Big Data Analytics Final Report
Flying Marco Polo

Jin Zhou, Chong Wang, Gongqian Li
Electrical Engineering Department, Fu Foundation School of Engineering and Applied Science
Columbia University, New York, NY 10027
jz2792@columbia.edu, cw2962@columbia.edu, gl2548@columbia.edu

Abstract—**In this paper, we introduce the technical details the the results of our final project: Flying Marco Polo, a real-time travel planner system designed for those who want to travel around the world by airplane. We process flight history dataset and international airport dataset with Spark and then use implement Genetic Algorithm and Simulated Annealing Algorithm to find a shortest-possible route to travel around the most capital cities in the world. A website is designed to enable the interactive result display and customized route planning.**

Keywords: Traveling Salesman Problem, Simulated Annealing Algorithm, Shortest Path, Genetic Algorithm, Travel Planner

I. Introduction

Almost everyone of us has a dream of traveling around the world, just like Marco Polo and Magellan. With the help of the widespread airline networks and easy access to the travel info, people start to think how to finish the global traveling with minimum total mileage, with a satisfactory airline and with more fun. Therefore, we want to utilize a variety of airline datasets to plan the multi-city travel itinerary for the modern Marco Polo.

We implement algorithms like Genetic Algorithm and Simulated Annealing Algorithm to design an itinerary visiting most of the capital cities in the world with minimum flying mileage. We also build the flight routes using a specific airline for the big fan of American Airlines, Air France and so on. Moreover, the itinerary programming with user-defined city-list is realized to maximize the planner's flexibility.

II. Related Works

The problem presented in the introduction can be abstracted and described in a graph: considering an undirected and weighted graph, such that the city is the node of the graph and the reachability and flying mileage between two cities is the edge and its weight, we have to find a shortest possible route that visits every city and returns to the origin. This graph obeys the triangle inequality and its adjacency matrix is symmetric because we consider all air routes to be round-way. Therefore, our problem is rather similar to the famous Traveling Salesman Problem (TSP) except that the two cities are not necessarily reachable through a direct flight. With the proper arrangement of the flying mileage (like transfer) between two arbitrary cities, a completed graph can be constructed so that the the problem can be solved as TSP.

The origin of TSP is a mystery but it is not surprising that this problem has long been studied by those real traveling salesmen to find a shortest route to visit every city in a given region. One of the earliest formal description of TSP was in an 1832 German traveling salesman handbook which said "*through an expedient choice and division of the tour so much time can be won that we feel compelled to give guidelines about this...it will not be possible to plan the tours through Germany in consideration of the distances and the traveling back and forth…The main thing to remember is always to visit as many localities as possible without having to touch them twice.*"

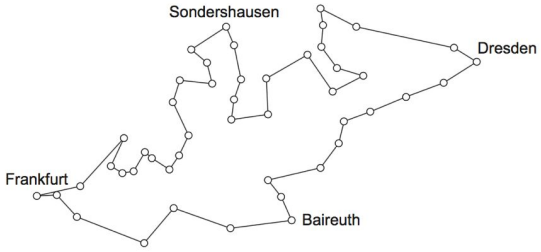

Figure 1. A route introduced in an 1832 German traveling salesman handbook. [1]

TSP is closely related to Hamiltonian Cycle problem and has been proved to belong to class of NP-hard. [2] To solve a TSP with n cities, by choosing an origin, we have $n-1$ choices for the second city, $n-2$ for the third one and so on. Therefore, the exact solution using brutal force, which enumerates all possible routes, takes time proportional to (n-1)!, which is an unimaginably large number even for the most powerful supercomputer in the world. Held and Karp discovered a more efficient algorithm with $O(n^2 2^n)$ complexity.[3] Obviously, solving a TSP with many cities

(thousands or even hundreds) using non-polynomial algorithm is not practical. Instead of the general solution approach, approximation TSP algorithms that takes polynomial time complexity and some heuristic algorithms are widely used in practical applications where the number of nodes is quite large. The best polynomial approximation is given by Christofides Algorithm.[4] The algorithm combines the minimum spanning tree of the graph and nodes with odd degree and finds a minimum-weight perfect matching in the multigraph, where a Hamiltonian circuit can be found. The solution can be within 3/2 of the optimal length.

