Traveling Plan Generator

Dong Xia, Boning Xing, Jiayi Bao, Wenyuan Shen

Abstract

In our study, we developed an advanced methodology to rank and optimize travel itineraries for various Chinese cities, utilizing a comprehensive dataset and advanced computational techniques from operations research. Our work is divided into two main parts: firstly, city ranking, and secondly, itinerary optimization through Integer Programming and the Nearest Neighbor Algorithm.

Introduction

Problem Statement

The primary problem we are addressing has two parts: firstly, identifying the optimal set of cities to visit, and secondly, determining the most efficient route to travel through these cities. This involves considering various factors such as personal preferences, budget constraints, travel duration, and the unique attributes of each city. The complexity of this problem lies in balancing these diverse elements to create an itinerary that maximizes overall travel satisfaction.

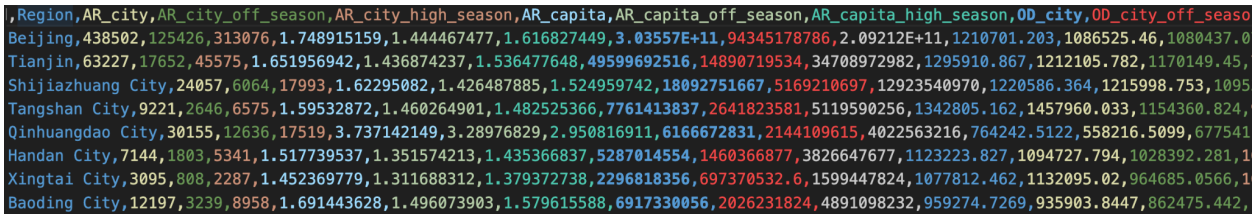
Objective

The objective of our project is to develop a comprehensive travel itinerary that optimizes the selection of cities and the duration of stay in each city. We aim to employ operations research techniques to assess and rank potential travel destinations based on various criteria tailored to our preferences. Additionally, we intend to utilize optimization algorithms to determine the most efficient travel route, ensuring that we make the most out of our time and budget. The end goal is to create a balanced, fulfilling travel experience that caters to our interests and maximizes the value of our graduation trip.

Data and Tools

The dataset described in "city.csv" offers a comprehensive statistical analysis of various cities in China, covering five key areas: basic city information, access and outbound

(AR&OD) data, demographic characteristics, social-spatial structure, and socioeconomic status. Each city is uniquely identified by a 'City_cd', representing its administrative division code. The 'Region' field provides the city's name. The dataset includes detailed AR information such as average yearly access frequency (AR_city), differentiated by off-season (AR_city_off_season) and high-season (AR_city_high_season), as well as per capita figures for these metrics. OD data covers average yearly outbound distances for each city (OD_city), again split into off-season and high-season figures, with corresponding per capita values. Demographic information includes the percentage of local residents with a hukou (household registration) in each city, age distribution data (percentages of populations under 14, between 15 and 64, and over 65), and resident and migrant population densities. Additionally, the dataset provides the number of attractions in each city, offering insights into its tourism and cultural significance. This rich dataset facilitates a multidimensional analysis of urban dynamics, demographics, and socioeconomic factors across Chinese cities. Below is an example of what our dataset looks like:



	Region	AR_city	AR_city_off_season	AR_city_high_season	AR_capita	AR_capita_off_season	AR_capita_high_season	OD_city	OD_city_off_seaso
Beijing	438502	125426	313076	1.748915159	1.444467477	1.616827449	3.03557E+11	94345178786	2.09212E+11
Tianjin	63227	17652	45575	1.651956942	1.436874237	1.536477648	49599692516	14890719534	34708972982
Shijiazhuang City	24057	6064	17993	1.62295082	1.426487885	1.524959742	18092751667	5169210697	12923540970
Tangshan City	9221	2646	6575	1.59532872	1.460264901	1.482525366	7761413837	2641823581	5119590256
Qinhuangdao City	30155	12636	17519	3.737142149	3.28976829	2.950816911	6166672831	2144109615	4022563216
Handan City	7144	1803	5341	1.517739537	1.351574213	1.435366837	5287014554	1460366877	3826647677
Xingtai City	3095	808	2287	1.452369779	1.311688312	1.379372738	2296818356	697370532.6	1599447824
Baoding City	12197	3239	8958	1.691443628	1.496073903	1.579615588	6917330056	2026231824	4891098232

	1123223.827	1094727.794	1028392.281	1
Beijing	764242.5122	558216.5099	677541	
Tianjin	1295910.867	1212105.782	1170149.45	
Shijiazhuang City	1220586.364	1215998.753	1095	
Tangshan City	1342805.162	1457960.033	1154360.824	
Qinhuangdao City				
Handan City				
Xingtai City	1077812.462	1132095.02	964685.0566	1
Baoding City	959274.7269	935903.8447	862475.442	

In our project, we aim to evaluate and rank various Chinese cities based on several key factors, leveraging the data from the 'city.csv' dataset, so we wrote the ‘city_score_generator’. Our approach involves a blend of data preprocessing, feature selection, and weighted scoring to create a comprehensive city ranking system.

