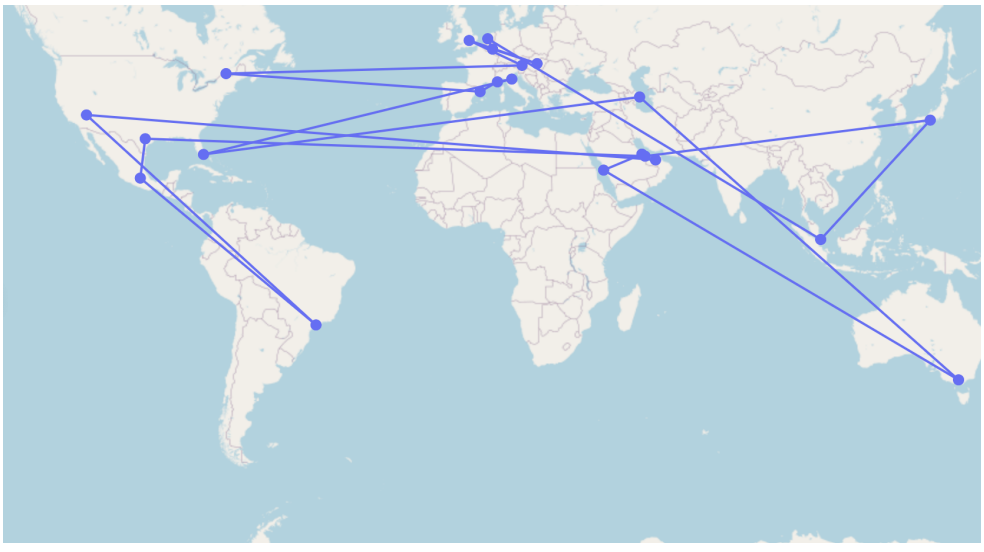
Kian Silva

Ross Parker

Calculus II

Formula One Calendar Optimized

Formula One Racing, a racing sport like no other, combines engineering feats and incredible driving ability. Each year a new car must be developed from scratch including everything from the engine to the brakes to the general shape and aerodynamics of the exterior. Every year further innovations and improvements to these vehicles make the racing miles more entertaining. As the innovations come, so do the vehicle's capacity to be more eco-friendly as they are transitioning from gas fuel to a battery-powered hybrid engine. Despite these advancements, Formula 1 still has a major flaw that creates more carbon emissions and costs for every team. This is the order in which they travel from city to city. The current Formula 1 calendar consists of traveling across the world starting in Bahrain, Saudi Arabia, Australia, Azerbaijan, Miami, Italy, Monaco, Spain, Canada, Austria, England, Hungary, Belgium, Netherlands, Italy, Singapore, Japan, Qatar, Texas, Mexico, Brazil, Las Vegas, Abu Dhabi. On a map, this looks like ridiculous back-and-forth travel and covers over one hundred sixty thousand kilometers of travel.



The travel is not only costly to the wallets of these multi-million dollar teams, but it makes the sport significantly more labor intensive. One of the more recent amendments to the rules of Formula One is that every team has a budget cap of one hundred forty million dollars to spend on whatever they may need throughout a calendar season. These expenditures spread from catering for the pit crews and admin to the engine repair costs and shipping. With cut travel expenses teams can allocate some of the funds previously designated for travel into the development of their vehicles making the competition closer and more exciting. For this reason, along with many others, I developed an interest to seek a more optimal calendar that could save some teams some more money.

In an effort to optimize this mess of a calendar, I looked at several optimization methods. I began with the Traveling Salesman Problem, which looks to optimize the route a salesman should travel from city to city to sell the most with the least amount of travel by trying every single possible order and simply seeing what is the best route. This method is an exponential method that for this purpose with 23 cities in the list would take 23! permutations to sort through. While I would ultimately get the best answer through the Traveling Salesman Problem, it would take significantly more time for any computer to compute the best route. Instead, I moved on to examining stochastic methods and decided to implement a Monte Carlo method known as simulated annealing to create the most efficient calendar to cut travel expenses. Simulated annealing will be more useful and time efficient even though this method won't find the true global minimum or the lowest distance between all of the cities. Instead this method accepts a local minimum which might be one of the shorter distances, but it may not be the shortest.