A family of heuristic algorithms becomes popular in the past several decades, among which the most notable ones are genetic algorithm and simulated annealing. Genetic Algorithm (GA) is based on a natural evolution process. The algorithm modifies a population of solutions in each generation and the solution finally evolves to global optimal. On the other hand, Simulated Annealing (SA) Algorithm works by simulating the physical process where a solid gradually cools down. A global minimum of total cost that may possess some local minima can be found when the algorithm terminates.

With the help of the computational capability of modern computers, these heuristic algorithms are quite effective and efficient to solve the optimization problem like TSP in moderate size. Other famous algorithms include a heuristic cutting-plane method (a kind of linear programming)[5], ant-colony optimization[6] and so on.

In terms of the progress of addressing more complicated TSP, computer scientists have been able to give the path based on more than one hundred thousand locations on a map, a big contrast to the capability of solving TSP with less than a hundred spots forty years ago. The graph illustrates such advancement.

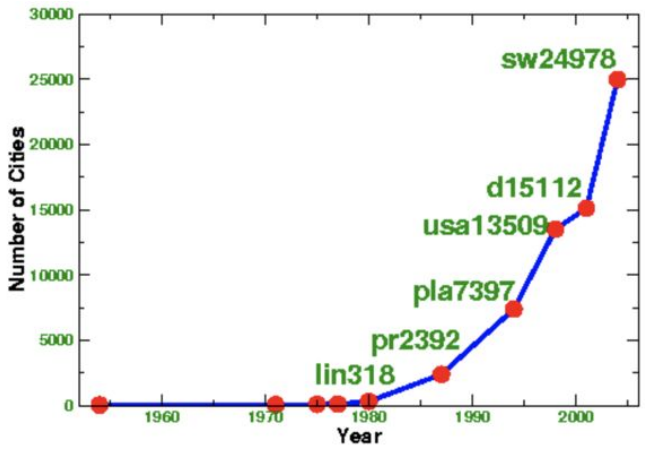

Figure 2. Progress in solving TSPs[7]

Recently in 2015, computer scientists successfully computed the shortest-possible tour to 49,603 sites from the National Register of Historic Places in the USA with cutting-plane method. The crazy tour takes 217,605 miles.[8]

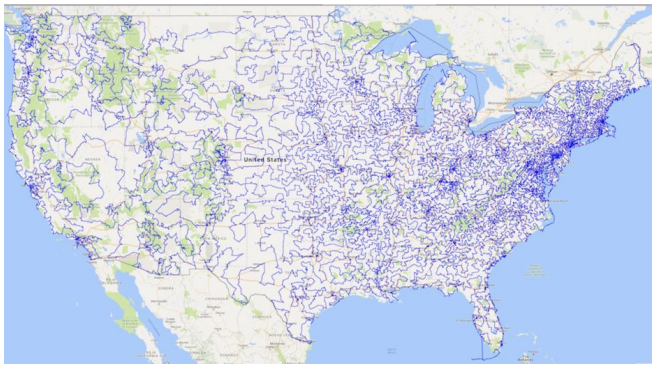

Figure 3. The shortest-possible walk to 49,603 National Historic Sites in the US

Most of the Traveling Salesman Problems are based on road-trips and there is few instance that concentrates on air transportation. Because the air route data and air history data are easily accessible, and the parallel query implementation with Spark can largely facilitate the processing speed, it is a practical and feasible idea to calculate the shortest-possible to visit all major capital cities in the world.

