Advanced Optimization Final Project: Traveling Salesman Problem

Ian Sinclair
ENGR 4622-1

November 17, 2022
Professor Joshua Betz

Abstract

The Traveling Salesman Problem seeks to find the faster tour around n points, or traditionally, is the fastest way a salesman can travel to n different cities and then return home. This problem has been the focus of research in many fields; however, one of its best solutions lies in linear programming / optimization. For an undirected graph, G_n, with n nodes, this corresponds to the shortest Hamiltonian Circuit incident with every node. Or in general, is a cycle that visits every node exactly once. Here the traveling salesman problem is used to plan a route around n cities, real world routing data is taken corresponding to the driving time between cities in minutes.

1 Introduction

Within the problem, a network optimization model is used to find the shortest cost Hamiltonian Circuit around the n cities. And so first we define how the underlying graph structure is established, then cover the functionality and purpose of the network model in relation to integer programming.

1.1 Graph Terminology

Within this report, let G be a graph, over n nodes, where each node, v, corresponds to the location (longitude/latitude) of a single city. Then, define an arc (i, j) as a route that exists between city i to city j, $i, j \in N$. Generally, in a TSP it is possible to travel from any city to any other city and so G is a complete graph, $G = K_n$. Next, define the distance metric between cities as the time in minutes is takes to drive from one city to another. And let $C_{i,j}$ reference this time from city i to city j.
Finally, a Hamiltonian Circuit is a cycle subgraph of G that spans N and visits every node exactly once. Unfortunately, in a complete graph K_n the number of possible Hamiltonian cycles, $|H(K_n)|$ is quite large,

$$|H(K_n)| = \frac{1}{2}(n-1)!$$

Which scales exponentially with n. This makes finding a polynomial time solution extremely difficult. And actually renders TSP among NP-complete problems. This means that a brute force solution for a graph with as few as 30 cities could take year. Critically; however, the problem can be cast as a network optimization problem similar to a shortest path problem which can potentially be solved much faster.

2 Problem Formulation

Correspondingly, we now examine how the problem can be converted into a network optimization task. And so consider the following, > Here we cover the basic formulation for a minimum tour problem.

2.0.1 Parameters

> Let I for a list of n cities.

> Let $C_{i,j}$ be the cost of moving from city i to city j (in travel time by road route).

2.0.2 Variables :

Let $X_{i,j} \in \{0,1\}$ be a binary variable determining if arc $C_{i,j}$ is used in the Hamiltonian circuit.

$$X_{i,j} = \begin{cases} 1 & \text{City } i \text{ connects to City } j \text{ is in the hamiltonian circuit} \\ 0 & \text{Otherwise} \end{cases}$$

Then the goal is to find the least cost way to visit every city starting and returning to city 1.

2.0.3 Objective Function

>Let the cost of any circuit be,

$$\min \sum_{i \in I} \sum_{j \in I} C_{i,j} X_{i,j}$$

2.0.4 Local Constraints

> Subject to the constraints,

» Entering every city only once:

$$\sum_{i \in I} X_{i,j} = 1, \qquad \forall j \in I$$

» Leaving every city only once:

$$\sum_{k \in I} X_{j,k} = 1, \qquad \forall j \in I$$

» Don't stay at one city:

$$X_{i,i} = 0, \qquad \forall i \in I$$

This method focuses on local constraints relevant to single cities. Or rather the only visiting/ leaving every city once. However, accounting for more global problems to optimality can be more difficult.

3 Sub Tours Problem

The local constraints above are not restrictive enough to eliminate all solutions that do not generate Hamiltonian circuits. In particular, it is still possible to get sub-tours that contribute to the final solution.

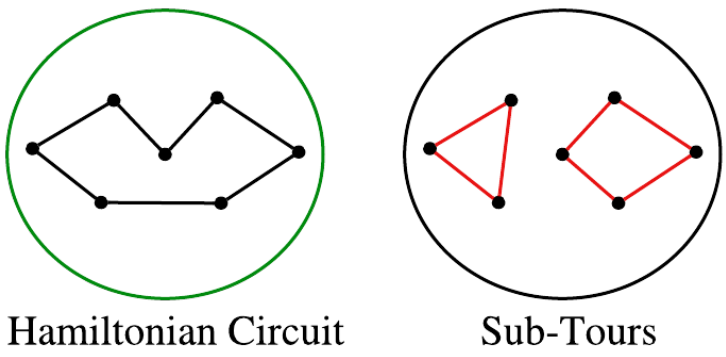

Figure 1: Concept art of two valid tours under local constraints. The Hamiltonian cycle on the left, and sub tours on the right.

