for (AFL).
Any feasible solution to (AFL) is also feasible for (FLP).
This demonstrates that the feasible regions for the two formulations are identical.
And since the objective functions are the same:

$$\min \quad \sum_{j=1}^{n} c_j y_j + \sum_{i,j=1}^{m,n} d_{i,j} x_{i,j}$$

We can conclude that:
The formulations (FLP) and (AFL) are mathematically equivalent.

1.3 Solution to Question(c)

To write the linear relaxation of (FLP), we simply replace the y_j binary constraints with $0 \le y_j \le 1$ box constraints.

$$
\begin{aligned}
\min \quad & \sum_{j=1}^{n} c_j y_j + \sum_{i,j=1}^{m,n} d_{i,j} x_{i,j} \\
\text{s.t.} \quad & \sum_{j=1}^{n} x_{i,j} = 1 \qquad \text{for all} \quad i \\
& x_{i,j} \le y_j \qquad \text{for all} \quad i, j \\
& 0 \le x_{i,j} \le 1 \\
& 0 \le y_j \le 1
\end{aligned}
\tag{FLP-LR}
$$

1.4 Solution to Question(d)

Similar idea for (AFL), replace binary y_j with box constraints.

$$
\begin{aligned}
\min \quad & \sum_{j=1}^{n} c_j y_j + \sum_{i,j=1}^{m,n} d_{i,j} x_{i,j} \\
\text{s.t.} \quad & \sum_{j=1}^{n} x_{i,j} = 1 \qquad \text{for all} \quad i \\
& \sum_{i=1}^{m} x_{i,j} \le m y_j \qquad \text{for all} \quad j \\
& 0 \le x_{i,j} \le 1 \\
& 0 \le y_j \le 1
\end{aligned}
\tag{AFL-LR}
$$

1.5 Solution to Question(e)

I. (FLP-Val) = (AFL-Val):
As shown in Question(b), (FLP) and (AFL) are equivalent integer programs, so their optimal values must be equal.
II. (FLP-LR-Val) $\le$ (FLP-Val):
The linear relaxation (FLP-LR) is a relaxation of (FLP), so its optimal value must be less than or equal to the optimal value of (FLP).
III. (AFL-LR-Val) $\le$ (AFL-Val):
The linear relaxation (AFL-LR) is a relaxation of (AFL), so its optimal value must be less than or equal to the optimal value of (AFL).
IV. (AFL-LR-Val) $\le$ (FLP-LR-Val):
Since y_j is continuous variable and from Question(b) we obtain that (FLP-LR) has more constraints than (AFL-LR). Therefore, the optimal value of (AFL-LR) must be less than or equal to the optimal value of (FLP-LR).
In summary, the optimal values can be arranged as:
(AFL-LR-Val) $\le$ (FLP-LR-Val) $\le$ (FLP-Val) = (AFL-Val)

1.6 Solution to Question(f)

The following code can solve this question:

```python
import gurobipy as gp
from gurobipy import GRB
from itertools import product
import numpy as np
```

```python
# Parameters
num_trials = 100

num_customers = 15
num_facilities = 10

# Generate random data
customers = [(np.round(np.random.rand(),2), np.round(np.random.rand(),2)) for i in
range(num_customers)]
facilities = [(np.round(np.random.rand(),2), np.round(np.random.rand(),2)) for i in
range(num_facilities)]

setup_cost = [1.0 for i in range(num_facilities)]
cost_per_mile = 1.0

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

[1] In this way, we generate random instances of the facility location problem and conclude the distance matrix between every facility and customer.

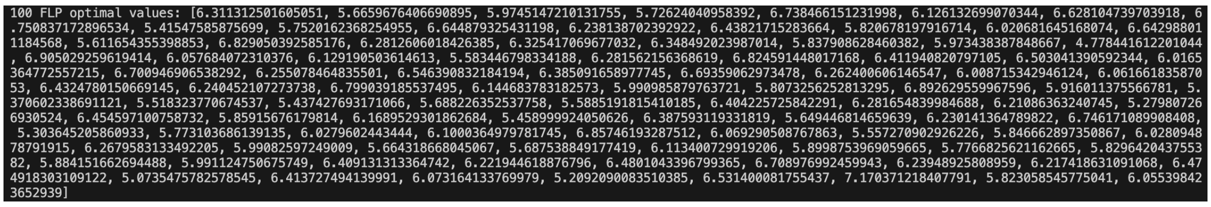
1.7 Solution to Question(g)

First, we solve the facility location problem using (FLP) and solve the linear relaxation (FLP-LR) corresponding to (FLP).
We can solve this by following code:

```python
import gurobipy as gp
from gurobipy import GRB
from itertools import product
import numpy as np

# Parameters
num_trials = 100

num_customers = 15
num_facilities = 10

FLP_vals = []
FLP_LR_vals = []

for t in range(num_trials):

  # Generate random data
  customers = [(np.round(np.random.rand(),2), np.round(np.random.rand(),2)) for i in
  range(num_customers)]
  facilities = [(np.round(np.random.rand(),2), np.round(np.random.rand(),2)) for i in
  range(num_facilities)]

  setup_cost = [1.0 for i in range(num_facilities)]
  cost_per_mile = 1.0

  # Compute distances
```

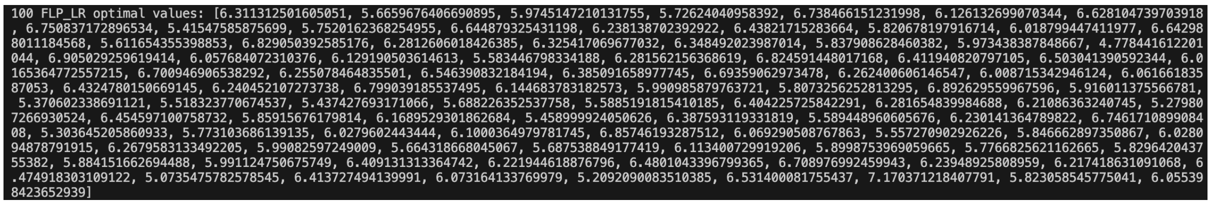
```
    def compute_distance(loc1, loc2):
        dx = loc1[0] - loc2[0]
        dy = loc1[1] - loc2[1]
        return np.sqrt(dx**2 + dy**2)

    # Compute parameters
    cartesian_prod = list(product(range(num_customers), range(num_facilities)))
    shipping_cost = {(c,f) : cost_per_mile*compute_distance(customers[c], facilities[f]) for c, f in
    cartesian_prod}

    # FLP MIP model
    m_FLP = gp.Model("FLP")
    select = m_FLP.addVars(num_facilities, vtype=GRB.BINARY, name="Select")
    assign = m_FLP.addVars(cartesian_prod, ub=1, vtype=GRB.CONTINUOUS, name="Assign")

    m_FLP.addConstrs((assign[(c,f)] <= select[f] for c,f in cartesian_prod), name="Setup2ship")
    m_FLP.addConstrs((gp.quicksum(assign[(c,f)] for f in range(num_facilities)) == 1 for c
    in range(num_customers)), name="Demand")

    m_FLP.setObjective(select.prod(setup_cost) + assign.prod(shipping_cost), GRB.MINIMIZE)

    m_FLP.optimize()
    FLP_vals.append(m_FLP.ObjVal)

    # FLP Linear Relaxation
    m_FLP_LR = gp.Model("FLP_LR")
    select = m_FLP_LR.addVars(num_facilities, ub=1, name="Select")
    assign = m_FLP_LR.addVars(cartesian_prod, ub=1, name="Assign")

    m_FLP_LR.addConstrs((assign[(c,f)] <= select[f] for c,f in cartesian_prod), name="Setup2ship")
    m_FLP_LR.addConstrs((gp.quicksum(assign[(c,f)] for f in range(num_facilities)) == 1 for c
    in range(num_customers)), name="Demand")

    m_FLP_LR.setObjective(select.prod(setup_cost) + assign.prod(shipping_cost), GRB.MINIMIZE)

    m_FLP_LR.optimize()
    FLP_LR_vals.append(m_FLP_LR.ObjVal)

print(len(FLP_vals), "FLP optimal values:", FLP_vals)
print(len(FLP_LR_vals), "FLP_LR optimal values:", FLP_LR_vals)
```

[1] After running this program, we obtain the optimal values to (FLP) and (FLP-LR).
For (FLP), the 100 optimal values lie below:



```
100 FLP optimal values: [6.311312501605051, 5.6659676406690895, 5.9745147210131755, 5.72624040958392, 6.738466151231998, 6.126132699070344, 6.628104739703918, 6
.750837172896534, 5.41547585875699, 5.7520162368254955, 6.644879325431198, 6.238138702392922, 6.43821715283664, 5.820678197916714, 6.020681645168074, 6.64298801
1184568, 5.611654355398853, 6.829050392585176, 6.2812606018426385, 6.325417069677032, 6.348492023987014, 5.837908628460382, 5.973438387848667, 4.778441612201044
, 6.905029259619414, 6.057684072310376, 6.129190503614613, 5.583446798334188, 6.281562156368619, 6.824591448017168, 6.411940820797105, 6.503041390592344, 6.0165
364772557215, 6.700946906538292, 6.255078464835501, 6.546390832184194, 6.385091658977745, 6.69359062973478, 6.262400606146547, 6.008715342946124, 6.061661835870
53, 6.4324780150669145, 6.240452107273738, 6.799039185537495, 6.144683783182573, 5.990985879763721, 5.8073256252813295, 6.892629559967596, 5.916011375566781, 5.
370602338691121, 5.518323770674537, 5.437427693171066, 5.688226352537758, 5.5885191815410185, 6.404225725842291, 6.281654839984688, 6.21086363240745, 5.27980726
6930524, 6.454597100758732, 5.85915676179814, 6.1689529301862684, 5.458999924050626, 6.387593119331819, 5.649446814659639, 6.230141364789822, 6.746171089908408,
 5.303645205860933, 5.773103686139135, 6.0279602443444, 6.1000364979781745, 6.85746193287512, 6.069290508767863, 5.557270902926226, 5.846662897350867, 6.0280948
78791915, 6.2679583133492205, 5.99082597249009, 5.664318668045067, 5.687538849177419, 6.113400729919206, 5.8998753969059665, 5.7766825621162665, 5.8296420437553
82, 5.884151662694488, 5.991124750675749, 6.409131313364742, 6.221944618876796, 6.4801043396799365, 6.708976992459943, 6.23948925808959, 6.217418631091068, 6.47
4918303109122, 5.0735475782578545, 6.413727494139991, 6.073164133769979, 5.2092090083510385, 6.531400081755437, 7.170371218407791, 5.823058545775041, 6.05539842
3652939]
```

For (FLP-LR), the 100 optimal values lie below:

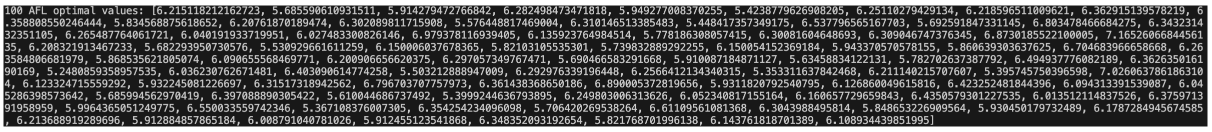


```
100 FLP_LR optimal values: [6.311312501605051, 5.6659676406690895, 5.9745147210131755, 5.72624040958392, 6.738466151231998, 6.126132699070344, 6.628104739703918
, 6.750837172896534, 5.41547585875699, 5.7520162368254955, 6.644879325431198, 6.238138702392922, 6.43821715283664, 5.820678197916714, 6.018799447411977, 6.64298
8011184568, 5.611654355398853, 6.829050392585176, 6.2812606018426385, 6.325417069677032, 6.348492023987014, 5.837908628460382, 5.973438387848667, 4.778441612201
044, 6.905029259619414, 6.057684072310376, 6.129190503614613, 5.583446798334188, 6.281562156368619, 6.824591448017168, 6.411940820797105, 6.503041390592344, 6.0
165364772557215, 6.700946906538292, 6.255078464835501, 6.546390832184194, 6.385091658977745, 6.69359062973478, 6.262400606146547, 6.008715342946124, 6.061661835
87053, 6.4324780150669145, 6.240452107273738, 6.799039185537495, 6.144683783182573, 5.990985879763721, 5.8073256252813295, 6.892629559967596, 5.916011375566781,
 5.370602338691121, 5.518323770674537, 5.437427693171066, 5.688226352537758, 5.5885191815410185, 6.404225725842291, 6.281654839984688, 6.21086363240745, 5.27980
7266930524, 6.454597100758732, 5.85915676179814, 6.1689529301862684, 5.458999924050626, 6.387593119331819, 5.589448960605676, 6.230141364789822, 6.7461710899084
08, 5.303645205860933, 5.773103686139135, 6.0279602443444, 6.1000364979781745, 6.85746193287512, 6.069290508767863, 5.557270902926226, 5.846662897350867, 6.0280
94878791915, 6.2679583133492205, 5.99082597249009, 5.664318668045067, 5.687538849177419, 6.113400729919206, 5.8998753969059665, 5.7766825621162665, 5.8296420437
55382, 5.884151662694488, 5.991124750675749, 6.409131313364742, 6.221944618876796, 6.4801043396799365, 6.708976992459943, 6.23948925808959, 6.217418631091068, 6
.474918303109122, 5.0735475782578545, 6.413727494139991, 6.073164133769979, 5.2092090083510385, 6.531400081755437, 7.170371218407791, 5.823058545775041, 6.05539
8423652939]
```

Second, we solve the facility location problem using (AFL) and solve the linear relaxation (AFL-LR) corresponding to (AFL).
We can solve this by following code:

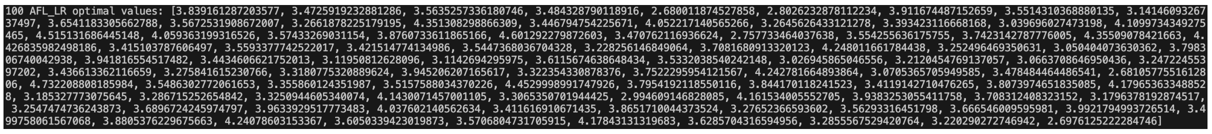
```python
import gurobipy as gp
from gurobipy import GRB
from itertools import product
import numpy as np

