確率解析特論 I 最終レポート

情報数学班

目次

1 はじめに

焼きなまし法を用いて、東京を起点に世界の首都を一度ずつ訪問し、再び東京へと戻る総移動距離の大域最適解を求めた。本レポートでは、使用したプログラムの紹介と説明、結果と考察について述べる。

2 プログラムの紹介と説明

本節では、焼きなまし法において重要な役割を果たす評価関数、温度関数、および焼きなまし法のコードの関連部分を紹介する。コード全体の説明については付録の章で解説する。

温度関数は以下の 2 種類を用いた。

一つ目は $T(t) = c_1 e^{-c_2 t}$ であり、対応するコードは以下の通りである。

```
1 def cooling_function1(T, init_temp, end_temp, start_time, max_time):
2     res = end_temp + (init_temp - end_temp)*math.exp(-((time() - start_time)/max_time)*5) #c1*
          exp(-c2*t)の実装
3     return res
```

二つ目は $T(t) = \frac{c}{\log(t+2)}$ であり、対応するコードは以下の通りである。

```
def cooling_function2(T, init_temp, end_temp, start_time, max_time):
    t = (time() - start_time)/max_time
    res = end_temp + (init_temp - end_temp) / math.log(t + 2)
    return res
```

評価関数の説明に先立ち、距離の計算方法について説明する。距離の計算には球面三角法を用いており、これにより円弧上の距離を求めた。コードは以下の通りである。

```
def spherical_trigonometry(lat_p, lon_p, lat_q, lon_q):
    res = np.sin(lat_p) * np.sin(lat_q) + np.cos(lat_p) * np.cos(lat_q) * np.cos(lon_q - lon_p)
    res = np.arccos(res)
    R = 6378 # 地球の半径 (km)
    res *= R

    return res
```

評価関数では都市の位置を入れ替えた際の移動距離の変化を計算する。この関数では入れ替え前と入れ替え後の距離をそれぞれ計算している。そして、その差を評価基準として用いる。対応するコードは以下の通りである。

```
def cost_function(x, y, Countries_list):
    now_cost, next_cost = 0, 0
    x_pre_country, x_country, x_next_country = Countries_list[x-1:x+2]
    y_pre_country, y_country, y_next_country = Countries_list[y-1:y+2]
    now_cost = spherical_trigonometry(lat[x_pre_country], lon[x_pre_country], lat[x_country],
        lon[x_country])\
            + spherical_trigonometry(lat[x_country], lon[x_country], lat[x_next_country], lon[
                x_next_country])\
            + spherical_trigonometry(lat[y_pre_country], lon[y_pre_country], lat[y_country], lon
                [y_country])\
            + spherical_trigonometry(lat[y_country], lon[y_country], lat[y_next_country], lon[
                y_next_country])
    next_cost = spherical_trigonometry(lat[x_pre_country], lon[x_pre_country], lat[y_country],
        lon[y_country])\
            + spherical_trigonometry(lat[y_country], lon[y_country], lat[x_next_country], lon[
                x_next_country])\
            + spherical_trigonometry(lat[y_pre_country], lon[y_pre_country], lat[x_country], lon
                [x_country])\
            + spherical_trigonometry(lat[x_country], lon[x_country], lat[y_next_country], lon[
                y_next_country])
    return now_cost, next_cost
```

3 結果

まず、実行方法について説明する。実行時間は 2 秒と設定し、初期状態から焼きなまし法を適用した。この時点での総距離は 1,424,968.4161400648 km である。なお、地球儀上にプロットした結果は、以下の URL から確認できる。

