Location-based Assignment Report

CS164 - Fall 2019

Team 4: Viet Hoang Tran Duong, Oscar Flores, Vy Tran

I. INTRODUCTION

Traveling Salesman Problem (TSP) is an optimization problem to find the minimum time or distance taken to travel between chosen locations. In this assignment, we will solve a TSP using integer programming then verify the optimal solution in Berlin.

1. **Engelbecken**: A really cool artificial lake close to us, which has swans and space to walk and take a break. Maybe not the most popular destination of Berlin, but it's a popular place for Minervans to walk and relax, a must for our list.
2. **St. Michael Kirche**: Unusual big church. The church has an important history because the wall was in front of it, so that one day people simply couldn't go to church anymore. It was since then restored and now serves also as an art gallery. Walking around it has several photos of how it looked when the wall was in place, as well as information on how it was almost destroyed for it. A hidden gem in the city.
3. **Berliner Dom**: Main Cathedral of Berlin, an icon of the architecture of the Neo-Renaissance movement of the 19th century. This is a must for anyone to appreciate the paintings and architecture, as well as the Catholic images found inside of it.
4. **AlexanderPlatz**: The modern center of Berlin. You can find here most shops and big malls, as well as a seasonal Christmas Market and the Berlin Tower. This place is one of the most visited places in the entire country.
5. **East Side Gallery**: Remaining pieces of the Berlin wall. Now it serves as a massive tourist attraction of people who want to admire the thought-provoking murals painted over the wall. One of the most famous graffiti paintings, titled "My God, Help Me to Survive this Deadly Love" is usually filled with tourists and street games/gamblers.
6. **Mercedes-Benz Arena**: Home of the Eisbären Berlin, the local Ice-Hockey team, and the Alba Berlin, the local basketball team, which are amongst the best teams in Germany and in the European League. The massive structure near the wall make it a quick stop for tourists, or for people visiting any event taking place there.

The table below is the timing, in minutes, between different locations. It was built by checking Google Maps for the shortest time from point A to B using a bike.

II. PROBLEM FORMULATION

1) Mathematical models

Let c be the cost matrix and x a matrix indicating the next destination to travel to. The time taken to travel from destination i to destination j is c_{ij} and the decision to travel from destination i to j is x_{ij}. Because x is a decision matrix, it has boolean values. $x_{ij} = 1$ means we will travel from i to j and $x_{ij} = 0$ otherwise. Let $V = \{1, 2, \ldots, n\}$ be the set of location names (vertex set, name of destinations) and n is the number of locations.

The objective function is:

$$\text{Minimize } \sum_{i=n}^{n} \sum_{i=n}^{n} c_{ij}x_{ij}$$

Subject to:

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \in V \qquad \textbf{(1)}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in V \qquad \textbf{(2)}$$

$$x_{ij} \in \{0, 1\} \quad \forall i,j \in V \qquad \textbf{(3)}$$

The first two constraints are to ensure that we only visit and leave a place exactly once. The third constraint ensures each element of x is a boolean value since x is a decision matrix.

2) Subtour elimination via lazy addition of constraints

In this approach, we initially solved a relaxed integer programming problem. We wrote a function in Python to identify if there is subtour in the optimal solution. If the tour does not include all destinations, we gradually added more constraints. Call the subtour S:

$$\sum_{i}\sum_{j} x_{ij} \geq 1 \quad \forall i \in S,\ j \notin S \quad \text{with } S\text{: the subtour} \qquad \textbf{(4)}$$

This constraint force a connection between the two subtours in our optimal solution. After adding this constraint, we solved the optimization problem again and get a new optima solution. We repeated the process until all subtour has been eliminated.

3) Subtour elimination via Miller-Tucker-Zemlin (MTZ) algorithm

With c, x defined as above. The MTZ algorithm adds a variable U_i, which defines the order in which vertex i is visited. Condition (6) requires the index to be from 1 to $n-1$. Condition (5) creates node potentials for all but one predetermined node (node 1), and use these to ensure that there are no closed paths (loop) in the solution, which don't include node 1. Condition (5) eliminates all possible subtours (because if there is a path from i to $j \Rightarrow x_{ij} = 1$. From (5), we then have $U_i - U_j \leq -1 \Leftrightarrow U_i + 1 \leq U_j$: forcing the index of j to be immediately after i) . This is expressed by adding the following constraints to the problem described in part 2.1.

$$U_i - U_j + (n-1)x_{ij} \leq n-2 \quad \forall i,j \in V \setminus \{1\} \qquad \textbf{(5)}$$

$$1 \leq U_i \leq n-1 \ \forall i \in V \setminus \{1\} \qquad \textbf{(6)}$$

4) Desrochers-Laporte (DL) formulation: the stronger form of MTX constraints.