Note, in the above figure, the image on the left shows a desired Hamiltonian Circuit; however, by moving two arcs the tour can be separated into two distinct cycles without violating any local constraints. And as a result, more constraints need to be added to prevent sub tours. One of the best methods to eliminate sub-tours is using the Miller-Tucker-Zemlin (MTZ) approach. Which adds a new time variable t, indicating at which point each city is visited in sequence.

4 Miller-Tucker-Zemlin (MTZ)

In MTZ, a new model is extended from the original local constraints that better accounts for globally related issues like sub tours. And so assuming the same model as for local constraints add the following.

4.0.1 New Parameter

$>$ First fix some ordering on the list of cities I, and let $V = \{1, 2, \cdots, n\}$ be that order. So for some $i \in V$, $i = 1, 2, \cdots, n$ for n cities.
Then, for $i \in V$ let i denote either an integer, or a city $i \in I$.

4.0.2 Variable

$>$ Next define the time variable, t_i by the time in sequence that city i is visited. $>$ As a result,

$$\text{if } X_{i,j} = 1, \quad \text{then} \quad t_j \geq t_i + 1, \qquad i, j \neq 1$$

$>$ This abuses notation a bit, but in general the i, j associated with $X_{i,j}$ are city names. And, the i, j associated with t_j, t_i are integers associated with the ordering on I.
Notice intuitively, this works by restricting a sequence on the order we visit each city. Or rather if we visit cities in the order, $i_1, i_2, \cdots, i_{n-1}$ then,

$$t_{i_1} = 1, t_{i_2} = 2, t_{i_3} = 3, \cdots, t_{i_{n-1}} = n - 1,$$

Then, suppose a sub tour exists, such that, $\exists a, b, c \in I$ where $X_{a,b} = X_{b,c} = X_{c,a} = 1$ (cycle C_3). and none of the cities a, b, c are the originating city. $(a, b, c \neq 1)$.
Then, the time sequence constraints impose,

$$\begin{cases} t_b \geq t_a + 1 \\ t_c \geq t_b + 1 \\ t_a \geq t_c + 1 \end{cases}$$

$>$ Which has no solution. Similar logic can be applied to any sub-tour, (C_k), and so this method effectively eliminates sub-tours from the optimal solution.
Lastly, consider how to encode the 'if' statement,

$$\text{if } X_{i,j} = 1, \quad \text{then} \quad t_j \geq t_i + 1, \qquad i, j \neq 1$$

As a constraint

$>$ In particular, consider,

$$\begin{cases} t_j \geq t_i + 1 & X_{i,j} = 1 \\ t_j \text{ Unrestricted} & X_{i,j} = 0 \end{cases}$$

And so consider the following constraint,

$$t_j \geq t_i + 1 - M(1 - X_{i,j})$$

For large M.
Actually because it has been shown that there exists an ordering that ensures

$$\max_{i \in V} t_i = n - 1$$

Then selecting $M = n$ is sufficient to guarantee that an optimal Hamiltonian Circuit can satisfy the constraints.

4.1 TSP MTZ Model

Reiterating the total model parameters for MTZ method.

4.1.1 Parameters

> I be a set of n cities.

> V be an ordering on I such that, $i \in I$ is either an integer i, or the $i'th$ city in the ordering of I.

> Let $C_{i,j}$ be the cost of moving from city i to city j (in travel time by road route).

4.1.2 Variables

> Let $X_{i,j} \in \{0, 1\}$ be a binary variable if we travel from city i to city j in the tour.

$$X_{i,j} = \begin{cases} 1 & \text{City } i \text{ connects to City } j \text{ is in the hamiltonian circuit} \\ 0 & \text{Otherwise} \end{cases}$$

> Let t_i be the time city i is visited in the tour. For $i \in V$ is an integer.