URL:https://belka-247d8560.github.io/StochasticAnalysis/template/init_root

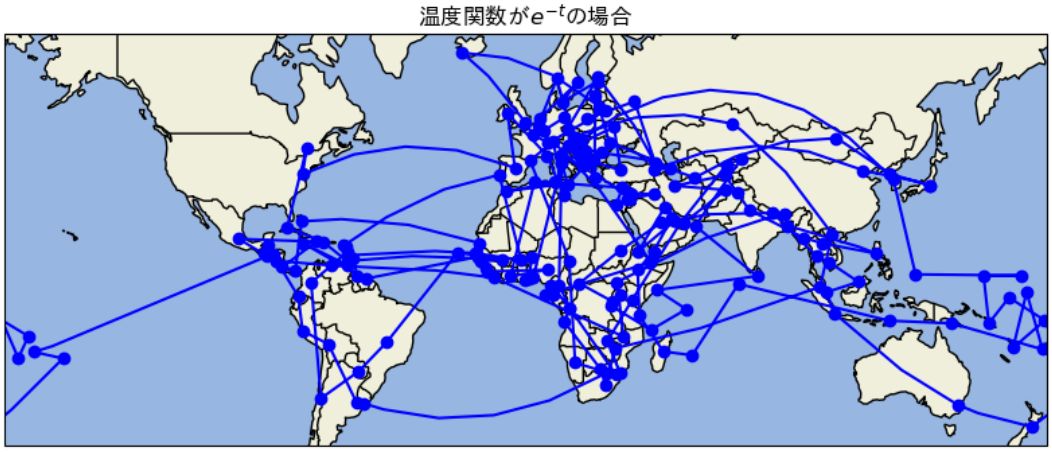
3.1 温度関数の比較

摂動は、経路から 2 都市をランダムに選び、その 2 都市を入れ替える方法を使用した。この際、異なる温度関数によって結果がどのように変化するかを観察した。

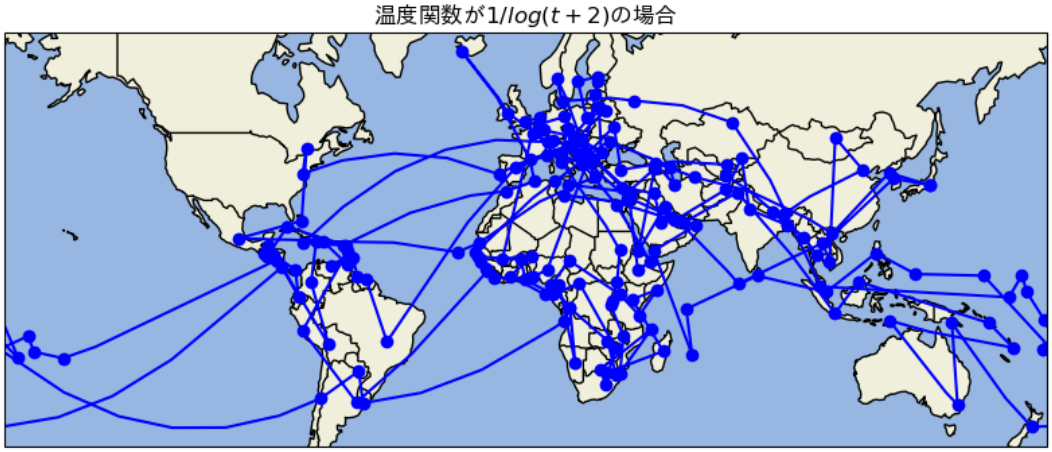
表 1　焼きなまし法を 100 回行った結果

温度関数	平均結果	最小の距離
e^{-t}	470040.5732654408	386820.6770684467
$\frac{1}{\log(t+2)}$	460445.19600290677	384470.9621612602

焼きなまし法を各温度関数で 100 回ずつ実行した結果が表 1 に示されている。この結果から、今回の条件では温度関数によって劇的な結果の改善が見られないことがわかった。