The MTZ formulation has $O(n^2)$ variables and $O(n^2)$ constraints, which is compact. However, its linear programming relaxation from the original constraints (condition (5)) is shown by Padberg & Sung (1991) to yields an extremely weak lower bound. To strengthen the MTZ constraints, Desrochers-Laporte proposes a stronger form:

Constrain (5) becomes:

$$U_i - U_j + (n-1)x_{ij} + (n-3)x_{ji} \leq n-2 \quad \forall i,j \in V \setminus \{1\} \qquad \textbf{(7)}$$

III. OPTIMAL SOLUTION VERIFICATION

Using three different approaches above, we obtained the same optimal path:

1. Engelbecken → 2. St. Michael Kirche → 5. East Side Gallery → 6. Mercedes-Benz Arena → 4. AlexanderPlatz → 3. Berliner Dom, which cost 40 minutes in total.

We also conducts the path in real-life to validate the results:

It took: 4 - 15 - 11 - 13 - 2: 45 minutes in total, which is very optimal and close to the predicted time by Google Maps. We took a bit more time because of the traffic and the time it took to wait for traffic lights to let us go.

Photos are attached in the appendix 4.

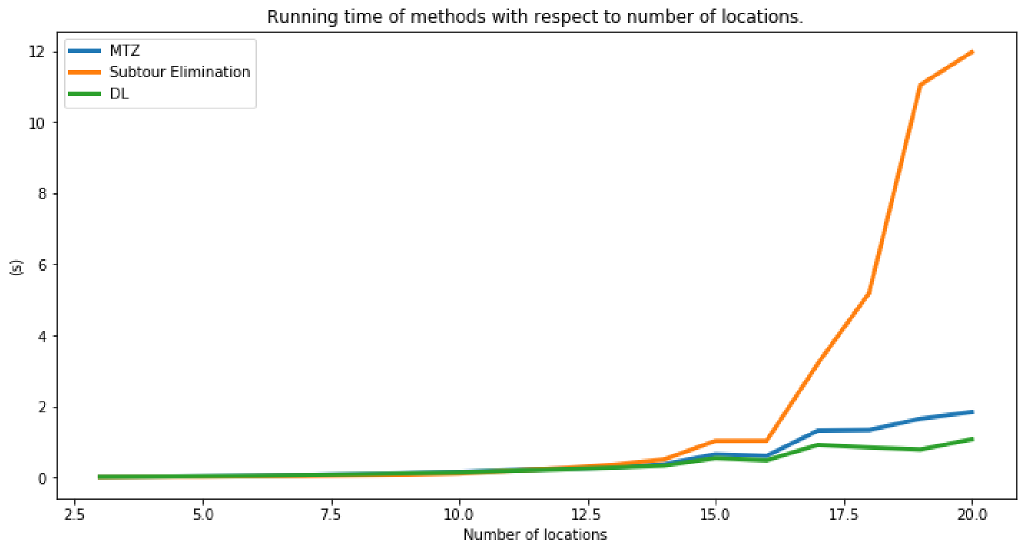
We also conduct another real test case for 6 different locations, which is in appendix 3.

