Introduction

The traveling salesman problem is a complex and interesting problem to solve. Given a set of destinations, the goal is to find the shortest route that visits all the destinations and ends at the destination you started from. It is important to note that the traveling salesman problem is fundamentally different from a shortest path problem.

To explain the difference, consider the motivation of the traveling salesman problem. A person living in Oklahoma City wants to visit the top ten largest cities in Oklahoma. This person, we can assume, wants their trip to end where they live. The traveling salesman problem will create an optimal route that minimizes the cost being evaluated (in this case, distance) and loops back to the original destination. A shortest path problem would only give a one-way path to visit all destinations, likely ending far away from the original destination. This would not be very convenient or efficient, because a very long drive would be required at the end to get back home.

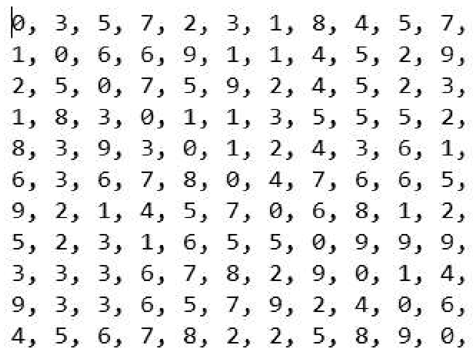
In this report, a 35-destination instance of the traveling salesman problem will be solved for Oklahoma's state parks. The results will give an optimal least-distance-traveled route/loop to visit all state parks.

Problem

The goal of this project is to solve an instance of the traveling salesman problem for all state parks in Oklahoma.

Preprocessing

A fundamental first step in solving the traveling salesman problem is creating a distance matrix for all destinations. A simple matrix is shown below.



```
0, 3, 5, 7, 2, 3, 1, 8, 4, 5, 7,
1, 0, 6, 6, 9, 1, 1, 4, 5, 2, 9,
2, 5, 0, 7, 5, 9, 2, 4, 5, 2, 3,
1, 8, 3, 0, 1, 1, 3, 5, 5, 5, 2,
8, 3, 9, 3, 0, 1, 2, 4, 3, 6, 1,
6, 3, 6, 7, 8, 0, 4, 7, 6, 6, 5,
9, 2, 1, 4, 5, 7, 0, 6, 8, 1, 2,
5, 2, 3, 1, 6, 5, 5, 0, 9, 9, 9,
3, 3, 3, 6, 7, 8, 2, 9, 0, 1, 4,
9, 3, 3, 6, 5, 7, 9, 2, 4, 0, 6,
4, 5, 6, 7, 8, 2, 2, 5, 8, 9, 0,
```

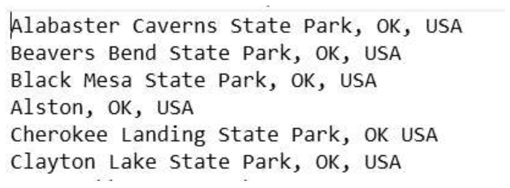
Each row represents a destination you are coming from, and each column represents a destination you are going to. Zero values represent the distance from a destination to itself, and all other values give the distance from destination *r* to destination *c*. In this example, the matrix is not symmetrical, but a distance matrix that represents real destinations on a map would be very close to symmetrical (excluding small differences for navigating one-way city streets or other anomalies).

The end goal is to create a distance matrix (numbers only) file that Python and the Gurobi solver can use. The steps to do this are presented here:

Collect Addresses

First, the addresses of each destination were collected. The geocoding provider that was used (discussed in the next section) works best when the addresses are written as landmarks, meaning either the name of the state park or the town closest to the state park was used.

All addresses/locations were written to a text file, each address/location getting its own line. Here is a sample from the text file for Oklahoma state parks:



```
Alabaster Caverns State Park, OK, USA
Beavers Bend State Park, OK, USA
Black Mesa State Park, OK, USA
Alston, OK, USA
Cherokee Landing State Park, OK USA
Clayton Lake State Park, OK, USA
```

Geocode Addresses

Nomatim is a service that can take addresses like the ones shown above and convert them into corresponding longitude and latitude pairs. This was performed on each line of the address file. Each pair of location values was then appended to a long string of pairs in Python.

Bing Maps provides a free API license to students, so their distance matrix API call was used for this project. The distance matrix call takes locations as a list of longitude and latitude pairs, and outputs a matrix of shortest-distances between all destinations. The URL for the API call was built in Python using this formula:

URLFirstHalf + longLatPairs + URLSecondHalf

After calling the API and retrieving the results, the .json text was pulled and made into a Python array object. The array was used to pull the distance values from each .json object and write to a file to create the distance matrix.

Process Distance Matrix

Before Gurobi can use this distance matrix, it must be processed again by the code.

First, a directed graph with *n* nodes was created using NetworkX. Then, a variable for each directed edge was created using Gurobi.