URL:https://belka-247d8560.github.io/StochasticAnalysis/template/temp1



URL:/https://belka-247d8560.github.io/StochasticAnalysis/template/temp2

3.2 摂動の比較

次に、摂動の選び方による結果の違いを検証した。温度関数としては e^{-t} を使用した。摂動の選び方として以下の 3 つを用いた。

1. 2 都市をランダムに選び、位置を入れ替える。
2. 都市とインデックスをランダムに選び、都市をインデックスに挿入する。
3. 2 都市をランダムに選び、その間の順路を逆順にする。

表 2　焼きなまし法を 100 回行った結果

摂動	平均結果	最小の距離	摂動回数
1 の方法	461837.5959320358	400347.08054371784	40418
2 の方法	435929.36188207375	352880.4459965957	52523
3 の方法	265037.0510349886	237233.0281093583	75053

摂動の選び方による結果の違いを検証した結果、以下のような傾向が確認された。

1 の方法の場合

この方法では、最小距離も他の方法に比べて大きく、探索の効率がやや低いことが示唆される。

2 の方法の場合

この方法では、摂動による改善が比較的効果的であり、平均距離と最小距離の両方が方法 1 よりも短縮されている。摂動回数も多いため、より広範囲に探索が行われたと考えられる。

3 の方法の場合

この方法では、最も大きな改善が見られ、平均距離と最小距離がいずれも他の方法に比べて大幅に短縮されている。摂動回数も最も多く、探索が効果的に行われたことがわかる。

総じて、3 つの摂動方法の中では「方法 3: 2 都市間の順路を逆順にする」が最も効果的であり、他の方法と比べて顕著に短い経路を見つけることができた。この結果から、摂動の選び方が焼きなまし法の性能に大きく影響することが確認できた。

URL:https://belka-247d8560.github.io/StochasticAnalysis/template/two-city-swap

URL:https://belka-247d8560.github.io/StochasticAnalysis/template/insert-swap

URL:https://belka-247d8560.github.io/StochasticAnalysis/template/two-city-reverse