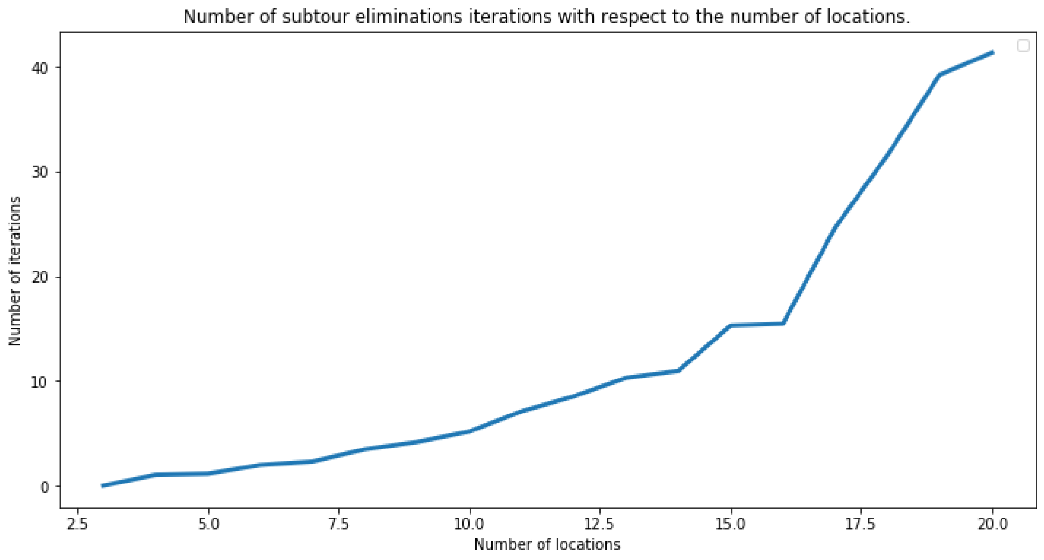
IV. METHOD EVALUATION

All these methods achieve optimality. However, depending on the size of the cost matrix (the number of locations), the running time of these methods varies. To evaluate the methods, for each location size from 3 to 20, we generate 30 random symmetry cost matrix and record the running time for each method.

As expected, the DL with the strongest constraints works the fastest on average as its feasible domain is smaller. MTZ with a weaker constraint compared to DL works slightly slower. The sub tour elimination method has the least number of constraints, but it has to run iteratively to remove sub tours, lead to having the slowest performance, and really sensitive to the input (as some cost matrix can lead to more feasible sub tours than others).

Overall, with a large number of locations, DL performs better than MTZ, and sub tour elimination has the worst performance. The y-axis is the running time in seconds





V. CONCLUSION

We solve the TSP problem using 3 different methods and mixed-integer linear programming. We create multiple test cases to track the performances of different methods, which yield that DL approach is the fastest, while the subtour elimination performs the worst, especially when the number of locations increase. We also conduct real-life test case by following the optimal path suggested by all three approaches (which all yield the same answer).

VI. APPENDICES

Contributions

Vy implemented the problem and the subtour elimination via lazy addition of constraints in cvxpy. Hoang and Oscar implemented the MTZ and DL algorithm and verified the optimal solution. All group members contributed to writing the final report.

Appendix 1: Table of distances to and from locations:

	From 1	From 2	From 3	From 4	From 5	From 6
To 1	N/A	4	11	9	12	11
To 2	3	N/A	10	7	9	8
To 3	13	10	N/A	4	16	14
To 4	10	7	4	N/A	11	10
TO 5	11	9	15	10	N/A	3
To 6	12	10	16	12	3	N/A

The diagonal means travelling from 1 location to itself, we assign them N/A. In the matrix for programming, to avoid null values or being stuck in 1 location, we assign the values for the diagonal of the matrix to be extensively big (diagonal values = 1000).

Appendix 2: The code:

- **TSP solution (subtour elimination, MTZ, and DL)**
  https://colab.research.google.com/drive/1G1vsF3wqIpXxupCiPKNA-XiDqLVX-tCu
- **Method evaluation**
  https://colab.research.google.com/drive/125ei2thGju7Kdu1ByZ8Wx8t1dgBoyLBH

Appendix 3: Another test case on 6 locations in Berlin

1. Alexanderplatz: The modern center of Berlin. You can find here most shops and big malls, as well as a seasonal Christmas Market and the Berlin Tower. This place is a must for anyone in Berlin.

2. Potsdamer Platz: Important public square, located near the center of Berlin, close to the Reichstag Building and the Brandenburg Gate. People tend to go there to admire the tall skyscrapers and the Sony Center, which is a mall with very unique architecture. For the holiday season, it has a Christmas market and an ice-slide next to it, which makes it good as family attraction.
3. Charlottenburg palace: Built at the 17th century, this place is known for its museum and its gardens. Located a little far from the center, the palace calls for people to go to the district of Charlottenburg, which proudly decorated with a Christmas Market its gardens.
4. Brandenburg Gate: The icon of Berlin, symbol of the fall of the wall and the reunification of Germany. It's located at the end of what is arguably the most important street in the country, which is a straight line from Alexanderplatz. Behind the gate, you can find the Tier Gardens, and the Reichstag Building.
5. Berlin Cathedral: Main Cathedral of Berlin, an icon of the architecture of the Neo-Renaissance movement of the 19th century. This is a must for anyone to appreciate the paintings and architecture, as well as the Catholic images found inside of it.
6. Reichstag Building: Government building. It has been a strong symbol of Germany, and has suffered as such. The building was set on fire after WW2, and was rebuilt after the reunification. Now it's the second place that tourists visit the most in Germany.

	From 1	From 2	From 3	From 4	From 5	From 6
To 1	N/A	15	34	13	6	14
To 2	18	N/A	27	7	14	13
To 3	35	25	N/A	33	39	36
To 4	15	7	33	N/A	11	6
TO 5	7	14	41	10	N/A	11
To 6	17	14	35	5	13	N/A

Table 1. Time cost matrix. Information was estimated by Google Maps, the time it takes to go from one place to the other at around 4pm, using public transportation.