Simulated Annealing is based on the metallurgical practice of welding where a material is heated to a high temperature and then cooled. In higher temperatures, atoms tend to shift

unpredictably, but when cooled the atoms are able to rid themselves of impurities. In this algorithm, a temperature is set to a high number that allows the system to the potential solutions randomly and accepts some of the less preferable solutions. As the algorithm proceeds, the temperature will decrease and the algorithm will become less likely to accept the inferior solutions. This starting temperature is cooled by a fraction, alpha, which is the factor that cools the starting temperature to a designated minimum that can be tuned to achieve the best solution.

The algorithm begins by defining an initial configuration, which for this instance is the current calendar list of cities. With this configured list the iterations begin, starting with selecting a random pair of cities in this list. Error functions of the current and neighboring lists are taken, where f(c) is the error formula for the current list of cities and f(n) is the error formula for the neighboring list. The error formula is the list as the distance of the route. The larger distance would mean a larger error since the objective is to get the shortest distance. If the error of the neighboring list is less than the current list, then the neighboring list becomes the current list and the temperature is cooled by the alpha fraction that was tuned for the best result at the beginning. If the neighboring route is larger then the current route, then using the error quantities a probability is created to calculate the chances of accepting the worse solution $e^{(f(c)-f(n))/(\,current\ temp)}$. This acceptance probability is compared to a randomly created number in each iteration. If the randomly generated number is less than the acceptance probability then the worse solution is accepted and the temperature is cooled and repeats itself. The algorithm stops when the maximum number of iterations are completed, which for this algorithm is 2,500 iterations. At the rate the temperature is set to cool at, the temperature tends to cool to its limit sooner than it takes the iterations to hit their maximum. When the temperature reaches its most

minimum temperature it stays at this minimum instead of continuously cooling to an incredibly small number.

Simulated annealing is a powerful optimization algorithm that can find high-quality solutions to difficult problems. By exploring the search space with a combination of random moves and gradual refinement, it can escape local optima and find globally optimal solutions.

In the algorithm I have written for this problem, after every 250 iterations, the program displays the current error value which is the random probability that the current iteration is accepted or denied. This is displayed alongside the temperature cooling value. As simulated annealing goes, the temperature will start at a manually tuned value and cool by a factor of alpha each time the algorithm accepts the new list up until a certain point; for this algorithm, I chose 10^{-4}. To best calculate the distances between each city I set the distance values to be in Kilometers. Below are the displayed values when running the algorithm I developed.

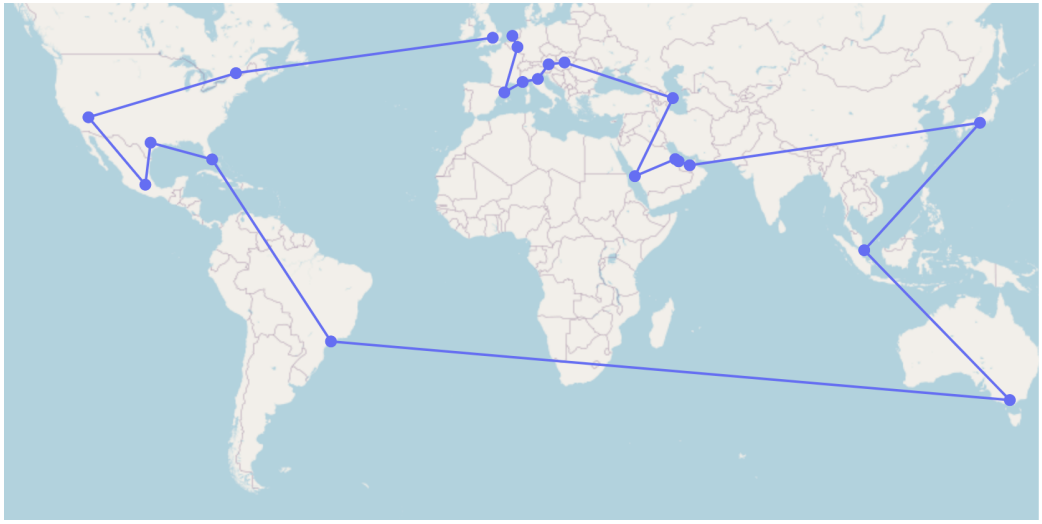
```
Current F1 calendar map
Current total distance of F1 Calendar Map = 166062.8
max_iter = 2500
start_temperature = 10000.0
alpha = 0.99
iter =      0 | curr error = 166041.78 | temperature = 10000.0000
iter =    250 | curr error = 110820.01 | temperature =   810.5852
iter =    500 | curr error = 92057.41 | temperature =    65.7048
iter =    750 | curr error = 74233.43 | temperature =     5.3259
iter =   1000 | curr error = 74233.43 | temperature =     0.4317
iter =   1250 | curr error = 73648.29 | temperature =     0.0350
iter =   1500 | curr error = 73648.29 | temperature =     0.0028
iter =   1750 | curr error = 73200.99 | temperature =     0.0002
iter =   2000 | curr error = 71925.33 | temperature =     0.0001
iter =   2250 | curr error = 71925.33 | temperature =     0.0001

Best solution found:
Zandvoort, Spa Francorchamps, Barcelona, Monte Carlo, Imola, Spielberg, Budapest, Baku, Jeddah, Sakhir, Lusail, Abu D
habi, Suzuka, Singapore, Melbourne, Sao Paulo, Miami, Austin, Mexico City, Las Vegas, Montreal, Silverstone

Total distance = 71946.3
```