Firstly, we define a set of criteria for our city ranking: Attractions and Culture, Cuisine, Accessibility, and Infrastructure and Safety. Each criterion will be assigned a specific personal preference weight, reflecting its importance in the heart of the person who is using this pipeline to select traveling cities. For example, these weights can be 0.2 for Attractions and Culture, 0.2 for Cuisine, 0.3 for Accessibility, and 0.3 for Infrastructure and Safety, which sum up to 1.

For each criterion, we select relevant features from the dataset. 'Attraction number' represents Attractions and Culture; 'Rural food expenditure' and 'Urban food expenditure' indicate Cuisine; 'AR_capita' and 'OD_capita' define Accessibility; and demographic features like '% local hukou rate', '% under age 14', and '% above age 65' are used for Infrastructure and Safety.

We then employ the MinMaxScaler from Scikit-learn to normalize these features. This transformation scales each feature to a given range, typically 0 to 1, enhancing comparability across different metrics.

The next step involves calculating individual scores for each criterion. We multiply the normalized feature values by 10, and for features with multiple indicators (like Cuisine), we take the mean across these indicators to obtain a single score per city per criterion.

Finally, we compute the overall city score by multiplying the individual scores by their respective weights for personal preference and summing these up. This score reflects a city's performance across all considered criteria.

We filter the results for a pre-selected list of cities and export the final ranked list to 'city_score.csv'. This list includes notable cities like Beijing, Shanghai, and Xi'an, among others. The final output provides a comprehensive ranking of these cities, based on the combined influence of cultural attractions, culinary offerings, accessibility, and infrastructural and safety parameters.

Methodology

Integer Programming

This project utilizes an Integer Programming (IP) approach to optimize the selection of travel destinations and the duration of stay in the selected destinations. The IP model is formulated to maximize the total score derived from visiting a set of cities within a specified number of travel days and a predefined budget. The city scores are derived from the city_score_generator that have taken customer's personal preferences into account as weights, and the budget for each city is derived from the city_expense_generator.

Variables:

- x_i: Number of days to stay in city i
- s_i: Score assigned to city i, reflecting its attractiveness or priority
- b_i: Expected expense in city i
- y_i: Binary variable indicating if city i is selected (1 if selected, 0 otherwise)

Objective Function:

The model seeks to maximize the traveler's satisfaction by choosing a set of cities that optimize the total score, adjusted by a quadratic penalty on the number of days to capture the phenomenon of diminishing marginal utility in reality thus to discourage disproportionately long stays.

$$max \sum_i (x_i * s_i * y_i) - 0.05 * s_i * x_i^2$$

Constraints:

- Time Constraint: The sum of days spent in all cities is less than or equal to the total available travel days.
- Minimum Stay Constraint: If a city is selected, the traveler must stay for at least two days to ensure an adequate experience.
- Budget Constraint: The sum of the product of days spent in each city and the corresponding expense does not exceed the traveler's budget.
- For the budget constraint, city_expense_generator first normalizes the city scores to a scale that matches our budget range. This normalization is done using a z-score calculation and then scaling these scores to fit within our maximum budget of ¥20,000. The budget constraint ensures that the total cost of our trip, calculated as the sum of the cost per day times the number of days in each city, does not exceed this maximum budget.

Decay function

The quadratic decay function $d(x) = 1 - \beta * x^2$ is applied to the objective function, where x denotes the duration of stay and β is the decay rate. A quadratic decay function is useful in scenarios where the rate of decay accelerates over time. This type of function models situations where the initial impact or value diminishes gradually at first and then more rapidly as time progresses. In this scenario, when a traveler first arrives in a city, their enjoyment level is high due to the novelty and excitement of new experiences. However, as travelers spend more time in a city, the initial high level of satisfaction decreases, and the rate at which enjoyment decreases starts to accelerate since what was novel and exciting becomes more familiar and less stimulating.