Optimal time: 91 minutes:

1 (Alexanderplatz) → 3 (Charlottenburg palace) → 2 (Potsdamer Platz) → 4 (Brandenburg Gate) → 6 (Reichstag Building) → 5 (Berlin Cathedral).

Appendix 4: Photos:

--- The End ---

References:

Letchford, A. D., Nasiri, S. O., & Theis, D. undefined. (2012). Compact formulations of the Steiner Traveling Salesman. Retrieved from https://www.sciencedirect.com/science/article/pii/S037722171300091X.

Padberg, M., & Sung, T.-Y. (1991). An analytical comparison of different formulations of the travelling salesman problem. Retrieved from https://link.springer.com/article/10.1007/BF01582894.

```python
import cvxpy as cvx
import numpy as np
import time
```

```python
def subtour(X):
    """Function to identify nodes that create a subtour
    input: Boolean matrix indicating whether the we go from one destination to another
    output: a list of nodes that make a subtour
    """
    # The number of nodes we have
    N = X.shape[0]
    subtour = [0] # List of nodes in subtour, starting with node 0
    while True:
        for i in range(N):
            if X[subtour[-1],i] == 1:
                # Add the node into our list of subtour
                if i not in subtour:
                    subtour.append(i)
                else:
                    return subtour
```

```python
# Define the problem in cvxpy
N = 6
# Cost matrix NxN
np.random.seed(1)
C1 = np.matrix([[1000, 15, 34, 13,  6, 14],
               [18, 1000, 27,  7, 14, 13],
               [35, 25, 1000, 33, 39, 36],
               [15,  7, 33, 1000, 11,  6],
               [ 7, 14, 41, 10, 1000, 11],
               [17, 14, 35,  5, 13, 1000]])

C2 = np.array([[1000,    4,   11,    9,   12,   11],
              [   3, 1000,   10,    7,    9,    8],
              [  13,   10, 1000,    4,   16,   14],
              [  10,    7,    4, 1000,   11,   10],
              [   11,    9,   15,   10, 1000,    3]
              [   11,    9,   15,   10, 1000,    5],
              [   12,   10,   16,   12,    3, 1000]])
```

The Miller -Tucker-Zemlin formulation:

```
# The Miller -Tucker-Zemlin formulation:

%%time

def lazy(N, C):
  start = time.time()
  x = cvx.Variable(shape=(N,N),boolean=True)

  # Come-from constraint, we must come from only 1 destination
  # Go-to constraint, we must go to exactly 1 destination for the next step
  ones = np.ones(N)
  constraints = [cvx.sum(x,axis=0)==ones]
  constraints.extend([cvx.sum(x,axis=1)==np.transpose(ones)])

  # location indexing variable: U[i]: the order in which vertex i is visited
  U = cvx.Variable(N)

  # MTZ sobtour eleimination constraint
  # Condition (5) in the document creates node potentials for all but one
  ## predetermined node (node 1 - in Python: node[0]),
  # #and use these to ensure that there are no closed paths (loop) in the solution,
  ## which don’t include node 1. Condition (5) eliminates all possible subtours.
  for i in range(1, N):
    for j in range(1,N):
      constraints.extend([U[i] - U[j] + (N-1)*x[i,j] <= N-2])

  # 1 <= U[i] <= n-1 for all i = 2,...,n -> 1, ..., n-1 in Python
  for i in range(1,N):
    constraints.extend([U[i] >= 1])
    constraints.extend([U[i] <= N-1])

  # objective function
  obj = cvx.Minimize(sum(C[i,:]*x[:,i] for i in range(N)))

  # define problem
  prob = cvx.Problem(obj,constraints)

  sol = prob.solve(cvx.GLPK_MI)
  print("Status:", prob.status)
  print("Optimal time:", sol)
  subt = subtour(x.value)
  print(subt)
  path = ""
  for i in range(N):
    path += "Location " + str(subt.index(i)+1) + " -> "
  print(path[:-3])
  run_time = time.time() - start
  print("Time taken to run:", run_time, "(s)")
  print('-----------------------------------\n')

print("Result for matrix 1:")
lazy(6, C1)
print("Result for matrix 2:")
lazy(6, C2)
```

