Group Travel Plan Optimization through Simulated Annealing Technique

Dr. C. Muthu
Associate Professor, Department of Statistics
St. Joseph's College, Tiruchirappalli
&
M. C. Prakash
PG Student, Bharathidasan University, Tiruchirappalli

Abstract

Stochastic Optimization Techniques are typically used for solving several business problems that have many possible solutions across many variables, and that have outcomes that can change greatly depending on the combinations of these variables. In this paper, the Simulated Annealing Optimization Technique is used to arrive at an optimal group travel plan for the players of the Indian Basket Ball Team.

Keywords: Stochastic Optimization Techniques, Simulated Annealing Technique

1. Introduction

The productivity and profitability of big organizations have recently witnessed a great impact due to the application of Machine Learning algorithms [1]. Hadoop and the related tools are now widely used for successfully implementing the advanced Machine Learning algorithms in the business organizations [2]. Price Predictors are often modeled by the Data Analysts by using the KNN algorithm [3]. Potential insights into the customer preferences are now obtained by the Data Analysts by applying the Hierarchical Clustering algorithm [4].

Optimization techniques enable us to find the best possible solution to a business problem by trying many different solutions and scoring them to determine their quality. Optimization Techniques are typically used in the cases where there are too many possible solutions to try them all. The simplest but least effective method of searching for solutions is just typing a few thousand random guesses and seeing which one is best. More effective Optimization techniques, such as the Simulated Annealing Technique, which is used in this paper for obtaining an optimal travel planfor the players of the Indian Basket Ball Team, try to intelligently modify the solutions in a way that is likely to improve them.

2. Data for Study

Planning a trip for a group of people from different locations all arriving at the same place is always a challenge, and it often requires an optimum solution. There are a lot of different inputs required, such as what every one's flight schedule should be, how many cars should be rented, and which airport is easiest. Many

outputs should also be considered, such as total cost, time spent waiting at airports, and time taken off work. Because the inputs cannot be mapped to the outputs with a simple formula, the need for obtaining an optimum solution arises in this case.

To begin, we create a new python file called **scheduleOptimization.py** and insert the following code:

```
import time
import random
import math
team = [('Aravind', 'MAA'), ('Rajvir', 'TRV'), ('Vishesh', 'BLR'), ('Amitpal', 'HYD'),
       ('Satnam', 'BOM'), ('Talwinder', 'CCU')]
# InternationalAirport in New Delhi
destination = 'DEL'
```

We are planning a trip for the members of the Indian Basket Ball Team who are from all over the country and wish to meet up in New Delhi for participating in the National Tournament. They will all arrive on the same day and leave on the same day, and they would like to share transportation to and from the airport. There are dozens of flights per day to New Delhi from any of the team members' locations, all leaving at different times. The flights also vary in price and in duration.

The **sample data** for this study are stored as **flightSchedule.txt** and this data file contains origin, destination, departure time, arrival time, and price for a set of flights in a comma-separated format:

```
DEL, MAA, 20:27, 23:42, 1690
MAA, DEL, 19:53, 22:21, 1730
DEL, BOM, 6:39, 8:09, 860
BOM, DEL, 6:17, 8:26, 890
DEL, BLR, 8:23, 10:28, 1490
BLR, DEL, 7:04, 9:11, 1280
```

We load this data into a dictionary with the origin and destination as the keys and a list of potential flight details as the values. Add this code to load the data into **scheduleOptimization.py**:

```
flights = { }
for line in file ('flightSchedule.txt.'):
    origin, destination, departure, arrival, fare = line.strip( ).split(',' )
    flights.setdefault ((origin,destination), [ ])
    #Details are added to the list of possible flights
    flights [(origin, destination)].append ((departure, arrival, int (fare)))
```

We also define the utility function **getTimeInMinutes( )**, which calculates how many minutes into the day a given time is. This makes it easy to calculate flight times and waiting times. We add this function to **scheduleOptimization.py**:

```
def getTimeInMinutes(t):
    x = time.strptime(t, '%H : %M')
    return x[3] * 60 + x[4]
```

The challenge now is to decide which flight each person in the team should take. Even though keeping total fare down is a goal, there are many other possible factors that the optimal solution will take into account and try to minimize, such as total waiting time at the airport or total flight time. We will take into account these factors through a Cost Function, which we will discuss soon.

When approaching an optimization problem like this, it is necessary to determine how a potential solution will be represented. A very common simple representation is a list of numbers. In this case, each number can represent which flight a team member chooses to take, where 0 is the first flight of the day, 1 is the second, and so on. Since each team member needs an outbound flight and a return flight, the length of this list is twice the number of team members. For example, the list

```
[1, 4, 3, 2, 7, 3, 6, 3, 2, 4, 5, 3]
```

represents a solution in which Aravind takes the second flight of the day from Chennai to New Delhi, and the fifth flight back to Chennai on the day he returns. Rajvir takes the fourth flight from Trivandrum to New Delhi, and the third flight back.