Python then iterated through the distance matrix file, adding each distance value to its corresponding edge variable. This gave each edge a “distance” attribute that can be accessed by the Gurobi methods.

Integer Programming Model

There are two models I used to solve the traveling salesman problem (TSP). I developed the first model on my own. This model produced an optimal solution, but had many subtours or isolated/disconnected loops within it. More details on what a subtour is specifically are included below. For the second model, I included the Miller-Tucker-Zemlin constraints to eliminate subtours.

First formulation:

Sets:

N is the set of nodes/destinations
C is the set of all combinations of edges. Combinations being non-ordered pairs. [i, j] is not repeated in the set as [j, i].

Variables:

$x_{ij} = \{1,\ 0\}\ 1 \text{ -- } \textit{the ordered edge } (i,\ j) \textit{ is used/traveled in the solution};\ 0 \text{ -- } \textit{otherwise}$
where edge (i, j) is the route from node i to node j, different from edge (j, i).

Parameters:

d_{ij} *= the road distance from node i to node j*
n = the number of nodes/destinations

$$\begin{array}{llll}
min & \sum_{i\in N}\sum_{j\in N} d_{ij}x_{ij} & & (1a) \\
s.t. & \sum_{i\in N} x_{iu} = 1 & \forall u = \{1,\ 2,\ ...,\ n\} & (1b) \\
 & \sum_{i\in N} x_{ui} = 1 & \forall u \in \{1,\ 2,\ ...,\ n\} & (1c) \\
 & x_{ii} = 0 & \forall i \in N & (1d) \\
 & x_{ij} + x_{ji} \leq 1 & \forall \{i,j\} \in C & (1e) \\
 & x_{ij} \in \{1,\ 0\} & \forall i \in N,\ \forall j \in N & (1f)
\end{array}$$

(1a) Minimize the distance between each edge traveled in the solution. Uses binary x variable to eliminate unused edges. (1b) Only one edge can be used to travel <u>to</u> a node/destination. (1c) Only one edge can be used to travel <u>from</u> a destination. (1d) Don't use edges that go from node i to itself. These are always zero, and the solver should not use them. (1e) For all unique combinations of edges, only travel one direction between those two nodes. In different words, if an edge is used/traveled on in the final solution, do not travel along its reverse. (1f) All *x* variables must be binary.

This formulation solves an incomplete TSP. Solutions generated by this model give multiple subtours instead of one full loop. A subtour is an isolated loop that is disconnected from other nodes in the model. In order to solve a true TSP with this formulation, the model must be solved multiple times. Each iteration gives a solution with multiple subtours, and the nodes contained in those loops should be collected. Then, a constraint like this should be used:

$$\sum_{i\in S}\sum_{j\in S} x_{ij} \leq L - 1$$

where S is the set of nodes contained in a subtour, and L is the length of S. This constraint says that for all edges contained in the subtour, only use L-1 of those edges, eliminating the possibility of that specific subtour appearing in the next iteration.

This can become extremely tedious, because each time constraints are written against certain subtours, other subtours are output by the next iteration of the solver. This process (recognizing subtours, writing constraints against them, and solving again) can take many iterations. The model below, using the Miller-Tucker-Zemlin constraints for eliminating subtours, solves the problem in one iteration.

Second formulation:

Sets:

N is the set of nodes/destinations

Variables:

$x_{ij} = \{1,\ 0\}\ 1 \text{ —— } \textit{the ordered edge } (i,\ j) \textit{ is used/traveled in the solution};\ 0 \text{ —— } \textit{otherwise}$

where edge (i, j) is the route from node i to node j, different from edge (j, i).

$y_i = \textit{the order placement of node i in the loop of destinations. First node visited is } 1,\ \textit{second is } 2,\ \textit{etc.}$

Parameters:

k = the number of nodes/destinations
d_{ij} = the road distance from node i to node j
n = the number of nodes/destinations

$$
\begin{array}{llll}
min & \sum\limits_{i \in N} \sum\limits_{j \in N} d_{ij} x_{ij} & & \text{(2a)} \\
s.t. & \sum\limits_{i \in N} x_{iu} = 1 & \forall u \in \{1,\ 2,\ ...,\ n\} & \text{(2b)} \\
 & \sum\limits_{i \in N} x_{ui} = 1 & \forall u \in \{1,\ 2,\ ...,\ n\} & \text{(2c)} \\
 & x_{ii} = 0 & \forall i \in N & \text{(2d)} \\
 & y_i + x_{ij} \le y_j + (n-1)(1 - x_{ij}) & \forall i \in N,\ \forall j \in \{2,\ 3,\ ...,\ n\} & \text{(2e)} \\
 & y_0 = 0 & & \text{(2f)} \\
 & x_{ij} \in \{1,\ 0\} & \forall i \in N,\ \forall j \in N & \text{(2g)} \\
 & y_i \quad integer & \forall i \in N & \text{(2h)}
\end{array}
$$