This example happens to be one of the better instances the algorithm returned. However, since simulated annealing commits to the trade-off of worse solutions at times, the algorithm can produce only one of the potential local minimum values. This means that at times the total distance of the new solution found can be upwards of one hundred thousand kilometers. Even

though this is still shorter than the current route, it still is not the best solution. To compare on a visual scale how much better this proposed solution is proposed here is the best solution mapped.



This route cuts travel down by 94,116.5 kilometers which can make the formula one season significantly more profitable and easier to manage for the hundreds of workers for each of the 10 teams to set up and break down each pit lane. With most of the optimized routes, there becomes the option to transport this cargo via boat or truck which would also minimize the cost and emissions to transport the equipment. It is also important to note that the map connecting each city is not a representation of the travel between each city. In some representations, the algorithm will display a connection from Las Vegas to Tokyo by zooming across the Atlantic Ocean, which is the least likely route a plane might take to get between those two cities.

Below is the code used for this algorithm, the starting order of the list is the current order of the Formula One calendar. In order to run this code, the numpy and geopy python packages must be installed.

```
import numpy as np
from numpy import random
from geopy import distance
```

```
import plotly.graph_objects as go


circuit_list = [{"lon": 50.512, "lat": 26.031, "location": "Sakhir",
"name": "Bahrain International Circuit"},
                {"lon": 39.104, "lat": 21.632, "location": "Jeddah",
"name": "Jeddah Corniche Circuit"},
                {"lon": 144.970, "lat": -37.846, "location": "Melbourne",
"name": "Albert Park Circuit"},
                {"lon": 49.842, "lat": 40.369, "location": "Baku", "name":
"Baku City Circuit"},
                {"lon": -80.239, "lat": 25.958, "location": "Miami",
"name": "Miami International Autodrome"},
                {"lon": 11.713, "lat": 44.341, "location": "Imola", "name":
"Autodromo Enzo e Dino Ferrari"},
                {"lon": 7.429, "lat": 43.737, "location": "Monte Carlo",
"name": "Circuit de Monaco"},
                {"lon": 2.259, "lat": 41.569, "location": "Barcelona",
"name": "Circuit de Barcelona-Catalunya"},
                {"lon": -73.525, "lat": 45.506, "location": "Montreal",
"name": "Circuit Gilles-Villeneuve"},
                {"lon": 14.761, "lat": 47.223, "location": "Spielberg",
"name": "Red Bull Ring"},
                {"lon": -1.017, "lat": 52.072, "location": "Silverstone",
"name": "Silverstone Circuit"},
                {"lon": 19.250, "lat": 47.583, "location": "Budapest",
"name": "Hungaroring"},
                {"lon": 5.971, "lat": 50.436, "location": "Spa
Francorchamps", "name": "Circuit de Spa-Francorchamps"},
                {"lon": 4.541, "lat": 52.389, "location": "Zandvoort",
"name": "Circuit Zandvoort"},
                {"lon": 103.859, "lat": 1.291, "location": "Singapore",
"name": "Marina Bay Street Circuit"},
                {"lon": 136.534, "lat": 34.844, "location": "Suzuka",
"name": "Suzuka International Racing Course"},
                {"lon": 51.454, "lat": 25.49, "location": "Lusail", "name":
"Losail International Circuit"},
                {"lon": -97.633, "lat": 30.135, "location": "Austin",
"name": "Circuit of the Americas"},
                {"lon": -99.091, "lat": 19.402, "location": "Mexico City",
"name": "Autódromo Hermanos Rodríguez"},
                {"lon": -46.698, "lat": -23.702, "location": "Sao Paulo",
"name": "Autódromo José Carlos Pace - Interlagos"},
```

```