4 付録

以下に今回使用したコードの全体を記載する。コードは https://github.com/Belka-247d8560/StochasticAnalysis にも掲載している。

```
"""
### ライブラリをインポートする上での注意
手元のPython環境次第ではインポートに失敗することがあります。その際はターミナルで
```
pip install -r requirements.txt
```
と試してください。
"""

# 必要なライブラリのインポート
import pandas as pd
import chardet
import numpy as np
import math
import random
from tqdm import tqdm
from time import time
import matplotlib.pyplot as plt
import japanize_matplotlib
import plotly.graph_objs as go
import plotly.io as pio
import cartopy.crs as ccrs
import cartopy.feature as cfeature

#データの読み込み
file_path = "capital_cities_2024.csv"

with open(file_path, 'rb') as f:
    result = chardet.detect(f.read())

encoding = result['encoding']
df = pd.read_csv(file_path, encoding=encoding)
df = df[["ja_ctry", "x", "y"]]
df = df.rename(columns={"ja_ctry":"国", "x":"lon", "y":"lat"})
#国・地域名をリストとして取得
Countries = df["国"].to_list()
#日本スタート、日本終了のリストにする。
Countries.remove('日本国')
Countries = ['日本国'] + Countries + ['日本国']
#緯度、経度を辞書として保存(コードの書きやすさのため)
lat, lon = dict(), dict()
for index, row in df.iterrows():
    lat[row["国"]] = np.radians(row["lat"])
    lon[row["国"]] = np.radians(row["lon"])

#球面三角法の定義
def spherical_trigonometry(lat_p, lon_p, lat_q, lon_q):
    res = np.sin(lat_p) * np.sin(lat_q) + np.cos(lat_p) * np.cos(lat_q) * np.cos(lon_q - lon_p)
    res = np.arccos(res)
    R = 6378 # 地球の半径 (km)
    res *= R

    return res

#総距離の計算関数の定義
def calculating_the_total_distance(Countries_list):
    N = len(Countries_list)
    res = 0
    for i in range(N-1):
        now_country, next_country = Countries_list[i], Countries_list[i+1]
        res += spherical_trigonometry(lat[now_country], lon[now_country], lat[next_country], lon
            [next_country])
    return res
```

```
63
64 #メルカトル図法での結果の描画
65 def mercator_plot(Countries_list, filename):
66     lats = [np.degrees(lat[Country]) for Country in Countries_list]
67     lons = [np.degrees(lon[Country]) for Country in Countries_list]
68     plt.figure(figsize=(10, 6))
69     ax = plt.axes(projection=ccrs.Mercator())
70     ax.add_feature(cfeature.COASTLINE)
71     ax.add_feature(cfeature.BORDERS)
72     ax.add_feature(cfeature.LAND)
73     ax.add_feature(cfeature.OCEAN)
74     plt.plot(lons, lats, marker='o', linestyle='-', color='b', transform=ccrs.Geodetic())
75     plt.xlabel("経度")
76     plt.ylabel("緯度")
77     plt.title(filename)
78     plt.grid(True)
79     plt.show()
80
81 #地球儀上への結果の描画
82 def globe_plot(Countries_list, filename=None):
83     lats = [df.loc[df['国']==Country, 'lat'].values[0] for Country in Countries_list]
84     lons = [df.loc[df['国']==Country, 'lon'].values[0] for Country in Countries_list]
85
86     line = go.Scattergeo(
87         lon=lons,
88         lat=lats,
89         mode='lines+markers',
90         line=dict(color='yellow', width=2),
91         marker=dict(size=8, color='red')
92     )
93
94     layout = go.Layout(
95         geo=dict(
96             projection=dict(type='orthographic'),
97             showcountries=True,
98         ),
99         margin=dict(r=10, l=10, b=10, t=10)
100    )
101
102    fig = go.Figure(data=[line], layout=layout)
103    pio.show(fig)
104
105    if filename:
106        pio.write_html(fig, file = filename)
107
108 #初期値の総距離の計算、描画
109 print("初期値:", *Countries)
110 print("総距離:", calculating_the_total_distance(Countries), "km")
111 # メルカトル図法での初期値の描画
112 mercator_plot(Countries, '初期値の経路')
113 # 地球儀上への初期値の描画
114 globe_plot(Countries, 'init_root')
115
116 # 温度関数による結果の比較
117 #温度関数の定義
118 def cooling_function1(init_temp, end_temp, start_time, max_time):
119     T_max = math.exp(0)
120     T_min  = math.exp(-max_time)
121     T = math.exp(-(time() - start_time))
122     res = end_temp  + (init_temp - end_temp)*((T - T_min) / (T_max - T_min))
123     return res
124 def cooling_function2(init_temp, end_temp, start_time, max_time):
125     T_max = 1/math.log(2)
126     T_min  = 1/math.log(max_time + 2)
127     T = 1/math.log((time() - start_time) + 2)