Therefore, incorporating a quadratic decay function into the travel itinerary optimization model effectively quantifies the intuitive understanding of human psychology and encapsulates the realistic scenario where extended stays in the same city offer diminishing returns after an optimal period. By applying this model, the objective function strategically addresses the diminishing returns of extended stays, thereby optimizing the allocation of time across multiple destinations. It ensures that travelers

spend an ideal amount of time in each city before the marginal enjoyment diminishes significantly, which makes each city visit remain engaging and fulfilling. This mathematical approach provides a pragmatic solution for solving the common issue of overextended stays and maximizing overall travel experience.

Nearest Neighbor Algorithm

1. **Introduction**

   The Nearest Neighbor Algorithm is a heuristic approach used to address the Traveling Salesman Problem (TSP), which involves identifying the shortest possible route that visits a set of cities and returns to the origin. This algorithm is noteworthy for its simplicity and efficiency in generating routes, especially when dealing with a large number of cities.

2. **Working Principle**

   The algorithm starts with an initial city and sequentially selects the nearest unvisited city as the next step. This process continues until all cities are visited. The final step involves returning to the starting city, completing the circuit. While this method does not always yield the shortest path, it is recognized for its relative accuracy and computational efficiency.

Results

We use 15 most popular cities (Beijing, Shanghai, Xi'an, Guilin, Chengdu, Hangzhou, Suzhou, Shenzhen, Guangzhou, Lhasa, Lijiang, Dali, Chongqing, Nanjing, Harbin) in China as a pre-selected pool from which our algorithm will select travel destinations. In this example, the traveler is planning for a 20-day trip with a total budget of 20,000 Chinese yuan.

The following examples illustrate how our algorithm generates different routes for travelers with different preferences.

In the first scenario, the traveler prefers safety and cuisine over accessibility and attractions. The personal preference weight vector is [Attractions and Culture: 0.1,

Cuisine: 0.3, Accessibility: 0.1, and Infrastructure and Safety: 0.5]. Based on our algorithm, the optimal travel plan is to spend 2 days in Beijing, 2 days in Shanghai, 2 days in Guilin, 3 days in Hangzhou, 2 days in Suzhou, 2 days in Guangzhou, 2 days in Dali, and 2 days in Chongqing. The shortest routes that visit each city exactly once and return to the origin city are shown below. Graph (1) on the left uses Beijing as the origin city, while graph (2) on the right chooses Shanghai as the starting point. It is clear from the graphs that the choice of starting point can greatly influence the output of the nearest neighbor algorithm. One future improvement we can do is to run the algorithm over all possible origin cities and return the route with the shortest total distance.

In the second scenario, the traveler prefers good food over all other factors. The personal preference weight vector then becomes [Attractions and Culture: 0.1, Cuisine: 0.5, Accessibility: 0.2, and Infrastructure and Safety: 0.2]. The optimal travel plan is altered to spend 2 days in Beijing, 2 days in Shanghai, 3 days in Guilin, 3 days in Hangzhou, 2 days in Suzhou, 2 days in Guangzhou, 2 days in Lhasa, and 2 days in Nanjing. Graph (3) below shows the shortest route that starts from Beijing, visits all the cities in the plan, and returns to Beijing.

Future research:

As we advance our research and development on route optimization algorithms, we acknowledge the importance of continuous improvement. Reflecting on the current implementation, we propose several enhancements to both the selection of cities and the construction of the traveling route.

1. **Enhancements in City Selection**

(1) Comprehensive Score Generator

Current methodologies for selecting cities rely on simplistic scoring systems that may not fully capture the complexities of city attributes. We recommend the development of a more nuanced score generator. For example, we don't need to use food expenditure in the city to estimate the food/restaurant quality. Instead, using the number of restaurants with high ratings would be a better estimation. This enhanced system would incorporate a broader range of data points, including economic indicators, cultural significance, and traveler reviews, to produce a multifaceted score reflective of a city's overall appeal for inclusion in travel routes.

(2) Advanced Decay Functions

The inclusion of decay functions in the city selection process can simulate diminishing returns, where the value of adding additional cities to the route decreases over time or distance. By incorporating more sophisticated decay functions, we can refine our model to better represent the real-world impact of extending travel itineraries, thus optimizing for time and resource expenditure.

During implementation of the decay function, we also found Gurobi doesn't support exponential function. If we have more time, we will try solving the integer programming without using Gurobi, so we can try other decay functions to improve the algorithm.