Lazy approach

```
# Lazy approach
# Define the problem in cvxpy

%%time

def MTZ(N, C):
  start = time.time()
  x = cvx.Variable(shape=(N,N),boolean=True)
  # Come-from constraint, we must come from only 1 destination
  # Go-to constraint, we must go to exactly 1 destination for the next step
  constraints = [cvx.sum(x,axis=0) <= 1, cvx.sum(x,axis=0) >= 1,
                 cvx.sum(x,axis=1)==1, cvx.sum(x)== N]
  #obj = cvx.Minimize(sum(C[i,:]*x[:,i] for i in range(N)))
  obj = cvx.Minimize(np.sum(np.array([C[i,:]*x[:,i] for i in range(N)])))
  #prob = cvx.Problem(obj,constraints)
  prob = cvx.Problem(objective=obj, constraints=constraints)
  # Solve the initial problem without caring about the subtour
  sol = prob.solve(cvx.GLPK_MI)
  # Lazily adding in the constraint as we solve the relaxed problem
  while True:
      subt = subtour(x.value)
      if len(subt) == N: # The tour include all the nodes, no subtour
          print("Time travelled: ",sol)
          print("Optimal path: ",subt)
          path = ""
          for i in range(N):
            path += "Location " + str(subt.index(i)+1) + " -> "
          print(path[:-3])
          run_time = time.time() - start
          print("Time taken to run:", run_time, "(s)")
          print('-----------------------------------\n')
          break
      else:
          not_in_subt = [i for i in range(N) if i not in subt]
          constraints.extend([sum(x[i,j] for i in subt for j in not_in_subt) >= 1])
          prob = cvx.Problem(obj,constraints)
          sol = prob.solve(cvx.GLPK_MI)

print("Result for matrix 1:")
MTZ(6, C1)
print("Result for matrix 2:")
MTZ(6, C2)
```

The Desrochers-Laporte formulation:

```
# The Desrochers-Laporte formulation:

%%time

def DL(N, C):
  start = time.time()
  x = cvx.Variable(shape=(N,N),boolean=True)

  # Come-from constraint, we must come from only 1 destination
  # Go-to constraint, we must go to exactly 1 destination for the next step
  ones = np.ones(N)
  constraints = [cvx.sum(x,axis=0)==1]
  constraints.extend([cvx.sum(x,axis=1)==1])

  # location indexing variable: U[i]: the order in which vertex i is visited
  U = cvx.Variable(N)

  # DL formulation: stronger form of MTZ
  for i in range(1, N):
    for j in range(1,N):
      if i != j:
        constraints.extend([U[i] - U[j] + (N-1)*x[i,j] + (N-3)*x[j,i] <= N-2])

  # 1 <= U[i] <= n-1 for all i = 2,...,n -> 1, ..., n-1 in Python
  for i in range(1,N):
    constraints.extend([U[i] >= 1])
    constraints.extend([U[i] <= N-1])

  # objective function
  obj = cvx.Minimize(sum(C[i,:]*x[:,i] for i in range(N)))

  # define problem
  prob = cvx.Problem(obj,constraints)

  sol = prob.solve(cvx.GLPK_MI)
  print("Status:", prob.status)
  print("Optimal time:", sol)
  subt = subtour(x.value)
  print(subt)
  path = ""
  for i in range(N):
    path += "Location " + str(subt.index(i)+1) + " -> "
  print(path[:-3])
  run_time = time.time() - start
  print("Time taken to run:", run_time, "(s)")
  print('-----------------------------------\n')


print("Result for matrix 1:")
```

```
DL(6, C1)
print("Result for matrix 2:")
DL(6, C2)
```

```
import cvxpy as cvx
import numpy as np
import time
import matplotlib.pyplot as plt

np.random.seed(2019)
```

```
def subtour(X):
    """Function to identify nodes that create a subtour
    input: Boolean matrix indicating whether the we go from one destination to another
    output: a list of nodes that make a subtour
    """
    # The number of nodes we have
    N = X.shape[0]
    subtour = [0] # List of nodes in subtour, starting with node 0
    while True:
        for i in range(N):
            if X[subtour[-1],i] == 1:
                # Add the node into our list of subtour
                if i not in subtour:
                    subtour.append(i)
                else:
                    return subtour
```

```
  # The Miller -Tucker-Zemlin formulation:

def MTZ(N, C):
  start = time.time()
  x = cvx.Variable(shape=(N,N),boolean=True)

  # Come-from constraint, we must come from only 1 destination
  # Go-to constraint, we must go to exactly 1 destination for the next step
  ones = np.ones(N)
  constraints = [cvx.sum(x,axis=0)==ones]
  constraints.extend([cvx.sum(x,axis=1)==np.transpose(ones)])

  # location indexing variable: U[i]: the order in which vertex i is visited
  # location indexing variable: U[i]: the order in which vertex i is visited
  U = cvx.Variable(N)

  # MTZ sobtour eleimination constraint
  # Condition (5) in the document creates node potentials for all but one
  ## predetermined node (node 1 - in Python: node[0]),
  # #and use these to ensure that there are no closed paths (loop) in the solution,
  ## which don’t include node 1. Condition (5) eliminates all possible subtours.
  for i in range(1, N):
    for j in range(1,N):
      constraints.extend([U[i] - U[j] + (N-1)*x[i,j] <= N-2])

  # 1 <= U[i] <= n-1 for all i = 2,...,n -> 1, ..., n-1 in Python
  for i in range(1,N):
    constraints.extend([U[i] >= 1])
    constraints.extend([U[i] <= N-1])

  # objective function
  obj = cvx.Minimize(sum(C[i,:]*x[:,i] for i in range(N)))

  # define problem
  prob = cvx.Problem(obj,constraints)

  sol = prob.solve(cvx.GLPK_MI)
  run_time = time.time()-start
  return sol, run_time
```