128     res = end_temp  + (init_temp - end_temp)*((T - T_min) / (T_max - T_min))
129     return res
130
131 #評価関数
132 def cost_function(x, y, Countries_list):
133     now_cost, next_cost = 0, 0
134     x_pre_country, x_country, x_next_country = Countries_list[x-1:x+2]
135     y_pre_country, y_country, y_next_country = Countries_list[y-1:y+2]
```

```
136     now_cost = spherical_trigonometry(lat[x_pre_country], lon[x_pre_country], lat[x_country],
            lon[x_country])\
137             + spherical_trigonometry(lat[x_country], lon[x_country], lat[x_next_country], lon[
                x_next_country])\
138             + spherical_trigonometry(lat[y_pre_country], lon[y_pre_country], lat[y_country], lon
                [y_country])\
139             + spherical_trigonometry(lat[y_country], lon[y_country], lat[y_next_country], lon[
                y_next_country])
140     next_cost = spherical_trigonometry(lat[x_pre_country], lon[x_pre_country], lat[y_country],
            lon[y_country])\
141             + spherical_trigonometry(lat[y_country], lon[y_country], lat[x_next_country], lon[
                x_next_country])\
142             + spherical_trigonometry(lat[y_pre_country], lon[y_pre_country], lat[x_country], lon
                [x_country])\
143             + spherical_trigonometry(lat[x_country], lon[x_country], lat[y_next_country], lon[
                y_next_country])
144     return now_cost, next_cost
145
146 #焼きなまし法
147 def simulated_annealing(Countries_list, cooling_function, cost_function, init_temp, end_temp,
        start_time, max_time):
148     T = init_temp
149     while time() - start_time <= max_time:
150         a, b = random.choices(range(1, len(Countries_list)-1), k=2)
151         now_cost, next_cost = cost_function(a, b, Countries_list)
152         if next_cost < now_cost or random.random() < np.exp(-np.abs(now_cost - next_cost)/T):
153             Countries_list[a], Countries_list[b] = Countries_list[b], Countries_list[a]
154         T = cooling_function(init_temp, end_temp, start_time, max_time)
155     return Countries_list
156
157 #1つ目の温度関数で焼きなまし法を100回行う
158 min_result = []
159 min_distance = float('inf')
160 average_distance = 0
161 for i in tqdm(range(100)):
162     result = simulated_annealing(Countries_list=Countries.copy(), cooling_function=
            cooling_function1, cost_function=cost_function, init_temp=10000, end_temp=0.001,
            start_time=time(), max_time=2)
163     distance = calculating_the_total_distance(result)
164     if distance < min_distance:
165         min_distance = distance
166         min_result = result.copy()
167     average_distance += distance
168 print("最小の距離:", min_distance)
169 print("最小の距離での巡回順番:", *min_result)
170 print("平均:", average_distance/100)
171
172 # メルカトル図法での最小の結果の描画
173 mercator_plot(min_result, '温度関数が$e^{-t}$の場合')
174 # 地球儀上への最小の結果の描画
175 globe_plot(Countries, 'temp1')
176
177 #2つ目の温度関数で焼きなまし法を100回行う
178 min_result = []
179 min_distance = float('inf')
180 average_distance = 0
181 for i in tqdm(range(100)):
182     result = simulated_annealing(Countries_list=Countries.copy(), cooling_function=
            cooling_function2, cost_function=cost_function, init_temp=10000, end_temp=0.001,
            start_time=time(), max_time=2)
183     distance = calculating_the_total_distance(result)
184     if distance < min_distance:
185         min_distance = distance
186         min_result = result.copy()
187     average_distance += distance
188 print("最小の距離:", min_distance)
189 print("最小の距離での巡回順番:", *min_result)
190 print("平均:", average_distance/100)
191
192 # メルカトル図法での最小の結果の描画
193 mercator_plot(min_result, '温度関数が$1/log(t+2)$の場合')
194 # 地球儀上への最小の結果の描画
195 globe_plot(Countries, 'temp2')
```

```
196
197 # 摂動による結果の比較
198 def cooling_function(init_temp, end_temp, start_time, max_time):
199     T_max = math.exp(0)
200     T_min  = math.exp(-max_time)
201     T = math.exp(-(time() - start_time))
202     res = end_temp  + (init_temp - end_temp)*((T - T_min) / (T_max - T_min))