                {"lon": -115.168, "lat": 36.116, "location": "Las Vegas",
"name": "Las Vegas Street Circuit"},
                {"lon": 54.601, "lat": 24.471, "location": "Abu Dhabi",
"name": "Yas Marina Circuit"}
               ]

# compute total distance of travelled between circuits using the current
calendar route
def total_distance(circuit_list):
    d = 0.0 # in Km
    n = len(circuit_list)
    nxt = 0
    for i in range(n):
        if i == n -1:
            b = 0
        else:
            b = i + 1
        cordsA = (circuit_list[i]["lat"], circuit_list[i]["lon"])
        cordsB = (circuit_list[b]["lat"], circuit_list[b]["lon"])
        if cordsA < cordsB:
            diff =  distance.distance(cordsA, cordsB).km *1.0
            d += diff
            circuit_list[i]["distance"] = round(diff,3)
        else:
            diff = distance.distance(cordsB, cordsA).km *1.5
            d += diff
            circuit_list[i]["distance"] = round(diff,3)
    return round(d,3)

#calculate an error to find the most optimal distance
def error(circuit_list):
    n = len(circuit_list)
    d = total_distance(circuit_list)
    min_dist = n - 1
    return d - min_dist

#swaps two random indices from the first route making a new route
def swap(circuit_list):
    n = len(circuit_list)
    shuffle = np.copy(circuit_list)
    i, j = np.random.randint(n), np.random.randint(n)
    hold = shuffle[i]
    shuffle[i],shuffle[j] = shuffle[j], hold
```

```
    return shuffle

def printVal(value, circ):
    val = []
    for i in range(len(circ)):
        val.append(circ[i][value])
    return ", ".join(val)

def solve(circuit_list, max_iter, start_temp, alpha, rnd):
    curr_temp = start_temp
    circ = circuit_list
    err = error(circ)
    iteration = 0
    interval = (int)(max_iter / 10)

    while iteration < max_iter and err > 0.0:
        adj_circ = swap(circ)
        adj_err = error(adj_circ)

        if adj_err < err:
            circ, err = adj_circ, adj_err
        else:
            accept_p = np.exp((err-adj_err) / curr_temp)
            p = rnd.random()
            if p < accept_p:
                circ, err = adj_circ, adj_err

        if iteration % interval == 0:
            print("iter = %6d | curr error = %7.2f | temperature = %10.4f "
% (iteration, err, curr_temp))

        if curr_temp < 0.0001:
            curr_temp = 0.0001
        else:
            curr_temp = curr_temp * alpha
        iteration += 1
    return circ

def mapCirc(circ):
    lonC = []
    latC = []
    locC = []
```

```
    for i in range(len(circ)):
        lonC.append(circ[i]["lon"])
        latC.append(circ[i]["lat"])
        locC.append(circ[i]["location"])

    fig = go.Figure(go.Scattermapbox(
        mode = "markers+lines",
        lon = lonC,
        lat = latC,
        marker = {'size': 10}))

    fig.update_layout(
        margin ={'l':0,'t':0,'b':0,'r':0},
        mapbox = {
            'center': {'lon': 10, 'lat': 10},
            'style': "open-street-map",
            'center': {'lon': -20, 'lat': -20},
            'zoom': 1 })
    fig.update_traces(
        text = locC)
    fig.show()

def main():
    mapCirc(circuit_list)
    print("Current F1 calendar map")
    circuit_dist = total_distance(circuit_list)
    print("Current total distance of F1 Calendar Map = %0.1f" %
circuit_dist)
    rnd = np.random.RandomState(4)
    max_iter = 2500
    start_temperature = 10000.0
    alpha = 0.99
    print("max_iter = %d " % max_iter)
    print("start_temperature = %0.1f " % start_temperature)
    print("alpha = %0.2f " % alpha)
    best = solve(circuit_list, max_iter, start_temperature, alpha,rnd)
    print("\nBest solution found: ")
    bestLoc = printVal('location', best)
    print(bestLoc)
    dist = total_distance(best)
    print("\nTotal distance = %0.1f " % dist)
    mapCirc(best)
```

```
if __name__ == "__main__":
    main()
```