3. The Cost Function

The **cost function** is the key to solving any optimization problem. The goal of our optimization algorithm is to find a set of flights that minimizes the cost function. The cost function in our Group Travel Optimization problem will involve the following variables:

Fare	:	The total fare of all the plane tickets.
Travel Time	:	The total time that everyone has to spend on a plane.
Waiting Time	:	Time spent at the airport waiting for the other members of the team to arrive.
Departure Time	:	Flights that leave too early in the morning may impose an additional cost by requiring travelers to miss out on sleep.
Car Rental Period	:	If the team rents a car, they should return it earlier in the day than when they rented it, or be forced to pay for a whole extra day.

We have to now determine how much money that time on the plane or time waiting in the airport is worth. The following **flightScheduleCost( )** function takes into account the total cost of the trip and the total time spent waiting at airports for the various members of the team. It also adds a penalty of Rs.500 if the car is

return at a later time of the day than when it was rented. We now add the following **flightScheduleCost()** function to **scheduleOptimization.py**:

```
def flightScheduleCost (soln):
    totalfare = 0
    latestarrival = 0
    earleistdeparture = 24 * 60
    for d in range (len(soln)/2):
        # Get the inbound and outbound flights
        origin = team [d][1]
        outbound = flights [(origin, destination)][int(soln[d])]
        returnf = flights [(destination, origin)] [int(soln[d+1])]
        # Total fare is the fare of all outbound and return flights
        totalfare += outbound[2]
        totalfare += returnf[2]
        # Track the latest arrival and earliest departure
        if latestarrival < getTimeInMinutes (outbound [1]):
            latestarrival = getTimeInMinutes (outbound[1])
        if earliestdeparture> geTimeInMinutes (returnf[0]):
            earliestdeparture = getTimeInMinutes (return[0])
    # Every person should wait at the airport until the latest person
    arrives.
    # They also should arrive at the same time and wait for their flights.
    totalWaitingTime = 0
    for d in range (len(soln)/2):
        origin = team [d][1]
        outbound = flights[(origin, destination)] [int(soln[d])]
        returnf = flights[(destination, origin)] [int(soln[d+1])]
        totalWaitingTime += latestarrival –
        getTimeInMinutes(outbound[1])
        totalwaitingTime += getTimeInMinutes (returnf[0] –
        earliestdeparture
    # Check whether this solution requires an extra day of car rental
    # That will be Rs. 500!
    if latestarrival > earliestdeparture: totalfare += 50
    return totalfare + totalwaitingTime
```

The logic involved in this cost function is simple. The total wait time assumes that all the team members will leave the airport together when the last person arrives, and will all go to the airport for the earliest departure. We can try this **flightScheduleCost()** function in the following Python session:

```
>>> reload (scheduleOptimization)
>>>scheduleOptimization.flightScheduleCost(s)
58230
```

4. Simulated Annealing Optimization Technique

The goal of our Group Travel Optimization problem is to minimize cost by choosing the correct set of numbers. In theory, we can try every possible combination, but in this study, we have 16 flights, all with 9 possibilities, giving a total of 9^{16} combinations (i.e. 300 billion combination). Testing every combination will guarantee that we get the best answer, but it will take a very long time on most types of computers. Trying a few thousand random guesses and seeing which one is best is another possible technique.

Randomly trying different solutions is very inefficient because it does not take advantage of the good solutions that have already been discovered. In our study, a flight schedule with a low overall cost is probably similar to other flight schedules that have a low cost. Because random optimization technique jumps around, it will not automatically look at similar flight schedules to locate the good ones that have already been found.

The Simulated Annealing Optimization Technique begins with a random solution to our Group Travel Plan optimization problem. It was a variable representing the willingness to accept a worse solution, which starts very high and gradually gets lower. In each iteration, one of the numbers in the solution is randomly chosen and changed in a certain direction. In our study, Aravind's return flight may be moved from the second of the day to the third. The cost is calculated before and after the change, and the costs are compared.

It the new cost is lower, the new solution becomes the current solution. However, if the cost is higher, the new solution can still become the current solution with a certain probability. This is done so as to avoid the local minimum problem.

In some cases, it is necessary to move to a worse solution before we can get to a better one. Simulated annealing works because it will always accept a move for the better, and because it is willing to accept a worse solution near the beginning of the process. As the process goes on, the algorithm becomes less and less likely to accept a worse solution, until at the end it will only accept a better solution. The probability of a higher-cost solution being accepted is given by the following formula:

$$p = e^{((-\text{highcost}-\text{lowcost})/\text{willingness to accept a worse solution})}$$

Since the willingness to accept a worse solution starts very high, the exponent will always be close to 0, so the probability will almost be 1. As the willingness to accept a worse solution decreases, the difference between the high cost and the low cost becomes more important – a bigger difference loads to a lower probability, so the algorithm will favour only slightly worse solutions over much worse ones.