203     return res
204
205 ### 2都市をランダムに選び、入れ替える方法の場合
206 def cost_function(x, y, Countries_list):
207     now_cost, next_cost = 0, 0
208     x_pre_country, x_country, x_next_country = Countries_list[x-1:x+2]
209     y_pre_country, y_country, y_next_country = Countries_list[y-1:y+2]
210     now_cost = spherical_trigonometry(lat[x_pre_country], lon[x_pre_country], lat[x_country],
          lon[x_country])\
211             + spherical_trigonometry(lat[x_country], lon[x_country], lat[x_next_country], lon[
                  x_next_country])\
212             + spherical_trigonometry(lat[y_pre_country], lon[y_pre_country], lat[y_country], lon
                  [y_country])\
213             + spherical_trigonometry(lat[y_country], lon[y_country], lat[y_next_country], lon[
                  y_next_country])
214     next_cost = spherical_trigonometry(lat[x_pre_country], lon[x_pre_country], lat[y_country],
          lon[y_country])\
215             + spherical_trigonometry(lat[y_country], lon[y_country], lat[x_next_country], lon[
                  x_next_country])\
216             + spherical_trigonometry(lat[y_pre_country], lon[y_pre_country], lat[x_country], lon
                  [x_country])\
217             + spherical_trigonometry(lat[x_country], lon[x_country], lat[y_next_country], lon[
                  y_next_country])
218     return now_cost, next_cost
219
220 def simulated_annealing(Countries_list, cooling_function, cost_function, init_temp, end_temp,
      start_time, max_time):
221     T = init_temp
222     # cnt = 0
223     while time() - start_time <= max_time:
224         # cnt += 1
225         a, b = random.choices(range(1, len(Countries_list)-1), k=2)
226         now_cost, next_cost = cost_function(a, b, Countries_list)
227         if next_cost < now_cost or random.random() < np.exp(-np.abs(now_cost - next_cost)/T):
228             Countries_list[a], Countries_list[b] = Countries_list[b], Countries_list[a]
229         T = cooling_function(init_temp, end_temp, start_time, max_time)
230     # print('摂動回数:', cnt)
231     return Countries_list
232
233 #焼きなまし法を100回行う
234 min_result = []
235 min_distance = float('inf')
236 average_distance = 0
237 for i in tqdm(range(100)):
238     result = simulated_annealing(Countries_list=Countries.copy(), cooling_function=
          cooling_function, cost_function=cost_function, init_temp=10000, end_temp=0.001,
          start_time=time(), max_time=2)
239     distance = calculating_the_total_distance(result)
240     if distance < min_distance:
241         min_distance = distance
242         min_result = result.copy()
243     average_distance += distance
244 print("最小の距離:", min_distance)
245 print("最小の距離での巡回順番:", *min_result)
246 print("平均:", average_distance/100)
247
248 # メルカトル図法での最小の結果の描画
249 mercator_plot(min_result, '2都市の入れ替え')
250 # 地球儀上への最小の結果の描画
251 globe_plot(Countries, 'two-city-swap')
252
253 # 挿入での方法の場合
254 def cost_function(x, y, Countries_list):
255     now_cost, next_cost = 0, 0
256     x_pre_country, x_country, x_next_country = Countries_list[x-1:x+2]
257     y_pre_country, y_next_country = Countries_list[y:y+2]
```

```
258     now_cost = spherical_trigonometry(lat[x_pre_country], lon[x_pre_country], lat[x_country],
            lon[x_country])\
259             + spherical_trigonometry(lat[x_country], lon[x_country], lat[x_next_country], lon[
                x_next_country])\
260             + spherical_trigonometry(lat[y_pre_country], lon[y_pre_country], lat[y_next_country
                ], lon[y_next_country])
261     Countries_list.pop(x)
262     Countries_list.insert(y, x_country)
263     next_cost = spherical_trigonometry(lat[x_pre_country], lon[x_pre_country], lat[
            x_next_country], lon[x_next_country])\
264             + spherical_trigonometry(lat[y_pre_country], lon[y_pre_country], lat[x_country], lon
                [x_country])\
265             + spherical_trigonometry(lat[x_country], lon[x_country], lat[y_next_country], lon[