III. System Overview

The core of our system include MySQL database and Spark SQL. The datasets we used include:
Airports data: (~1MB)
https://raw.githubusercontent.com/jpatokal/openflights/master/data/airports.dat
Air routes data: (~2MB)
https://raw.githubusercontent.com/jpatokal/openflights/master/data/routes.dat
Air Carrier dataset (All carrier): (~1.2GB)
http://www.transtats.bts.gov/Tables.asp?DB_ID=111
We store these datasets in MySQL, then use Spark to query the information we want and implement several algorithms by python to get the ideal path for the trip. We show our results through website constructed by Flask (a python microframework). The frontend is HTML+CSS, embedded with Google Maps JavaScipt API. The server is built on Amazon AWS. The interactive part on the website include the display of Basic Marco Polo mode (visit ~150 cities), Airline mode (American Airlines) and Holiday Mode. Holiday mode enables traveller to finish the global travel in several segments with fixed number of days. Our default setting is 7 days, which means the traveller will leave New

York to explore 7 cities and go back after 7 days. This mode is processed in a different way in our algorithm. The website also provides customized plannning. Users can input a list of cities and the system will calculate and output the result.

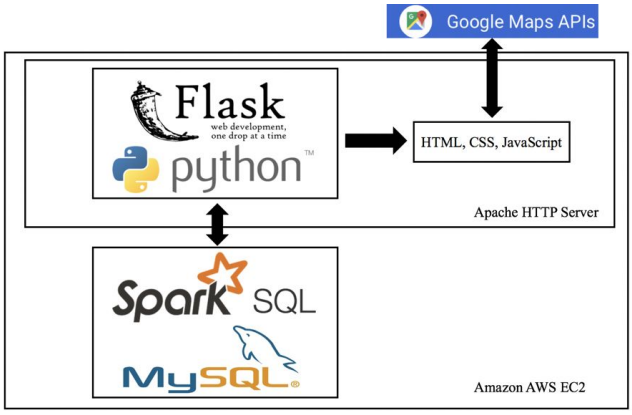

Figure 4. Structure of our system

IV. ALGORITHM

As is mentioned before, we use SQL to process raw data and build our data interface between database and python. Basically, we follow three steps to generate the input graph information which we use in GA and SA.

1. Extract geographic information of airports including longtitude, latitude and nearby cities.
2. Generate city list table including city's zip in case of duplication.
3. Join two city list table with history flight information data table. In terms of time consummation, this step cost most in SQL procedure. So my adopt spark SQL to accelerate and optimize our SQL query.

After data processing, we can get a large matrix showing whether flight exists between every pairs of city or not, and the average flying time based on history flight information.

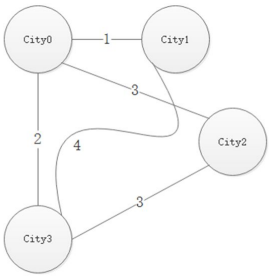

Figure 5. Distance between 4 cities

	City0	City1	City2	City3
City0		1	3	2
City1	1			4
City2	3			3
City3	2	4	3	

Figure 6. Distance dictionary D

We can get a distance dictionary D through our input data. D[City_i][City_j] represents the average flight distance from City_i to City_j. The vacancy in the dictionary represents that there is no direct flight between its two corresponding cities.

Dijkstra's Algorithm

	City0	City1	City2	City3
City0	0	1	3	2
City1	1	0	4	4
City2	3	4	0	3
City3	2	4	3	0

Figure 7. Full connected distance dictionary A

	City0	City1	City2	City3
City0	City0	City0	City0	City0
City1	City1	City1	City0	City1
City2	City2	City0	City2	City2
City3	City3	City3	City3	City3

Figure 8. Shortest path dictionary B

We implement Dijkstra's Algorithm to the distance dictionary D, and get a full connected distance dictionary A and shortest path dictionary B.

If D[City_i][City_j] is not vacant
A[City_i][City_j]=D[City_i][City_j]
Else
A[City_i][City_j]=the distance of the shortest path from City_i to City_j calculated by Dijkstra's Algorithm.