4.1.3 Objective Function

>Let the cost of any circuit be,

$$\min \sum_{i \in I} \sum_{j \in I} C_{i,j} X_{i,j}$$

4.1.4 Local Constraints

» Entering every city only once:

$$\sum_{i \in I} X_{i,j} = 1, \qquad \forall j \in I$$

» Leaving every city only once:

$$\sum_{k \in I} X_{j,k} = 1, \qquad \forall j \in I$$

» Don't stay at one city:

$$X_{i,i} = 0, \qquad \forall i \in I$$

4.1.5 MTZ Constraints

» Sub-tour elimination

$$t_j \geq t_i + 1 - n(1 - X_{i,j}), \qquad \forall i, j \in V, \quad j > i$$

» Circuit completeness

$$t_1 = 1$$

Within the score of the report, solutions using only local constraints are compared to solutions using MTZ method in search for the optimal shortest Hamiltonian Cycle.

5 Analysis

5.1 Cities

We specifically solve a version of the traveling salesman problem with cities,

```
1 ['Denver', 'New York', 'Houston', 'Dallas', 'Philadelphia', 'Phoenix', 'Miami', 'Cleveland', 'San
      Francisco', 'Nashville', 'Greensboro', 'Lincoln', 'Seattle']
```

From here an API called DistanceMatrix.ai is used to find the travel time duration in minutes from any city i to every other city j.
Note, that the rest of the program is adaptive and so any cities can be selected. However, the API may not be able to find a route between distant cities, in this case, the corresponding arc is neglected by getting an untractable value.

$$C_{i,j}^{neglected} = 100000 \text{ minutes}$$

This removes the arc from any optimal tour. If it looks like this affects the optimal solution, run the API request again and it will probably find a route for the neglected arc.

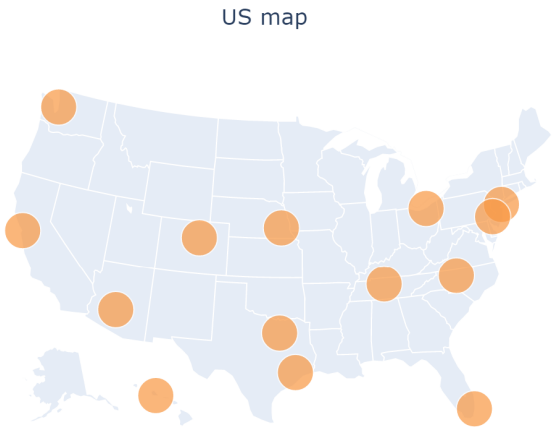

Figure 2: Display of city locations that will be used for the TSP model

After calculating the coordinates of each city, the complete K_n graph connecting any city to every other city can be constructed.

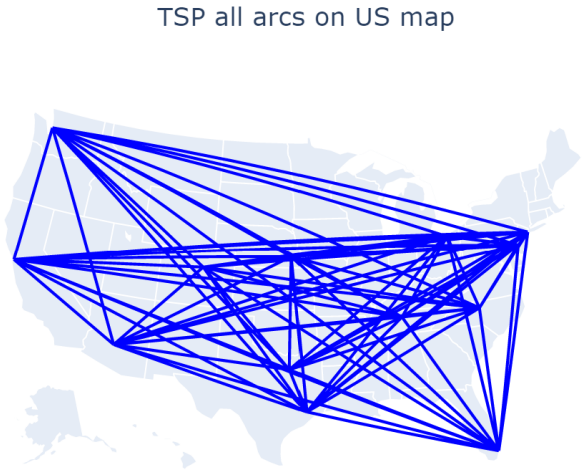

Figure 3: Complete graph network between cities.

NOTE: lines between cities are shown as linear (straight segments) however, are weighted by road travel routing information. I don't have a good way to get the geometry of a route just how long it takes to travel.