                y_next_country])
266     return now_cost, next_cost
267
268 def simulated_annealing(Countries_list, cooling_function, cost_function, init_temp, end_temp,
        start_time, max_time):
269     T = init_temp
270     # cnt = 0
271     while time() - start_time <= max_time:
272         # cnt += 1
273         a, b = random.choices(range(1, len(Countries_list)-1), k=2)
274         now_cost, next_cost = cost_function(a, b, Countries_list)
275         if not (next_cost < now_cost or random.random() < np.exp(-np.abs(now_cost - next_cost)/T
                )):
276             Country = Countries_list.pop(b)
277             Countries_list.insert(a, Country)
278         T = cooling_function(init_temp, end_temp, start_time, max_time)
279     # print('摂動回数:', cnt)
280     return Countries_list
281
282 #焼きなまし法を100回行う
283 min_result = []
284 min_distance = float('inf')
285 average_distance = 0
286 for i in tqdm(range(100)):
287     result = simulated_annealing(Countries_list=Countries.copy(), cooling_function=
            cooling_function, cost_function=cost_function, init_temp=10000, end_temp=0.001,
            start_time=time(), max_time=2)
288     distance = calculating_the_total_distance(result)
289     if distance < min_distance:
290         min_distance = distance
291         min_result = result.copy()
292     average_distance += distance
293 print("最小の距離:", min_distance)
294 print("最小の距離での巡回順番:", *min_result)
295 print("平均:", average_distance/100)
296
297 # メルカトル図法での最小の結果の描画
298 mercator_plot(min_result, '挿入による入れ替え')
299 # 地球儀上への最小の結果の描画
300 globe_plot(Countries, 'insert-swap')
301
302 # 順路を逆順にする方法の場合
303 def cost_function(x, y, Countries_list):
304     now_cost, next_cost = 0, 0
305     x_pre_country, x_country = Countries_list[x-1:x+1]
306     y_country, y_next_country = Countries_list[y:y+2]
307     now_cost = spherical_trigonometry(lat[x_pre_country], lon[x_pre_country], lat[x_country],
            lon[x_country])\
308             + spherical_trigonometry(lat[y_country], lon[y_country], lat[y_next_country], lon[
                y_next_country])
309     next_cost = spherical_trigonometry(lat[x_pre_country], lon[x_pre_country], lat[y_country],
            lon[y_country])\
310             + spherical_trigonometry(lat[x_country], lon[x_country], lat[y_next_country], lon[
                y_next_country])
311     return now_cost, next_cost
312
313 def simulated_annealing(Countries_list, cooling_function, cost_function, init_temp, end_temp,
        start_time, max_time):
314     T = init_temp
315     # cnt = 0
```

```
316         while time() - start_time <= max_time:
317             # cnt += 1
318             a, b = random.choices(range(1, len(Countries_list)-1), k=2)
319             now_cost, next_cost = cost_function(a, b, Countries_list)
320             if next_cost < now_cost or random.random() < np.exp(-np.abs(now_cost - next_cost)/T):
321                 Countries_list[a:b+1] = Countries_list[a:b+1][::-1]
322             T = cooling_function(init_temp, end_temp, start_time, max_time)
323         # print('摂動回数:', cnt)
324         return Countries_list
325
326 #焼きなまし法を100回行う
327 min_result = []
328 min_distance = float('inf')
329 average_distance = 0
330 for i in tqdm(range(100)):
331         result = simulated_annealing(Countries_list=Countries.copy(), cooling_function=
                cooling_function, cost_function=cost_function, init_temp=10000, end_temp=0.001,
                start_time=time(), max_time=2)
332         distance = calculating_the_total_distance(result)
333         if distance < min_distance:
334             min_distance = distance
335             min_result = result.copy()
336         average_distance += distance
337 print("最小の距離:", min_distance)
338 print("最小の距離での巡回順番:", *min_result)
339 print("平均:", average_distance/100)
340
341 # メルカトル図法での最小の結果の描画
342 mercator_plot(min_result, '2都市間の順路を逆順')
343 # 地球儀上への最小の結果の描画
344 globe_plot(Countries, 'two-city-reverse')
```