B[City_i][City_j] in the shortest distance dictionary B records the penultimate city of the shortest path from City_i to City_j. With the help of dictionary B, we can find the shortest path between two arbitrary cities.

Since A can represent a full connected distance matrix of all cities. The Flying Marco Polo Problem is transformed to a classical TSP. We choose three algorithms to get the result.

Greedy Algorithm

In a Greedy Algorithm, the program first pick out the city nearest to the start city and add that city as the second stop in the path. Secondly, it finds a unvisited city which is nearest to the last city in the path and adds the city to the end. By doing the second process repeatedly until every city is visited, we can get the final path. Greedy Algorithm is very fast, but cannot achieve a desirable result.

Genetic Algorithm (GA)

In a Genetic Algorithm, a set of candidate solutions is evolved toward better solutions by mutating and altering certain properties. In each iteration (generation), the optimality of the solution is evaluated. The better solutions are stochastically selected, and each individual's genome is recombined or randomly mutated to form a new generation. The algorithm stops when either a maximum number of iteration has been reached, or the optimality is good enough. When specially applied to TSP, GA does the evolution by shuffling cities to generate another valid route and evaluates its total distance. Different representation methods like binary representation and path representation, and different operators like OX, PMX and CX can affect the effectiveness of GA. [9]

```
BEGIN AGA
  Make initial population at random.
  WHILE NOT stop DO
    BEGIN
      Select parents from the population.
      Produce children from the selected parents.
      Mutate the individuals.
      Extend the population adding the children to it.
      Reduce the extend population.
    END
  Output the best individual found.
END AGA
```

Figure 9. Pseudo-code of a general GA [10]

Simulated Annealing(SA)

In Simulated Annealing, we can select an arbitrary initial tour from all valid routes. Then we move around to check random neighboring tours to evaluate the optimality. If it is better, we choose it as the next state; otherwise, we have the exponential probability which is related to the temperature at that time to decide whether to accept it or not. Finally, we find a global minimum cost with several

```
Select an initial state i ∈ S;
Select an initial temperature T > 0;
Set temperature change counter t = 0;
Repeat
Set repetition counter n = 0;
  Repeat
   Generate state j, a neighbour of i;
   Calculate δ = f(j) − f(i);
   If δ < 0 then i := j
     else if random(0, 1) < exp(− δ/T) then i := j;
  n := n + 1;
  until n = N(t);
t := t + 1;
T := T(t);
until stopping criterion true.
```

Figure 10. Pseudo-code of Simulated Annealing[11]

The result path is a list: *[City0, City1,City2,City3]*. Every city shows once in the list, and we assume that the traveller will go back to the first city after visiting the last city in the list.

There are three modes (Basic MarcoPolo Mode, Airline Company Mode and Holiday Mode). The difference between them is the calculating method of their total travelling distance.

In Basic MarcoPolo Mode and Airline Company Mode, the total distance of a given path *[City0,City1,City2,City3]* is equal to

A[City3][City0] + A[City0][City1] + A[City1][City2] + A[City2][City3].

In Holiday Mode, the traveller have to go back to the start city after travelling n cities. If *n=2* the total distance of path *[City0,City1,City2,City3]* is equal to

A[City3][City0] + A[City0][City1] + A[City1][City2] + A[City2][City0] + A[City0][City3] + A[City3][City0].

The list getting from the above algorithms is not the final path. Firstly, for Holiday Mode, we have to add the start city into the path. If *n=2*, *[City0,City1,City2,City3]* will become *[City0,City1,City2,City0,City3]*. Secondly, since there may not be a direct flight between two neighboring cities in the list. For example, in the path *[City0,City1,City2,City3]*, a traveller cannot fly from *City1* to *City2* directly according to distance dictionary D. Therefore, we have to add the shortest path from *City1* to *City2* into the list by using the shortest path dictionary B. And the final path is *[City0,City1,City0,City2,City3]*.