We now create the function **annealingoptimization( )** and add it to **scheduleOptimization.py**:

```
def annealingOptimization (domain, costf, T=10000.0, coolingRate = 0.95, step = 1):
        # The values are initialized randomly
        vec = [float (random.randint (domain[i] [0], domain[i][1]))
            for i in range (len(domain))]
        while T > 0.1 :
            # One of the indices is chosen
            i = random.randint (0, len(domain) – 1
            # A direction to change the index is chosen
            dir = random.randint (–step, step)
            # A new list is created with one of the values changed
            vecb = vec[:]
            vecb [i] += dir
            if vecb [i] < domain[i][0] : vecb[i] = domain[i][0]
            elif vecb[i] > domain [i][1] : vecb[i] = domain [i][1]
            # The current cost and the new cost are calculated
            ea = costf (vec)
            eb = costf (vecb)
            p = pow (math.e, (–eb–ea) / T)
            # Check whether it is better, or it makes the probability cutoff
            if (eb < ea or random.random( ) < p) :
                vec = vecb
            # The willingness to accept a worse solution is decreased
            T = T * coolingRate
        return vec
```

To perform annealing, the above function first creates a random solution of the right length with all the values in the range specified by the domain parameter. The **willingness to accept a worse solution** and the **coolingRate** are optional parameters. In each iteration, **i** is set to a random index of the solution and **dir** is set to a random number between –**step** and **step**. It calculates the current function cost and the cost if it were to change the value at **i** by **dir**.

The line of code **p = pow (math.e, (–eb–ea) / T)** shows the probability calculation, which gets lower as T gets lower. If a random float between 0 and 1 is less than this value, or if the new solution is better, the function accepts the new solution. The function loops until the willingness to accept a worse solution has almost reached 0, each time multiplying it by the cooling rate.

We now execute the **annealingoptimization( )** function is the following python session in order to get the optimum cost for our Group Travel Planning problem.

```
>>> reload (scheduleOptimization)
>>> s = scheduleOptimization.annealingOptimization (domain,

scheduleOptimization.flightScheduleCost)
>>>scheduleOptimization.flightScheduleCost(s)
22780
```

For printing all the flights that the Basket Ball Team Members need to take in order to ensure the above mentioned optimum cost, we add the following **printFlightSchedule( )** function to **scheduleOptimization.py**:

```
def printFlightSchedule(r) :
for d in range (len(r)/2):
name = team [d] [0]
origin = team [d] [1]
out = flights [(origin, destination)] [r[d]]
ret = flights [(destination, origin)] [r[d+1]]
print '%10s %10s %5s – %5s $%3s %5s %5s $%3s'
    % (name, origin, out[0], out[1], out[2], ret[0], ret[1], ret[2])
```

The optimum Group Travelling Plan obtained by using simulated Annealing Technique shall now be obtained by executing **printFlightSchedule( )** in a python session:

```
>>> scheduleOptimization.printFlightSchedule(s)
Aravind Chennai      11:44 – 14:12  Rs 1090  10:53 – 12:23  Rs 7400
Rajvir Trivandrum    11:32 – 15:59  Rs 2900  11:54 – 15:19  Rs 2560
Vishesh Bengaluru    10:57 – 13:40  Rs 1890  10:42 – 13:26  Rs 1390
Amitpal Hyderabad    10:29 – 13:41  Rs 2480  11:47 – 14:15  Rs 1700
Satnam Mumbai        10:54 – 12:27  Rs 1340  10:53 – 13:31  Rs 1320
Talwinder Kolkata    11:28 – 13:27  Rs 175   14:17 – 16:31  Rs 1290
```

It is obvious that the Simulated Annealing Optimization did a good job of reducing the overall waiting times while keeping the costs considerably down.

References

1. Jacques Bughin, "Big Data, Big Bang?", *Journal of Big Data*, 2016, Vol.3, Iss. 2, pp. 1-14.
2. Muthu, C. and Prakash, M.C., "Impact of Hadoop Ecosystem on Big Data Analytics", *International Journal of Exclusive Management Research* - Special Issue, 2015, Vol. 1, pp. 88-90.
3. Muthu, C. and Prakash, M.C., "Building a Price Predictor for an Auctioning Website", RETELL, 2015, Vol. 15, Iss. 1, pp. 135-137.
4. Muthu, C. and Prakash, M.C., "Hierarchical Clustering of Users' Preferences", RETELL, 2016, Vol. 16, Iss. 1, pp. 135-136.
5. Muthu, C. and Prakash, M.C., "Matching the users of a Website using SVM Technique", RETELL, 2017, Vol. 17, Iss. 1, pp. 53-56.
6. Muthu, C. and Prakash, M.C., "Using Bayesian Classifier for Email Sorting", RETELL, 2017, Vol. 17, Iss. 1, pp. 57-60.

———