# Parameters
num_trials = 100

num_customers = 15
num_facilities = 10

AFL_vals = []
AFL_LR_vals = []

for t in range(num_trials):

  # Generate random data
  customers = [(np.round(np.random.rand(),2), np.round(np.random.rand(),2)) for i in
  range(num_customers)]
  facilities = [(np.round(np.random.rand(),2), np.round(np.random.rand(),2)) for i in
  range(num_facilities)]

  setup_cost = [1.0 for i in range(num_facilities)]
  cost_per_mile = 1.0

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

  # Compute parameters
  cartesian_prod = list(product(range(num_customers), range(num_facilities)))
  shipping_cost = {(c,f) : cost_per_mile*compute_distance(customers[c], facilities[f]) for c, f
  in cartesian_prod}

  # AFL MIP model
  m_AFL = gp.Model("AFL")
  select = m_AFL.addVars(num_facilities, vtype=GRB.BINARY, name="Select")
  assign = m_AFL.addVars(cartesian_prod, ub=1, vtype=GRB.CONTINUOUS, name="Assign")

  m_AFL.addConstrs((gp.quicksum(assign[(c,f)] for c in range(num_customers)) <= num_customers*
  select[f] for f in range(num_facilities)), name="Setup2ship")
  m_AFL.addConstrs((gp.quicksum(assign[(c,f)] for f in range(num_facilities)) == 1 for c
  in range(num_customers)), name="Demand")

  m_AFL.setObjective(select.prod(setup_cost) + assign.prod(shipping_cost), GRB.MINIMIZE)

  m_AFL.optimize()
  AFL_vals.append(m_AFL.ObjVal)

  # AFL Linear Relaxation
  m_AFL_LR = gp.Model("AFL_LR")
  select = m_AFL_LR.addVars(num_facilities, ub=1, name="Select")
  assign = m_AFL_LR.addVars(cartesian_prod, ub=1, name="Assign")

  m_AFL_LR.addConstrs((gp.quicksum(assign[(c,f)] for c in range(num_customers)) <= num_customers*
  select[f] for f in range(num_facilities)), name="Setup2ship")
  m_AFL_LR.addConstrs((gp.quicksum(assign[(c,f)] for f in range(num_facilities)) == 1 for c
  in range(num_customers)), name="Demand")

  m_AFL_LR.setObjective(select.prod(setup_cost) + assign.prod(shipping_cost), GRB.MINIMIZE)

  m_AFL_LR.optimize()
  AFL_LR_vals.append(m_AFL_LR.ObjVal)

print(len(AFL_vals), "AFL optimal values:", AFL_vals)
print(len(AFL_LR_vals), "AFL_LR optimal values:", AFL_LR_vals)
```

After running this program, we obtain the optimal values to (AFL) and (AFL-LR).
For (AFL), the 100 optimal values lie below:



```
100 AFL optimal values: [6.215118212162723, 5.685590610931511, 5.914279472766842, 6.282498473471818, 5.949277008370255, 5.4238779626908205, 6.25110279429134, 6.218596511009621, 6.362915139578219, 6
.358808550246444, 5.834568875618652, 6.20761870189474, 6.302089811715908, 5.576448817469004, 6.310146513385483, 5.448417357349175, 6.537796565167703, 5.692591847331145, 6.803478466684275, 6.3432314
32351105, 6.265487764061721, 6.040191933719951, 6.027483300826146, 6.979378116939405, 6.135923764984514, 5.778186308057415, 6.30081604648693, 6.309046747376345, 6.8730185522100005, 7.16526066844561
35, 6.208321913467233, 5.682293950730576, 5.530929661611259, 6.150006037678365, 5.82103105535301, 5.739832889292255, 6.150054152369184, 5.943370570578155, 5.860639303637625, 6.704683966658668, 6.26
3584806681979, 5.868535621805074, 6.090655568469771, 6.200906656620375, 6.297057349767471, 5.690466583291668, 5.910087184871127, 5.63458834122131, 5.782702637387792, 6.494937776082189, 6.3626350161
90169, 5.2480859358957535, 6.036230762671481, 6.403090614774258, 5.503212888947009, 6.292976339196448, 6.2566412134340315, 5.353311637842468, 6.211140215707607, 5.395745750396598, 7.026063786186310
4, 6.123324715559292, 5.932245081226697, 6.31517318942562, 6.796703707757973, 6.361438368650186, 6.890005372819656, 5.9311820792540795, 6.126860049615816, 6.423252481844396, 6.094313391539087, 6.04
5286398573642, 5.685994562970419, 6.397088890305422, 5.610044686737492, 5.3999244636793895, 6.249803006313626, 6.052340817155164, 6.160657729659843, 6.4350579301227535, 6.013512114837526, 6.3759713
91958959, 5.9964365051249775, 6.550033559742346, 5.367108376007305, 6.354254234096098, 5.706420269538264, 6.61109561081368, 6.3043988495814, 5.848653226909564, 5.930450179732489, 6.1787284945674585
, 6.213688919289696, 5.912884857865184, 6.008791040781026, 5.912455123541868, 6.348352093192654, 5.821768701996138, 6.143761818701389, 6.108934439851995]
```

For (AFL-LR), the 100 optimal values lie below:



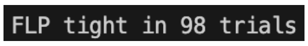
```
100 AFL_LR optimal values: [3.839161287203577, 3.4725919232881286, 3.5635257336180746, 3.484328790118916, 2.680011874527858, 2.8026232878112234, 3.911674487152659, 3.5514310368880135, 3.14146093267
37497, 3.6541183305662788, 3.5672531908672007, 3.2661878225179195, 4.351308298866309, 3.446794754225671, 4.052217140565266, 3.2645626433121278, 3.393423116668168, 3.039696027473198, 4.1099734349275
465, 4.515131686445148, 4.059363199316526, 3.57433269031154, 3.8760733611865166, 4.601292279872603, 3.470762116936624, 2.757733464037638, 3.554255636175755, 3.7423142787776005, 4.35509078421663, 4.
426835982498186, 3.415103787606497, 3.5593377742522017, 3.421514774134986, 3.5447368036704328, 3.228256146849064, 3.7081680913320123, 4.248011661784438, 3.252496469350631, 3.050404073630362, 3.7983
06740042938, 3.941816554517482, 3.4434606621752013, 3.11950812628096, 3.1142694295975, 3.6115674638648434, 3.5332038540242148, 3.026945865046556, 3.2120454769137057, 3.0663708646950436, 3.247224553
97202, 3.436613362116659, 3.275841615230766, 3.3180775320889624, 3.945206207165617, 3.322354330878376, 3.7522295954121567, 4.242781664893864, 3.0705365705949585, 3.478484464486541, 2.68105775516128
06, 4.732208808185984, 3.5486302772061653, 3.355860124351987, 3.5157588034370226, 4.4529998991747926, 3.7954192118550116, 3.844170118241523, 3.4119142710476265, 3.8073974651835085, 4.17965363348852
8, 3.185327773075645, 3.286715252654842, 3.3250944605340074, 4.1430071457001105, 3.3065350701944425, 2.994609146828085, 4.161534005552705, 3.9383253055411758, 3.708312408323152, 3.1796378192874517,
 3.2547474736243873, 3.6896724245974797, 3.9633929517773483, 4.037602140562634, 3.411616910671435, 3.8651710044373524, 3.27652366593602, 3.56293316451798, 3.666546009595981, 3.9921794993726514, 3.4
99758061567068, 3.8805376229675663, 4.24078603153367, 3.6050339423019873, 3.5706804731705915, 4.17843131319683, 3.6285704316594956, 3.2855567529420764, 3.220290272746942, 2.6976125222284746]
```

1.8 Solution to Question(h)

To compare and comment on the difference between the optimal values of all four optimization instances, we just need to count how often is (FLP-Val) equal to (FLP-LR-Val) and how often is (AFL-Val) equal to (AFL-LR-Val). To fulfill this purpose, we add 2 lines of code to the codes used in Question(g).
For (FLP) and (FLP-LR), the added code is:

```
num_tight = sum(v1 == v2 for v1, v2 in zip(FLP_vals, FLP_LR_vals))
print("\nFLP tight in", num_tight, "trials")
```

We can get the outcome below after 100 trials:

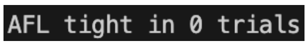


```
FLP tight in 98 trials
```

Similarly,for (AFL) and (AFL-LR), the added code is:

```python
num_tight = sum(v1 == v2 for v1, v2 in zip(AFL_vals, AFL_LR_vals))
print("\nAFL tight in", num_tight, "trials")
```

We can get the outcome below after 100 trials:



```
AFL tight in 0 trials
```

All in all, from what we have done before, we can conclude that the difference between the optimal values of (FLP) and (FLP-LR) is small. In this way, we can consider (FLP-Val) and (FLP-LR-Val) as the same. However, when it comes to (AFL) and (AFL-LR), (AFL-Val) is totally different from (AFL-LR-Val).In this way, we cannot consider (AFL-Val) and (AFL-LR-Val) as the same.

1.9 Solution to Question(i)

The goal of this question is to formulate a capacitated facility location problem (CFLP) as a MILP and write down the linear relaxation.
The key difference between (CFLP) and the regular (FLP) is that each facility now has a limited capacity that constrains the number of customers it can serve.
To write the (CFLP), we simply add a constraint:

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} c_j y_j + \sum_{i,j=1}^{m,n} d_{i,j} x_{i,j} \\ \text{s.t.} \quad & \sum_{j=1}^{n} x_{i,j} = 1 \qquad \text{for all} \quad i \\ & x_{i,j} \le y_j \qquad \text{for all} \quad i,j \\ & \sum_{i=1}^{m} x_{i,j} \le r_j y_j \\ & 0 \le x_{i,j} \le 1, y_j \in \{0,1\}\,. \end{aligned} \tag{CFLP}$$

To write the linear relaxation of (CFLP), we simply replace the y_j binary constraints with $0 \le y_j \le 1$ box constraints.

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} c_j y_j + \sum_{i,j=1}^{m,n} d_{i,j} x_{i,j} \\ \text{s.t.} \quad & \sum_{j=1}^{n} x_{i,j} = 1 \qquad \text{for all} \quad i \\ & x_{i,j} \le y_j \qquad \text{for all} \quad i,j \\ & \sum_{i=1}^{m} x_{i,j} \le r_j y_j \\ & 0 \le x_{i,j} \le 1 \\ & 0 \le y_j \le 1 \end{aligned} \tag{CFLP-LR}$$

1.10 Solution to Question(j)

From question(i) we know that what we need to do is just add a constraint.
To formulate this, we introduce a new parameter capacity that indicates the maximum number of customers that can be served by each facility.
Then we add the following new constraint to the (FLP) MILP model:

```
# Capacity constraint
m.addConstr(sum(assign[(c,f)] for c in customers) <= capacity*select[f] for f in facilities)
```

This requires that the total number of customers assigned to a facility is less than the capacity if the facility is selected.
After 100 trials, we obtain the following outcome:

```
CFLP tight in 0 trials
```

2 Traveling Salesman Problem

2.1 Solution to Question(a)

To solve this question, we realized we would need to loop through the tour to calculate the total distance. To look up the distances between cities, we would need their indices to index into the distance matrix. So we decided to convert the city tour into a list of indices by subtracting 1 from each city number.
Next, to loop through the tour, we knew we would need to access the next index after the current one. So we decided to use `'np.roll'` to shift the index list by -1 to get the next city index. Then, we indexed the distance matrix using these index lists to get the actual distances between consecutive cities. Finally, we summed all the retrieved distances to get the total tour distance. We also handled the edge case from last city back to first city.
The code is as follows:

```
# Distance calculation
def calculate_tour_distance(tour, dist_matrix):
    indices = np.array(tour) - 1
    dist = dist_matrix[indices, np.roll(indices, -1)]
    return dist.sum()
```

The `'calculate_tour_distance'` function calculates the total length of a city tour. It takes two parameters: `'tour'`, which is a list containing the order of cities, and `'dist_matrix'`, which is the matrix of distances between cities. This function first subtracts 1 from the city indices in the `'tour'` to match array indexing. Then, it uses NumPy array operations to calculate the distances between adjacent cities and returns the sum of these distances as the total tour length.

2.2 Solution to Question(b)

To solve this question, we knew we needed to modify the tour to create a candidate. Reversing a section of the tour seemed like a good option for perturbation.We decided to make a copy of the tour first to avoid modifying the original tour.
Next, we needed to randomly select two indices to define the section to reverse. `'np.random.choice'` is useful for sampling indices without replacement. Then, we used `'np.flip'` to neatly reverse the slice between the two selected indices. Finally, we return the new candidate tour with the reversed section.
The code is as follows:

```
# Tour perturbation
def generate_candidate(tour):
    n = len(tour)
    i, j = sorted(np.random.choice(range(n), 2, replace=False))
    candidate = tour.copy()
    candidate[i:j] = np.flip(candidate[i:j])
    return candidate
```

The `'generate_candidate'` function is responsible for creating a candidate tour by perturbing the current tour. It randomly selects two cities (i and j) from the tour and reverses the order of the cities between them, creating a new tour.

2.3 Solution to Question(c)

To solve this question, we first initialized the algorithm by randomly generating a tour and setting an initial temperature. Then, we knew we would need a loop where each iteration generates and evaluates a candidate tour. Inside the loop, we call the 'generate_candidate' function to get a new candidate tour. Next, we calculate the tour lengths for the current and candidate tours to get the change in cost delta_f.
Using delta_f and the temperature T, we probabilistically decide whether to move to the candidate tour or stay at the current tour. We also track the best tour found so far within the loop. Finally, we reduce the temperature T gradually according to the annealing schedule. After many iterations, the algorithm converges to a good tour.
The code is as follows:

```python
# (c) Simulated annealing
def simulated_annealing(dist_matrix, max_iters=10000, T0=100, eta=0.99):

    n = len(dist_matrix)
    current = np.random.permutation(n) + 1

    best = current.copy()
    f_curr = calculate_tour_distance(current, dist_matrix)
    T = T0

    for i in range(max_iters):
        candidate = generate_candidate(current)
        f_cand = calculate_tour_distance(candidate, dist_matrix)

        delta_f = f_cand - f_curr
        if delta_f < 0 or np.random.rand() < math.exp(-delta_f / T):
            current = candidate
            f_curr = f_cand

        if f_cand < calculate_tour_distance(best, dist_matrix):
            best = candidate

        T *= eta

    return best
```

The 'simulated_annealing' function is the core of the entire solution. It takes the distance matrix 'dist_matrix' and several algorithm parameters, including the maximum number of iterations, initial temperature (T0), and cooling rate (eta). Initially, it randomly generates a starting tour and uses the simulated annealing algorithm to find a shorter tour.
In each iteration, it generates a new candidate tour and calculates the total length of the new tour.
It compares the difference (delta_f) between the current tour and the candidate tour's length with the temperature (T) to decide whether to accept the new tour. If the new tour is shorter or with a certain probability, even if it's longer, it may be accepted to explore the search space.
It also keeps track of the best tour found.
The temperature (T) is reduced at each iteration to simulate the annealing process.

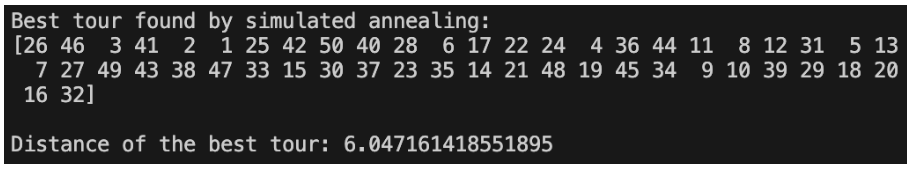
2.4 Solution to Question(d)

To test the implementation in Question(c), we first generate random coordinates for 50 cities and then calculate the distance matrix using the 'pairwise_distances' function.
Then, we print out the distance matrix that represents the distances between the randomly generated cities for inspection.

Finally, we run the simulated annealing algorithm to find the best city tour, and then print out the found tour and its total length.
The code is as follows:

```
# (d) Generate random 50 city instance and run simulated annealing

num_cities = 50
cities = np.random.rand(num_cities, 2)
dist_matrix = pairwise_distances(cities)

print(f"Distance matrix for {num_cities} randomly generated cities:")
print(dist_matrix)

best_tour = simulated_annealing(dist_matrix)

print(f"\nBest tour found by simulated annealing:")
print(best_tour)

best_distance = calculate_tour_distance(best_tour, dist_matrix)
print(f"\nDistance of the best tour: {best_distance}")
```

After running this program, we have the following outcome:



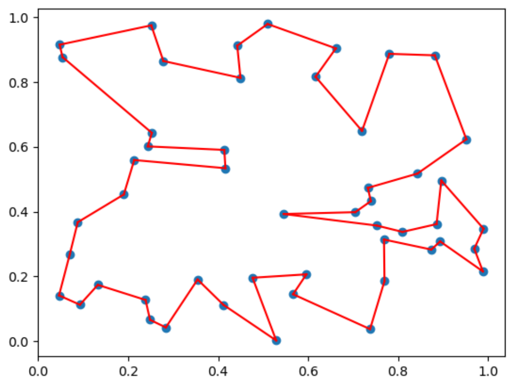
```
Best tour found by simulated annealing:
[26 46  3 41  2  1 25 42 50 40 28  6 17 22 24  4 36 44 11  8 12 31  5 13
  7 27 49 43 38 47 33 15 30 37 23 35 14 21 48 19 45 34  9 10 39 29 18 20
 16 32]

Distance of the best tour: 6.047161418551895
```

2.5 Solution to Question(e)

We use `'matplotlib'` to create a visualization. We plot the cities as points, and then connect them in the order specified by the best tour with red lines, forming a visual representation of the city tour.
The code is as follows:

```
# Visualization
plt.plot(cities[:, 0], cities[:, 1], 'o')
for i in range(len(best_tour)):
    plt.plot([cities[best_tour[i] - 1, 0], cities[best_tour[(i + 1) % num_cities] - 1, 0]],
             [cities[best_tour[i] - 1, 1], cities[best_tour[(i + 1) % num_cities] - 1, 1]], 'r-')

plt.show()
```

After running this program, we have the following outcome:



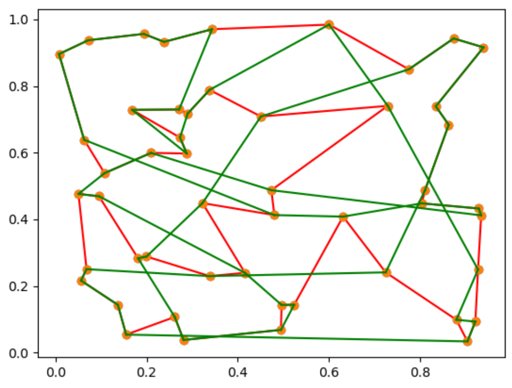
From the outcome, we can find that this tour doesn't have edges that intersect. Therefore, the outcome is an optimal tour.

2.6 Solution to Question(f)

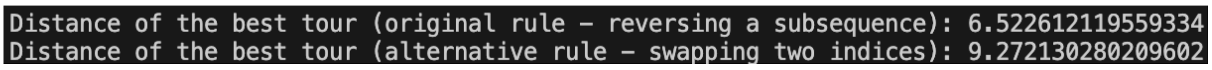
The key modification is in the `'generate_candidate'` function. Instead of reversing a subsequence of cities, we now randomly select two indices i and j, and we swap the positions of the cities at these indices in the `'candidate tour'`. This creates a different proposal rule where you swap two cities while keeping all other cities intact.
The code is as follows:

```python
# Tour perturbation (alternative rule: swapping two indices)
def generate_candidate_swap(tour):
    n = len(tour)
    i, j = np.random.choice(range(n), 2, replace=False)
    candidate = tour.copy()
    candidate[i], candidate[j] = candidate[j], candidate[i]
    return candidate
```

After running this modified program, we have the following outcome:



The figure means that the new proposal rule also obtains an optimal tour. Then we have two distance of best tour from the reversing and swapping:



```
Distance of the best tour (original rule - reversing a subsequence): 6.522612119559334
Distance of the best tour (alternative rule - swapping two indices): 9.272130280209602
```

Comparing with the two proposal rules, we can find that reversal finds slightly shorter tours. However, after several trials, we can find that both methods work well to explore the search space and converge to good solutions.

References

[1] Facility location problem. https://colab.research.google.com/drive/1YWJ8MT8t9-9b0OZKEkrchMfu0YLZ4zJv?usp=sharing.