V. Software Package Description

We mainly use following software package and tools to do our project.

Spark : Run query commands to process raw data.
MySQL : Store prepared data.
Flask : Construct a website framework for Python.
Python : Implement algorithm.
Google Map API : Display the route and transition cities on a Map.

Spark SQL:
Spark introduces a programming module for structured data processing called Spark SQL. It provides a programming abstraction called DataFrame and can act as distributed SQL query engine.We use Spark SQL in our project because its efficiency towards big data processing. Spark SQL has more advanced mechnism to plan queries.

To better display the results and provide an interactive interface for self-defined travel routes, we constructed a website embedded with Google Maps using Flask. Flask is a lightweight framework for Python. With Flask, user's input can be easily collected and processed with python program in backend. Jinja2 template engine enables the generation of HTML which correctly instructs Google Maps API to display the results.

In the frontend, HTML and CSS are used to define the structure and some advanced style features of our website. How Google Map is called is also straightforward with Maps JavaScript API. All geolocation information and route-drawing instructions are passed to the API. To make this process easier, we use Flask Google Maps package that integrates basic functionality to display locations, show labels and draw lines on Google Map.

The website is intended to running on Apache HTTP server on Amazon AWS EC2. But for easy deployment, the primitive version used the Flask's built-in server function and was running on port 80 for direct public access through IP address or flyingmarcopolo.info. It is definitely not recommended to deploy a website in this way for safety consideration. We will soon move it to a permanent place if time is allowed.

VI. Experiment Results

We build an interactive website to show results and process customized input city list. Three modes are implemented on our website. Here is introduction of function of different modes and corresponding screenshots.

1. Basic MarcoPolo Mode:

We choose most capital cities in the world as our objective destinations.

Figure 11. Basic Mode

It takes 193778 kilometers to travel all these cities. Since this route is generated by real history flight data, some cities are only connected to few other cities. Thus, it's reasonable to visit French Guiana across the Atlantic Ocean from Paris.

2. Airline Company Mode:

In this mode, we take airline company attribute into consideration. We choose American Airline and its flight to generate the shortest route. This function could help those who want to choose an airline company credit card.

Figure 12. Airline Company Mode

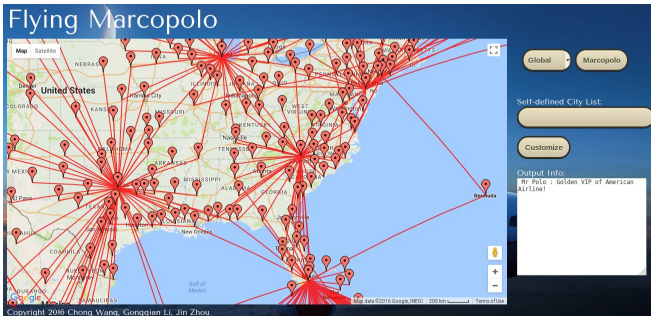

Figure 13. Zoom in map of United States

3. Holiday Mode:

The holiday mode gives a more practical solution. We assume that traveler can only use Christmas holiday week to travel around the world, which means in the end of the holiday week he must come back to New York.

Figure 14. Holiday Mode

If he spends all his holiday week traveling, it will take 15 years.

4. Customized Mode:

We allow users to input their own city list. Our system generate optimized route based on customized input city list. Because we automatically run a new query to acquire data and run simulated annealing algorithm, it may take a little more time.

Figure 15. Customized Mode

To be clear, there is a complex set of algorithms and judgment process to decide whether a given route is shortest or not. Though finding room for improvement remains a big issue, it is safe to implement several popular heuristic algorithms on our processed dataset and compare their outcomes.

To deal with join function between large table test several queries on Spark SQL and MySQL. Here is our experimental test result.

Platform	Query function	Number of airports	Number of flights (k)	Running time (min)
MySQL	Join	300	190	29
Spark SQL	Join	300	190	4
MySQL	Outer join	300	190	34
Spark SQL	Outer join	300	190	4
MySQL	Join	1200	190	84
Spark SQL	Join	1200	190	12

Figure 18.