5.2 AMPL Data Generation

The next section follows methods to automatically generate .dat files in AMPL. This is necessary because the list of cities is intended to be adaptive, and so data to the linear model in AMPL must also be adaptive or easily generative. This is done by creating wrapper are standard AMPL formatting and reading in Distance Matrix information from a saved JSON from the previous section.

```
1 import json
2
3 def readJsonData(filename) :
4     with open(filename, 'r') as f :
```

```
        data = json.load(f)
    return data

def AMPLT_Result_to_path(inputFile : str) :
    city_info = []
    with open(inputFile, 'r') as f:
        city_info += f.readline()
    print(city_info)

def makeDatFile(data : dict, outfile='TSP_DATA.dat') :
    if type(data) == type(' ') :
        data = readJsonData(data)

    #  Generate SET
    datFileString = 'set ORIG := \n'
    nextSET = 'set DEST := \n'
    for city in data.keys() :
        datFileString += '\t' + city.replace(" ", "_") + ' \n'
        nextSET += '\t' + city.replace(" ", "_") + ' \n'
    datFileString += ';\n'
    nextSET += ';\n'
    datFileString += nextSET
    paramString = 'param DISTANCE := \n'
    for origin, travel_info in data.items() :
        paramString += '\t[' + str(origin.replace(" ", "_")) + ',*] '
        for dest, info in travel_info.items() :
            paramString += dest.replace(" ", "_") + ' ' + str(info['duration']) + ' '
        paramString += '\n'
    paramString += ';\n'
    datFileString += paramString

    with open(outfile, 'w', encoding='utf-8') as f:
        f.write(datFileString)
```

Which generates an AMPL .dat called '$TSP_DATA.dat$. (The contents of this file are to large to display in pdf form. However, the file is included in the project folder.)

6 Results Local Constraints

After generating the .dat data file, an AMPL solver is used to establish both local and MTZ constraints for the respective models. Beginning with just local constraints, the corresponding .mod and .run files are as follows,

```
set ORIG;
set DEST;

param DISTANCE{i in ORIG, j in DEST};


var X{i in ORIG, j in DEST} binary;

minimize tour_cost: sum{i in ORIG, j in DEST} DISTANCE[i,j]*X[i,j];

subject to Flow_IN_constraint{j in DEST}: sum{i in ORIG} X[i,j] = 1;
subject to Flow_OUT_constraint{i in ORIG}: sum{j in DEST} X[i,j] = 1;

subject to Dont_Stay_at_city{j in ORIG}: X[j,j] = 0;
```

```
reset;

model TSP_traditional.mod;
data TSP_DATA.dat;

option solver cplex;
option cplex_options 'mipdisplay 2';

solve;
option omit_zero_rows 0;
```

```
option display_lcol 10000;

display X, tour_cost;

#print {i in ORIG, j in DEST} : if X[i,j]==1 :  i,j >'TSP_traditional_TOUR_Result.
    txt';

for{i in ORIG, j in DEST: X[i,j] = 1 } {
     printf "%s %s \n", i,j >'TSP_traditional_TOUR_Result.txt';
}
```

Which has a terminal output of, Or in an easier format to read consider the image below, (the terminal output file

	Cleveland	Dallas	Denver	Greensboro	Houston	Lincoln	Miami	Nashville	New$_{York}$	Philadelphia	Phoenix	San$_{Francisco}$	Seatt
Cleveland	0	0	0	0	0	0	0	1	0	0	0	0	0
Dallas	0	0	0	0	1	0	0	0	0	0	0	0	0
Denver	0	0	0	0	0	1	0	0	0	0	0	0	0
Greensboro	0	0	0	0	0	0	1	0	0	0	0	0	0
Houston	0	1	0	0	0	0	0	0	0	0	0	0	0
Lincoln	0	0	1	0	0	0	0	0	0	0	0	0	0
Miami	0	0	0	1	0	0	0	0	0	0	0	0	0
Nashville	1	0	0	0	0	0	0	0	0	0	0	0	0
New$_{York}$	0	0	0	0	0	0	0	0	0	1	0	0	0
Philadelphia	0	0	0	0	0	0	0	0	1	0	0	0	0
Phoenix	0	0	0	0	0	0	0	0	0	0	0	0	1
San$_{Francisco}$	0	0	0	0	0	0	0	0	0	0	1	0	0
Seattle	0	0	0	0	0	0	0	0	0	0	0	1	0

Table 1: Tour chart terminal output.

is included in the project folder.)