(3) Balancing Collective Goals and Individual Preferences

The preference vector inputted in this algorithm represents one traveler or one group's preference. However, preferences of travelers within a group are usually different, so the algorithm should aim to harmonize the collective and individual preferences.. We can add more constraints to the model so that while the collective goal is optimized, individuals preferences are also well considered.

2. Improvements in Traveling Route Construction

(1) Advanced Algorithms for TSP

The Nearest Neighbor Algorithm, while effective for initial route planning, is fundamentally greedy and can be shortsighted. For future iterations, we propose exploring more advanced algorithms that can provide better solutions for the TSP. Techniques such as Genetic Algorithms, Simulated Annealing, or Ant Colony Optimization have shown promise in finding near-optimal solutions and could significantly enhance our route optimization.

(2) Incorporation of Travel Time Metrics

Distance has been a traditional metric for route planning; however, travel time is often a more relevant measure for travelers. We advocate for the integration of travel time calculations in route planning, accounting for varying speeds and transportation modes. This shift in metrics will likely result in more practical and time-efficient travel routes.

(3) Data Collection from Multiple Transportation Modes

To further refine the travel time estimates and provide more comprehensive routing options, we plan to integrate diverse transportation data, including airline schedules, railway timetables, and public transit systems. By collecting and

utilizing data from these various modes, the algorithm can offer routes that are not only optimized for distance or time but also provide a range of options catering to different preferences and constraints.

In conclusion, the proposed enhancements aim to augment the algorithm's robustness and adaptability. By embracing a more holistic approach to city selection and route construction, we can deliver sophisticated solutions that align more closely with the multifaceted nature of travel planning and individual traveler needs.

Code

city_score_generator

```
import pandas as pd

from sklearn.preprocessing import MinMaxScaler




df = pd.read_csv('city.csv')



# Self Defined Weight

weights = {

   'Attractions and Culture': 0.2,

   'Cuisine': 0.2,

   'Accessibility': 0.3,

   'Infrastructure and Safety': 0.3
```

```
}



attractions_culture_feature = 'Attraction number'

cuisine_features = ['Rural food expenditure', 'Urban food expenditure']

accessibility_features = ['AR_capita', 'OD_capita']

infrastructure_safety_features = ['  % local hukou rate', '% under age 14', '% above
age 65']




scaler = MinMaxScaler()

df[attractions_culture_feature] =
scaler.fit_transform(df[attractions_culture_feature].values.reshape(-1, 1))

df[cuisine_features] = scaler.fit_transform(df[cuisine_features])

print(df[cuisine_features])

df[accessibility_features] = scaler.fit_transform(df[accessibility_features])

df[infrastructure_safety_features] =
scaler.fit_transform(df[infrastructure_safety_features])



df['Attractions and Culture'] = 10* df[attractions_culture_feature]

df['Cuisine'] = 10*df[cuisine_features].mean(axis=1)

df['Accessibility'] = 10* df[accessibility_features].mean(axis=1)

df['Infrastructure and Safety'] = 10* df[infrastructure_safety_features].mean(axis=1)
```

```
df['score'] = df[list(weights.keys())].mul(list(weights.values())).sum(axis=1)



cities = ['Beijing', 'Shanghai', "Xi'an City", 'Guilin City', 'Chengdu City',
'Hangzhou City', 'Suzhou City', 'Shenzhen City', 'Guangzhou City', 'Lhasa City',
'Lijiang City', 'Dali Bai A.P', 'Chongqing', 'Nanjing City', 'Harbin City']



result = df[['Region','score', 'Attractions and Culture', 'Cuisine', 'Accessibility',
'Infrastructure and Safety']]

filtered_result = result[result['Region'].isin(cities)]



filtered_result.to_csv('city_score.csv', index=False)
```