	Greedy Algorithm	Genetic Algorithm	Annealing Algorithm
Time/min	-	1	1
Distance/km	-	803k	284k
Time/min	-	10	10
Distance/km	-	247k	197k
Time/min	3	20	20
Distance/km	937k	241k	193k

Figure 16. Results of different TSP algorithms

Greedy algorithm is easy to implement with given seeds. In average, it gives a distance of around 900 thousand kilometers and the results are poorly organized. In terms of the genetic algorithm, the distance is improved from 800 thousand kilometers down to 241 thousand kilometers by increasing the number of generations. Simulated Annealing is our most effective and efficient algorithm. The setting of appropriate initial and terminated temperature can decrease the mileage to 200 thousand kilometers in 10 minutes running time.

VII. Conclusion

In this project, we collect open source airline data to build database, use Spark SQL to process raw data, experiment with different group algorithm, and implement a real-time traveling plan website.

Gongqian Li, Chong Wang and Jin Zhou contributed equally to the fantastic idea of Flying Marco Polo. Gongqian Li was mainly responsible for processing raw data with MySQL and Spark SQL. Chong Wang studied different algorithms of TSP and wrote our python code to compute the shortest path. Jin Zhou constructed the entire website with Flask and configured the web server and DNS

to allow public access. Jin Zhou and Gongqian Li recorded the interesting demo video for this project.

Acknowledgment

The authors would like to thank the mother nature who created this beautiful world with inspirations and surprises. The authors appreciate the lectures given by Prof. Lin and his leading teaching assistants Eric Johnson that help everybody understand all kinds of topics and tools of big data analytics. Also, our TA Edward Cheng helped us make the first step into the final project. The authors would also like to thank Eugene Wu who introduce us Flask as a convenient tool to build our website.

Many thanks should also go to the author of Flask-GoogleMaps, Bruno Rocha who developed such an easy-to-use package for Flask.

Appendix

our website:

www.flyingmarcopolo.com

The Git of Flask-GoogleMap package:

https://github.com/rochacbruno/Flask-GoogleMaps

References

[1] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook, The traveling salesman problem: A computational study.New Jersey: Princeton University Press, 2007.
[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, Second edition, 2nd ed. Cambridge, MA: MIT Press, 2001, pp.1096-1097
[3] M. Held and R. M. Karp, "A dynamic programming approach to Sequencing problems," Journal of the Society for Industrial and Applied Mathematics, vol. 10, no. 1, pp. 196–210, Mar. 1962.
[4] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem." Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, No. RR-388. 1976.
[5] M. Dorigo and L. M. Gambardella, "Ant colony system: A cooperative learning approach to the traveling salesman problem," IEEE Transactions on Evolutionary Computation, vol. 1, no. 1, pp. 53–66, Apr. 1997.
[6] M. Padberg and G. Rinaldi, "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems," SIAM Review, vol. 33, no. 1, pp. 60–100, Mar. 1991.
[7] [Online]. Available: http://www.math.uwaterloo.ca/tsp/methods/progress/progress.htm. Accessed: Dec. 23, 2016.
[8] "US history traveling salesman problem,". [Online]. Available: http://www.math.uwaterloo.ca/tsp/us/index.html. Accessed: Dec. 23, 2016.
[9] P. Larrañaga, , C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic, "Genetic algorithms for the travelling salesman problem: A review of representations and operators." Artificial Intelligence Review 13.2, pp. 129-170. 1999
[10] J.-Y. Potvin, "Genetic algorithms for the traveling salesman problem," Annals of Operations Research, vol. 63, no. 3, pp. 337–370, Jun. 1996.
[11] L. Davis. Genetic algorithms and simulated annealing. Los Altos, CA: Morgan Kaufman Publisher, 1987, pp.272