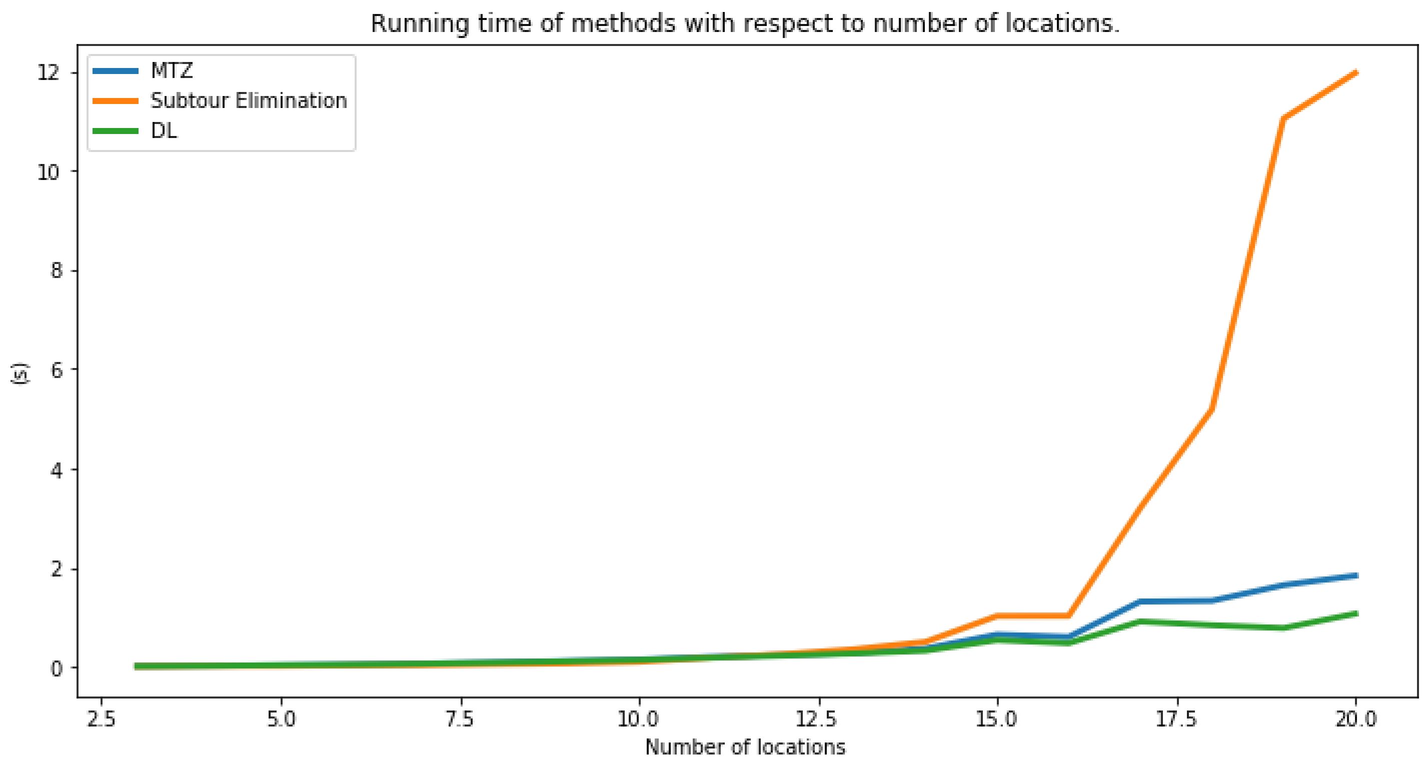
```

# Lazy approach
# Define the problem in cvxpy
def lazy(N, C):
  start = time.time()
  x = cvx.Variable(shape=(N,N),boolean=True)
  # Come-from constraint, we must come from only 1 destination
  # Go-to constraint, we must go to exactly 1 destination for the next step
  constraints = [cvx.sum(x,axis=0) <= 1, cvx.sum(x,axis=0) >= 1,
                 cvx.sum(x,axis=1)==1, cvx.sum(x)== N]
  #obj = cvx.Minimize(sum(C[i,:]*x[:,i] for i in range(N)))
  obj = cvx.Minimize(np.sum(np.array([C[i,:]*x[:,i] for i in range(N)])))
  #prob = cvx.Problem(obj,constraints)
  prob = cvx.Problem(objective=obj, constraints=constraints)
  # Solve the initial problem without caring about the subtour
  sol = prob.solve(cvx.GLPK_MI)
  # Lazily adding in the constraint as we solve the relaxed problem
  count = 0
  while True:
      subt = subtour(x.value)
      if len(subt) == N: # The tour include all the nodes, no subtour
          run_time = time.time() - start
          return sol, run_time, count
          break
      else:
          count += 1
          not_in_subt = [i for i in range(N) if i not in subt]
          constraints.extend([sum(x[i,j] for i in subt for j in not_in_subt) >= 1])
          prob = cvx.Problem(obj,constraints)
          sol = prob.solve(cvx.GLPK_MI)
```

```
# The Desrochers-Laporte formulation:
def DL(N,C):
  start = time.time()
  x = cvx.Variable(shape=(N,N),boolean=True)

  # Come-from constraint, we must come from only 1 destination
  # Go-to constraint, we must go to exactly 1 destination for the next step
  ones = np.ones(N)
  constraints = [cvx.sum(x,axis=0)==1]
  constraints.extend([cvx.sum(x,axis=1)==1])