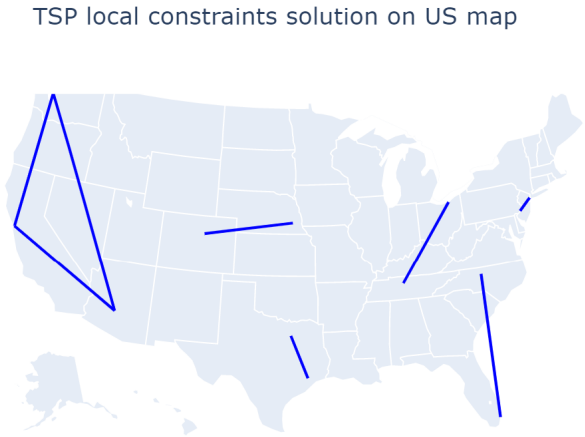

Figure 4: TSP solution with only local constraints.

Notice the tour with local constrains is dis-connected into many small segments. This is as a consequence of sub tours, or rather lack of constraints against sub-tours. And so the integer model with only local constraints is unable to find a minimum cost Hamiltonian Circuit for the candidate cities.

7 Results MTZ Constraints

Next consider the MTZ constraints which are instantiated by the following .mod and .run files,

```
set ORIG ordered;
set DEST ordered;

param DISTANCE{i in ORIG, j in DEST};
```

```
5
6
7  var X{i in ORIG, j in DEST} binary;
8  var t{1 .. card(ORIG)} >=0;
9
10 minimize tour_cost: sum{i in ORIG, j in DEST} DISTANCE[i,j]*X[i,j];
11
12 subject to Flow_IN_constraint{j in DEST}: sum{i in ORIG} X[i,j] = 1;
13 subject to Flow_OUT_constraint{i in ORIG}: sum{j in DEST} X[i,j] = 1;
14
15 subject to Dont_Stay_at_city{j in ORIG}: X[j,j] = 0;
16
17 subject to Time_Sequence_Constraint{i in 1..card(ORIG) , j in 1..card(DEST) : j !=
       1}:  t[j] >= t[i]+1-100*(1-X[member(i, ORIG),member(j, DEST)]);
18
19 subject to Initialize_Time_Sequence: t[1] = 1;
```

```
1  reset;
2
3  model TSP_MTZ.mod;
4  data TSP_DATA.dat;
5
6  option solver cplex;
7  option cplex_options 'mipdisplay 2';
8
9  solve;
10 option omit_zero_rows 0;
11 option display_lcol 10000;
12
13 display X,t, tour_cost;
14
15 print {i in 1..card(ORIG)} : member(i, ORIG), t[i] >'TSP_MTZ_TOUR_Result.txt';
```

Here additional care is taken to prevent sub tours in accordance with the MTZ method.
Ultimately resulting in the tour,

```
1  'Denver --> Lincoln --> Nashville --> Cleveland --> New_York --> Philadelphia --> Greensboro --> Miami
       --> Houston --> Dallas --> Phoenix --> San_Francisco --> Seattle --> Denver'
```

Which corresponds to the plot,

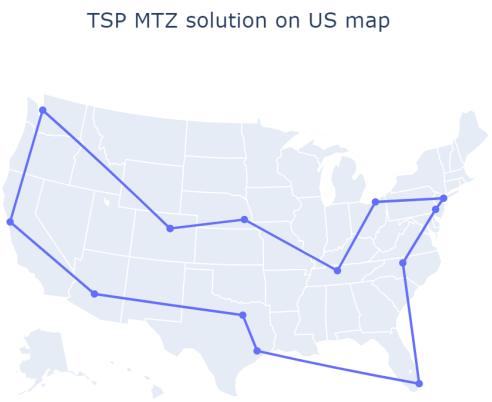

Figure 5: Plot of tour under MTZ constraints.

Here the final tour satisfies the conditions for a Hamiltonian Circuit. It is also important to note that this implementation is linear, has only 13 variables and constraints that scale linearly with N, and so simplex method is able to find minimum Hamiltonian Circuits extremely quickly (less than 1 seconds), which is significantly less than the brute force method.