IP_select_city & city_expense_generator

```
from gurobipy import Model, GRB

import pandas as pd

import math



df = pd.read_csv('city_score.csv')

model = Model("CityTravel")

model.params.NonConvex = 2



cities = ['Beijing', 'Shanghai', "Xi'an City", 'Guilin City', 'Chengdu City',
'Hangzhou City', 'Suzhou City', 'Shenzhen City', 'Guangzhou City', 'Lhasa City',
'Lijiang City', 'Dali Bai A.P', 'Chongqing', 'Nanjing City', 'Harbin City']
```

```
x = {city: model.addVar(vtype=GRB.INTEGER, name=f"x_{city}") for city in cities}

y = {city: model.addVar(vtype=GRB.BINARY, name=f"y_{city}") for city in cities}

cities_scores_dict = dict(zip(df['Region'], df['score']))




# Objective with decay

objective = sum(cities_scores_dict[city] * x[city] * y[city] for city in cities)

for city in cities:

   objective -= 0.05 * cities_scores_dict[city] * x[city]**2

model.setObjective(objective, GRB.MAXIMIZE)

# Time Constraint

max_total_days = 20

model.addConstr(sum(x[city] for city in cities) <= max_total_days,
name="total_days_constraint")



for city in cities:

   model.addConstr(x[city] >= 2 * y[city], name=f"x_{city}_not_1_constraint")



# Budget constraint

city_score_list = []

for index, row in df.iterrows():

   city_score_list.append((row['Region'], row['score']))
```

```
# generate mean

mean = df['score'].mean()



# generate standard deviation

std = df['score'].std()



# generate z score for each city

city_z_score_list = []

for city, score in city_score_list:

   city_z_score_list.append((city, (score - mean) / std))

# print(city_z_score_list)



# get the max and min z score

max_z_score = max(city_z_score_list, key=lambda x: x[1])[1]

min_z_score = min(city_z_score_list, key=lambda x: x[1])[1]

print(max_z_score, min_z_score)



# normalize the z score of each city to the range of -1 to 1, make sure all z scores
are in this range

city_z_score_scaled_list = []

for city, z_score in city_z_score_list:

   city_z_score_scaled_list.append((city,1 + 0.4 * (z_score - min_z_score) /
(max_z_score - min_z_score) - 0.2))
```

```
max_budget = 20000

base = max_budget/max_total_days

city_z_score_scaled_list = [(city, scale * base) for city, scale in
city_z_score_scaled_list]

cities_costs_dict = dict(city_z_score_scaled_list)




model.addConstr(sum(cities_costs_dict[city] * x[city] for city in cities) <=
max_budget, name="budget_constraint")



model.optimize()



if model.status == GRB.OPTIMAL:

   print("Optimal solution found!")

   for city in cities:

       print(f"{city}: {x[city].x}")



   print(f"Total score: {model.objVal}")

   print(f"Total cost: {sum(cities_costs_dict[city] * x[city].x for city in cities)}")

else:

   print("No optimal solution found.")
```

TSP

```python
import math
import pandas as pd
# https://simplemaps.com/data/cn-cities

df = pd.read_csv('cn.csv')

# Dropping duplicates based on the 'city' column while keeping the first occurrence
df_unique_cities = df.drop_duplicates(subset='city')

# Now constructing the dictionary
city_location_dict = df_unique_cities.set_index('city')[['lat', 'lng']].to_dict(orient='index')

# Convert the dictionary values from single-key dictionaries to tuples
city_location_dict = {city: (info['lat'], info['lng']) for city, info in city_location_dict.items()}

# Check if works
# print(list(city_location_dict.items())[:5])
# print(city_location_dict["Guangzhou"])

import math

def haversine(lat1, lon1, lat2, lon2): # calculate distance from latitude and longitude
    # Convert decimal degrees to radians
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])
    # Haversine formula
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = math.sin(dlat / 2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon / 2)**2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    r = 6371  # Radius of Earth in km
    return c * r

def find_nearest_city(current_city, unvisited_cities, cities):
    nearest_city = None
    min_distance = float('inf')
    for city in unvisited_cities:
        distance = haversine(cities[current_city][0], cities[current_city][1],
                             cities[city][0], cities[city][1])
        if distance < min_distance:
            min_distance = distance
            nearest_city = city
    return nearest_city

def tsp_knn(cities, start):
    path = [start]
    visited = {start}
    current_city = start
    while visited != cities.keys():
        nearest_city = find_nearest_city(current_city, set(cities.keys()) - visited, cities)
        visited.add(nearest_city)
        path.append(nearest_city)
        current_city = nearest_city
    path.append(start)  # Return to start
    return path

# # Example usage
cities = {
    'A': (0, 0),  # Replace with actual latitudes and longitudes
    'B': (1, 5),
    'C': (5, 1),
    'D': (3, 3),
    'E': (4, 4)
}

```

```
start_city = 'A'
tsp_path = tsp_knn(cities, start_city)
print("TSP Path:", tsp_path)


target_cities = ['Beijing', 'Shanghai', 'Guilin', 'Hangzhou', 'Suzhou', 'Guangzhou', 'Lhasa', 'Nanjing' ]
# Sub-dictionary for target cities
cities = {city: city_location_dict[city] for city in target_cities if city in city_location_dict}
start_city = 'Shanghai'
tsp_path = tsp_knn(cities, start_city)
print("TSP Path:", tsp_path)

```