(2a) Minimize the distance between each edge traveled in the solution. Uses binary x variables to eliminate unused edges. (2b) Only one edge can be used to travel to a node/destination. (2c) Only one edge can be used to travel from a destination. (2d) Don't use edges that go from node i to itself. These are always zero, and the solver should not use them. (2e) Miller-Tucker-Zemlin constraint that says the node i you are coming from should have an order lower than the node j you are going to. If you imagine the map of nodes, the first node visited has order 1, the second 2, and so on. (2f) The order of the first node must be zero. This allows constraint (2e) to work as intended. (2g) All *x* variables must be binary. (2h) All y variables must be integers.

Since this formulation has constraints that eliminate subtours, the problem is solved in one iteration. Gurobi solve time takes exponentially longer than the first formulation.

Code

A link to the github repository is included in the appendix. There are three code files, one for preprocessing and two for the Gurobi models.

Results

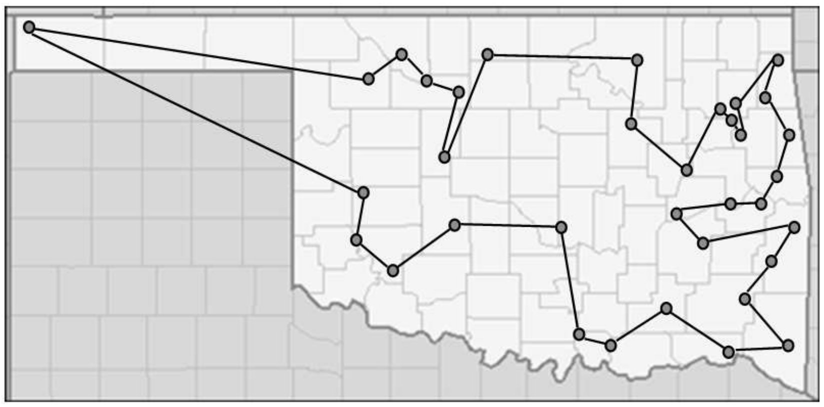
Due to its limitations, the free Bing Maps API was not able to output an image with the full route. There is a limit of 25 “waypoints” or destinations for generated maps with routes. Instead of a map, here is the list of state parks in the order in which they should be visited. This order achieves the shortest possible path to all state parks in Oklahoma:

1. Alabaster Caverns State Park
2. Boiling Springs State Park
3. Black Mesa State Park
4. Foss State Park
5. Quartz Mountain State Park
6. Great Plains State Park
7. Fort Cobb State Park
8. Lake Thunderbird State Park
9. Lake Murray State Park
10. Lake Texoma State Park
11. McGee Creek State Park
12. Raymond Gary State Park
13. Beavers Bend State Park
14. Clayton Lake State Park
15. Talimena State Park
16. Wister Lake State Park
17. Robbers Cave State Park
18. Lake Eufaula State Park
19. Greenleaf State Park
20. Tenkiller State Park
21. Cherokee Landing State Park
22. Natural Falls State Park
23. Honey Creek at Grand Lake
24. Twin Bridges at Grand Lake
25. Bernice Area at Grand Lake
26. Little Blue at Grand Lake
27. Cherokee Area at Grand Lake

28. Spavinaw State Park
29. Sequoyah State Park
30. Keystone State Park
31. Osage Hills State Park
32. Salt Plains State Park
33. Roman Nose State Park
34. Gloss Mountain State Park
35. Little Sahara State Park
36. Loop back to Alabaster Caverns

Starting at any item on this list and continuing down the list (looping back when at the end) will provide a route with the shortest distance to all of Oklahoma's state parks.

Here is a rough idea of what the route looks like:



Conclusion

This problem took some trial and error and a decent amount of research, especially to formulate the integer program. References to the websites used are included in the next section.

Gurobi took less than one minute to find the optimal solution with a 0% margin. After a couple different formulations of the integer program, this is a good solution. In the future, the goal is to improve on the preprocessing code to attempt a 300+ instance problem. A few steps of this specific project were done manually, and the preprocessing must be improved before a larger instance is solved.

References

Traveling Salesman Problem: Solver-Based - MATLAB & Simulink (mathworks.com)
Traveling Salesman Problem (TSP) with Miller-Tucker-Zemlin (MTZ) in CPLEX/OPL – CoEnzyme (co-enzyme.fr)

Appendix

Code and files specific to Oklahoma State Parks:
https://github.com/KyleHumphreys/IEM-4013-Traveling-Salesman-Problem

Reader-friendly blog post about this project:
https://kylehumphreys.wordpress.com/2021/04/30/the-traveling-salesman-problem-for-oklahoma-state-parks/