  # location indexing variable: U[i]: the order in which vertex i is visited
  U = cvx.Variable(N)

  # DL formulation: stronger form of MTZ
  for i in range(1, N):
    for j in range(1,N):
      if i != j:
        constraints.extend([U[i] - U[j] + (N-1)*x[i,j] + (N-3)*x[j,i] <= N-2])
  # 1 <= U[i] <= n-1 for all i = 2,...,n -> 1, ..., n-1 in Python
  for i in range(1,N):
    constraints.extend([U[i] >= 1])
    constraints.extend([U[i] <= N-1])

  # objective function
  obj = cvx.Minimize(sum(C[i,:]*x[:,i] for i in range(N)))

  # define problem
  prob = cvx.Problem(obj,constraints)

  sol = prob.solve(cvx.GLPK_MI)
  run_time = time.time() - start
  return sol, run_time
```

```
sol_MTZ = []
sol_laz = []
sol_DLa = []
time_MTZ = []
time_laz = []
time_DLa = []
count_laz = []
for i in range(3, 21):
  print("Current size:", i)
  N = i
  cur_time_MTZ = []
  cur_time_laz = []
  cur_time_DLa = []
  cur_count_laz = []
  cur_sol_MTZ = []
  cur_sol_laz = []
  cur_sol_DLa = []
  trial = 30
  for j in range(trial):
    C = np.random.randint(50, size=(N,N))
    np.fill_diagonal(C, 1000)
    C = (C + C.T)//2
    sol_1, run_time_1 = MTZ(N, C)
    sol_2, run_time_2, count = lazy(N, C)
    sol_3, run_time_3 = DL(N, C)

    cur_time_MTZ.append(run_time_1)
    cur_time_laz.append(run_time_2)
    cur_time_DLa.append(run_time_3)
    cur_count_laz.append(count)
    cur_sol_MTZ.append(sol_1)
    cur_sol_laz.append(sol_2)
    cur_sol_DLa.append(sol_3)
  time_MTZ.append(np.mean(cur_time_MTZ))
  time_laz.append(np.mean(cur_time_laz))
  time_DLa.append(np.mean(cur_time_DLa))
  count_laz.append(np.mean(cur_count_laz))
  sol_MTZ.append(cur_sol_MTZ)
  sol_laz.append(cur_sol_laz)
  sol_DLa.append(cur_sol_DLa)
```

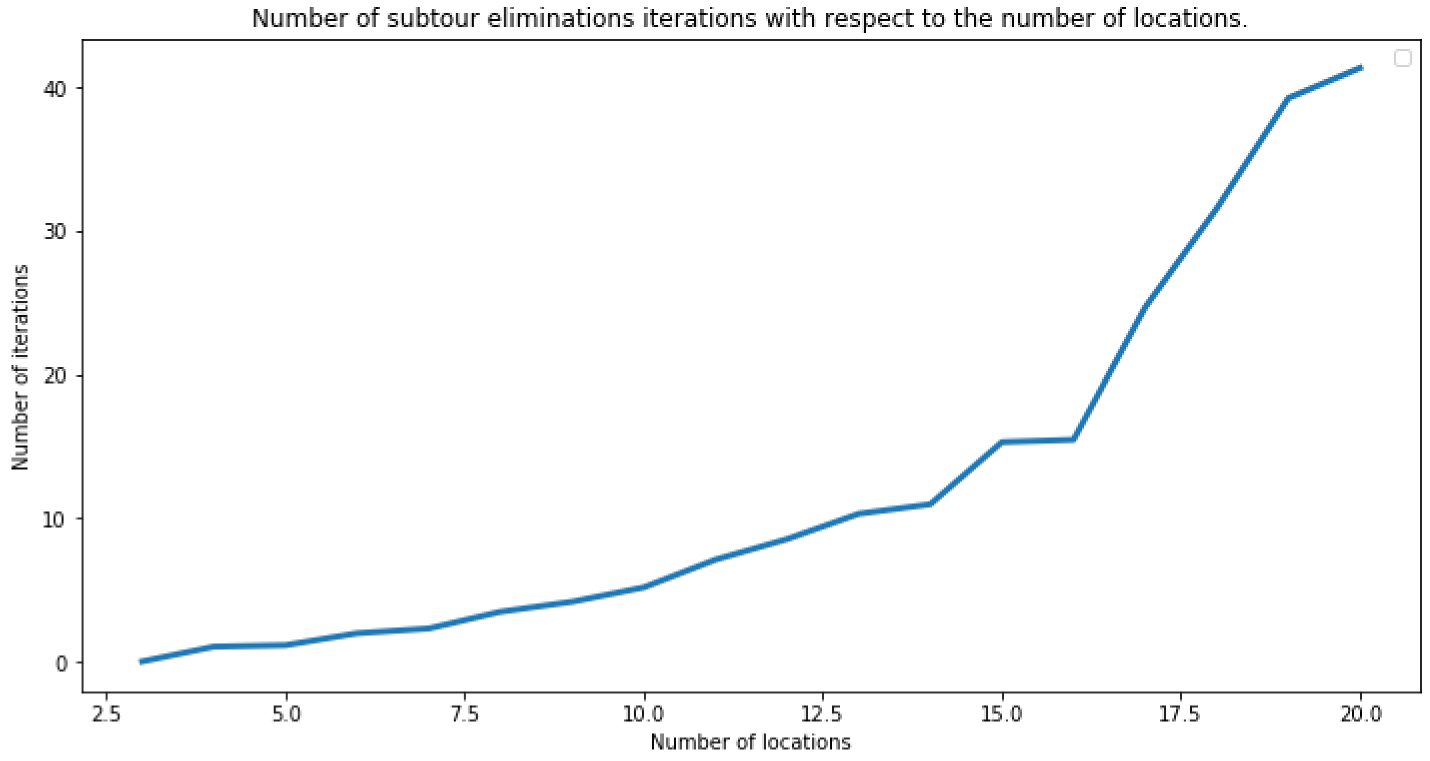
```
Current size: 3
Current size: 4
Current size: 5
Current size: 6
Current size: 7
Current size: 8
Current size: 9
Current size: 10
Current size: 11
Current size: 12
Current size: 13
Current size: 14
Current size: 15
Current size: 16
Current size: 17
Current size: 18
Current size: 19
Current size: 20
```

```
plt.figure(figsize = (12,6))
x = list(range(3,21))
plt.plot(x, time_MTZ, label = "MTZ", linewidth = 3)
plt.plot(x, time_laz, label = "Subtour Elimination", linewidth = 3)
plt.plot(x, time_DLa, label = "DL", linewidth = 3)
plt.title("Running time of methods with respect to number of locations.")
plt.ylabel("(s)")
plt.xlabel("Number of locations")
plt.legend()
plt.show()
```



```
plt.figure(figsize = (12,6))
x = list(range(3,21))
plt.plot(x, count_laz, linewidth = 3)
plt.title("Number of subtour eliminations iterations with respect to the number of locations.")
plt.ylabel("Number of iterations")
plt.xlabel("Number of locations")
plt.show()
```

```
No handles with labels found to put in legend.
```



```
1
```